

ACT-R 7 Reference Manual

Dan Bothell

Includes material adapted from the ACT-R 4.0 manual by Christian Lebiere, documentation on the perceptual motor components by Mike Byrne and the Introduction is a shortened version of the ACT-R description written by Raluca Budiu for the ACT-R web site.

Notice

ACT-R 7 is not completely backwards compatible with ACT-R 6.0. There is a large conceptual difference between ACT-R 7 and 6.0 (chunks are no longer categorized into fixed types), but programmatically the two systems are still very similar. Many models written for ACT-R 6.0 will still run as-is in ACT-R 7, but not every model will. This document does not include a comprehensive discussion of the differences, but there will be some notes in sections where the differences are significant. For a discussion of the differences there is a set of slides available with the software and on the ACT-R website.

Table of Contents

Notice.....	3
Table of Contents.....	4
Preface.....	17
Introduction.....	18
Document Overview.....	20
General Software Description	21
Case sensitivity.....	21
Functions vs. Macros.....	22
Compatibility issues.....	22
Notations in the Documentation.....	23
Command Syntax.....	23
Examples.....	23
ACT-R Software Distribution.....	25
Distribution Contents.....	26
Subdirectories.....	26
commands.....	26
core-modules.....	26
devices.....	26
docs.....	26
environment.....	28
examples.....	28
extras.....	28
framework.....	28
modules.....	28
other-files.....	28
support.....	29
tools.....	29
tutorial.....	29
user-loads.....	29
Loading and Running the ACT-R System.....	30
Compiler Optimizations.....	30
Logical Host.....	31
Load order.....	31
Recompiling.....	32
Packaging.....	32
Clean.....	33
Packaged.....	33
Requiring files.....	33
Overall Software Design.....	34
Model files.....	35
Meta-process.....	37
Commands.....	37
clear-all.....	37
reset.....	38
reload.....	39

mp-time.....	40
mp-time-ms.....	40
Events.....	42
Commands.....	42
mp-show-queue	42
mp-queue-count.....	43
mp-show-waiting.....	43
mp-modules-events.....	44
Module.....	46
Commands.....	46
mp-print-versions.....	46
all-module-names.....	47
Buffers.....	49
Commands.....	50
buffers.....	50
buffer-chunk.....	50
buffer-status.....	52
Models.....	55
Commands.....	56
define-model.....	56
delete-model.....	57
Chunks & Chunk-types.....	59
Default Chunk-types.....	60
chunk.....	60
constant-chunks.....	60
query-slots.....	60
clear.....	60
Default Chunks.....	61
Commands.....	61
Chunk-type Commands.....	61
chunk-type.....	61
pprint-chunk-type.....	64
chunk-type-p.....	65
all-chunk-type-names.....	65
chunk-type-possible-slot-names-fct.....	66
Chunk Commands.....	68
define-chunks.....	68
pprint-chunks & pprint-chunks-plus.....	70
chunk-p.....	73
chunk-documentation.....	73
chunk-slot-value.....	74
set-chunk-slot-value.....	75
mod-chunk.....	77
copy-chunk.....	78
chunk-copied-from.....	79
chunks.....	81
chunk-slot-equal.....	81
equal-chunks.....	82
eq-chunks.....	84

delete-chunk.....	85
purge-chunk.....	86
merge-chunks.....	87
create-chunk-alias.....	89
true-chunk-name.....	90
normalize-chunk-names.....	91
General Parameters.....	94
Commands.....	95
sgp.....	95
get-parameter-default-value.....	97
with-parameters.....	97
System Parameters.....	100
Commands.....	100
ssp.....	100
Printing and Output.....	102
Model Output.....	102
Command Output.....	102
Warnings.....	102
Commands.....	103
model-output	103
meta-p-output.....	104
command-output.....	105
no-output.....	106
capture-model-output.....	106
print-warning.....	107
model-warning.....	108
one-time-model-warning.....	109
suppress-warnings.....	110
Running the system.....	112
Commands.....	112
run.....	112
run-full-time.....	114
run-until-time.....	116
run-until-condition.....	118
run-n-events.....	120
run-step.....	122
Scheduling Events.....	125
Details of events.....	125
time.....	125
priority.....	126
action.....	126
parameters.....	126
model.....	126
module.....	126
destination.....	126
details.....	126
output.....	127
Event Accessors.....	127
General Event Commands.....	128

event-displayed-p.....	128
format-event.....	129
Scheduling Commands.....	129
schedule-event.....	130
schedule-event-relative.....	132
schedule-event-now.....	133
schedule-event-after-module.....	134
schedule-event-after-change.....	136
schedule-periodic-event.....	138
schedule-break.....	139
schedule-break-relative.....	140
schedule-break-after-module.....	141
schedule-break-after-all.....	144
delete-event.....	145
Event Hooks.....	145
add-pre-event-hook.....	146
add-post-event-hook.....	147
delete-event-hook.....	149
About the Included Modules.....	151
Printing module.....	152
Parameters.....	152
:cbct.....	152
:cmdt.....	152
:model-warnings.....	153
:trace-detail.....	153
:trace-filter.....	153
:v.....	154
Naming Module.....	155
Parameters.....	155
:ncnar.....	155
Commands.....	157
new-name.....	157
release-name.....	159
new-symbol.....	160
Random module.....	162
Parameters.....	162
:randomize-time.....	162
:seed.....	162
Commands.....	163
act-r-random.....	163
act-r-noise.....	164
randomize-time.....	165
randomize-time-ms.....	166
Buffer trace module.....	169
Parameters.....	171
:buffer-trace.....	171
:buffer-trace-hook.....	171
:buffer-trace-step.....	171
:save-buffer-trace.....	171

:traced-buffers.....	171
Commands.....	172
get-current-buffer-trace.....	172
add-buffer-trace-notes.....	175
Central Parameters Module.....	177
Parameters.....	177
:er.....	177
:esc.....	178
:ol.....	178
System Parameters.....	178
:starting-parameters.....	178
Commands.....	179
register-subsymbolic-parameters.....	179
The Procedural System.....	180
Procedural Module.....	181
Conflict Resolution.....	181
Parameters.....	182
:crt.....	183
:cst.....	183
:cycle-hook.....	183
:dat.....	183
:do-not-harvest.....	184
:lhst.....	184
:ppm.....	184
:ppm-hook.....	185
:rhst.....	185
:style-warnings.....	186
:use-tree.....	186
:vpft.....	186
Production buffer.....	186
Commands.....	187
p/define-p/p*/define-p*.....	187
all-productions.....	205
pp.....	206
pbreak/punbreak.....	208
pdisable/penable.....	209
whynot.....	210
production-firing-only.....	213
un-delay-conflict-resolution.....	214
clear-productions.....	215
declare-buffer-usage.....	215
Utility module.....	218
Parameters.....	218
:alpha.....	218
:egs.....	219
:iu.....	219
:nu.....	219
:reward-hook.....	219
:ul.....	220

:ult.....	220
:ut.....	220
:utility-hook.....	220
:utility-offsets.....	221
Commands.....	221
trigger-reward.....	221
spp.....	223
Production Compilation Module.....	228
Parameters.....	230
:epl.....	230
:pct.....	230
:tt.....	230
Commands.....	230
show-compilation-buffer-types.....	230
compilation-buffer-type.....	231
specify-compilation-buffer-type.....	232
Goal Module.....	234
Goal buffer.....	234
Queries	234
Requests.....	234
Modification Requests.....	235
Commands.....	235
goal-focus	236
mod-focus.....	237
Imaginal Module.....	240
Parameters	240
:imaginal-delay	240
:vidt.....	240
Chunk-types & Chunks.....	240
Imaginal buffer.....	240
Requests.....	241
Modification requests.....	242
Imaginal-action buffer.....	242
Requests.....	243
Commands.....	245
set-imaginal-free.....	245
set-imaginal-error.....	245
Declarative module.....	247
Activation.....	248
Base-level.....	248
Spreading Activation.....	250
Partial Matching.....	251
Noise.....	252
Retrieval time.....	253
Declarative finsts.....	253
Parameters.....	254
:act.....	254
:activation-offsets.....	255
:ans.....	255

:bl-hook.....	255
:blc.....	255
:bll.....	255
:chunk-add-hook.....	256
:chunk-merge-hook.....	256
:declarative-finst-span.....	256
:declarative-num-finsts.....	257
:declarative-stuffing.....	257
:le.....	257
:lf.....	257
:mas	257
:md.....	257
:mp.....	258
:ms	258
:noise-hook.....	258
:nsji	258
:partial-matching-hook.....	258
:pas.....	258
:retrieval-request-hook.....	259
:retrieval-set-hook.....	259
:retrieved-chunk-hook.....	259
:rt.....	260
:sact.....	260
:sim-hook.....	260
:sji-hook.....	261
:spreading-hook.....	261
:w-hook.....	261
Retrieval buffer.....	261
Queries.....	261
Requests.....	262
Commands.....	263
add-dm.....	263
dm.....	265
sdm.....	266
print-dm-finsts.....	269
sdp.....	270
sji/add-sji.....	277
similarity/set-similarities.....	279
get-base-level/set-base-levels/set-all-base-levels.....	282
clear-dm.....	285
reset-declarative-finsts.....	285
merge-dm.....	286
print-activation-trace.....	288
print-chunk-activation-trace.....	290
saved-activation-history.....	292
whynot-dm.....	294
simulate-retrieval-request.....	297
Perceptual & Motor modules.....	300
The Device Module.....	301

Parameters.....	301
:mouse-fitts-coeff.....	301
:needs-mouse.....	302
:pixels-per-inch.....	302
:process-cursor.....	302
:show-focus.....	302
:stable-loc-names.....	302
:trace-mouse.....	303
:viewing-distance.....	303
:vwt.....	303
Commands.....	303
install-device.....	303
current-device.....	304
get-mouse-trace.....	305
model-generated-action.....	305
Vision module.....	307
The model's visual world.....	307
The Where System.....	307
Finsts.....	308
The What System.....	308
Re-encoding	309
Scene change.....	309
Tracking	310
Parameters	310
:auto-attend.....	310
:delete-visicon-chunks.....	310
:optimize-visual	311
:overstuff-visual-location.....	311
:scene-change-threshold.....	311
:test-feats.....	311
:unstuff-visaul-location.....	312
:visual-attention-latency.....	312
:visual-finst-span.....	312
:visual-movement-tolerance.....	312
:visual-num-finsts.....	313
:visual-onset-span.....	313
Visual-location buffer.....	313
Queries	313
Requests.....	314
Visual buffer.....	318
Queries.....	319
Requests.....	320
Chunks & Chunk-types.....	324
Commands.....	325
proc-display.....	325
print-visicon.....	327
remove-visual-finsts.....	327
set-visloc-default.....	329
add-word-characters.....	330

set-visual-center-point.....	332
set-char-feature-set.....	333
Audio module.....	334
Auditory world.....	334
The Where System.....	335
The What System.....	335
Parameters	335
:digit-detect-delay	335
:digit-duration.....	336
:digit-recode-delay	336
:overstuff-aural-location.....	336
:sound-decay-time	336
:tone-detect-delay.....	336
:tone-recode-delay	336
:unstuff-aural-location.....	336
Aural-location buffer.....	337
Queries	337
Requests.....	338
Aural buffer.....	341
Queries.....	341
Requests.....	342
Chunks & Chunk-types.....	344
Commands.....	344
new-digit-sound/new-tone-sound/new-other-sound/new-word-sound.....	345
print-audicon.....	347
set-audloc-default.....	348
Motor module.....	350
Physical world.....	350
Operation.....	350
Fitts's Law.....	352
Parameters.....	352
:cursor-noise.....	353
:default-target-width.....	353
:incremental-mouse-moves	353
:min-fitts-time.....	353
:motor-burst-time.....	353
:motor-feature-prep-time	353
:motor-initiation-time.....	353
:peck-fitts-coeff	354
Manual Buffer.....	354
Requests.....	356
Chunks & Chunk-types.....	367
Commands.....	368
start-hand-at-mouse	368
start-hand-at-keypad.....	369
set-cursor-position.....	369
set-hand-location.....	370
extend-manual-requests.....	371
remove-manual-request.....	373

Speech module.....	375
The vocal world.....	375
Operation.....	375
Parameters	376
:char-per-syllable.....	376
:subvocalize-detect-delay.....	376
:syllable-rate.....	377
Vocal buffer.....	377
Queries.....	377
Requests.....	379
Chunks & Chunk-types.....	380
Commands.....	381
get-articulation-time/register-articulation-time.....	381
Temporal Module.....	383
Parameters	383
:record-ticks.....	383
:time-master-start-increment.....	383
:time-mult	384
:time-noise.....	384
Temporal buffer.....	384
Requests.....	384
Modification requests.....	386
Chunks & Chunk-types.....	386
Commands.....	386
Advanced Topics.....	387
Extending Possible Chunk Slots.....	388
Commands.....	388
extend-possible-slots.....	388
Chunk-Specs.....	390
Commands.....	391
define-chunk-spec.....	391
chunk-name-to-chunk-spec.....	394
pprint-chunk-spec.....	395
match-chunk-spec-p.....	395
find-matching-chunks	398
chunk-spec-slots.....	400
slot-in-chunk-spec-p.....	400
chunk-spec-slot-spec.....	401
slot-spec-modifier/slot-spec-slot/slot-spec-value.....	402
chunk-spec-variable-p.....	403
chunk-spec-to-chunk-def.....	404
verify-single-explicit-value.....	405
test-for-clear-request.....	407
chunk-difference-to-chunk-spec.....	408
Using Buffers.....	410
Commands.....	411
buffer-read.....	412
query-buffer.....	414
clear-buffer.....	417

set-buffer-chunk.....	419
overwrite-buffer-chunk.....	421
mod-buffer-chunk.....	424
set-buffer-failure.....	428
module-request.....	430
module-mod-request.....	433
buffer-spread.....	437
buffers-module-name.....	438
request-completed-p.....	439
complete-request.....	440
complete-all-buffer-requests.....	442
complete-all-module-requests.....	443
Extending Chunks.....	445
Commands.....	447
extend-chunks.....	448
Defining New Modules.....	451
Documentation.....	451
Buffers.....	452
spreading activation weight.....	452
request parameters.....	452
queries.....	453
query printing.....	453
multi-buffer.....	453
request tracking.....	453
Parameters.....	454
name.....	454
owner.....	454
documentation.....	454
default-value.....	454
valid-test.....	455
warning.....	455
Interface functions.....	455
creation.....	455
reset.....	456
delete.....	456
parameters.....	456
queries.....	457
requests.....	457
buffer modification requests.....	458
notify upon clearing.....	458
notify at the start of a new call to run the system.....	459
notify upon a completion of a call to run the system.....	459
warning of an upcoming request.....	460
search and offset.....	460
Common Class of Modules – Goal Style.....	461
Writing Module Code.....	462
Module examples.....	463
Commands.....	463
get-module.....	463

define-buffer.....	464
define-parameter.....	465
define-module.....	467
undefine-module.....	473
goal-style-query.....	474
goal-style-request.....	475
create-new-buffer-chunk.....	476
goal-style-mod-request.....	477
Multi-buffers.....	479
Commands.....	480
store-m-buffer-chunk.....	481
get-m-buffer-chunks.....	482
remove-m-buffer-chunk.....	483
remove-all-m-buffer-chunks.....	485
erase-buffer.....	486
Searchable buffers.....	488
Multiple Models.....	491
Synchronous models.....	492
Commands.....	494
current-model.....	494
mp-models.....	495
delete-model.....	495
with-model.....	497
Asynchronous models.....	499
Commands.....	500
define-meta-process.....	500
meta-process-names.....	501
delete-meta-process.....	502
current-meta-process.....	503
with-meta-process.....	503
Combining synchronous and asynchronous models.....	505
Other multiple model examples.....	506
Configuring Real Time Operation.....	507
Dynamic events.....	509
Commands.....	510
mp-real-time-management.....	510
Module Activity and Brain Predictions.....	514
Configure a model to record module activity.....	514
Commands for getting module activity.....	514
What counts as buffer activity?.....	515
module-demand-times.....	515
module-demand-functions.....	517
module-demand-proportion.....	519
BOLD module.....	521
Parameters.....	523
:bold-param-mode.....	523
:bold-exp & :neg-bold-exp.....	524
:bold-scale & :neg-bold-scale.....	524
:bold-positive & :bold-negative.....	524

:bold-settle	525
:bold-inc.....	525
:point-predict.....	525
Commands.....	526
predict-bold-response.....	526
Adding new production compilation types.....	529
References.....	530

Preface

This document is a work in progress. The content is accurate for the components covered, but it contains references to sections on advanced materials which have not yet been included. The hope is that although it is not yet complete, this working version will be of some use to ACT-R modelers.

Introduction

ACT-R is a cognitive architecture: a theory about how human cognition works. Its constructs reflect assumptions about human cognition which are based on numerous facts derived from psychology experiments. It is a hybrid cognitive architecture – it has both symbolic and subsymbolic components. Its symbolic structure is a production system and its subsymbolic structure is represented by a set of massively parallel processes that can be summarized by a number of mathematical equations. The subsymbolic equations control many of the symbolic processes, and are also responsible for most learning processes in ACT-R.

Using ACT-R, researchers can create models that incorporate ACT-R's view of cognition and their own assumptions about a particular task. These assumptions can be tested by comparing the results of the model performing the task with the results of people doing the same task. By "results" we mean the traditional measures of cognitive psychology: time to perform the task, accuracy in the task, and, (more recently) neurological data such as those obtained from fMRI.

One important feature of ACT-R that distinguishes it from other theories in the field is that it allows researchers to collect quantitative measures that can be directly compared with the quantitative measures obtained from human participants.

ACT-R has been used successfully to create models in domains such as learning and memory, problem solving and decision making, language and communication, perception and attention, cognitive development, or individual differences.

Beside its applications in cognitive psychology, ACT-R has also been used in other fields including:

- human-computer interaction to produce user models that can assess different computer interfaces
- education (cognitive tutoring systems) to "guess" the difficulties that students may have and provide focused help
- computer-generated forces to provide cognitive agents that inhabit training environments
- neuropsychology, to interpret fMRI data.

For more detailed information, please refer to the description of the ACT-R theory in the paper "An Integrated Theory of the Mind" (2004) which is available from the ACT-R web site at: <http://act->

r.psy.cmu.edu/papers/403/IntegratedTheory.pdf, or to the book “How Can the Human Mind Occur in the Physical Universe?”.

Document Overview

This manual is a guide and reference for the ACT-R 7 software implementation. It is not meant to be a tutorial or a textbook on the ACT-R theory or a “how to” on writing models using ACT-R. The ACT-R Tutorial, which accompanies the software, is designed to introduce the theory and techniques for modeling with ACT-R. This document is intended to be a compliment to the tutorial, and it describes the components of the implementation, how they are connected, the commands available to the user, and some recommended practices for use.

This manual is a reference for the ACT-R 7 implementation only. It does not describe mechanisms from older implementations nor does it thoroughly discuss how commands may differ from similar commands in previous versions.

General Software Description

ACT-R 7 (hereafter referred to only as ACT-R) is written in Common Lisp. It was implemented and tested using Allegro Common Lisp by Franz Inc. <http://www.franz.com/>, Clozure CL <http://www.clozure.com/clozurecl.html>, LispWorks by LispWorks Ltd <http://www.lispworks.com>, CMUCL <http://www.cons.org/cmucl/>, CLISP <http://www.clisp.org>, SBCL <http://sbcl.sourceforge.net/>, ABCL <http://common-lisp.net/project/armedbear/>, and Embeddable Common-Lisp <https://common-lisp.net/project/ecl/>. It should run in any ANSI compliant implementation of Common Lisp, but has only been tested with those listed above. If you have problems loading or running ACT-R in any Lisp please contact Dan Bothell (db30@andrew.cmu.edu) with the details. We also make the ACT-R system available as a standalone application for those that do not have Lisp software. The standalone versions include a command line only version of Clozure CL and thus are as complete a system as one has when using the ACT-R source code, but it may not be as easy to use as a Lisp which has a nice IDE or which is used through an interface like SLIME, SLIMV, or with the Inferior Lisp mode of Emacs.

It is not necessary for one to be a Lisp programmer to be able to use ACT-R for basic modeling work. However, because ACT-R is running in Lisp, some basic understanding of how to program in Lisp can be helpful, and for those looking to extend the capabilities of a model it will be essential. An introduction to Lisp is beyond the scope of this document, but there are many introductory Lisp books available as well as many online resources. Two online resources where you can find additional information about Lisp are The Association of Lisp Users, <http://www.alu.org/alu/home>, and CLiki, the Common Lisp wiki, <http://www.cliki.net/>.

The primary means of interacting with ACT-R is through the Lisp read-eval-print loop - a command line interface. All of the commands described in this manual are available through that interface, and the manual assumes that that is how one will be using the system. However, there is also a set of GUI tools available (included with the main distribution) called the ACT-R Environment. The ACT-R Environment provides an alternate interface to a subset of the commands and is described in its own manual. The ACT-R Environment is useful for beginners, and because it uses separate windows to display the information, can be helpful to coordinate viewing model data for advanced users as well.

Case sensitivity

Commands and names in ACT-R are not case-sensitive. The ACT-R software should only be used with an ANSI compliant case insensitive Lisp system. A Lisp like the modern mode provided with

ACL may result in problems when trying to use ACT-R and a warning will be displayed if the software is loaded into such a system to indicate that problems could occur.

Functions vs. Macros

Many ACT-R user commands are implemented as macros which do not evaluate their arguments so that one does not have to quote the arguments. This makes it generally easier to work with the ACT-R software without having to worry as much about Lisp syntax, but does mean that if one wants to use ACT-R commands programmatically more effort is required to either explicitly evaluate the macro command with its arguments or to use a corresponding ACT-R function. Most of the macro based ACT-R commands also have a corresponding function which will have the same name, but with a `-fct` appended to it e.g. `add-dm` and `add-dm-fct`. The functions occasionally require a slightly different specification of the parameters relative to the macro, for example requiring that a list of items be provided instead of just specifying an arbitrary number of items. A command's description and examples will indicate any such differences.

Compatibility issues

While the software should run in any ANSI compliant Common Lisp, there are occasionally minor issues with particular Lisp versions and there are patches in the ACT-R code to address those. Those changes are described here because it may affect other Lisp code which one writes in those Lisps while using ACT-R.

Three such issues occur with SBCL. The first is that SBCL takes a very strict interpretation of how `defconstant` should operate which prevents some code which contains `defconstant` calls from being compiled and then loaded within the same instance of SBCL. To avoid that problem the ACT-R code shadows the `defconstant` macro in SBCL with a macro which calls `cl::defconstant` and wraps it with an `ignore-errors` when the variable specified is already bound. The second is that the internal SBCL code already defines a function called `reset`. To fix that the ACT-R code just shadows the `reset` function. The final patch for SBCL occurs for the Windows version of the directory function because it does not handle wildcard characters in the same way as other Lisps (or even other OS versions of SBCL). Again, to address this the default function is shadowed with one in the ACT-R code which handles things as needed.

There is one such issue with CLisp because it has a function named `execute` defined internally. The fix for that is to shadow the internal function with the one defined in ACT-R.

Notations in the Documentation

In the description of some commands it will describe a parameter or return value as a “generalized boolean”. What that means is that the value is used to represent a truth value – either true/successful or false/failure. If the value is the symbol **nil** then it represents false and all other values represent true. When a generalized boolean is returned by one of the commands, one should not make any assumptions about the returned value for the true case. Sometimes the true value may look like it provides additional information, but if that is not specified in the command’s description then it is not guaranteed to hold for all cases or across updates to the command.

Command Syntax

When describing a command’s syntax the following conventions will be used:

- items appearing in **bold** are to be entered verbatim
- items appearing in *italics* take user-supplied values
- items enclosed in {curly braces} are optional
- * indicates that any number of items may be supplied
- + indicates that one or more items may be supplied
- | indicates a choice between options which are enclosed in [square brackets]
- (parentheses) denote that the enclosed items are to be in a list
- a pair of items enclosed in <angle brackets> denote a cons cell with the first the car and the second the cdr
- -> indicates that calling the command on the left of the "arrow" will return the item to the right of the "arrow"
- ::= indicates that the item on the left of that symbol is of the form given by the expression on the right

Examples

When examples are provided for the commands they are shown as if they have been evaluated at a Lisp prompt. The prompt that is shown prior to the command indicates additional information about the example. There are three types of prompts that are used in the examples:

- A prompt with just the character ‘>’ indicates that it is an individual example – independent of those preceding or following it.

- A prompt with a number followed by '>', for example "2>" means that the example is part of a sequence of calls which were evaluated and the result depends on the preceding examples. For any given sequence of calls in an example the numbering will start at 1 and increase by 1 with each new example in the sequence.
- A prompt with the letter E preceding the '>', "E>", indicates that this is an example which is either incorrect or was evaluated in a context where the call results in an error or warning. This is done to show examples of the warnings and errors that can occur.

ACT-R Software Distribution

There are two primary means of acquiring the ACT-R software. The first is from the ACT-R web site <http://act-r.psy.cmu.edu>. The software page of the web site has the most recently released version of ACT-R available as either a .zip of the source files or built into a standalone application for Windows or Mac OS. The released versions have been tested against the reference models and the output of the models should be consistent with respect to what is printed in the tutorial. New releases are made when there are significant updates or patches and typically happen two or three times a year. The other method for acquiring the software is via version control software called Subversion. More information on Subversion can be found at <http://subversion.apache.org/>. The ACT-R archive is located at <svn://act-r.psy.cmu.edu/actr7>. The Subversion archive contains the most up to date version of ACT-R, and often contains minor changes or bug fixes not yet available in a released version. Note however that the minor changes made to the sources available through Subversion are not all tested thoroughly against the tutorial models and there may be discrepancies with respect to the tutorial documentation until the next released version.

Distribution Contents

The ACT-R distribution consists of: the Lisp source code files which implement ACT-R, the ACT-R Environment application along with its corresponding Tcl/Tk source code files, the ACT-R Tutorial unit texts and models, documentation for the software and tools, examples showing some advanced capabilities not covered in the tutorial, and some extensions of the system which are not included by default. All of the files are distributed in a single directory, called `actr7`, which contains two files and several subdirectories. The files are `load-act-r.lisp` which is the file that should be loaded to load the main ACT-R software (see [Loading and Running the ACT-R System](#)) and a file called `readme.txt` which contains some information about the distributed files. Here is a listing of the subdirectories along with a general description of their purpose and some of their specific contents.

Subdirectories

commands

This directory contains the Lisp code for user commands for some of the central modules. One feature of this directory is that any file with a `.lisp` extension placed into this folder will be compiled and loaded with the rest of the system.

core-modules

This directory contains the Lisp code that defines the modules which instantiate the main ACT-R system described in the theory. They are assumed to always be available, but are not absolutely required. The base modules are Procedural, Declarative, Goal, Vision, Auditory, Motor, Speech and Imaginal and are all described in this manual.

devices

This directory contains subdirectories which each contain two files. Those files contain the Lisp code which control the interaction between an ACT-R device and some particular Lisp's GUI components (as well as a generic virtual system which simulates a GUI that is available for all Lisps) and a set of tools for interacting and creating interfaces called the ACT-R GUI interface (AGI). A [device](#) is described later in this manual and the AGI has a separate manual.

docs

This directory contains the documentation files for ACT-R. They include details on using the system, as well as documents describing particular features. Most are either Microsoft Word documents (`.doc` files) or PDFs depending on how the files were acquired (the Subversion repository holds the `.docs`

but they are converted to .pdf for the releases found on the website). Here are descriptions of some of the files found there:

- AGI
 - o The manual for the AGI tools
- compilation
 - o A document describing the details of how the production compilation mechanism works.
- compilation.xls
 - o An Excel Spreadsheet that is used to define the operation of the production compilation mechanism for different buffer styles.
- device-interaction-issues
 - o A document indicating some issues to be aware of when building and working with devices in ACT-R which connect to real GUI items or external systems.
- differences-between-6-and-7
 - o Slides covering the most significant changes between ACT-R 6.0 and ACT-R 7.
- EnvironmentManual
 - o A manual for the GUI tools provided by the ACT-R Environment application.
- extending-actr
 - o A set of slides which describe mechanisms through which the ACT-R software may be extended with a focus on devices and modules which was presented at the ICCM tutorial in 2007.
- LGPL.txt
 - o The Lesser Gnu Public License text. That is the license under which the ACT-R software is distributed.
- new-utility
 - o A document describing the current utility learning mechanism which replaced the prior “PG-C” mechanism in 2006.
- QuickStart.txt
 - o A text file with instructions on how to load ACT-R and start the ACT-R Environment.
- reference-manual
 - o This document.
- template.lisp

- o A lisp file which contains a documentation header, the LGPL license notice, and some switches used for packaging the ACT-R source code that is a recommended starting point for writing any new modules or additions which are for general use with ACT-R.

environment

The environment directory contains all of the files necessary for using the ACT-R Environment. There are several items in this directory: Lisp files that define the tools and the communication between ACT-R and the Environment, the Environment applications for both Windows and Mac OS X, and a GUI directory that contains the Tcl/Tk files used by the Environment application.

examples

Several Lisp files and directories which contain examples of using more advanced components of the software like multiple models, visual tracking, creating new devices, and adding new modules.

extras

The extras directory contains additional modules and other files that have been contributed to the distribution, but which are not part of the default ACT-R system and are not loaded by default. Each addition is included in its own subdirectory and directions for using the extension should be found within the files themselves or the documentation which accompanies them.

framework

The framework directory contains the Lisp files that define the software framework upon which the ACT-R system is based. The software framework is a basic discrete event simulation system that was designed to implement ACT-R, but is not based on the theory of the ACT-R.

modules

The modules directory contains Lisp files which implement additional modules which are loaded into the default ACT-R system. As with the commands directory, any files with a .lisp name placed into this directory will be loaded automatically when ACT-R is loaded.

other-files

This directory contains Lisp files that add additional tools for generating BOLD response data and providing some of the graphic information for the ACT-R Environment. Like the commands and modules directories, any file with a .lisp extension placed into this directory will be automatically loaded with the system.

support

The support directory contains Lisp files that may be needed for certain Lisp implementations, by certain modules of the system, or for particular extensions or tools. These files are loaded when required by other files. Files with a .lisp name placed here can be compiled and loaded using the [require-compiled](#) command.

tools

The tools directory contains Lisp files that define user functions and modeling tools for ACT-R. Like the commands and modules directories, all files with a .lisp name placed into this folder will be automatically loaded.

tutorial

The tutorial directory contains several subdirectories. Each one holds the files for a unit of the ACT-R Tutorial. Each unit consists of a text on a particular aspect of ACT-R, one or more demonstration models, a partial model which provides a starting point for an assignment, a text describing the Lisp code in the provided models, and in some units an additional text with details on how to debug models along with a broken model to work through debugging.

user-loads

The user-loads directory contains no files in the distribution. It is provided as a place for users to add files which will be loaded automatically after ACT-R has finished loading and initializing. All of the files in the user-loads directory with a .lisp name will be compiled and loaded in order based on the file names sorted using the Lisp `string<` function. Because this occurs after the system has been initialized it is safe to put a model file into this directory.

Loading and Running the ACT-R System

To start ACT-R all one needs to do is load the load-act-r.lisp file into a supported Lisp system. That will load all of the necessary files for ACT-R. The file should only be loaded into a given Lisp session once.

The files are compiled before loading and that will usually generate a lot of warnings from the compiler. Those warnings can be safely ignored. The files are only compiled the first time you load ACT-R. The compiled files are saved with the source files and on subsequent loadings there is no need to recompile everything. Thus, on all loadings after the first one, it should load faster and produce fewer warnings.

Once the loading is complete, you will see a print out describing the current version of ACT-R being used which looks like this:

```
#####  
ACT-R 7 Version Information:  
Software           : 7.0-<internal>
```

The software version indicates the specific version of ACT-R 7 which has been loaded. The tag <internal> means that it isn't a specific release and was checked out from the repository. If it is a specific release then there will be a tag indicating the date that release was first made available. After that there will be a listing of all the modules that are defined for ACT-R along with their versions and brief descriptions. That will be followed by a line that looks like this:

```
##### Loading of ACT-R 7 is complete #####
```

At that point, ACT-R is ready to use.

If there were any files in the user-loads directory then there will be at least two additional lines displayed. The first will be:

```
##### Loading user files #####
```

That will be followed by any information displayed while the files in that directory are compiled and loaded. After all of those files have been loaded this will then be displayed:

```
##### User files loaded #####
```

Compiler Optimizations

Normally, when ACT-R compiles its files it uses the current optimization settings of the Lisp. However, if **:actr-fast** is on the features list when load-act-r.lisp is loaded it will apply these settings before compiling the ACT-R sources:

```
(proclaim '(optimize (speed 3) (safety 1) (space 0) (debug 0)))
```

and will remain after the loading is finished. With those settings ACT-R should run faster, but how much faster will vary from Lisp to Lisp. If the ACT-R source files were compiled without the switch then setting the switch and recompiling them will be required to see the improved performance.

Logical Host

As part of loading ACT-R a logical host of “ACT-R” is defined which maps to the directory where the load-act-r.lisp file is located. That logical host is available for use by the user. It can be useful when working with the tutorial in a command-line only Lisp to load the tutorial models. Here is an example which will load the count model from unit1 (assuming that one has not moved the tutorial files):

```
> (load "ACT-R:tutorial;unit1;count.lisp")
; Loading C:\Users\db30\Desktop\actr7\tutorial\unit1\count.lisp
T
```

Note however that some Lisps will not allow a logical pathname to be passed to commands like load and one will have to translate that into a physical pathname using translate-logical-pathname:

```
> (load (translate-logical-pathname "ACT-R:tutorial;unit1;count.lisp"))
; Loading C:\Users\db30\Desktop\actr7\tutorial\unit1\count.lisp
T
```

or instead of explicitly translating the pathname, the command actr-load may be used which performs the two steps indicated in the last example directly i.e. this:

```
> (actr-load "ACT-R:tutorial;unit1;count.lisp")
```

is equivalent to the previous example and will work in all Lisps.

Load order

For those considering adding extensions or just having files loaded automatically, the files and directories are loaded in the following order:

- framework directory files in a predefined order

- core-modules directory files in a predefined order
- all .lisp files from the commands directory in no particular order
- the virtual device files
- any Lisp specific device files
- all .lisp files from the modules directory in no particular order
- all .lisp files from the tools directory in no particular order
 - o the ACT-R Environment files are loaded as part of this step
- all .lisp files from the other-files directory in no particular order
- all .lisp files from the user-loads directory in order based on file name sorted using the Lisp `string<` function.

Recompiling

If one of the source files in the distribution changes (the date on the .lisp file is newer than the date on the compiled version of that file) then it will automatically be recompiled the next time it is loaded. However, there may be times when you need to force all of the ACT-R files to be recompiled. For instance, if you upgrade or change your Lisp system you will likely need to recompile everything. Also, if you get an update to your current set of ACT-R files it is often best to force a recompile the next time you load it because there may be some interdependencies that will require more than just the updated file to be recompiled.

To force ACT-R to recompile all of its files you should add the `:actr-recompile` switch to the features list which can be done with this call before loading the `load-act-r.lisp` file:

```
(push :actr-recompile *features*)
```

Packaging

By default, the ACT-R files are loaded into which ever package is current at the time they are loaded i.e. there are no package specifications. However, there are two features which can be set that will change the package into which ACT-R is loaded. If the ACT-R files have been compiled previously, then it will be necessary to also force the recompiling of the sources when changing the package into which they are loaded. Note that these options may not work properly in all systems – please contact Dan if you have any problems or questions with using the packaged ACT-R code.

Clean

The first option is to add the `:clean-actr` switch to the features list before loading. That will force the files to be loaded into the `:cl-user` package in most Lisps (the only exception is that in ACL with the IDE it will load them into the `:cg-user` package).

Packaged

The other option is to add the `:packaged-actr` switch to the features list. That will create a new package called `:act-r` when the `load-act-r.lisp` file is loaded and the ACT-R code will be loaded into that package. Nothing is exported from that package by the ACT-R code.

Requiring files

Files placed into the support directory of the distribution can be compiled and loaded when needed by using an ACT-R extension of the Lisp `require` function. The function is named `require-compiled` and it takes two parameters. The first is the Lisp module name string as would be passed to the `require` function (note this is not the same as an ACT-R module name) and the second is the pathname to the file to be loaded. Since the pathname can be anywhere this is not technically specific to the support directory of ACT-R, but of the ACT-R source code files only those in the support directory use the `provide` function to specify a module name for use with the `requiring`. There is another logical host created when ACT-R is loaded called “ACT-R-support” which refers to the support directory of the current ACT-R source code tree. Here is an example which would compile and load the `central-parameters.lisp` file from the support directory of the distribution:

```
(require-compiled "CENTRAL-PARAMETERS" "ACT-R-support:central-parameters")
```

The reason for using `require-compiled` compared to just loading the file directly is that `require-compiled` will only compile and load the file if it has not been loaded previously. That way one can specify that a particular support file is necessary in multiple files and only the first one will actually compile and load it.

Overall Software Design

The ACT-R software is composed of two major components. From the perspective of the user, these components operate together seamlessly to form the ACT-R system, but it is worth noting from the perspective of the theory of ACT-R that there are really two separate pieces to the system.

The first is a discrete event simulation system which controls the timing and coordination of operations within ACT-R. It was designed to provide all of the support necessary to implement the current ACT-R theory, but is not itself a part of the theory. It defines the abstractions and tools which underlie the operations of the system, namely a [meta-process](#), a [model](#), a [module](#), a [buffer](#), a [chunk](#) and a [parameter](#). Some of those items are components of the theory of ACT-R, for example buffers and chunks, but their specific implementation in the software is not prescribed by the theory.

The other component is the set of modules that instantiate the theory of ACT-R. These modules contain the components that are used to model human cognition as described in the paper “Integrated Theory of the Mind” and the book “How Can the Human Mind Occur in the Physical Universe?”. The actions and timing profiles generated by these modules when a model is run are the actual predictions of the theory. Anything else, for instance the actual time it takes the software to run the simulation, is not based on the theory. Most of the time, the distinction is immaterial to the user, but sometimes it is important to distinguish between what is a psychological claim of a model and what is just a consequence of the current software implementation.

Model files

Generally, when working with ACT-R one will generate text files that contain the description of a model along with any necessary control and support code which will be loaded into a Lisp that has the ACT-R system already loaded. This is not the only way to develop models in ACT-R, but is by far the most typical usage.

An ACT-R model file is a text file of Lisp source code. It can be generated in any text editor. Because it will be loaded into Lisp it must be syntactically correct Lisp code. Thus, it can be useful to use an editor that helps with that. The editors built into the GUI based Lisp systems (like CCL on a Mac, LispWorks, or ACL with its IDE) are good choices if using such a Lisp, but if not, an editor like Emacs which has automatic Lisp indenting and parentheses matching will also help.

A typical model file will have the following structure:

```
(clear-all)  
{Lisp functions for presenting an experiment, data collection or other support needs}  
(define-model model-name  
  (sgp {parameter value}*)  
  {chunk-type definitions}  
  {initial chunks are defined}  
  {productions are specified}  
  {any additional model set-up commands}  
  {additional model parameter settings}  
)
```

The ACT-R commands shown above and the model components referenced ([chunk-types](#), [chunks](#), and [productions](#)) will be described in detail in later sections of this document, but for now here is a basic description of what the components of the model file do.

- **(clear-all)**
The clear-all command completely resets ACT-R's state to a clean slate. This does not have to be the first thing in the file, but it should occur before defining any models.
- *{Lisp functions for presenting an experiment, data collection or other support needs}*
The Lisp commands that define any experiment for the model or other code typically come before the ACT-R model description. The amount and nature of the code can vary significantly between models.
- **(define-model model-name)**
The define-model command is used to specify exactly what constitutes the components of the model and to give it a name for reference. Everything between the name specified for the model

and the closing parenthesis of this command are considered the model's initial configuration. The commands are processed sequentially, thus the order of things does matter.

- **(sgp {parameter value}*)**
The sgp command is used to set parameters that control the general operation of the system. This is typically the first command in the model's definition so that all of the conditions are properly set before anything else occurs.
- *{chunk-type definitions}*
Descriptions are given for declaring the configuration of slots that will be used in the chunks in the model.
- *{initial chunks are defined}*
The initial chunks for the model are created and typically placed into the model's declarative memory.
- *{productions are specified}*
The productions that control how the model will act are written here.
- *{any additional model set-up commands}*
Any other commands necessary to configure components of the model or modules are specified.
- *{additional model parameter settings}*
Parameters for chunks and productions specified above are set.
-)
The define-model call is ended with a closing parenthesis.

Meta-process

The main component of the simulation system is called a meta-process. It is essentially the clock and event coordinator for the system. It maintains a schedule of events that the other components have initiated and executes them at the desired time. It also maintains a list of events that are waiting for specific events to occur before they are added to the main event queue. It is not part of the ACT-R theory – it is purely a component of the software system.

It is possible to have more than one meta-process defined in a running ACT-R system. Doing so would allow one to have multiple asynchronous models. That is an advanced topic discussed in a [later section of the manual](#). Typically there is only one meta-process (the default that exists when the system is loaded) and users will not need to create any others. For most users, the meta-process is invisible – it runs the system behind the scene.

Regardless of how many meta-processes are defined, only one is accessible at any given time. This is referred to as the current meta-process. Only the current meta-process will be manipulated by the commands. If there is only one meta-process, then it will always be the current meta-process. If there is more than one meta-process defined, then it is up to the modeler to specify which is current before executing any commands.

Commands

clear-all

Syntax:

```
clear-all -> nil
```

Arguments and Values:

none

Description:

Clear-all restores ACT-R to its initial state. It removes all meta-processes except for the default meta-process and all attributes of the default meta-process are set to their initial values: no model defined, the time is set to 0.0, the event queue is cleared, waiting events are removed and the event hooks are cleared.

In addition, the current binding of the Lisp variable `*load-truename*` is recorded for use by the reload command.

Typically usage is to place `clear-all` at the top of a model file to ensure that when the model is defined it starts in a clean system and that the reload command can be used.

Examples:

```
> (clear-all)
NIL
```

reset

Syntax:

```
reset -> [meta-process-name | nil]
```

Arguments and Values:

meta-process-name ::= a symbol which is the name of the meta-process that was reset

Description:

The reset command is similar to `clear-all` except that it only affects the current meta-process and instead of removing all currently defined models they are restored to their initial conditions. Specifically, for the current meta-process the time is set to 0.0, the event queue is cleared, all waiting events are removed and then each of the currently defined models is reset. The details of what happens when a model is reset are described in the [models section](#). The name of the meta-process which was reset is returned.

There are also two special situations which can result when there is a single empty model in the current meta-process. If the empty model was loaded from a file, then a warning is displayed and that file is reloaded (using the reload command). If the empty model was not loaded from a file then the call to reset has no effect on the system and a warning is displayed to indicate that.

If there is no current meta-process, then a warning is displayed and `nil` is returned.

Examples:

```
> (reset)
DEFAULT

> (reset)
#|Warning: Resetting an empty model results in a reload|#
```

```

; Loading C:\model.cl
DEFAULT

> (reset)
#|Warning: CANNOT RESET an empty model that wasn't loaded. |#
#|Warning: RESET had no effect! |#
DEFAULT

E> (reset)
#|Warning: reset called with no current meta-process. |#
NIL

```

reload

Syntax:

```
reload {compile?} -> [load-return-value | :none]
```

Arguments and Values:

compile? ::= a generalized boolean indicating whether or not to compile the file
load-return-value ::= a generalized boolean returned from calling the load command

Description:

The reload command calls the Lisp load command to load the file recorded during the last call to the [clear-all](#) command. If the compile? parameter is specified with a true value and that file has a type of “lisp” it will be compiled before loading. The return value from reload is the value returned from the call to load which is a generalized boolean that indicates true for a successful load or false if there was an error.

If the compile? parameter is specified as true but the recorded file is not of type “lisp” then it is not compiled. A warning is printed and the file is just loaded.

If the recorded value from clear-all is **nil** then no file is loaded and the keyword **:none** is returned.

Additional information may be printed by the call to load depending on the Lisp implementation and current settings.

Reload is essentially a shortcut for reloading a model file that has been edited to incorporate those changes.

Examples:

```

> (reload)
; Loading C:\model.lisp
T

```

```
> (reload t)
;;; Compiling file C:\model.lisp
;;; Writing fasl file C:\model.fasl
;;; Fasl write complete
; Fast loading C:\model.fasl
T

> (reload t)
#|Warning: To use the compile option the pathname must have type lisp.|#
; Loading C:\model.txt
T

E> (reload)
#|Warning: No load file recorded|#
:NONE
```

mp-time

Syntax:

mp-time -> [current-time | **nil**]

Arguments and Values:

current-time ::= a number representing time in seconds

Description:

Mp-time returns the current time of the current meta-process in seconds.

If there is no current meta-process, then a warning is displayed and **nil** is returned.

This is generally used for two purposes, either debugging a model or collecting response time data from a model.

Examples:

```
> (mp-time)
0.3

E> (mp-time)
#|Warning: mp-time called with no current meta-process.|#
NIL
```

mp-time-ms

Syntax:

mp-time-ms -> [current-time | **nil**]

Arguments and Values:

current-time ::= a number representing time in milliseconds

Description:

Mp-time-ms returns the current time of the current meta-process like mp-time. The difference is that mp-time-ms returns the time as an integer count of milliseconds.

If there is no current meta-process, then a warning is displayed and **nil** is returned.

Examples:

```
> (mp-time-ms)
300
```

```
E> (mp-time-ms)
#|Warning: mp-time-ms called with no current meta-process.|#
NIL
```

Events

As indicated in the description of the meta-process, the simulation system for ACT-R is implemented using discrete events. The system runs by executing a sequence of events each occurring at a specific simulated time. Effectively, every action of the model is performed by an event in the system, and the model's trace when it runs is just a printing of those events as they occur.

Basically, each event consists of the time at which it should occur, an indication of which, if any, module made the request, and an action to perform. There are some additional details and capabilities of events (like the ability to have them wait for some other action to occur instead of having a pre-specified time) and those will be described in the section on [creating events](#). This section will only describe the commands for inspecting the events which exist in the current meta-process.

Commands

mp-show-queue

Syntax:

```
mp-show-queue {indicate-traced} -> [event-count | nil]
```

Arguments and Values:

indicate-traced ::= a generalized boolean indicating whether to mark events that will occur in the trace
event-count ::= a number indicating how many items are in the event queue

Description:

Mp-show-queue prints all of the events that are on the event queue of the current meta-process to the Lisp stream ***standard-output*** in the order that they will be executed. If the *indicate-traced* value is provided and is non-**nil** then the character “*” will be displayed before the events which will be displayed in the trace with the current parameter settings. It returns the number of events in the queue.

If there is no current meta-process then a warning is displayed and **nil** is returned.

This command can be useful for debugging a model as well as when working on creating modules and experiments which generate events.

Examples:

```
> (mp-show-queue)
Events in the queue:
  0.000  NONE          CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS :ESC T :OL T :ER NIL)
  0.000  GOAL          SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000  PROCEDURAL   CONFLICT-RESOLUTION
  0.000  GOAL          CLEAR-DELAYED-GOAL
4

> (mp-show-queue t)
Events in the queue:
  0.000  NONE          CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS :ESC T :OL T :ER NIL)
*  0.000  GOAL          SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
*  0.000  PROCEDURAL   CONFLICT-RESOLUTION
  0.000  GOAL          CLEAR-DELAYED-GOAL
4

E> (mp-show-queue)
#|warning: mp-show-queue called with no current meta-process.|#
NIL
```

mp-queue-count

Syntax:

```
mp-queue-count -> [event-count | nil]
```

Arguments and Values:

event-count ::= a number indicating how many items are in the event queue

Description:

Mp-queue-count returns the number of events which are in the queue of the current meta-process. If there is no current meta-process then a warning is displayed and **nil** is returned.

Examples:

```
> (mp-queue-count)
4

E> (mp-queue-count)
#|warning: mp-show-queue called with no current meta-process.|#
NIL
```

mp-show-waiting

Syntax:

```
mp-show-waiting -> [event-count | nil]
```

Arguments and Values:

event-count ::= a number indicating how many items are in the waiting queue

Description:

Mp-show-waiting prints all of the events that are on the waiting queue of the current meta-process to the Lisp stream ***standard-output*** along with the description of the condition for which it is waiting to occur to be added to the main event queue. The first element in the waiting description indicates whether it is waiting for any event or a particular module. If it is waiting on a module then the second element indicates which module, and the last element of the description indicates whether a maintenance event will satisfy the condition. It returns the number of events that are in the waiting queue.

If there is no current meta-process a warning is displayed and **nil** is returned.

This command can be useful for debugging, but is generally more important when working on creating modules and experiments than when debugging a model.

Examples:

```
> (mp-show-waiting)
Events waiting to be scheduled:
      NIL   PROCEDURAL          CONFLICT-RESOLUTION Waiting for: (ANY NIL)
1

E> (mp-show-waiting)
#|Warning: mp-show-waiting called with no current meta-process.|#
NIL
```

mp-modules-events

Syntax:

mp-modules-events *module-name* -> [event-list | nil]

Arguments and Values:

module-name ::= a symbol which should be the name of a module

event-list ::= a list of events scheduled for the named module

Description:

Mp-module-events returns a list of all of the events from both the regular and waiting queues of the current meta-process which have a module specified that matches the *module-name* provided. The

elements of the list are the actual event structures which should only be accessed with the commands described in the [event accessors](#) section.

If there is no current meta-process a warning is displayed and **nil** is returned.

Examples:

```
> (mp-modules-events 'procedural)
(#S(ACT-R-EVENT ...))
```

```
> (mp-modules-events 'not-a-module)
NIL
```

```
E> (mp-modules-events 'procedural)
#|Warning: mp-modules-events called with no current meta-process.|#
NIL
```

Module

Module is an overloaded term in ACT-R because it has different connotations when talking about the software and the theory. At the software level a module is a basic component of the system. It can serve any number of purposes, for instance there is a printing module, a random number generator module, as well as a vision module, a declarative memory module and many others. Each module is essentially an independent component, and it is the modules which provide the functionality of the overall system. There are basically no restrictions on what a software module can do and adding new modules is the primary way of extending or modifying the overall system.

From the theory perspective however a module is a reference to some cognitive faculty which can typically be ascribed to a particular region of the brain. Thus, something like the random number module in the software obviously would not be considered a module of the theory. Another issue is that the implementation of the cognitive modules as software modules is not always one-to-one. For example, the vision system of ACT-R, which is implemented in the software as one module, is more appropriately considered as two cognitive modules – one for location information and one for object information. In the other direction, the theory’s procedural module is actually implemented as three modules in the software (one that controls production definition and matching, one to handle the utility computations and one to implement the production compilation mechanism).

Often the context in which one encounters “module” with respect to ACT-R makes it clear what is being discussed (the software or the theory) so it is not usually as confusing as it might seem. For clarity, from this point on in the manual, when it says module, the reference will always be to the actual software modules unless explicitly stated otherwise.

Each of the specific modules of the ACT-R software will be described in its own section of this manual. Other sections will include details on the components of a module, like [buffers](#) and [parameters](#), as well as [how to implement a new module](#). This section will only describe the commands for printing the currently defined modules in the system and getting their names.

Commands

mp-print-versions

Syntax:

mp-print-versions -> nil

Arguments and Values:

none

Description:

mp-print-versions prints the specific version number of the ACT-R software with a tag indicating whether the code is an [official release or an internal version checked out of the source repository](#) followed by the name, version number, and documentation string of each module which is currently defined in the system in alphabetical order by module name. It always returns **nil**.

Examples:

```
> (mp-print-versions)
ACT-R 7 Version Information:
Software          : 7.0-<internal>
AGI               : 2.0      Module based manager for AGI windows
AUDIO            : 3.2      A module which gives the model an auditory ...
BOLD             : 3.0      A module to produce BOLD response predictions ...
BUFFER-HISTORY   : 1.0      Module to record buffer change history for ...
BUFFER-PARAMS    : 1.0      Module to hold and control the buffer parameters
BUFFER-TRACE     : 2.0      A module that provides a buffer based tracing ...
CENTRAL-PARAMETERS : 1.2    a module that maintains parameters used by other ...
DECLARATIVE      : 2.0      The declarative memory module stores chunks from ...
DEVICE           : 2.0      The device interface for a model
ENVIRONMENT      : 3.0      A module to handle the environment connection if opened
GOAL             : 2.0      The goal module creates new goals for the goal buffer
IMAGINAL         : 2.1      The imaginal module provides a goal style buffer ...
MOTOR           : 3.0      Module to provide a model with virtual hands
NAMING-MODULE    : 1.3      Provides safe and repeatable new name generation ...
PRINTING-MODULE  : 1.1      Coordinates output of the model.
PROCEDURAL       : 3.0      The procedural module handles production definition ...
PRODUCTION-COMPILATION: 2.0  A module that assists the primary procedural module ...
PRODUCTION-HISTORY : 1.1    Module to record production history for display in ...
RANDOM-MODULE     : 1.0      Provide a good and consistent source of ...
RETRIEVAL-HISTORY : 1.1    Module to record retrieval history for display in ...
SPEECH          : 3.0      A module to provide a model with the ability to speak
TEMPORAL        : 2.1      The temporal module is used to estimate short time ...
UTILITY         : 2.2      A module that computes production utilities
VISION          : 4.2      A module to provide a model with a visual ...
NIL
```

all-module-names

Syntax:

all-module-names -> (module-name*)

Arguments and Values:

module-name ::= a symbol which names a module in the system

Description:

All-module-names returns a list of the names of the currently defined modules in the system in alphabetical order.

Examples:

```
> (all-module-names)
(AGI :AUDIO BOLD BUFFER-HISTORY BUFFER-PARAMS BUFFER-TRACE CENTRAL-PARAMETERS ...)
```

Buffers

Buffers in ACT-R are the interfaces between modules. Each buffer is connected to a specific module and has a unique name by which it is referenced e.g. the goal buffer or the retrieval buffer which are associated with the goal and declarative modules respectively. A buffer is used to relay requests for actions to its module, to query its module about the module's state, it can hold one chunk which is visible to all other modules, and it will respond directly to queries about the contents of the buffer itself.

A module will respond to a query through its buffer with a generalized-boolean indicating the result. In response to a request, the module will usually generate one or more events to perform some action(s) and may place a chunk into the buffer to indicate the result of that action. Any module may access or modify the chunk in any buffer at any time, but typically a module will only manipulate its own buffer(s).

The buffer itself responds to five queries all specified with the name `buffer` and the possible values: `empty`, `full`, `failure`, `requested`, and `unrequested`. Each query will return `t` if the condition is true and `nil` if it is false. The first three query values, `empty`, `full`, and `failure`, are determined based on whether there is a chunk in the buffer and whether or not the failure flag for the buffer has been set. Only one of those queries will be true at a time. If the buffer contains a chunk then `full` will be true. If the failure flag has been set then `failure` will be true (it is not possible to set the failure flag if there is a chunk in the buffer). If neither of those is true then `empty` will be true. The other two queries indicate whether the chunk which is in the buffer or the failure flag, if it is set, was the result of a request or not. The determination of whether a chunk or failure flag in the buffer was the result of a request is determined by the call to the function which sets that item (see the [Using Buffers](#) section for details). If there is no chunk in the buffer and the failure flag is not set then both of those queries will be false.

An important thing to note is that when a chunk is placed into a buffer the buffer makes a copy of that chunk which it then makes available. Any changes made to the chunk in the buffer only affect the copy that it holds – they do not affect the original chunk from which it was copied.

One of the current research areas with ACT-R is in using the buffers to track the activity of their associated modules and then comparing that activity to data from neuroimaging studies (fMRI, MEG, or EEG) to find correlations between regions of the brain and particular buffer/module activity in

ACT-R models. Thus providing a mechanism for mapping cognitive modeling work onto actual brain regions and even being able to make predictions about where activation should show up in future neuroimaging research. Details on how that is done can be found in the [Module Activity and Brain Predictions](#) section.

The commands described here provide general information about buffers which is most often needed for modeling in ACT-R. The [Using Buffers](#) section describes the commands one can use for more low-level interaction with the buffers which would be necessary for creating a new module.

Commands

buffers

Syntax:

buffers -> (buffer-name*)

Arguments and Values:

buffer-name ::= a symbol which is the name of a currently existing buffer

Description:

The buffers command will return a list with the names of all the currently defined buffers in no particular order.

Examples:

```
> (buffers)
(VISUAL-LOCATION MANUAL RETRIEVAL IMAGINAL VISUAL AURAL PRODUCTION VOCAL AURAL-LOCATION
IMAGINAL-ACTION GOAL)
```

buffer-chunk

Syntax:

buffer-chunk *buffer-name** -> [(<buffer-name [chunk-name | **nil**]>*) | ([chunk-name | **:error**]*) **nil**]
buffer-chunk-fct (*buffer-name**) -> [(<buffer-name [chunk-name | **nil**]>*) | ([chunk-name | **:error**]*) **nil**]

Arguments and Values:

buffer-name ::= a symbol that names a buffer

chunk-name ::= a symbol that names a chunk

Description:

Generally, the `buffer-chunk` command prints out the names of buffers along with the chunks those buffers hold in the current model of the current meta-process.

If no buffer names are specified, then it prints all the buffers and the chunk name of the chunk in that buffer (or `nil` if the buffer is empty) one per line to the model's command output stream and returns a list of cons cells where the car of each cons cell is the name of a buffer and the cdr is the name of the chunk it holds (or `nil`) in no particular order.

If specific buffers are provided, then for each of those buffers, in the order provided, it prints the buffer name followed by the name of the chunk in that buffer (or `nil` if the buffer is empty) and if there is a chunk in the buffer that chunk is also printed. In this case it returns a list of the names of the chunks in the buffers provided in the same order as they were specified in the call. If an invalid buffer name is provided the corresponding value in the return list will be the keyword `:error` and nothing will have been printed.

If there is no current model or current meta-process then a warning is printed and `nil` is returned.

Examples:

These examples were generated after running the count model in unit 1 of the tutorial.

```
> (buffer-chunk)
RETRIEVAL: E-0 [E]
IMAGINAL: NIL
MANUAL: NIL
GOAL: NIL
IMAGINAL-ACTION: NIL
VOCAL: NIL
AURAL: NIL
PRODUCTION: NIL
VISUAL-LOCATION: NIL
AURAL-LOCATION: NIL
TEMPORAL: NIL
VISUAL: NIL
((VISUAL-LOCATION) (MANUAL) (RETRIEVAL . E-0) (IMAGINAL) (VISUAL) (AURAL) (PRODUCTION)
(VOCAL) ...)
```

```
> (buffer-chunk-fct '(retrieval goal))
RETRIEVAL: E-0 [E]
E-0
  FIRST 4
  SECOND 5
```

```
GOAL: NIL
(E-0 NIL)
```

```
E> (buffer-chunk-fct '(bad-buffer-name))
(:ERROR)
```

```
E> (buffer-chunk retrieval bad-name goal)
RETRIEVAL: E-0 [E]
```

```
E-0
  FIRST 4
  SECOND 5
```

```
GOAL: NIL
(E-0 :ERROR NIL)
```

```
E> (buffer-chunk)
#|Warning: buffer-chunk called with no current model.|#
NIL
```

```
E> (buffer-chunk)
#|Warning: buffer-chunk called with no current meta-process.|#
NIL
```

buffer-status

Syntax:

```
buffer-status buffer-name* -> [ ([buffer-name | :error]*) | nil]
buffer-status-fct (buffer-name*) -> [ ([buffer-name | :error]*) | nil]
```

Arguments and Values:

buffer-name ::= a symbol that names a buffer

Description:

The `buffer-status` command prints out the current values for the possible queries to which the buffers and their modules respond from the current model in the current meta-process using the current model's command output stream. For each buffer specified (or all buffers if none are specified) the buffer name is printed followed by the current result of the buffer's queries and the required module queries, one per line, indicating **t** for a true query and **nil** for a false result. That is followed by any module specific status the module prints (the module specific status is not constrained by the system and could be any type of output). It returns a list of the buffer names of the buffers for which the status was printed.

If specific buffers are provided, and an invalid buffer name is specified, the corresponding value in the return list will be the keyword **:error** and nothing will have been printed.

If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

```
> (buffer-status)
RETRIEVAL:
  buffer empty      : T
  buffer full       : NIL
  buffer failure    : NIL
  buffer requested  : NIL
```

```

buffer unrequested : NIL
state free        : T
state busy        : NIL
state error       : NIL
recently-retrieved nil: NIL
recently-retrieved t : NIL
IMAGINAL:
buffer empty      : T
buffer full       : NIL
buffer failure    : NIL
buffer requested  : NIL
buffer unrequested : NIL
state free        : T
state busy        : NIL
state error       : NIL
MANUAL:
buffer empty      : T
buffer full       : NIL
buffer failure    : NIL
buffer requested  : NIL
buffer unrequested : NIL
state free        : T
state busy        : NIL
state error       : NIL
preparation free  : T
preparation busy  : NIL
processor free    : T
processor busy    : NIL
execution free    : T
execution busy    : NIL
last-command     : NONE
...

(RETRIEVAL IMAGINAL MANUAL VISUAL AURAL PRODUCTION VOCAL ...)

> (buffer-status goal)
GOAL:
buffer empty      : T
buffer full       : NIL
buffer failure    : NIL
buffer requested  : NIL
buffer unrequested : NIL
state free        : T
state busy        : NIL
state error       : NIL
(GOAL)

> (buffer-status-fct '(retrieval))
RETRIEVAL:
buffer empty      : T
buffer full       : NIL
buffer failure    : NIL
buffer requested  : NIL
buffer unrequested : NIL
state free        : T
state busy        : NIL
state error       : NIL
recently-retrieved nil: NIL
recently-retrieved t : NIL
(RETRIEVAL)

E> (buffer-status goal non-buffer)
GOAL:
buffer empty      : T
buffer full       : NIL
buffer failure    : NIL
buffer requested  : NIL

```

```
buffer unrequested : NIL
state free         : T
state busy         : NIL
state error        : NIL
(GOAL :ERROR)
```

```
E> (buffer-status)
#|Warning: buffer-status called with no current model.|#
NIL
```

```
E> (buffer-status)
#|Warning: buffer-status called with no current meta-process.|#
NIL
```

Models

An ACT-R model is one simulated cognitive agent. Theoretically the software can have any number of models loaded simultaneously (practically there is a limit that will depend on the hardware and Lisp software) and they can be run either individually or synchronized. However, the most common usage of ACT-R is to work with only one model at a time. Most of this manual assumes that one is working with only one model at a time. Information about dealing with more than one model simultaneously is covered in the [multiple models](#) section.

A model is referenced by a name specified when it is defined, and that name must be unique within the current meta-process. A model consists of the code specified in its definition, an instance of each module in the system (which is independent of any other model's copy of that module unless a specific module indicates otherwise), and its set of chunks (which are always independent of the chunks of any other model).

A model is created using the `define-model` command which specifies the model's name and its initial conditions and results in the creation of a new instance of each module for that model to use.

Specifically, when a model is created the following sequence of actions occur:

- A new instance of each module is created
- The default [chunk-types](#) and [chunks](#) are created
- All of the buffers are set to empty
- The model's modules have their primary reset functions called (in no specific order)
- The parameters of all the modules are set to their default values (in no specific order)
- The model's modules have their secondary reset functions called (in no specific order)
- The model's definition code is evaluated in the order given (left to right)
- The model's modules have their tertiary reset functions called (in no specific order)

When a model is reset a similar sequence of actions occurs. The difference is that new module instances are not created and instead any chunks and chunk-types in the model are deleted first.

As with meta-processes, it is possible to define more than one model at a time, and regardless of how many models are defined in a meta-process, only one is accessible at any given time. This is referred to as the current model (note that different meta-processes will each have their own current model).

Only the current model in the current meta-process may be manipulated by the commands. If there is only one model in the current meta-process, then it will be the current model. If there is more than one model defined in the current meta-process, then it is up to the modeler to specify which is current before executing any commands (see the multiple models section for more details).

Commands

define-model

Syntax:

```
define-model model-name {model-code*} -> [model-name | nil]  
define-model-fct model-name ({model-code*}) -> [model-name | nil]
```

Arguments and Values:

model-name ::= a symbol that will be the name of the model

model-code ::= a Lisp expression that will be evaluated when the model is created and when it is reset

Description:

The `define-model` command creates a new model with the given *model-name* in the current meta-process. Its initial conditions are specified by the *model-code* provided.

If there is not already a model by that name in the current meta-process and there are no errors in evaluation of the *model-code* forms then the new model is created and *model-name* is returned.

If there is not a current meta-process, *model-name* is already the name of a model in the current meta-process, or an error occurs during the evaluation of the *model-code* then a warning is printed and **nil** is returned.

Special case:

If no *model-code* is specified when the model is defined, then the new model is considered an “empty model”. That has some implications when using the [reset](#) command, and generally empty models are not recommended.

Examples:

Only basic usage of `define-model` is shown here – see the tutorial for definition of actual cognitive models that perform meaningful tasks.

```

> (define-model-fct 'model-10 (list '(chunk-type start slot)))
MODEL-10

1> (define-model model-1 (chunk-type goal state))
MODEL-1

2E> (define-model-fct 'model-1 nil)
#|Warning: MODEL-1 is already the name of a model in the current meta-process. Cannot be
redefined.|#
NIL

E> (define-model model-2)
#|Warning: define-model called with no current meta-process.|#
NIL

E> (define-model model-3 (pprint "start") (pprint "end"))

"start"
#|Warning: Error encountered in model form:
(PPRNT "end")
Invoking the debugger.|#
#|Warning: You must exit the error state to continue.|#
Debug: attempt to call `PPRNT' which is an undefined function.
[condition type: UNDEFINED-FUNCTION]
#|Warning: Model MODEL-3 not defined.|#
NIL

```

delete-model

Syntax:

```

delete-model {model-name} -> [t | nil]
delete-model-fct model-name -> [t | nil]

```

Arguments and Values:

model-name ::= a symbol that should be the name of a model

Description:

The `delete-model` command removes the model with the specified *model-name* from the current meta-process. If *model-name* is not provided the current model is deleted. Deleting a model removes all events generated by that model from the event queues, deletes each of the model's instances of the modules, and removes the model from the set of models currently defined. If a model is successfully deleted then **t** is returned.

If there is not a current meta-process, *model-name* is not the name of a model in the current meta-process, or no *model-name* is given and there is no current model, then a warning is printed and **nil** is returned.

The `delete-model` command is typically only useful when working with multiple models.

Examples:

```
> (delete-model)
T
```

```
> (delete-model-fct 'model)
T
```

```
E> (delete-model)
#|Warning: delete-model called with no current meta-process.
No model deleted. |#
NIL
```

```
E> (delete-model)
#|Warning: No current model to delete. |#
NIL
```

```
E> (delete-model-fct 'model)
#|Warning: No model named MODEL in current meta-process. |#
NIL
```

Chunks & Chunk-types

Chunks are the elements of declarative knowledge in the ACT-R theory and are used to communicate information among modules through the buffers. A chunk consists of a set of named slots each holding a single value. A chunk also has a name which is used to reference it, however, the name is not considered to be a part of the chunk itself from a theoretical standpoint. Previous versions of ACT-R also required each chunk to have a specific chunk-type associated with it that defined the set of slots which it had. As of ACT-R 6.1 chunks are no longer cast into specific types. A chunk may have any combination of slots desired without having to first specify an appropriate type, and a chunk may also now have slots removed as well as added. Chunk-types still exist as a tool which the modeler may use, but they are no longer a required component of chunks.

In the ACT-R software, chunks exist at the “model level” and may be created and used by any module or additional code – they do not have to be associated with a model’s declarative memory module unlike older versions of ACT-R. In addition to the slots and values of a chunk, in the software each chunk also maintains a set of parameters which contain additional information needed by the modules or the modeler. A modeler may add chunk parameters for recording additional information as needed. See the section on [extending chunks](#) for more information on adding and manipulating chunk parameters. Along with the parameters, there is an additional feature associated with a chunk in the ACT-R software which indicates whether the slots and values of the chunk can be modified or not. When a chunk is created it is initially able to have its content modified, but the modeler (or a module) may mark a chunk as “immutable” at any point after it has been created. Once a chunk is marked as immutable it cannot be changed back to modifiable. The final thing to note about chunks is that in the software they must always be referenced by their name – the underlying representation of a chunk in the code is not considered to be part of the API.

Chunk-types in ACT-R are used essentially as a pre-processing mechanism in the definition of a model. A chunk-type associates a name with a set of slots and optional default values for those slots. That chunk-type name may then be used when creating chunks and specifying productions to indicate the slots which are intended and to include the default slot values from that chunk-type for any slots not specified in the chunk or production. The chunk-type name itself does not get recorded in the model with the chunk or production it is used in – it only serves as a declaration which will provide warnings if the chunk or production does not conform to the chunk-type indicated. If a chunk-type does not include default values for its slots then whether it is specified or not in the creation of a

chunk or production will not affect the resulting item – it will be the same with or without the chunk-type specification.

When creating chunk-types they may be organized into a hierarchy, with new chunk-types inheriting slots from one or more parent chunk-types. If a chunk-type is created with one or more parent chunk-types then that new type will include all the slots and default values from all of the parent chunk-types along with those specified in the new chunk-type. A default value specified for a slot directly in a chunk-type overrides any default value which would be inherited from a parent type. If multiple parent chunk-types specify a default value for the same slot then unless those default values are all the same such a chunk-type cannot be created. When specifying a chunk-type in creating a chunk or production a parent type may be specified and use any of the slots specified for the children of that chunk-type, but only the default values for the specific chunk-type named will be applied.

Default Chunk-types

There are a few chunk-types created with a model automatically. Those are, chunk, constant-chunks, query-slots, and clear. These are used to create some of the default chunks for the model and may be used freely in creating additional chunks or productions. Many of the provided modules also specify new chunk-types which may be used and those will be indicated with the module description.

chunk

The chunk-type named chunk has no slots specified for it. Its purpose is to serve as the root of the chunk-type hierarchy i.e. all chunk-types which are created implicitly inherit from the chunk-type named chunk.

constant-chunks

The constant-chunks chunk-type specifies one slot named name.

query-slots

The query-slots chunk-type specifies three slots named state, buffer, and error. Those are the names of the queries which all buffers will respond to and by creating a chunk-type with those slot names one can create [chunk-specs](#) which use those slots to perform queries from code.

clear

The clear chunk-type has one slot named clear which has a default value of t. It is provided as a convenient and consistent mechanism for use when creating a module which needs to provide a request which performs some sort of “clearing” of the module.

Default Chunks

There are several chunks created for each model automatically which may be used as needed. All of these default chunks are marked as immutable. There will also be many chunks created by the provided modules and the details of those can be found in the specific module sections.

Chunks named free, busy, error, empty, full, failure, requested, and unrequested are created each with a slot named name that is set to the chunk’s name i.e. the chunk named free has one slot called name with the value free.

A chunk named clear is created having one slot named clear with a value of t.

Commands

The commands described here are general chunk and chunk-type actions which can be used for any chunk. Some modules provide additional commands for manipulating or inspecting the chunks that are being used by that module and the details of those commands can be found in the specific module sections.

Chunk-type Commands

chunk-type

Syntax:

```
chunk-type { [ new-name | (new-name (:include parent-name)* ) ] {doc-string} [slot-name | (slot-name default-value)]* } -> [ type-name | (type-name*) | nil ]
```

```
chunk-type-fct [ nil | ( [ new-name | (new-name (:include parent-name)* ) ] {doc-string} [slot-name | (slot-name default-value)]* ) ] -> [ type-name | (type-name*) | nil ]
```

Arguments and Values:

new-name ::= a symbol that is to be the name of the new chunk-type

type-name ::= a symbol that is the name of a chunk-type

parent-name ::= a symbol that names a chunk-type to be a parent type of this new chunk-type

doc-string ::= a string that is used as documentation for this chunk-type

slot-name ::= a symbol that names a slot which will be part of this chunk-type

default-value ::= any Lisp value which specifies the value that will be used as the default value for the corresponding slot-name

Description:

The chunk-type command creates a new chunk-type for the current model of the current meta-process or displays all of the currently defined chunk-types for the current model.

If no parameters are passed to the chunk-type command (or **nil** to the function) then all of the existing chunk-types in the current model are printed to the command output stream as described by [pprint-chunk-type](#) and a list of their names is returned (in no particular order).

If a valid chunk-type specification is provided then a new chunk-type is created for the current model and the name of that chunk-type is returned. If the chunk-type provides the same specification as an existing chunk-type in the model then a warning will be displayed.

If there is no current model, no current meta-process, the given new-name already names an existing chunk-type in the current model, or there is an error in the specification of the chunk-type then a warning is printed and **nil** is returned.

Examples:

```
> (chunk-type)
```

```
SOUND  
  KIND  
  CONTENT  
  EVENT
```

```
AUDIO-COMMAND
```

```
MOVE-CURSOR <- MOTOR-COMMAND  
  OBJECT  
  LOC  
  DEVICE
```

```
...
```

```
(SOUND AUDIO-COMMAND MOVE-CURSOR ...)
```

```
1> (chunk-type goal slot1 state)  
GOAL
```

```
2> (chunk-type-fct '(other-type slot1 slot2))  
OTHER-TYPE
```

```
3> (chunk-type (subgoal1 (:include goal)) new-slot)  
SUBGOAL1
```

```
4> (chunk-type-fct '((subgoal2 (:include goal) (:include other-type))))
```

```

SUBGOAL2

5> (chunk-type new-type (slot default-value) (other-slot 4))
NEW-TYPE

6> (chunk-type detailed-type "The chunk-type detailed-type" slot)
DETAILED-TYPE

7> (chunk-type-fct nil)
...

GOAL
  SLOT1
  STATE

OTHER-TYPE
  SLOT1
  SLOT2

SUBGOAL1 <- GOAL
  NEW-SLOT
  SLOT1
  STATE

SUBGOAL2 <- GOAL, OTHER-TYPE
  SLOT2
  SLOT1
  STATE

NEW-TYPE
  SLOT (DEFAULT-VALUE)
  OTHER-SLOT (4)

DETAILED-TYPE "The chunk-type detailed-type"
  SLOT

(CHUNK CONSTANT-CHUNKS CLEAR AUDIO-EVENT SET-AUDLOC-DEFAULT SOUND GENERIC-ACTION ...)

8E> (chunk-type repeat slot1 state)
#|Warning: Chunk-type REPEAT has the same specification as the chunk-type GOAL.|#
REPEAT

E> (chunk-type (new-type (:include bad-type)))
#|Warning: Non-existent chunk-type BAD-TYPE specified as an :include in NEW-TYPE chunk-
type definition.|#
NIL

E> (chunk-type (bad :include goal))
#|Warning: Invalid modifier list specified with the chunk-type name: (:INCLUDE GOAL)|#
NIL

E> (chunk-type no-model)
#|Warning: chunk-type called with no current model.|#
NIL

E> (chunk-type no-meta-process)
#|Warning: chunk-type called with no current meta-process.|#
NIL

```

pprint-chunk-type

Syntax:

pprint-chunk-type *type-name* -> [*type-name* | **nil**]

pprint-chunk-type-fct *type-name* -> [*type-name* | **nil**]

Arguments and Values:

type-name ::= a symbol that is the name of a chunk-type

Description:

The `pprint-chunk-type` command is used to print a description of a chunk-type. The output is sent to the current model's command output stream.

If the parameter provided is the name of a chunk-type in the current model of the current meta-process then that chunk-type is printed like this: the chunk-type name is printed, if the chunk-type had any parent types specified then the name is followed by “<-” and the names of the parent chunk-types separated by commas, the documentation string for the chunk-type, if it has one, is printed, and then the slot names of the chunk-type are printed one per line with the slot's default value in parenthesis after the slot name if one was provided.

If there is no current model, current meta-process, or the given *type-name* does not name an existing chunk-type in the current model then a warning is printed and **nil** is returned.

Examples:

```
1> (chunk-type test slot1 (slot2 2))  
TEST
```

```
2> (chunk-type (subtest (:include test)) "a subtype of test" slot3)  
SUBTEST
```

```
3> (pprint-chunk-type test)  
TEST  
  SLOT1  
  SLOT2 (2)
```

```
TEST
```

```
4> (pprint-chunk-type-fct 'subtest)  
SUBTEST <- TEST "a subtype of test"  
  SLOT1  
  SLOT2 (2)  
  SLOT3
```

SUBTEST

```
E> (pprint-chunk-type not-a-chunk-type)
#|Warning: NOT-A-CHUNK-TYPE does not name a chunk-type in the current model. |#
NIL
```

```
E> (pprint-chunk-type chunk)
#|Warning: pprint-chunk-type called with no current model. |#
NIL
```

```
E> (pprint-chunk-type chunk)
#|Warning: pprint-chunk-type called with no current meta-process. |#
NIL
```

chunk-type-p

Syntax:

```
chunk-type-p chunk-type-name? -> [ t | nil ]
chunk-type-p-fct chunk-type-name? -> [ t | nil ]
```

Arguments and Values:

chunk-type-name? ::= a symbol to be tested to determine if it names a chunk-type

Description:

The `chunk-type-p` command returns **t** if *chunk-type-name?* is a symbol that names a chunk-type in the current model of the current meta-process and returns **nil** if it does not. If there is no current model or no current meta-process then a warning is printed and **nil** is returned.

Examples:

```
> (chunk-type-p clear)
T
```

```
> (chunk-type-p-fct 'chunk)
T
```

```
> (chunk-type-p bad-name)
NIL
```

```
> (chunk-type-p-fct 'non-chunk-type)
NIL
```

```
E> (chunk-type-p chunk)
#|Warning: get-chunk-type called with no current model. |#
NIL
```

```
E> (chunk-type-p clear)
#|Warning: get-chunk-type called with no current meta-process. |#
```

all-chunk-type-names

Syntax:

all-chunk-type-names -> (type-name*)

Arguments and Values:

type-name ::= a symbol that names a chunk-type

Description:

All-chunk-type-names takes no parameters and returns a list of all the chunk-type names in the current model of the current meta-process. The type names are ordered based on the order in which they were defined (first name in the list is the first one defined).

If there is no current model or current meta-process a warning is printed and **nil** will be returned.

Examples:

```
> (all-chunk-type-names)
(CHUNK CONSTANT-CHUNKS CLEAR AUDIO-EVENT SET-AUDLOC-DEFAULT SOUND GENERIC-ACTION ...)
```

```
E> (all-chunk-type-names)
#|Warning: all-chunk-type-names called with no current model.|#
NIL
```

```
E> (all-chunk-type-names)
#|Warning: all-chunk-type-names called with no current meta-process.|#
NIL
```

chunk-type-possible-slot-names-fct

Syntax:

chunk-type-possible-slot-names-fct *chunk-type-name* -> [(slot-name*) | nil]

Arguments and Values:

slot-name ::= a symbol which names a slot usable in the specified chunk-type

Description:

Chunk-type-possible-slot-names-fct takes one parameter which should be the name of a chunk-type. It returns a list of all the names of slots which are valid for use with that chunk-type in the current model of the current meta-process along with all of the request parameters which have been defined for the buffers. A slot is valid for the chunk-type if it is specified in that chunk-type's definition, is specified in a chunk-type which has the specified chunk-type as a parent, or is a slot which has been added to chunks through extension with [extend-possible-slots](#).

If there is no current model or current meta-process or the specified name does not name a chunk-type in the current model then a warning is printed and **nil** is returned.

Examples:

```
> (chunk-type-possible-slot-names-fct 'clear)
(CLEAR :MP-VALUE :RECENTLY-RETRIEVED :CENTER :NEAREST :FINISHED :ATTENDED)

> (chunk-type-possible-slot-names-fct 'chunk)
(SCALE RIGHT LEFT TYPE SET-VISLOC-DEFAULT SIZE DISTANCE SCREEN-Y SCREEN-X COLORS ...)

E> (chunk-type-possible-slot-names-fct 'bad-name)
#|Warning: Invalid chunk-type name BAD-NAME passed to chunk-type-possible-slot-names-fct.
|#
NIL

E> (chunk-type-possible-slot-names-fct 'chunk)
#|Warning: get-chunk-type called with no current model.|#
#|Warning: Invalid chunk-type name CHUNK passed to chunk-type-possible-slot-names-fct.|#
NIL

E> (chunk-type-possible-slot-names-fct 'clear)
#|Warning: get-chunk-type called with no current meta-process.|#
#|Warning: Invalid chunk-type name CLEAR passed to chunk-type-possible-slot-names-fct.|#
NIL
```

Chunk Commands

define-chunks

Syntax:

```
define-chunks ({chunk-name}{doc-string isa chunk-type | isa chunk-type} {slot value}*)* -> (chunk*)  
define-chunks-fct (({chunk-name}{doc-string isa chunk-type | isa chunk-type} {slot value}*)*) -> (chunk*)
```

Arguments and Values:

chunk-name ::= a symbol that will be the name of the chunk

doc-string ::= a string that will be the documentation for the chunk

chunk-type ::= a symbol that names a chunk-type in the model

slot ::= a symbol that names a slot for the chunk

value ::= any Lisp value which will be the contents of the correspondingly named slot for this chunk

chunk ::= a symbol which names a chunk that was created

Description:

The `define-chunks` command creates a new chunk in the current model of the current meta-process for each valid chunk description list provided and returns a list of the names of the chunks that were created.

Within a chunk description list the chunk name is optional. If a *chunk-name* is provided, it must not name an existing chunk in the current model and must be a non-keyword, non-nil symbol that begins with an alphanumeric character. If a *chunk-name* is not provided, a new name will be generated for the chunk, and that name is guaranteed to be unique.

The name may be followed by an optional *chunk-type* specification and if it does it may also include a documentation string for the chunk. If a *chunk-type* is specified it must name a valid chunk-type in the current model.

That may then be followed by any number of slot and value pairs for the chunk. If a given slot is named more than once in the definition then the last value it is given (rightmost) will be the one set for the chunk. If the value for a slot is non-nil then that slot is added to the chunk being created with the value specified.

If a *chunk-type* is specified then each of the slots specified will be tested to see if it is a valid slot for the type indicated. If it is not then a warning will be printed, but the chunk will still be created. If it

is not a valid slot for any chunk-type then it will automatically extend the valid slots for chunks with that slot name.

If a chunk-type is specified and there are slots with default values in that chunk-type's definition which are not included in the chunk definition the chunk will get all of those unspecified slots with their corresponding default values.

If a value for a slot is a non-nil symbol other than the Lisp true symbol `t` then it is assumed to be the name of a chunk. If there is not already a chunk by that name, then one is created automatically which has no slots and a warning is printed to indicate that.

However, within a call to `define-chunks` one can use the names of the chunks that are being defined in the other chunks without having them created as default chunks. Here is an example to clarify that:

```
(define-chunks (a slot b)
              (b slot a)
              (c slot d))
```

Because both `a` and `b` are being defined in the same call to `define-chunks` neither will need to be created automatically. However, unless `d` already names a chunk in the model a chunk with that name will be created automatically in the process of creating chunk `c`.

If the syntax is incorrect or any of the components are invalid in a description list then a warning is displayed and no chunk is created for that chunk description, but any other valid chunks defined will still be created.

If there is no current model or current meta-process then a warning is printed, no chunks are created and `nil` is returned.

Examples:

```
1> (chunk-type test1 slot1)
TEST1
```

```
2> (chunk-type test2 slot1 (slot2 2))
TEST2
```

```
3> (define-chunks (a isa test1) (b isa test2))
(A B)
```

```
4> (define-chunks-fct '((c slot1 a) (d slot2 c)))
(C D)
```

```
5> (define-chunks ("this is chunk e" isa test1 slot1 100))
(TEST10)
```

```

6E> (define-chunks-fct '((isa test1 slot2 "value")))
#|Warning: Invalid slot SLOT2 specified when creating chunk with type TEST1, but creating
chunk TEST11 anyway. |#
(TEST11)

7> (define-chunks ())
(CHUNK0)

8E> (define-chunks (slot1 new-name))
#|Warning: Creating chunk NEW-NAME with no slots |#
(CHUNK1)

9E> (define-chunks ("not allowed" slot2 t))
#|Warning: Invalid chunk definition: ("not allowed" SLOT2 T) chunk name is not a valid
symbol. |#
NIL

10E> (define-chunks (slot1 t new-slot 10))
#|Warning: Extending chunks with slot named NEW-SLOT because of chunk definition (SLOT1 T
NEW-SLOT 10) |#
(CHUNK2)

E> (define-chunks (z isa chunk))
#|Warning: define-chunks called with no current model. |#
NIL

E> (define-chunks (z isa chunk))
#|Warning: define-chunks called with no current meta-process. |#
NIL

```

pprint-chunks & pprint-chunks-plus

Syntax:

```

pprint-chunks chunk-name* -> [chunk-name-list | nil ]
pprint-chunks-fct (chunk-name*) -> [chunk-name-list | nil ]
pprint-chunks-plus chunk-name* -> [chunk-name-list | nil ]
pprint-chunks-plus-fct (chunk-name*) -> [chunk-name-list | nil ]

```

Arguments and Values:

chunk-name ::= a symbol that should be the name of a chunk
chunk-name-list ::= ([chunk-name | :error]*)

Description:

The pprint-chunks family of commands are used to print a description of each of the chunks specified, or all of the chunks in the model if no names are provided. The output is sent to the current model's command output stream.

For each chunk specified, on one line it will print the chunk's name followed by its "true name" in parentheses if the chunk's true name differs from the chunk's current name (see [merge-chunks](#) and [true-chunk-name](#) below for more details on a chunk's true name). If a documentation string was

provided for the chunk that is printed on the next line. Then one per line, each of the chunk's slots is printed followed by that slot's value.

The `pprint-chunks-plus` command prints all of the chunk's parameters after the description of the chunk is printed. The parameters are printed one per line with the name of the parameter and its current value. Note, that these chunk parameters are the ones that have been added to the chunks (typically by a module as described in the [extending chunks](#) section) and may not have any direct significance to the model or modeler. For example, the declarative memory parameters of chunks used by the [declarative module](#) to compute and record the activation of chunks should be viewed using the declarative module's `sdp` command because the values shown with `pprint-chunks-plus` are values used internally by the declarative module and may not adequately reflect the current value of activations as would be shown by calling the module's command for inspecting the declarative memory chunk's parameters ([sdp](#)).

These commands return a list with the names of all the chunks that were printed in the same order as they were specified. If an invalid chunk-name is given nothing is printed for that item and the value **:error** is returned in its place in the list.

If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

These examples assume that the chunks created in the examples for `define-chunks` exist.

```
> (pprint-chunks)
TEST10
"this is chunk e"
  SLOT1 100

BUSY
  NAME  BUSY

SPEAK
  NAME  SPEAK

CURRENT
  NAME  CURRENT

SUBVOCALIZE
  NAME  SUBVOCALIZE

AUDIO-EVENT
  NAME  AUDIO-EVENT

D
  SLOT2  C

CHUNK0
```

```

...

(TEST10 BUSY SPEAK CURRENT SUBVOCALIZE AUDIO-EVENT D CHUNK0 B MAGENTA ...)

> (pprint-chunks a b test10)
A

B
  SLOT2 2

TEST10
"this is chunk e"
  SLOT1 100

(A B TEST10)

> (pprint-chunks-fct '(c d))
C
  SLOT1 A

D
  SLOT2 C

(C D)

> (pprint-chunks-plus chunk0)
CHUNK0

--chunk parameters--
VISUAL-APPROACH-WIDTH-FN NIL
REAL-VISUAL-VALUE NIL
SPECIAL-VISUAL-OBJECT NIL
VISUAL-OBJECT NIL
VISUAL-TSTAMP NIL
VISUAL-FEATURE-NAME NIL
VISICON-ENTRY NIL
VISUAL-NEW-P NIL
SYNTH-FEAT NIL
FAST-MERGE-KEY NIL
RETRIEVAL-TIME NIL
RETRIEVAL-ACTIVATION NIL
SJIS NIL
PERMANENT-NOISE 0.0
SIMILARITIES NIL
REFERENCE-COUNT 0
REFERENCE-LIST NIL
SOURCE-SPREAD 0
LAST-BASE-LEVEL 0
BASE-LEVEL NIL
CREATION-TIME 0
FAN-IN NIL
C-FAN-OUT 0
FAN-OUT 0
IN-DM NIL
ACTIVATION 0
BUFFER-SET-INVALID NIL

(CHUNK0)

E> (pprint-chunks a bad-name b)
A

B
  SLOT2 2

(A :ERROR B)

```

```
E> (pprint-chunks)
#|Warning: pprint-chunks called with no current model. |#
NIL

E> (pprint-chunks)
#|Warning: pprint-chunks called with no current meta-process. |#
NIL
```

chunk-p

Syntax:

```
chunk-p chunk-name? -> [ t | nil ]
chunk-p-fct chunk-name? -> [ t | nil ]
```

Arguments and Values:

chunk-name? ::= a symbol which is being tested to determine if it names a chunk

Description:

The `chunk-p` command returns **t** if *chunk-name?* is a symbol that names a chunk in the current model of the current meta-process and returns **nil** if it does not. If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

These examples assume that chunks named `a` and `b` have been created.

```
> (chunk-p a)
T

> (chunk-p not-chunk)
NIL

> (chunk-p-fct 'b)
T

E> (chunk-p-fct 'a)
#|Warning: get-chunk called with no current model. |#
NIL

E> (chunk-p a)
#|Warning: get-chunk called with no current meta-process. |#
NIL
```

chunk-documentation

Syntax:

```
chunk-documentation chunk-name -> [ doc-string | nil ]
```

chunk-documentation-fct *chunk-name* -> [doc-string | **nil**]

Arguments and Values:

chunk-name ::= a value which should be a symbol that names a chunk

doc-string ::= a string of the documentation provided when *chunk-name* was created

Description:

Chunk-documentation returns the documentation string of the chunk *chunk-name* from the current model of the current meta-process if it names a valid chunk and has a documentation string. If it does not have a documentation string it returns **nil**. If *chunk-name* is not the name of a chunk in the current model, there is no current model, or no current meta-process then a warning is printed and **nil** is returned.

Examples:

These examples assume that the chunks created in the examples for `define-chunks` exist.

```
> (chunk-documentation test10)
"this is chunk e"
```

```
> (chunk-documentation a)
NIL
```

```
> (chunk-documentation-fct 'test10)
"this is chunk e"
```

```
E> (chunk-documentation-fct 'not-a-chunk)
#|Warning: NOT-A-CHUNK does not name a chunk in the current model.|#
NIL
```

```
E> (chunk-documentation c)
#|Warning: get-chunk called with no current model.|#
NIL
```

```
E> (chunk-documentation c)
#|Warning: get-chunk called with no current meta-process.|#
NIL
```

chunk-slot-value

Syntax:

chunk-slot-value *chunk-name slot-name* -> [slot-value | **nil**]

chunk-slot-value-fct *chunk-name slot-name* -> [slot-value | **nil**]

Arguments and Values:

chunk-name ::= a symbol which should be the name of a chunk

slot-name ::= a symbol that should be the name of a slot in the chunk *chunk-name*

slot-value ::= the Lisp value from *slot-name* in chunk *chunk-name*

Description:

Chunk-slot-value is used to get the value of a slot in a chunk. If chunk-name is the name of a chunk in the current model of the current meta-process and slot-name is the name of a slot in chunk-name then the value in the slot-name slot of the chunk chunk-name is returned. If chunk-name is the name of a chunk but it does not have a slot named slot-name then **nil** will be returned.

If chunk-name does not name a chunk in the model, there is no current model, or there is no current meta-process then a warning is printed and **nil** is returned.

Examples:

These examples assume that the chunks created in the examples for define-chunks exist.

```
> (chunk-slot-value c slot1)
A
> (chunk-slot-value c slot2)
NIL
>(chunk-slot-value-fct 'd 'slot2)
C
> (chunk-slot-value a other-slot)
NIL
E> (chunk-slot-value bad-chunk-name slot1)
#|Warning: BAD-CHUNK-NAME does not name a chunk in the current model.|#
NIL
E> (chunk-slot-value c slot1)
#|Warning: get-chunk called with no current model.|#
NIL
E> (chunk-slot-value c slot1)
#|Warning: get-chunk called with no current meta-process.|#
NIL
```

set-chunk-slot-value

Syntax:

```
set-chunk-slot-value chunk-name slot-name slot-value -> [ slot-value | nil ]
set-chunk-slot-value-fct chunk-name slot-name slot-value -> [ slot-value | nil ]
```

Arguments and Values:

chunk-name ::= a symbol which should be the name of a chunk
slot-name ::= a symbol that names a slot
slot-value ::= a Lisp value for slot-name in chunk chunk-name

Description:

Set-chunk-slot-value is used to set the value of the slot slot-name in the chunk chunk-name in the current model of the current meta-process to the value slot-value. If successful, slot-value is returned.

If slot-value is a symbol and not the name of a current chunk in the current model then it is created as a new chunk with no slots and a warning is displayed.

If a slot-value of **nil** is specified for a slot that will remove that slot from the chunk.

If the chunk chunk-name has been marked as immutable then a warning is printed, no changes are made to the chunk and **nil** is returned.

If either chunk-name or slot-name is invalid or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

These examples assume that the chunks created in the examples for define-chunks exist.

```
> (set-chunk-slot-value a slot1 10)
10

> (set-chunk-slot-value-fct 'b 'slot2 "value")
"value"

> (set-chunk-slot-value a slot2 new-chunk)
#|Warning: Creating chunk NEW-CHUNK with no slots|#
NEW-CHUNK

1> (make-chunk-immutable 'b)
T

2E> (set-chunk-slot-value b slot1 10)
#|Warning: Cannot change contents of chunk B.|#
NIL

E> (set-chunk-slot-value-fct 'not-a-chunk 'slot1 t)
#|Warning: NOT-A-CHUNK does not name a chunk in the current model.|#
NIL

E> (set-chunk-slot-value c not-a-slot b)
#|Warning: NOT-A-SLOT is not a valid slot name. You can use extend-possible-slots to add
it first if needed.|#
NIL

E> (set-chunk-slot-value a slot1 b)
#|Warning: get-chunk called with no current model.|#
NIL

E> (set-chunk-slot-value a slot1 b)
#|Warning: get-chunk called with no current meta-process.|#
NIL
```

mod-chunk

Syntax:

mod-chunk *chunk-name* {*slot-name slot-value*}* -> [*chunk-name* | **nil**]
mod-chunk-fct *chunk-name* ({*slot-name slot-value*}*) -> [*chunk-name* | **nil**]

Arguments and Values:

chunk-name ::= a symbol which should be the name of a chunk
slot-name ::= a symbol that should be the name of a slot in the chunk *chunk-name*
slot-value ::= a Lisp value for the corresponding *slot-name* in chunk *chunk-name*

Description:

Mod-chunk is used to set the value of multiple slots in the chunk named *chunk-name* in the current model in the current meta-process. It is essentially a short hand for multiple calls to set-chunk-slot-value.

If *chunk-name* is the name of a chunk in the current model and there are an even number of items specified thereafter, then those items are considered pair-wise to be the name of a slot and a value for that slot in that chunk. All of those slots in the chunk are set to the values specified and *chunk-name* is returned.

If any *slot-value* is a symbol and not the name of a chunk in the current model then it is created as a new chunk with no slots and a warning is displayed.

A slot name may only be specified once in the set of slot-names. If a slot name is specified more than once a warning is printed, no changes are made to the chunk, and **nil** is returned.

If any slot name provided is not a valid slot for chunks no changes are made to the chunk, and **nil** is returned.

If the chunk *chunk-name* has been marked as immutable then a warning is printed, no changes are made to the chunk and **nil** is returned.

If *chunk-name* does not name a chunk in the current model, there is no current model or current meta-process, or there are an odd number of items provided after the *chunk-name* then a warning is displayed, no changes are made, and **nil** is returned.

Examples:

These examples assume that the chunks created in the examples for define-chunks exist.

```
> (mod-chunk a slot1 b slot2 c)
A

> (mod-chunk-fct 'b '(slot1 10 new-slot t))
B

> (mod-chunk a slot1 new-chunk-name)
#|Warning: Creating chunk NEW-CHUNK-NAME with no slots|#
A

1> (make-chunk-immutable 'b)
T

2E> (mod-chunk b slot1 10)
#|Warning: Cannot modify chunk B because it is immutable.|#
NIL

E> (mod-chunk a slot1 1 slot1 2)
#|Warning: Slot name used more than once in modifications list.|#
NIL

E> (mod-chunk-fct 'a '(slot1 b slot2))
#|Warning: Odd length modifications list in call to mod-chunk.|#
NIL

E> (mod-chunk a slot1 b)
#|Warning: get-chunk called with no current model.|#
NIL

E> (mod-chunk a slot1 b)
#|Warning: get-chunk called with no current meta-process.|#
NIL
```

copy-chunk

Syntax:

```
copy-chunk chunk-name -> [new-name | nil ]
copy-chunk-fct chunk-name -> [new-name | nil ]
```

Arguments and Values:

chunk-name ::= a symbol which should be the name of a chunk
new-name ::= a symbol which will be a unique name for a new chunk

Description:

Copy-chunk creates a copy of the chunk *chunk-name* in the current model of the current meta-process and returns the name of the newly created chunk. The newly created copy has the same slots and values as the chunk *chunk-name*. The values of the parameters defined for the new chunk will have

the default value unless the parameter was specified with a copy-function, in which case, the value will be the one returned by that function.

If chunk-name does not name a chunk in the current model or there is no current model or meta-process then a warning is displayed and **nil** is returned.

Examples:

These examples assume that there are chunks named a and b in the current model.

```
> (copy-chunk a)
A-0

> (copy-chunk-fct 'b)
B-0

E> (copy-chunk not-a-chunk)
#|Warning: NOT-A-CHUNK does not name a chunk in the current model. |#
NIL

E> (copy-chunk a)
#|Warning: get-chunk called with no current model. |#
NIL

E> (copy-chunk a)
#|Warning: get-chunk called with no current meta-process. |#
NIL
```

chunk-copied-from

Syntax:

```
chunk-copied-from chunk-name -> [original-name | nil] {other}
chunk-copied-from-fct chunk-name -> [original-name | nil] {other}
```

Arguments and Values:

chunk-name ::= a symbol which should be the name of a chunk

original-name ::= a symbol which is the name of a chunk

other ::= a generalized boolean indicating whether this chunk was a copy and is still unmodified

Description:

The chunk-copied-from command returns one or two values. If chunk-name is the name of a chunk in the current model of the current meta-process then two values will be returned. If chunk-name was created with copy-chunk, it has not been modified since its creation, and the original chunk has not been modified such that it now differs from the copy (the original could have been modified but if it is still a match using [equal-chunks](#) it is still considered the same) then the name of the chunk from

which chunk-name was copied is returned as both the first and second value. If it was not created using copy-chunk, it has since been modified, or the original chunk has been modified in such a way that the two now differ (including being deleted) then **nil** is returned for the first value. If the chunk was created using copy-chunk and has not been modified then the second value will be the name of the chunk from which it was copied (which may no longer be an interned symbol if that chunk was purged) otherwise the second value will also be **nil**.

If chunk-name does not name a chunk in the current model or there is no current model or meta-process then a warning is displayed and a single value of **nil** is returned.

This command is rarely used by modelers because needing to copy chunks and keep track of how they came about are not typical actions. However, it can be important to those creating new modules where it can be used to determine if a chunk passed in as part of a request is a copy of a chunk which the module had placed into a buffer i.e. the request is using a copy of a chunk for which the module has created the original.

Examples:

```
1> (chunk-type test slot1 slot2)
TEST

2> (define-chunks (a slot1 10 slot2 "answer") (b slot1 a))
(A B)

3> (copy-chunk a)
A-0

4> (copy-chunk b)
B-0

5> (chunk-copied-from a-0)
A
A

6> (chunk-copied-from a)
NIL
NIL

7> (mod-chunk a slot1 5)
A

8> (chunk-copied-from-fct 'a-0)
NIL
A

9> (chunk-copied-from-fct 'b-0)
B
B

10> (mod-chunk b-0 slot2 10)
B-0

11> (chunk-copied-from b-0)
```

```
NIL
NIL
```

```
E> (chunk-copied-from-fct 'not-a-chunk)
#|Warning: NOT-A-CHUNK does not name a chunk in the current model.|#
NIL
```

```
E> (chunk-copied-from b-0)
#|Warning: get-chunk called with no current model.|#
NIL
```

```
E> (chunk-copied-from b-0)
#|Warning: get-chunk called with no current meta-process.|#
NIL
```

chunks

Syntax:

chunks -> [(chunk-name*) | nil]

Arguments and Values:

chunk-name ::= the name of a chunk in the current model

Description:

The `chunks` command returns a list of the names of all the chunks defined in the current model of the current meta-process in no particular order.

If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

```
> (chunks)
(GREEN CYAN SPEECH C DARK-CYAN RED-COLOR B-0 OVAL ...)
```

```
E> (chunks)
#|Warning: chunks called with no current model.|#
NIL
```

```
E> (chunks)
#|Warning: chunks called with no current meta-process.|#
NIL
```

chunk-slot-equal

chunk-slot-equal *val-1 val-2* -> equal-result

Arguments and Values:

val-1 ::= any value

val-2 ::= any value

equal-result ::= a generalized boolean indicating whether the chunks are equal

Description:

The chunk-slot-equal function is used to determine if two values are considered equivalent for the contents of slots in chunks. The values will be equivalent if one of the following is true:

- the values are eq
- both values are symbols which name chunks in the current model and those chunks return true from [eq-chunks](#)
- both values are strings and those strings return true from string-equal
- if the values are not both chunk names or not both strings and they return true from equalp

If the two values are equivalent, then a true value is returned. Otherwise, **nil** will be returned.

Examples:

```
> (chunk-slot-equal 1 1)
T

> (chunk-slot-equal 1 1.5)
NIL

> (chunk-slot-equal "String1" "strING1")
T

> (chunk-slot-equal 'not-a-chunk 'not-a-chunk)
T

> (chunk-slot-equal 'not-a-chunk :not-a-chunk)
NIL

1> (define-chunks (c1)(c2))
(C1 C2)

2> (chunk-slot-equal 'c1 'c2)
NIL

3> (merge-chunks c1 c2)
C1

4> (chunk-slot-equal 'c1 'c2)
T
```

equal-chunks

equal-chunks *chunk-name-1 chunk-name-2* -> equal-result
equal-chunks-fct *chunk-name-1 chunk-name-2* -> equal-result

Arguments and Values:

chunk-name-1 ::= a symbol that should be the name of a chunk
chunk-name-2 ::= a symbol that should be the name of a chunk
equal-result ::= a generalized boolean indicating whether the chunks are equal

Description:

The `equal-chunks` command can be used to determine if the chunks named by `chunk-name-1` and `chunk-name-2` in the current model of the current meta-process are equivalent chunks in the current model. They will be equivalent if they are [eq-chunks](#) or if they have the same set of slots and for each slot the values of those slots in the two chunks are the same as determined by the [chunk-slot-equal](#) function. If the two chunks are equivalent, then a true value is returned. Otherwise, **nil** will be returned.

If either name does not name a chunk or there is no current model or current meta-process, then a warning is printed and **nil** is returned.

Examples:

```
1> (chunk-type test1 slot1 slot2)
TEST1

2> (chunk-type test2 slot2 slot3)
TEST2

3> (define-chunks (c1 isa test1)
                  (c2 isa test2)
                  (c3 isa test1 slot1 10 slot2 "value")
                  (c4 isa test1 slot1 10 slot2 "VALUE")
                  (c5 isa test1 slot2 10)
                  (c6 isa test2 slot2 10))
(C1 C2 C3 C4 C5 C6)

4> (equal-chunks c1 c2)
T

5> (equal-chunks-fct 'c1 'c3)
NIL

6> (equal-chunks-fct 'c3 'c4)
T

7> (equal-chunks c5 c6)
T

8> (mod-chunk c5 slot3 t)
C5

9> (equal-chunks c5 c6)
NIL

E> (equal-chunks not-a-chunk free)
#|Warning: NOT-A-CHUNK does not name a chunk in the current model.|#
NIL
```

```
E> (equal-chunks c1 c2)
#|warning: get-chunk called with no current model.|#
#|warning: get-chunk called with no current model.|#
NIL
```

```
E> (equal-chunks c1 c2)
#|warning: get-chunk called with no current meta-process.|#
#|warning: get-chunk called with no current meta-process.|#
NIL
```

eq-chunks

Syntax:

```
eq-chunks chunk-name-1 chunk-name-2 -> equal-result
eq-chunks-fct chunk-name-1 chunk-name-2 -> equal-result
```

Arguments and Values:

chunk-name-1 ::= a symbol that should be the name of a chunk
chunk-name-2 ::= a symbol that should be the name of a chunk
equal-result ::= a generalized boolean indicating whether the chunks are the same

Description:

Eq-chunks is used to determine if the chunks named by *chunk-name-1* and *chunk-name-2* are the exact same chunk in the current model of the current meta-process. They will be the same chunk if *chunk-name-1* and *chunk-name-2* are the same symbol or if the two named chunks have been merged. If they are the same chunk, then a true value is returned. Otherwise **nil** will be returned.

If either name does not name a chunk or there is no current model or current meta-process, then a warning is printed and **nil** is returned.

Examples:

```
1> (chunk-type test1 slot1 slot2)
TEST1

2> (chunk-type test2 slot2 slot3)
TEST2

3> (define-chunks (c1) (c2)
      (c3 isa test1 slot2 10)
      (c4 isa test2 slot2 10))
(C1 C2 C3 C4)

4> (eq-chunks c1 c1)
T

5> (eq-chunks-fct 'c1 'c2)
NIL
```

```

6> (merge-chunks c1 c2)
C1

7> (eq-chunks c1 c2)
T

8> (eq-chunks c3 c4)
NIL

E> (eq-chunks c1 not-a-chunk)
#|warning: NOT-A-CHUNK does not name a chunk in the current model. |#
NIL

E> (eq-chunks c1 c2)
#|warning: get-chunk called with no current model. |#
#|warning: get-chunk called with no current model. |#
NIL

E> (eq-chunks c1 c2)
#|warning: get-chunk called with no current meta-process. |#
#|warning: get-chunk called with no current meta-process. |#
NIL

```

delete-chunk

Syntax:

```

delete-chunk chunk-name -> [ chunk-name | nil ]
delete-chunk-fct chunk-name -> [ chunk-name | nil ]

```

Arguments and Values:

chunk-name ::= a symbol that should be the name of a chunk

Description:

Delete-chunk removes the chunk named *chunk-name* from the set of chunks in the current model of the current meta-process. If *chunk-name* is the name of a chunk in the current model then after that chunk is deleted *chunk-name* is returned.

If *chunk-name* has been marked as immutable then it cannot be deleted, a warning will be printed and **nil** will be returned.

If *chunk-name* does not name a chunk in the current model or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Note: there is no additional clean-up done in conjunction with deleting the chunk. Thus, if it is used as a slot value in other chunks or currently residing in a buffer the consequences of deleting it are undefined and warnings or errors could result from later actions involving such chunks or buffers.

Delete-chunk should be used rarely and only when it is certain that the chunk being deleted is not referenced elsewhere.

Examples:

These examples assume that chunks named a, b, and c exist.

```
> (delete-chunk a)
A

1> (delete-chunk-fct 'b)
B

2E> (delete-chunk b)
#|Warning: B does not name a chunk in the current model. |#
NIL

1> (make-chunk-immutable 'c)
T

2E> (delete-chunk c)
#|Warning: Cannot delete chunk C because it is marked as immutable. |#
NIL

E> (delete-chunk c)
#|Warning: get-chunk called with no current model. |#
NIL

E> (delete-chunk c)
#|Warning: get-chunk called with no current meta-process. |#
NIL
```

purge-chunk

Syntax:

```
purge-chunk chunk-name -> [ t | nil ]
purge-chunk-fct chunk-name -> [ t | nil ]
```

Arguments and Values:

chunk-name ::= a symbol that should be the name of a chunk

Description:

Purge-chunk removes the chunk named *chunk-name* from the set of chunks in the current model of the current meta-process using [delete-chunk](#) and releases the name of that chunk using the [release-name](#) command as described under the [naming module](#). If *chunk-name* is the name of a chunk in the current model and its name was released then **t** is returned.

If `chunk-name` does not name a chunk in the current model or there is no current model or meta-process then a warning is printed and **nil** is returned.

If the chunk is deleted, but the name is not released **nil** is returned without a warning being printed.

As with `delete-chunk`, there is no additional clean-up done in conjunction with purging the chunk. Thus, if it is used as a slot value in another chunk or currently residing in a buffer undefined consequences could arise.

Because `purge-chunk` also attempts to unintern the name of the chunk it should only be used for chunks for which the name was automatically generated by ACT-R or explicitly generated with the [new-name](#) command. This is not going to be a command used by most modelers. However, in situations where (computer) memory usage is important in long running models or models which generate a lot of temporary chunks explicitly freeing some of that space may be necessary.

Examples:

These examples assume that there are chunks named a and b.

```
1> (copy-chunk b)
B-0

2> (purge-chunk-fct 'b-0)
T

3E> (purge-chunk b-0)
#|warning: B-0 does not name a chunk in the current model.|#
NIL

> (purge-chunk a)
NIL

E> (purge-chunk a)
#|warning: get-chunk called with no current model.|#
NIL

E> (purge-chunk a)
#|warning: get-chunk called with no current meta-process.|#
NIL
```

merge-chunks

Syntax:

```
merge-chunks chunk-name-1 chunk-name-2 -> [ chunk-name-1 | nil ]
merge-chunks-fct chunk-name-1 chunk-name-2 -> [ chunk-name-1 | nil ]
```

Arguments and Values:

chunk-name-1 ::= a symbol that should be the name of a chunk
chunk-name-2 ::= a symbol that should be the name of a chunk

Description:

If the chunks named by chunk-name-1 and chunk-name-2 are equivalent chunks as determined by [equal-chunks](#) then both chunks are replaced by a single chunk. Effectively, the two chunks are merged into one chunk. The “true name” of the merged chunk will be chunk-name-1, but references which use either name will still be valid and now refer to the single chunk resulting from the merge.

If the chunks are merged, then any additional chunk parameters that have been added to the chunks will remain those that existed for chunk-name-1 unless there is a merge-function defined for the parameter.

If either chunk is later deleted, both of the chunks will become unavailable i.e. deleting any one of a set of merged chunks deletes all of those merged chunks since there is only one underlying chunk.

If the chunks are equivalent as tested by [eq-chunks](#) then no actions are taken and chunk-name-1 is returned.

If the chunks are successfully merged, then chunk-name-1 is returned.

If the chunks are not equal-chunks **nil** is returned.

If either name does not name a chunk in the current model of the current meta-process or there is no current model or current meta-process then a warning is displayed and **nil** is returned.

The merge-chunks command is primarily for use by the declarative memory module, and it is not expected to be used elsewhere but is available if one finds a need. As with delete-chunk, it should be used carefully to avoid circumstances where chunks to which other modules already have references are merged which could result in unexpected consequences.

Examples:

```
1> (chunk-type test slot1 slot2)
TEST

2> (define-chunks (a slot1 10)
                  (b slot1 10)
                  (c slot1 10 slot2 t))
(A B C)

3> (merge-chunks a a)
```

```

A

4> (eq-chunks a b)
NIL

5> (merge-chunks-fct 'a 'b)
A

6> (eq-chunks a b)
T

7> (merge-chunks a c)
NIL

E> (merge-chunks a not-a-chunk)
#|Warning: NOT-A-CHUNK does not name a chunk in the current model. |#
NIL

E> (merge-chunks a b)
#|Warning: get-chunk called with no current model. |#
#|Warning: get-chunk called with no current model. |#
NIL

E> (merge-chunks a b)
#|Warning: get-chunk called with no current meta-process. |#
#|Warning: get-chunk called with no current meta-process. |#
NIL

```

create-chunk-alias

Syntax:

```

create-chunk-alias chunk-name alias -> [ alias | nil ]
create-chunk-alias-fct chunk-name alias -> [alias | nil ]

```

Arguments and Values:

chunk-name ::= a symbol that should be the name of a chunk
alias ::= a symbol that should not be the name of a chunk

Description:

If the chunk specified by *chunk-name* exists in the current model of the current meta-process and the symbol provided as *alias* is not the name of a chunk in the current model of the current meta-process then *alias* will be added as a reference to the chunk *chunk-name*. This works essentially the same as if a chunk named *alias* had been merged with the chunk named *chunk-name*.

If the *alias* is successfully created, then *alias* is returned.

If *chunk-name* does not name a chunk in the current model, *alias* is not a symbol, *alias* is already the name of a chunk in the model, or there is no current model or meta-process then a warning is displayed and **nil** is returned.

This command is not likely to be used often, but it may be helpful if a chunk which has a long name is generated automatically by the model and one needs to perform lots of actions or tests using that chunk.

Examples:

```
1> (define-chunks (a) (b))
(A B)

2> (chunk-p alias)
NIL

3> (create-chunk-alias a alias)
ALIAS

4> (eq-chunks a alias)
T

5E> (create-chunk-alias-fct 'a 'b)
#|Warning: B is already the name of a chunk in the current model and cannot be used as an
alias. |#
NIL

E> (create-chunk-alias c new-chunk-name)
#|Warning: C is not the name of a chunk in the current model. |#
NIL

E> (create-chunk-alias a alias)
#|Warning: create-chunk-alias called with no current model. |#
NIL

E> (create-chunk-alias a alias)
#|Warning: create-chunk-alias called with no current meta-process. |#
NIL
```

true-chunk-name

Syntax:

```
true-chunk-name chunk-name -> [ true-name | chunk-name ]
true-chunk-name-fct chunk-name -> [ true-name | chunk-name ]
```

Arguments and Values:

chunk-name ::= any Lisp value

true-name ::= a symbol that is the name of a chunk in the current model

Description:

True-chunk-name is used to find the “true name” of a chunk in the current model of the current meta-process. The true name of a chunk which has not been merged with another chunk is its own name. The true name of a chunk that has been merged with another chunk is the true name of the chunk that

was returned from a merging of that chunk with another. The true name of a chunk alias is the true name of the chunk to which it was aliased.

If chunk-name is the name of a chunk in the current model then its true name is returned. If chunk-name is any other value, then chunk-name is returned.

If there is no current model or current meta-process then a warning is printed and chunk-name is returned.

Examples:

```
1> (define-chunks (a) (b) (c))
(A B C)

2> (merge-chunks a b)
A

3> (create-chunk-alias b alias)
ALIAS

4> (true-chunk-name a)
A

5> (true-chunk-name-fct 'c)
C

6> (true-chunk-name b)
A

7> (true-chunk-name alias)
A

8> (true-chunk-name d)
D

> (true-chunk-name 100)
100

E> (true-chunk-name t)
#|Warning: get-chunk called with no current model.|#
T

E> (true-chunk-name "Name")
#|Warning: get-chunk called with no current meta-process.|#
"Name"
```

normalize-chunk-names

normalize-chunk-names { *unintern* } -> nil

Arguments and Values:

unintern ::= a generalized boolean indicating whether to delete the merged chunks and release the names

Description:

The `normalize-chunk-names` command will iterate through all chunks in the current model and replace all chunk references in slots with the true name of that chunk. That may be useful for debugging purposes and the [naming module](#) has a parameter (`:ncnar`) which can trigger this call automatically.

In addition, if the `unintern` parameter is true then all chunks which have been merged with other chunks (those for which their name is not the chunk's true name) will be deleted from the model and the chunk name will be released using [release-name](#).

The command will always return **nil**. If there is no current model or current meta-process, then a warning will be printed indicating that.

Notes: This command may take a long time to run if the model has a large number of chunks. Also, the `unintern` option is generally not recommended because it may cause problems for modules which have stored internal references to those temporary names. However, in some extreme circumstances (a very long continuous run or a model which does a lot of buffer manipulations over a long run) a model can generate so many chunk name symbols that it can become unable to continue running (the Lisp heap or the physical memory of the machine is exhausted) thus calling `normalize-chunk-names` periodically with the `unintern` option would be necessary to continue running. If you are encountering such situations, please let me know about it because there may be other options or changes that could be made.

Examples:

To show the command in use, there must be a chunk which has been merged with another and also used in a slot value.

```
1> (chunk-type test slot1 slot2)
TEST

2> (define-chunks (c1 slot1 10 slot2 t)
                  (c2 slot1 1 slot2 t)
                  (c3 slot1 c2))
(C1 C2 C3)

3> (pprint-chunks c3)
C3
  SLOT1  C2

(C3)

4> (merge-chunks c1 c2)
```

C1

```
5> (normalize-chunk-names)
NIL
```

```
6> (pprint-chunks c3)
C3
  SLOT1  C1
```

(C3)

```
7> (pprint-chunks c2)
C2 (C1)
  SLOT1  1
  SLOT2  T
```

(C2)

```
8> (normalize-chunk-names t)
NIL
```

```
9> (pprint-chunks c2)
(:ERROR)
```

```
E> (normalize-chunk-names)
#|Warning: No current model in which to normalize chunk names.|#
NIL
```

General Parameters

General parameters are the primary means of configuring the operation of ACT-R both from a usability standpoint and at the level of controlling the performance of a model. They can be used to control how much output is shown when a model runs or to adjust how long it takes a model to retrieve a chunk from its declarative memory as well as many other things. Each module in the system can make available any number of general parameters that are relevant to its operation. The specific parameters of each module will be described in that module's section. In this section, the common aspects of those parameters will be described along with the command that is used to set, get, and show them.

The general parameters are each referenced by a name which is a Lisp keyword e.g. `:v` or `:trace-detail`, and can be set to some value which is meaningful to the module that owns the parameter. Each one has a default value specified by the owning module and often there are limits as to what values can be given to a particular parameter. Attempting to set an invalid value will result in a warning and no change in the parameter. In most cases the parameters are independent between models i.e. two concurrent models could have different values for the same general parameter. However, there can be exceptions to that. For example, a module that allows models to connect to some external simulation might provide parameters to specify where to connect to that simulation, but may require that all models connect to the same simulation. None of the provided modules operate that way, but it is worth noting that modules could be added which have parameters that are linked between models.

One final thing to note about general parameters is that it is possible for modules other than the parameter's owning module to monitor the parameter setting and possibly modify the parameter. The details of doing that are covered in the [module creation](#) section. Because of that one should be aware that it is possible for parameters to start with values other than their specified default after a model is reset or to be set to a value different than one the user requests if a monitoring module changes it. An example of the first situation (a starting value other than the default) exists in the main ACT-R system with the `:do-not-harvest` parameter. The procedural module owns that parameter and specifies a default value of `nil`, but the goal module will change that parameter at reset time to include the goal buffer. It is also the case that if one starts the ACT-R Environment then several of the tracing and hook parameters will be set automatically to values which allow the Environment to operate. There are no modules in the provided set which modify the values a user specifies, but an example where such a situation could be used might be a module which provides support for modeling alertness or

sleepiness. It could automatically adjust the parameters that a user specifies for controlling other modules to take into account the current alertness setting. Of course it does not have to work that way, but it is a possibility which modelers and module writers should know about.

Commands

sgp

Syntax:

```
sgp {[ param-name* | param-value-pair* ] -> [ nil | ([param-value | :bad-parameter-name | :invalid-value]* ) ]  
sgp-fct ({[ param-name* | param-value-pair* ]}) -> [ nil | ([param-value | :bad-parameter-name |  
:invalid-value]* ) ]
```

Arguments and Values:

param-name ::= a keyword which names a parameter

param-value-pair ::= *param-name new-param-value*

new-param-value ::= a Lisp value to which the preceding *param-name* is to be set

param-value ::= the current value of a *param-name*

Description:

Sgp is used to set or get the value of the parameters from the modules of the current model in the current meta-process.

If no parameters are provided, all of the current model's parameters are printed to the model's command output stream. They are organized alphabetically by module name and then by parameter name within a module, and **nil** is returned. For each parameter, its name and current value is printed followed by the default value and any documentation provided by the module that owns the parameter.

If all of the parameters passed to sgp are keywords, then it is a request for the current values of those general parameters' values in the current model. Those parameters are printed and a list of their values in the order requested is returned. If any of the names are not of valid parameters then a warning is displayed and the keyword **:bad-parameter-name** is returned for that position in the list. Note: because the test to determine that a call to sgp is a request for parameter values is that all the values passed to sgp are keywords, a module should never have a parameter accept a keyword as a possible value because it will not be possible to set such a parameter value on its own.

If there are any non-keyword parameters in the call to `sgp` and the total number of elements is even, then they are assumed to be pairs of a parameter name and a parameter value. Each of those parameter values will be passed to the corresponding parameter's owning module and all monitoring modules. The return value will be the current settings of those parameters in the order given (the values may or may not be the same as the values passed in to set them depending on the module) unless a parameter value was not of the appropriate type as required by the module. In that case, a warning is printed and the value returned in that position will be the keyword `:invalid-value`.

If there are non-keywords passed to `sgp` and the number of items is odd, or if there is no current model or current meta-process at the time of the call, then a warning is displayed and **nil** is returned.

Examples:

```
> (sgp)
-----
:AUDIO module
-----
:DIGIT-DETECT-DELAY 0.3  default: 0.3  : Lag between onset and detectability for digits
:DIGIT-DURATION     0.6  default: 0.6  : Default duration for digit sounds.
:DIGIT-RECODE-DELAY 0.5  default: 0.5  : Recoding delay for digit sound content.
:HEAR-NEWEST-ONLY  NIL  default: NIL   : Whether to stuff only the newest unattended
audio-event from the audicon into the aural-location buffer.
:SOUND-DECAY-TIME   3.0  default: 3.0  : The amount of time after a sound has finished
it takes for the sound to be deleted from the audicon
:TONE-DETECT-DELAY  0.05 default: 0.05  : Lag between sound onset and detectability for
tones
:TONE-RECODE-DELAY  0.285 default: 0.285 : Recoding delay for tone sound content.
-----
BOLD module
-----
...

> (sgp :v :lf)
:V T (default T) : Verbose controls model output
:LF 1.0 (default 1.0) : Latency Factor
(T 1.0)

> (sgp-fct '(:v nil :lf 4.5))
(NIL 4.5)

E> (sgp-fct '(:v t :lf nil))
#|warning: Parameter :LF cannot take value NIL because it must be a positive number.|#
(T :INVALID-VALUE)

E> (sgp :not-a-parameter 10)
#|warning: Parameter :NOT-A-PARAMETER is not the name of an available parameter|#
(:BAD-PARAMETER-NAME)

E> (sgp :esc t :v)
#|warning: Odd number of parameters and values passed to sgp.|#
NIL

E> (sgp)
#|warning: sgp called with no current model.|#
NIL
```

```
E> (sgp)
#|Warning: sgp called with no current meta-process.|#
NIL
```

get-parameter-default-value

Syntax:

get-parameter-default-value *param-name* -> [param-default | **:bad-parameter-name**]

Arguments and Values:

param-name ::= a keyword which should name a parameter

param-default ::= the default value specified for *param-name* when it was defined

Description:

The `get-parameter-default-value` command is used to get the default value that a parameter was given when it was defined. If *param-name* is the name of a general parameter then the default value specified for that parameter is returned. If *param-name* does not name a general parameter then a warning is printed and **:bad-parameter-name** is returned.

Examples:

```
> (get-parameter-default-value :v)
T
```

```
E> (get-parameter-default-value :not-a-param)
#|Warning: Invalid parameter name :NOT-A-PARAM in call to get-parameter-default-value.|#
:BAD-PARAMETER-NAME
```

with-parameters

Syntax:

with-parameters *parameter-list form** -> [result | nil]

with-parameters-fct *parameter-list form** -> [result | nil]

Arguments and Values:

parameter-list ::= (*{param-name value}**)

param-name ::= a keyword which should name a parameter

value ::= a Lisp value to which the preceding parameter should temporarily be set

form ::= a valid Lisp expression to evaluate

result ::= the value returned from the last form evaluated

Description:

The `with-parameters` command is used to temporarily set some parameters in the current model of the current meta-process and then execute some commands. If all of the `param-name` values provided name valid parameters then each will be set to the corresponding value given before executing the forms. After those forms have been evaluated each of those parameters will be set back to the value it had previously and the result of the last form evaluated will be returned. The forms are evaluated in an `unwind-protect` so that the restoring of the parameters occurs even if the forms result in an error.

If any of the `param-name` values do not name a valid parameter or there is no current model or current meta-process then a warning will be printed, the forms will not be evaluated, and `nil` is returned.

The difference between `with-parameters` and `with-parameters-fct` is not quite the same as it is for other commands. In this case both are macros, but `with-parameters-fct` evaluates the items on the `parameter-list` and `with-parameters` does not. Thus the `parameter-list` for `with-parameters` will look similar to what one would provide to `sgp` whereas the `parameter-list` for `with-parameters-fct` may contain expressions and variables which need to be evaluated.

Examples:

This example sequence assumes that the `count` model from unit 1 of the tutorial is loaded.

```
1> (reset)
DEFAULT

2> (run .05)
  0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000 PROCEDURAL CONFLICT-RESOLUTION
  0.000 PROCEDURAL PRODUCTION-SELECTED START
  0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
  0.050 PROCEDURAL PRODUCTION-FIRED START
  0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
  0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
  0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
  0.050 DECLARATIVE START-RETRIEVAL
  0.050 PROCEDURAL CONFLICT-RESOLUTION
  0.050 ----- Stopped because time limit reached

0.05
13
NIL

3> (with-parameters (:v nil)
   (run .05))
0.05
8
NIL

4> (with-parameters-fct (:trace-detail 'low)
   (run .05))
  0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT

2
```

```

0.150  ----- Stopped because time limit reached
0.05
6
NIL

5> (run .05)
0.200  DECLARATIVE      RETRIEVED-CHUNK D
0.200  DECLARATIVE      SET-BUFFER-CHUNK RETRIEVAL D
0.200  PROCEDURAL       CONFLICT-RESOLUTION
0.200  PROCEDURAL       PRODUCTION-SELECTED INCREMENT
0.200  PROCEDURAL       BUFFER-READ-ACTION GOAL
0.200  PROCEDURAL       BUFFER-READ-ACTION RETRIEVAL
0.200  ----- Stopped because time limit reached
0.05
7
NIL

E> (with-parameters (:not-valid 10)
    (run .05))
#|Warning: :NOT-VALID is not the name of a parameter. with-parameters body ignored.|#
NIL

E> (with-parameters-fct (:v)
    (run .05))
#|Warning: Odd length parameters list in call to with-parameters. The body is ignored.|#
NIL

E> (with-parameters (:v t)
    (run 1))
#|Warning: with-parameters called with no current model.|#
NIL

E> (with-parameters (:v t)
    (run 1))
#|Warning: with-parameters called with no current meta-process.|#
NIL

```

System Parameters

System parameters are similar to general parameters, but they are only used for configuring the operation of the ACT-R software itself. They do not have to be connected to any particular module or model and changing one will affect all models. They also retain their settings across a reset or clear-all therefore they will generally only need to be set once if one needs to use them and probably will not be included in a model definition.

Commands

ssp

Syntax:

```
ssp {[ param-name* | param-value-pair*]} -> [ nil | ([param-value | :bad-parameter-name | :invalid-value]*) ]  
ssp-fct ({[ param-name* | param-value-pair*]) -> [ nil | ([param-value | :bad-parameter-name |  
:invalid-value]*) ]
```

Arguments and Values:

param-name ::= a keyword which names a parameter

param-value-pair ::= *param-name new-param-value*

new-param-value ::= a Lisp value to which the preceding param-name is to be set

param-value ::= the current value of a param-name

Description:

The ssp command is used to set or get the value of the system parameters.

If no parameters are provided, all of the current system parameters are printed to **standard-output** and **nil** is returned. For each parameter, its name and current value is printed followed by the default value and any documentation it has.

If all of the parameters passed to ssp are keywords, then it is a request for the current values of those parameters. Those parameters are printed and a list of their values in the order requested is returned. If any of the names are not of valid parameters then a warning is displayed and the keyword **:bad-parameter-name** is returned for that position in the list.

If there are any non-keyword parameters in the call to ssp and the total number of elements is even, then they are assumed to be pairs of a parameter name and a parameter value. Each of those parameters will be set to the provided value. The return value will be the current settings of those

parameters in the order given unless a parameter value was not an appropriate value. In that case, a warning is printed and the value returned in that position will be the keyword `:invalid-value`.

If there are non-keywords passed to `ssp` and the number of items is odd then a warning is displayed and `nil` is returned.

Examples:

```
> (ssp)
:DEFAULT-ENVIRONMENT-PORT 2621          default: 2621          : Default port ...
:STARTING-PARAMETERS      NIL          default: NIL          : Parameter ...
:SAFE-PERCEPTUAL-BUFFERS (TEMPORAL ...) default: (TEMPORAL ...) : Perceptual ...
:MCTRT                    NIL          default: NIL          : rehash-threshold ...
:DEFAULT-ENVIRONMENT-HOST "127.0.0.1" default: "127.0.0.1"  : Default address ...
:MCTS                     NIL          default: NIL          : initial size of ...
:CLOSE-EXP-WINDOWS-ON-RESET NIL        default: NIL          : Whether the AGI ...

> (ssp :default-environment-port)
(2621)

> (ssp :default-environment-port 242)
(242)

E> (ssp :default-environment-port 'x)
#|Warning: System parameter :DEFAULT-ENVIRONMENT-PORT cannot take value (QUOTE X) because
it must be positive number.|#
(:INVALID-VALUE)

E> (ssp :bad-name)
(:BAD-PARAMETER-NAME)

E> (ssp :default-environment-port 100 :mctrt)
#|Warning: Odd number of parameters and values passed to ssp.|#
NIL
```

Printing and Output

Many of the commands in ACT-R result in output being printed. There is a [printing module](#) which can be used to control where and when certain things are printed, and that is described in detail in a separate section. For now the general aspects of the output will be described as well as the commands that are used to generate the output.

There are three basic types of output that ACT-R generates: model output, command output, and warnings. They are generated by different output functions described below and it is up to module writers to use the appropriate ones for any output which they create so as to conform to what is expected.

Model Output

Model output is essentially all the things that are printed by a running model. The trace of the model is considered model output as are various internal module specific traces and notices. The default is to send the model output to the Lisp stream `*standard-output*`, but the [printing module](#) has a [parameter named :v](#) which allows one to send the output elsewhere or disable it.

Command Output

Command output is what gets printed when one calls one of the ACT-R commands, for example the parameter listing when one calls `sgp`. By default this is also sent to the `*standard-output*` stream. It is controlled by a separate printing module [parameter named :cmdt](#). Thus, one could have model output going one place and command output going elsewhere if desired. Often, one does not need or want the printed output from an ACT-R command because only the returned value is important. In those situations, there is a [command called no-output](#) that can be used to temporarily disable command output.

Warnings

Warnings from ACT-R are always enclosed inside of a Lisp comment block (between the characters `#|` and `|#`) and start with “Warning:”. The reason they are inside a comment block is so they do not create a problem if someone is using Lisp to read an output file generated by a model trace which might contain warnings. It also distinguishes them from any other warnings that may occur within the Lisp. There are two general classes of warnings and they are created with different commands. The first is referred to as model warnings. These are things like “undefined chunk FOO being

created with no slots.” They inform the modeler of something that was assumed or may be unusual within a model. They are generally just hints or suggestions and can often be ignored. In fact, there is a parameter switch to actually suppress such warnings if desired (though if the model is not working as one would expect turning the model warnings back on and reading them carefully is probably a good first thing to look at). The other type is just referred to as a warning, and they are generated when an ACT-R command receives invalid parameters or a more serious issue has occurred e.g. the “... called with no current model” warning. These are usually more important issues and cannot be turned off with a simple switch. Warnings are sent to the Lisp stream **error-output**. Model warnings are sent to both the stream **error-output** as well as to the current model’s model output stream if it differs from **error-output**.

Below are the commands that are used by the system for outputting information and which are available for use if one wants their model or experiments using ACT-R to print in the ways described above. Those adding new modules to the system should use the appropriate commands for any output that their module generates.

Commands

model-output

Syntax:

model-output *control-string* {*args**} -> **nil**

Arguments and Values:

control-string ::= a Lisp format control (a format string or function as returned by the formatter macro)
args ::= arguments as required by the control-string provided

Description:

Model-output is used to print output to the current model’s model output stream, which defaults to **standard-output** and is controlled by the [printing module](#). It passes the provided control-string and args to format for output followed by a new line if model output is enabled for the current model. It does not test the control-string or args for correctness, so any problems will likely trigger an error or warning from the format function.

If there is no current model a warning is printed.

It always returns **nil**.

Examples:

```
> (model-output "This is ~A the ACT-R ~d model-output command" "output from" 6)
This is output from the ACT-R 6 model-output command
NIL
```

```
E> (model-output "This is ~A the ACT-R ~d model-output command" "output from" 6)
#|warning: get-module called with no current model.|#
NIL
```

meta-p-output

Syntax:

meta-p-output *control-string* {*args**} -> **nil**

Arguments and Values:

control-string ::= a Lisp format control (a format string or function as returned by the formatter macro)
args ::= arguments as required by the control-string provided

Description:

Meta-p-output is used to print output to all of the models in the current meta-process. It uses each model's model output stream as if model-output were used, but only prints once to a given stream in the event that more than one model is using the same stream for model output. It does not test the control-string or args for correctness, so any problems will likely trigger an error or warning from the format function.

If there is no current meta-process a warning is printed.

It always returns **nil**.

Meta-p-output is currently used for actually printing the trace because there can be multiple models running concurrently. It is not likely that users or module writers will have need for meta-p-output because it is above the level of a model or module, but it is described because its results are seen when using simultaneous multiple models.

One thing to note about meta-p-output is that it will evaluate the args separately for each stream to which the output is written. If there are no output streams (all models have :v set to **nil** for example) then the args are not evaluated. Thus, if there are any actions with side effects in the args the results could differ when the number of streams to which output is written changes.

Examples:

```
> (meta-p-output "This is from ~s" "meta-p-output")
This is from "meta-p-output"
NIL

E> (meta-p-output "This is from ~s" "meta-p-output")
#|Warning: No current meta-process in call to meta-p-output |#
NIL
```

command-output

Syntax:

command-output *control-string* {*args**} -> **nil**

Arguments and Values:

control-string ::= a Lisp format control (a format string or function as returned by the formatter macro)
args ::= arguments as required by the control-string provided

Description:

Command-output is used to print output to the current model's command output stream, which defaults to **standard-output** and is controlled by the [printing module](#). It passes the provided control-string and args to format for output followed by a new line if command output is enabled for the current model. It does not test the control-string or args for correctness, so any problems will likely trigger an error or warning from the format function.

If there is no current model a warning is printed.

It always returns **nil**.

Command-output is intended for use by things that print in response to being called outside of a model run, like the display of parameters from *sgp* or the chunk printing from *pprint-chunks* and can be turned off by the modeler using a parameter or the [no-output](#) command.

Examples:

```
> (command-output "A command-output ~s" 'example)
A command-output EXAMPLE
NIL

E> (command-output "A command-output ~s" 'example)
#|Warning: get-module called with no current model. |#
NIL
```

no-output

Syntax:

no-output {forms*} -> [result | nil]

Arguments and Values:

forms ::= a Lisp form to evaluate

result ::= the return value from the last form evaluated

Description:

No-output is used to disable the command output stream of the current model while evaluating the forms provided. It returns the value returned by the last form evaluated.

If there is no current model a warning is printed and **nil** is returned.

No-output can be useful if one wants to get the results from some other ACT-R command without having to see any of its output and without needing to explicitly disable and then possibly re-enable the command output parameter.

Examples:

```
> (no-output (pprint-chunks))
(EXTERNAL LIGHT-GRAY INTERNAL DIGIT CURRENT FULL FREE BLACK ...)

> (no-output (sgp-fct '(:v :lf)))
(T 1.0)

E> (no-output (sgp-fct '(:v :lf)))
#|Warning: get-module called with no current model.|#
NIL
```

capture-model-output

Syntax:

capture-model-output {forms*} -> [output-string | nil]

Arguments and Values:

forms ::= a Lisp form to evaluate

result ::= a string containing all the model output generated by the forms

Description:

Capture-model-output is used to save a model's output to a string and return it. It sets both the model output and command output streams to a string-output stream before evaluating the forms provided. After evaluating the forms the output streams are returned to the values they had when the command started. It returns the string generated by the string-output stream.

If there is no current model a warning is printed and **nil** is returned.

Examples:

```
> (capture-model-output (pprint-chunks free busy))
"FREE
  ISA CHUNK

BUSY
  ISA CHUNK

"

> (capture-model-output (model-output "Model...") (command-output "Command..."))
"Model...
Command...
"

E> (capture-model-output (pprint-chunks))
#|warning: get-module called with no current model.|#
NIL
```

print-warning

Syntax:

print-warning *control-string* {*args**} -> [result | nil]

Arguments and Values:

control-string ::= a Lisp format control (a format string or function as returned by the formatter macro)
args ::= arguments as required by the control-string provided
result ::= a string containing the output

Description:

Print-warning is used to print a warning message to **error-output**. It passes the provided control-string and args to format for output after printing “#| Warning:” and followed by “|#” and a new line. It does not test the control-string or args for correctness, so any problems will likely trigger an error or warning from the format function.

If **error-output** is **nil** then the output string from the call to format is returned, otherwise it returns **nil**.

Print-warning is intended for use in printing important notices of problems or errors that occurred within a module or command.

Examples:

```
> (print-warning "This is a warning from ACT-R ~a" "!!")
#|Warning: This is a warning from ACT-R !! |#
NIL
```

model-warning

Syntax:

model-warning *control-string* {*args**} -> **nil**

Arguments and Values:

control-string ::= a Lisp format control (a format string or function as returned by the formatter macro)
args ::= arguments as required by the control-string provided

Description:

Model-warning is used to print a warning to the current model's model output stream and to *standard-error* if it is a different stream. It passes the provided control-string and args to format for output after printing “#| Warning:” and followed by “|#” and a new line if model output is enabled for the current model and model-warnings are not disabled. It does not test the control-string or args for correctness, so any problems will likely trigger an error or warning from the format function.

If there is more than one model currently defined then the warning will also include the name of the model in which the warning was generated. If there is no current model or meta-process then a warning about that situation is printed instead of the specified warning.

It always returns **nil**.

Model-warning is intended for use when the model causes a problem within a module or a less serious situation has occurred which the modeler might want to be informed about but which may often be safely ignore.

Examples:

```
> (model-warning "This may not be what you wanted: ~s" 'bad-value)
#|Warning: This may not be what you wanted: BAD-VALUE |#
```

NIL

```
> (with-model bar (model-warning "There is more than one model defined."))
#|Warning (in model BAR): There is more than one model defined. |#
NIL
```

```
E> (model-warning "This may not be what you wanted: ~s" 'bad-value)
#|Warning: get-module called with no current model. |#
NIL
```

```
E> (model-warning "Too many meta-processes")
#|Warning: get-module called with no current meta-process. |#
NIL
```

one-time-model-warning

Syntax:

one-time-model-warning *tag control-string {args*}* -> nil

Arguments and Values:

tag ::= any Lisp value

control-string ::= a Lisp format control (a format string or function as returned by the formatter macro)

args ::= arguments as required by the control-string provided

Description:

One-time-model-warning is used to print a warning to the current model's model output stream and to **standard-error** if it is a different stream like [model-warning](#) does. It passes the provided control-string and args to format for output after printing “#| Warning:” and followed by “|#” and a new line if model output is enabled for the current model, model-warnings are not disabled, and this is the first time since that model has been reset that a warning with the specified tag has been issued as tested using the Lisp equal function. It does not test the control-string or args for correctness, so any problems will likely trigger an error or warning from the Lisp format function.

If there is more than one model currently defined then the warning will also include the name of the model in which the warning was generated. If there is no current model or meta-process then a warning about that situation is printed instead of the specified warning.

It always returns **nil**.

Like [model-warning](#), [one-time-model-warning](#) is intended for use when the model causes a problem within a module or a less serious situation has occurred which the modeler might need to be informed

about but which may often be safely ignore. One-time-model-warning can be used if that situation may occur repeatedly to avoid a new warning being printed each time.

Examples:

```
1> (one-time-model-warning :demo "This is a warning")
#|Warning: This is a warning |#
NIL

2> (one-time-model-warning :demo "This is a warning")
NIL

3> (one-time-model-warning "different tag" "Another warning")
#|Warning: Another warning |#
NIL

> (with-model bar (one-time-model-warning :other "the ~s warning" 'first))
#|Warning (in model BAR): the FIRST warning |#
NIL

E> (one-time-model-warning :demo "This is a warning")
#|Warning: get-module called with no current model. |#
NIL

E> (one-time-model-warning :demo "This is a warning")
#|Warning: get-module called with no current meta-process. |#
NIL
```

suppress-warnings

Syntax:

```
suppress-warnings {form*} -> result
```

Arguments and Values:

form ::= a Lisp form to evaluate
result ::= the return value from the last form evaluated

Description:

Suppress-warnings is used to turn off all ACT-R warnings which would normally be shown while evaluating the forms provided. It returns the value returned by the last form evaluated.

Note: suppress-warnings may also stop the output of Lisp warnings and errors because it binds ***error-output*** to a null stream during the evaluation of the forms. Because of that, suppress-warnings is only recommended for use in situations where one is certain that the code is correct and the warnings can be ignored.

Examples:

```
1> (progn (add-dm (a isa visual-object value b)) (sgp :lf .5))
#|Warning: Creating chunk B of default type chunk |#
#|Warning: Changing declarative parameters with chunks in dm not supported. |#
#|Warning: Results may not be what one expects. |#
(0.5)

2> (reset)
DEFAULT

3> (suppress-warnings (add-dm (a isa visual-object value b)) (sgp :lf .5))
(0.5)
```

Running the system

Running the system means executing the events that are on the queue of the current meta-process. Those events may lead to other events being scheduled and that will continue until the condition specified for the command used to run the system is met. There are several commands for running the system which specify various stopping conditions as well as allowing users to specify arbitrary end conditions.

The system can run in either a simulated time frame where the events are processed as fast as possible or in “real time” where the execution of the events is synchronized with the passing of time from some other source. Generally, running in real time is associated with the actual passage of time (with a scale factor which can be specified by the modeler) and the model is constrained to that, but it is possible to synchronize it with [other time sources](#). For now, we will focus mostly on the simulated time operation.

When running in simulated time the time stamps on the events control the advancement of the clock in the meta-process. When a meta-process is created or whenever it is reset the current time is set to 0.0. The event with the lowest time stamp is always the next one executed and if that time is greater than the current time the clock is updated. This allows the system to run much faster than real time for most models. The important thing to remember is that the timing of the events is produced by the modules which instantiate the ACT-R theory and thus the predictions do not depend on how the model is run or the source of the clock.

An important thing to note is that it is a meta-process which is run. All of the models that are included in that meta-process will be running simultaneously. The same commands are used to run the system regardless of how many models or meta-processes are defined.

Commands

run

Syntax:

```
run run-time {:real-time real-time?} -> [ nil | time-passed event-count break? ]
```

Arguments and Values:

run-time ::= a number greater than 0 indicating the number of seconds to run

real-time? ::= a generalized boolean to indicate whether to run in real time and possibly the scale for

the real time clock (default is **nil**)

time-passed ::= a number indicating the number of seconds in model time which passed during the run

event-count ::= a number indicating how many events were executed during this run

break? ::= [**t** | **nil**] indicating whether the run terminated due to a break event

Description:

Run will run the current meta-process until there are either no events remaining to execute, run-time seconds have passed, or a break event is executed. If the real-time? keyword parameter is provided with a non-**nil** value then the model will be run in real time mode. If real-time? is a positive number that will be used as a scale for the real time clock (a number greater than one would cause the meta-process to run faster than real time and a number less than one would cause it to run slower than real time).

If run-time is not a number greater than 0, or there is no current meta-process then a warning is printed and **nil** is returned.

If there is a meta-process to run, then when one of the end conditions has been met, run will output a line in the trace to indicate which condition terminated the run and it will return three values.

The first value is the number of seconds that passed for the model during this run. The second is a count of the number of events that were executed (which could be greater than the number of lines shown in the trace because some events may have no output), and the last indicates whether or not the run was terminated by a break event. If it was, then the third value will be **t** otherwise it will be **nil**.

This is the primary function used for running the system and the one you are most likely to encounter when looking at model files.

Examples:

For this example the count model from unit 1 of the ACT-R tutorial is the only model that is loaded.

```
> (run 10)
  0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000 PROCEDURAL CONFLICT-RESOLUTION
  0.000 PROCEDURAL PRODUCTION-SELECTED START
  ...
  0.300 PROCEDURAL CLEAR-BUFFER GOAL
  0.300 PROCEDURAL CONFLICT-RESOLUTION
  0.300 ----- Stopped because no events left to process
0.3
50
NIL
```

```

> (run .1 :real-time t)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK C
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL C
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 PROCEDURAL PRODUCTION-SELECTED INCREMENT
0.100 PROCEDURAL BUFFER-READ-ACTION GOAL
0.100 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.100 ----- Stopped because time limit reached

0.1
19
NIL

```

```

1> (schedule-break .075)
#S(ACT-R-BREAK-EVENT ...)

```

```

2> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.075 ----- BREAK-EVENT

0.075
14
T

```

```

E> (run 0)
#|Warning: run-time must be a number greater than zero.|#
NIL

```

```

E> (run 'foo)
#|Warning: run-time must be a number greater than zero.|#
NIL

```

```

E> (run 10)
#|Warning: run called with no current meta-process.|#
NIL

```

run-full-time

Syntax:

```
run-full-time run-time {:real-time real-time?} -> [ nil | time-passed event-count break? ]
```

Arguments and Values:

run-time ::= a number greater than 0 indicating the number of seconds to run

real-time? ::= a generalized boolean to indicate whether to run in real time and possibly the scale for the real time clock (default is **nil**)

time-passed ::= a number indicating the number of seconds in model time which passed during the run

event-count ::= a number indicating how many events were executed during this run

break? ::= [**t** | **nil**] indicating whether the run terminated due to a break event

Description:

Run-full-time will run the current meta-process until either run-time seconds have passed or a break event is executed. This differs from the run command because unless there is a break event run-full-time will always run for the full run-time specified. If the real-time? keyword parameter is provided with a non-**nil** value then the model will be run in real time mode. If real-time? is a positive number that will be used as a scale for the real time clock (a number greater than one would cause the meta-process to run faster than real time and a number less than one would cause it to run slower than real time).

If run-time is not a number greater than 0, or there is no current meta-process then a warning is printed and **nil** is returned.

If there is a meta-process to run, then when one of the end conditions has been met, run-full-time will output a line in the trace to indicate which condition terminated the run and it will return three values.

The first value is the number of seconds that passed for the model. The second is a count of the number of events that were executed (which could be greater than the number of lines shown in the trace because some events may have no output), and the last indicates whether or not the trial was terminated by a break event. If it was, then the third value will be **t** otherwise it will be **nil**.

This is also a commonly used function for running models and it is useful when fixed time sequences are desired.

Examples:

For this example the count model from unit 1 of the ACT-R tutorial is the only model that is loaded.

```
> (run-full-time 1.0)
  0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000 PROCEDURAL CONFLICT-RESOLUTION
  0.000 PROCEDURAL PRODUCTION-SELECTED START
  0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
  ...
  0.300 PROCEDURAL CONFLICT-RESOLUTION
  1.000 ----- Stopped because time limit reached
1.0
51
```

NIL

```
1> (schedule-break .55)
#S(ACT-R-BREAK-EVENT ...)
```

```
2> (run-full-time 2.0 :real-time t)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
...
0.300 PROCEDURAL CONFLICT-RESOLUTION
0.550 ----- BREAK-EVENT
0.55
51
T
```

```
E> (run-full-time -1)
#|Warning: run-time must be a number greater than zero. |#
NIL
```

```
E> (run-full-time "2.0")
#|Warning: run-time must be a number greater than zero. |#
NIL
```

```
E> (run-full-time 1.0)
#|Warning: run-full-time called with no current meta-process. |#
NIL
```

run-until-time

Syntax:

run-until-time *end-time* {:**real-time** *real-time?*} -> [**nil** | time-passed event-count break?]

Arguments and Values:

end-time ::= a number greater than 0 indicating the explicit time at which the run should stop

real-time? ::= a generalized boolean to indicate whether to run in real time and possibly the scale for the real time clock (default is **nil**)

time-passed ::= a number indicating the number of seconds in model time which passed during the run

event-count ::= a number indicating how many events were executed during this run

break? ::= [**t** | **nil**] indicating whether the run terminated due to a break event

Description:

Run-until-time will run the current meta-process until either the specified end-time is reached (including if the current time is greater than the specified time) or a break event is executed. This differs from the run and run-full-time commands because an explicit time is specified instead of a duration. If the real-time? keyword parameter is provided with a non-**nil** value then the model will be run in real time. If real-time? is a positive number that will be used as a scale for the real time clock (a number greater than one would cause the meta-process to run faster than real time and a number less than one would cause it to run slower than real time).

If end-time is not a number greater than 0, or there is no current meta-process then a warning is printed and **nil** is returned.

If there is a meta-process to run, then when one of the end conditions has been met, run-until-time will output a line in the trace to indicate which condition terminated the run and it will return three values.

The first value is the number of seconds that passed for the model. The second is a count of the number of events that were executed (which could be greater than the number of lines shown in the trace because some events may have no output), and the last indicates whether or not the trial was terminated by a break event. If it was, then the third value will be **t** otherwise it will be **nil**.

Examples:

For this example the count model from unit 1 of the ACT-R tutorial is the only model that is loaded.

```
1> (run-until-time .125)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK C
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL C
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 PROCEDURAL PRODUCTION-SELECTED INCREMENT
0.100 PROCEDURAL BUFFER-READ-ACTION GOAL
0.100 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.125 ----- Stopped because time limit reached
0.125
21
NIL

2> (run-until-time .100)
0.125 ----- Stopped because end time already passed
0
0
NIL

3> (run-until-time 10.0)
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT
2
0.150 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.150 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.150 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.150 DECLARATIVE START-RETRIEVAL
...
0.300 PROCEDURAL CONFLICT-RESOLUTION
10.000 ----- Stopped because time limit reached
9.875
```

```
31
NIL
```

```
1> (schedule-break .4)
#S(ACT-R-BREAK-EVENT ...)
```

```
2> (run-until-time .5)
  0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000 PROCEDURAL CONFLICT-RESOLUTION
  0.000 PROCEDURAL PRODUCTION-SELECTED START
  0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
  0.050 PROCEDURAL PRODUCTION-FIRED START
  0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
  ...
  0.300 PROCEDURAL CONFLICT-RESOLUTION
  0.400 ----- BREAK-EVENT
```

```
0.4
51
T
```

```
E> (run-until-time -10)
#|Warning: end-time must be a number greater than zero. |#
NIL
```

```
E> (run-until-time 1)
#|Warning: run-until-time called with no current meta-process. |#
NIL
```

run-until-condition

Syntax:

```
run-until-condition condition {:real-time real-time?} -> [ nil | time-passed event-count break? ]
```

Arguments and Values:

condition ::= a function or the name of a function that takes no parameters

real-time? ::= a generalized boolean to indicate whether to run in real time and possibly the scale for the real time clock (default is **nil**)

time-passed ::= a number indicating the number of seconds in model time which passed during the run

event-count ::= a number indicating how many events were executed during this run

break? ::= [**t** | **nil**] indicating whether the run terminated due to a break event

Description:

Run-until-condition will run the current meta-process until either the provided condition function returns non-**nil**, there are no events left to process, or a break event is executed. The condition function will be called before every event is executed and as soon as it returns non-**nil** the run is terminated. If the *real-time?* keyword parameter is provided with a non-**nil** value then the model will be run in real time mode. If *real-time?* is a positive number that will be used as a scale for the real time clock (a number greater than one would cause the meta-process to run faster than real time and a number less than one would cause it to run slower than real time).

If condition is not a function or the name of a function, or there is no current meta-process then a warning is printed and **nil** is returned.

If there is a meta-process to run, then when one of the end conditions has been met, run-until-condition will output a line in the trace to indicate which condition terminated the run and it will return three values.

The first value is the number of seconds that passed for the model. The second is a count of the number of events that were executed (which could be greater than the number of lines shown in the trace because some events may have no output), and the last indicates whether or not the trial was terminated by a break event. If it was, then the third value will be **t** otherwise it will be **nil**.

Examples:

For this example the count model from unit 1 of the ACT-R tutorial is the only model that is loaded.

```
> (run-until-condition (lambda () nil))
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
...
0.300 PROCEDURAL CONFLICT-RESOLUTION
0.300 ----- Stopped because no events to process
0.3
50
NIL

1> (schedule-break .275)
#S(ACT-R-BREAK-EVENT ...)

2> (run-until-condition (lambda () nil))
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
...
0.250 PROCEDURAL BUFFER-READ-ACTION GOAL
0.275 ----- BREAK-EVENT
0.275
45
T

1> (defvar *count* 0)
*COUNT*

2> (defun stop-at-10 ()
  (> (incf *count*) 10))
STOP-AT-10

3> (run-until-condition 'stop-at-10)
```

```

0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 ----- Stopped because condition is true
0.05
10
NIL

E> (run-until-condition "not-a-function")
#|Warning: condition must be a function.|#
NIL

E> (run-until-condition (lambda () nil))
#|Warning: run-until-condition called with no current meta-process.|#
NIL

```

run-n-events

Syntax:

```
run-n-events num-events {:real-time real-time?} -> [ nil | time-passed event-count break? ]
```

Arguments and Values:

num-events ::= a number greater than 0 indicating the number of events to run

real-time? ::= a generalized boolean to indicate whether to run in real time and possibly the scale for the real time clock (default is **nil**)

time-passed ::= a number indicating the number of seconds in model time which passed during the run

event-count ::= a number indicating how many events were executed during this run

break? ::= [**t** | **nil**] indicating whether the run terminated due to a break event

Description:

Run-n-events will run the current meta-process until either *num-events* events have been processed, there are no events remaining, or a break event is executed. If the *real-time?* keyword parameter is provided with a non-**nil** value then the model will be run in real time mode. If *real-time?* is a positive number that will be used as a scale for the real time clock (a number greater than one would cause the meta-process to run faster than real time and a number less than one would cause it to run slower than real time).

If *num-events* is not a number greater than 0, or there is no current meta-process then a warning is printed and **nil** is returned.

If there is a meta-process to run, then when one of the end conditions has been met, run-n-events will output a line in the trace to indicate which condition terminated the run and it will return three values.

The first value is the number of seconds that passed for the model. The second is a count of the number of events that were executed (which could be greater than the number of lines shown in the trace because some events may have no output), and the last indicates whether or not the trial was terminated by a break event. If it was, then the third value will be **t** otherwise it will be **nil**.

Examples:

For this example the count model from unit 1 of the ACT-R tutorial is the only model that is loaded.

```
> (run-n-events 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 ----- Stopped because event limit reached
0.05
10
NIL
```

```
> (run-n-events 100)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
...
0.300 PROCEDURAL CONFLICT-RESOLUTION
0.300 ----- Stopped because no events to process
0.3
50
NIL
```

```
1> (schedule-break .225)
#S(ACT-R-BREAK-EVENT ...)
```

```
2> (run-n-events 50)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
...
0.200 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.225 ----- BREAK-EVENT
0.225
35
T
```

```
E> (run-n-events 'a)
#|warning: event-count must be a number greater than zero.|#
NIL
```

```
E> (run-n-events 10)
#|warning: run-n-events called with no current meta-process.|#
NIL
```

run-step

Syntax:

run-step -> [**nil** | time-passed event-count break?]

Arguments and Values:

time-passed ::= a number indicating the number of seconds in model time which passed during the run

event-count ::= a number indicating how many events were executed during this run

break? ::= [**t** | **nil**] indicating whether the run terminated due to a break event

Description:

Run-step will run the current meta-process one event at a time. For each event a summary of the event is printed to **standard-output** and the user is prompted to respond as to whether that event should be executed, deleted, or the run terminated. This will stop for all events, even those which do not get displayed in the trace. The user can also show various debugging information before deciding what to do with the current event. The response is read from **standard-input** and should be one of the characters indicated in the prompt. It will continue to run the model until the user requests it to stop, there are no events remaining, or a break event is executed. Using run-step cannot run the model in real time mode.

If there is no current meta-process then a warning is printed and **nil** is returned.

If there is a meta-process to run, then when one of the end conditions has been met, run-step will output a line in the trace to indicate which condition terminated the run and it will return three values.

The first value is the number of seconds that passed for the model. The second is a count of the number of events that were executed (which could be greater than the number of lines shown in the trace because some events may have no output), and the last indicates whether or not the trial was terminated by a break event. If it was, then the third value will be **t** otherwise it will be **nil**.

Run-step can be a useful function for debugging a model because it allows one to walk through the events one at a time and stop to inspect the state of the system before or after any one.

Examples:

For this example the count model from unit 1 of the ACT-R tutorial is the only model that is loaded.

```

> (run-step)
Next Event:      0.000  NONE  CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS :ESC T :OL T :ER NIL)
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
s
Events in the queue:
  0.000  NONE  CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS :ESC T :OL T :ER NIL)
  0.000  GOAL  SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000  PROCEDURAL  CONFLICT-RESOLUTION
  0.000  GOAL  CLEAR-DELAYED-GOAL
Next Event:      0.000  NONE  CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS :ESC T :OL T :ER NIL)
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
e
Next Event:      0.000  GOAL  SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
e
  0.000  GOAL  SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
Next Event:      0.000  PROCEDURAL  CONFLICT-RESOLUTION
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
e
  0.000  PROCEDURAL  CONFLICT-RESOLUTION
Next Event:      0.000  PROCEDURAL  PRODUCTION-SELECTED START
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
b
RETRIEVAL: empty
IMAGINAL: empty
MANUAL: empty
GOAL:
FIRST-GOAL-0
  START 2
  END 4

IMAGINAL-ACTION: empty

```

```

VOCAL: empty
AURAL: empty
PRODUCTION: empty
VISUAL-LOCATION: empty
AURAL-LOCATION: empty
TEMPORAL: empty
VISUAL: empty
Next Event:      0.000  PROCEDURAL          PRODUCTION-SELECTED START
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
e
      0.000  PROCEDURAL          PRODUCTION-SELECTED START
Next Event:      0.000  PROCEDURAL          START
[A]bort (or [q]uit)
[D]elete
[S]how event queue
[W]aiting events
[B]uffer contents
[V]isicon
[R]eport buffer status
[E]xecute
q
      0.000  -----          Stepping stopped
0.0
4
NIL

E> (run-step)
#|Warning: run-step called with no current meta-process.|#
NIL

```

Scheduling Events

The event system that drives ACT-R models is also available to the modeler for use in writing experiments or other interactive actions for the models. In fact, because ACT-R relies on the events to trigger actions such as conflict-resolution, this is the preferred mechanism for creating experiments or making other run-time changes. It is also essential when adding new modules because for proper integration with other components of the system any changes which a module has on buffers as well as when changing its internal state or affecting outside actions should occur through scheduled events.

When writing experiments for a model, one useful approach is to have the model's actions trigger the events that make changes in such a way that one only needs to call one of the ACT-R "run" functions to execute both the model and the task. That has the benefit of not introducing any discrepancies into the model timing relative to the task and also allows for the task to be run using the provided stepping tools or continued after a break in the model. That is not always practical for a simple model/task and often one may want to use a "run, stop, change, run again" style (also referred to as run-stop style). Even when using the run-stop style, it is still important to schedule any direct effects that one makes to buffers or chunks so that the model properly notes the changes.

In general, most of the module commands will schedule some event as a response, but many of the general commands which perform similar actions may not. For example, [mod-chunk](#) (a general command) does not generate an event, but [mod-focus](#) (a similar command specific to the goal module) does.

Details of events

Each event has several attributes associated with it that are specified when the event is created with one of the scheduling functions provided. Most of the time the user will not need to work with the events directly, but there are some situations where access to the details of an event may be useful (for instance the event hooks allow one to add functions which see each event before or after it is executed). Here are the attributes which an event has.

time

The simulation time at which the event will occur. All times are rounded to the millisecond when the event is created.

priority

When multiple events are scheduled to occur at the same time they are ordered by their priorities. The priority is either a number or one of the keywords :max or :min. An event with a priority of :max will occur before any event at the same time which has a priority other than :max. An event with a priority of :min will occur after any event at the same time which has a priority other than :min. An event with a numeric priority will be executed before any other events at the same time which have a lower numeric priority i.e. numeric priorities are ordered from highest to lowest with no bounds on the numbers given. Events which have both the same time and same priority will occur in the order which they were scheduled -- the earlier scheduled item will occur before the later scheduled one.

action

The function that will be called when this event is executed.

parameters

The list of values which will be passed to the action function when the event is executed.

model

The name of the model in which the event was generated.

module

The name of the module which generated the event or the keyword **:none** if no module was specified when the event was created.

destination

If this action is to be sent *to* a specific module, then that module's name can be given as the destination and the instance of that module will be passed as the first parameter to the action. Using this is can be simpler and more informative than just making the instance of the module the first element in the parameters list.

details

The details can be a string which will be output in the trace for the event. If details are specified that is all that is printed after the time, model and module. If the details are not specified then the action and parameters are printed in the trace.

output

Output controls under which trace-detail levels the event will be displayed. It can have a value of **t**, **high**, **medium**, **low**, or **nil**. A value of **t** or **high** means it will be displayed only for the high trace detail setting. **Nil** means not to show it at all. **Medium** means that it should be shown under both medium and high trace details, and a value of **low** means it will be shown for any trace detail setting. The output value effectively specifies the lowest detail setting for which the output will be displayed.

Event Accessors

The specific implementation of an event is not part of the API for ACT-R. One should not assume anything about the structure or object that is returned as an event nor attempt to make any modifications to it. To be able to read the information from an event a set of accessors are provided. Because all of the accessors operate the same way they are all presented in one description. Note that the accessors are not intended to be used to change a value of an event only to read the value that it has - even though the specific implementation may allow one to use them with `setf` to change a value that is not recommended and could result in unexpected consequences within the model.

Syntax:

```
evt-time event -> time  
evt-priority event -> priority  
evt-action event -> action  
evt-params event -> parameters  
evt-model event -> model  
evt-module event -> module  
evt-destination event -> destination  
evt-details event -> details  
evt-output event -> output
```

Arguments and Values:

`event` ::= an ACT-R event
time, priority, action, parameters, model, module, destination, details, output ::= the corresponding attribute of event

Description:

Each of the event accessors returns the corresponding attribute from the event provided. There is no error checking to make sure that event is a valid ACT-R event. If it is not a valid event, then a Lisp error is likely to occur with the current implementation of events.

Examples:

```
> (let ((event (schedule-event 30 'model-output :params '("some text"))))
      (format t "~S-%" (evt-time event))
      (format t "~S-%" (evt-action event))
      (format t "~S-%" (evt-module event))
      (format t "~S-%" (evt-params event)))
30.0
MODEL-OUTPUT
:NONE
("some text")
NIL

E> (evt-time 'not-an-event)
Error: Non-structure argument NOT-AN-EVENT passed to ref of structure slot 1
[condition type: SIMPLE-ERROR]
```

General Event Commands

These commands allow for getting additional information about events related to whether they will be displayed in the trace and how their output will look.

event-displayed-p

Syntax:

```
event-displayed-p event -> [t | nil]
```

Arguments and Values:

event ::= an ACT-R event

Description:

Event-displayed-p can be used to determine whether or not an event will be printed in the trace given the current setting of the [:trace-detail](#) and [:trace-filter](#) parameters for the model in which it was generated. If the event will be printed with the current settings of that model's parameters, then **t** is returned and if not then **nil** is returned. If event is not an ACT-R event then **nil** is returned.

This command might be useful when working with the event hooks or for developing an interactive stepper or tracing tool.

Examples:

```
> (let ((event (schedule-event 30 'model-output :params '("some text") :output 'medium)))
      (no-output (sgp :trace-detail high))
      (format t "trace-detail high output medium: ~S~%" (event-displayed-p event))
      (no-output (sgp :trace-detail medium))
      (format t "trace-detail medium output medium: ~S~%" (event-displayed-p event))
      (no-output (sgp :trace-detail low))
      (format t "trace-detail low output medium: ~S~%" (event-displayed-p event)))
```

```
trace-detail high output medium: T
trace-detail medium output medium: T
trace-detail low output medium: NIL
NIL
```

```
E> (event-displayed-p 'not-an-event)
NIL
```

format-event

Syntax:

format-event *event* -> [event-string | **nil**]

Arguments and Values:

event ::= an ACT-R event

event-string ::= a string that contains the text that would be printed for this event in the trace

Description:

Format-event can be used to get a string with the representation of what the provided event will look like in the trace when it is executed. If *event* is not a valid ACT-R event then **nil** is returned.

This would likely be used with the development of additional stepping tools or a data logger which was tied into the event hooks to be able to record and/or display an event independently of the trace.

Examples:

```
> (let ((event (schedule-event 30 'model-output :params '("some text"))))
      (format-event event))
" 30.000 NONE MODEL-OUTPUT some text "
```

```
E> (format-event 'not-an-event)
NIL
```

Scheduling Commands

Events can be generated using a variety of scheduling functions described here, as well as automatically by certain module commands. There are three different types of events that can be generated: normal (or model) events, maintenance events, and break events. Model events are things

that are generated by the cognitive modules or outside actions which the model may need to detect (in particular, the conflict resolution mechanism of the procedural module is sensitive only to model events). Maintenance events are generated by the non-theory parts of the system or things which are not of importance to the model (for instance an event which signals the time at what a particular run is going to stop). Practically, the only difference between model and maintenance events is in whether events that are waiting to be scheduled consider any event or only model events (they are generated with the same functions). Break events are a special type of maintenance event in that whenever a break event is executed the current run is terminated. Break events are generated with a separate set of functions because they do not perform any actions.

When an event's action is executed the current model will be set to the model which generated the event (if there was one). When working with a single model that does not make a difference, but in the context of multiple models it means that the action function does not need to use [with-model](#) or make any explicit checks to ensure that it is working in the proper context.

schedule-event

Syntax:

schedule-event *time action {event-descriptors}* -> [*actr-event* | **nil**]

Arguments and Values:

time ::= a number representing an absolute time for the event in seconds or milliseconds

action ::= a function name or function which specifies the action to be evaluated

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:module** *module-value* |
:destination *destination-value* |
:priority *priority-value* |
:params *params-value* |
:time-in-ms *time-units* |
:maintenance *maintenance-value* |
:details *details-value* |
:output *output-value*]

module-value ::= a symbol which names the module which is scheduling the event (default **:none**)

destination-value ::= a symbol which names a module (default **nil**)

priority-value ::= [**:max** | **:min** | a number] (default 0)

params-value ::= a list of values to pass to the action (default **nil**)

time-units ::= a generalized boolean indicating whether the time is set in milliseconds
(default is **nil**)

maintenance-value ::= generalized boolean indicating whether this is a maintenance event (default **nil**)

details-value ::= a string to output in the trace or **nil** (default **nil**)

output-value ::= [**t** | **high** | **medium** | **low** | **nil**] (default **t**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-event creates a new event using the supplied parameters for its corresponding attributes and the current model for its model. It will be added to the event queue of the current meta-process to occur at the specific time provided and the event will be returned.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

Examples:

```
>(schedule-event 10 'goal-focus-fct :priority :min :params '(new-goal-chunk) :output nil)
#S(ACT-R-EVENT ...)
```

```
> (schedule-event 45 (lambda (module) (do-something module)) :maintenance t
:destination :some-module :details "Do Something")
#S(ACT-R-MAINTENANCE-EVENT ...)
```

```
> (mp-show-queue)
Events in the queue:
    10.000  NONE                                GOAL-FOCUS NEW-GOAL-CHUNK
    45.000  NONE                                Do Something
```

```
E> (schedule-event 'bad-time (lambda ()))
#|Warning: Time must be non-negative number. |#
NIL
```

```
E> (schedule-event 0 'bad-function-name)
#|Warning: Can't schedule BAD-FUNCTION-NAME not a function or function name. |#
NIL
```

```
E> (schedule-event 10 'goal-focus :priority :min :params '(new-goal-chunk) :output nil)
#|Warning: Can't schedule GOAL-FOCUS because it is a macro and not a function. |#
NIL
```

```
E> (schedule-event 0 (lambda ()) :priority 'value)
#|Warning: Priority must be a number or :min or :max. |#
NIL
```

```
E> (schedule-event 0 (lambda (x)) :params 10)
#|Warning: params must be a list. |#
NIL
```

```
E> (schedule-event 0 'pprint)
#|Warning: schedule-event called with no current model. |#
NIL
```

```
E> (schedule-event 30 (lambda ()))
#|Warning: schedule-event called with no current meta-process. |#
NIL
```

schedule-event-relative

Syntax:

schedule-event-relative *delta-time action {event-descriptors}* -> [*actr-event* | **nil**]

Arguments and Values:

delta-time ::= a number in seconds or milliseconds indicating the delay before executing the event

action ::= a function name or function which specifies the action to be evaluated

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:module** *module-value* |
 :destination *destination-value* |
 :priority *priority-value* |
 :params *params-value* |
 :time-in-ms *time-units* |
 :maintenance *maintenance-value* |
 :details *details-value* |
 :output *output-value*]

module-value ::= a symbol which names the module which is scheduling the event (default **:none**)

destination-value ::= a symbol which names a module (default **nil**)

priority-value ::= [**:max** | **:min** | a number] (default 0)

params-value ::= a list of values to pass to the action (default **nil**)

time-units ::= a generalized boolean indicating whether the time is set in milliseconds
(default is **nil**)

maintenance-value ::= generalized boolean indicating whether this is a maintenance event (default **nil**)

details-value ::= a string to output in the trace or **nil** (default **nil**)

output-value ::= [**t** | **high** | **medium** | **low** | **nil**] (default **t**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-event-relative creates a new event using the supplied parameters for its corresponding attributes and the current model for its model. It will then be added to the event queue of the current meta-process to occur *delta-time* seconds (or milliseconds if *:time-in-ms* is specified as true) from the current time and the event will be returned.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

Examples:

see [schedule-event](#) for related examples

schedule-event-now

Syntax:

schedule-event-relative *action* {*event-descriptors*} -> [*actr-event* | **nil**]

Arguments and Values:

action ::= a function name or function which specifies the action to be evaluated

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:module** *module-value* |
 :destination *destination-value* |
 :priority *priority-value* |
 :params *params-value* |
 :maintenance *maintenance-value* |
 :details *details-value* |
 :output *output-value*]

module-value ::= a symbol which names the module which is scheduling the event (default **:none**)

destination-value ::= a symbol which names a module (default **nil**)

priority-value ::= [**:max** | **:min** | a number] (default 0)

params-value ::= a list of values to pass to the action (default **nil**)

maintenance-value ::= generalized boolean indicating whether this is a maintenance event (default **nil**)

details-value ::= a string to output in the trace or **nil** (default **nil**)

output-value ::= [**t** | **high** | **medium** | **low** | **nil**] (default **t**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-event-now creates a new event using the supplied parameters for its corresponding attributes and the current model for its model. It will then be added to the event queue of the current meta-process to occur at the current time and the event will be returned.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

Examples:

see [schedule-event](#) for related examples

schedule-event-after-module

Syntax:

schedule-event-after-module *after-module action {event-descriptors}* -> [*actr-event* | **nil**] [**t** | **nil** | **:abort**]

Arguments and Values:

after-module ::= a symbol which names a module

action ::= a function name or function which specifies the action to be evaluated

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:module** *module-value* |
:destination *destination-value* |
:params *params-value* |
:maintenance *maintenance-value* |
:details *details-value* |
:output *output-value* |
:delay *delay-value* |
:include-maintenance *include-maintenance-value* |
:dynamic *dynamic-value*]

module-value ::= a symbol which names the module which is scheduling the event (default **:none**)

destination-value ::= a symbol which names a module (default **nil**)

params-value ::= a list of values to pass to the action (default **nil**)

maintenance-value ::= generalized boolean indicating whether this is a maintenance event (default **nil**)

details-value ::= a string to output in the trace or **nil** (default **nil**)

output-value ::= [**t** | **high** | **medium** | **low** | **nil**] (default **t**)

delay-value ::= [**t** | **nil** | **:abort**] (default **t**)

include-maintenance-value ::= a generalized boolean indicating whether to consider maintenance events when determining when to schedule this event (default **nil**)

dynamic-value ::= generalized boolean indicating whether to allow rescheduling under real time mode (default **nil**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-event-after-module creates a new event using the supplied parameters for its corresponding attributes and the current model for its model.

If there is an event currently in the event queue with the module name of *after-module* and the same model as the current model and either *include-maintenance-value* is **t** or the event is not a maintenance event, then this new event is placed into the event queue at the time of the next such matching event (lowest time) with a priority of **:min**. If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If there is no event in the event queue that matches on model, module and of the appropriate event type (model or maintenance), then the value of delay-value determines what happens to the new event.

If delay-value is **t** then the new event is placed into the set of waiting events to be scheduled after an event which matches the conditions necessary to schedule this new event.

If delay-value is **nil** then the new event is added to the event queue for immediate execution. Its time will be set to the current time and its priority will be **:max**.

If delay-value is **:abort** then the new event is discarded without being scheduled or placed onto the waiting queue.

The setting of dynamic-value does not matter under normal circumstances, but may be useful if a custom clock is provided for real time operations. See the [Configuring Real Time Operations](#) section for details of how it works.

Schedule-event-after-module returns two values. If there is no current model or current meta-process or any of the parameters are invalid, then no event is scheduled and both values are **nil**.

If an event is scheduled then the first value will be the event and the second value will be **t** if the event is in the waiting queue or **nil** if it is in the event queue.

If the event is aborted, the first value will be **nil** and the second value will be **:abort**.

Examples:

```
1> (mp-show-queue)
Events in the queue:
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
1

2> (schedule-event-after-module 'procedural (lambda ()) :details "has a matching event")
#S(ACT-R-EVENT ...)
NIL

3> (mp-show-queue)
Events in the queue:
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.000  NONE                has a matching event
2

4> (schedule-event-after-module :vision (lambda ()) :details "no event so wait" :delay t)
#S(ACT-R-EVENT ...)
T

5> (mp-show-queue)
Events in the queue:
```

```

0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 NONE has a matching event
2

6> (mp-show-waiting)
Events waiting to be scheduled:
NIL NONE no event so wait Waiting for: (MODULE VISION NIL)
1

7> (schedule-event-after-module :motor (lambda ()) :details "no event so go now" :delay
nil)
#S(ACT-R-EVENT ...)
NIL
> (mp-show-queue)
Events in the queue:
0.000 NONE no event so go now
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 NONE has a matching event
3

8> (schedule-event-after-module :audio (lambda ()) :details "aborted" :delay :abort)
NIL
:ABORT

E> (schedule-event-after-module 'bad-module-name (lambda ()))
#|Warning: after-module must name a module.|#
NIL
NIL

```

schedule-event-after-change

Syntax:

schedule-event-after-change *action* {*event-descriptors*} -> [*actr-event* | **nil**] [**t** | **nil** | **:abort**]

Arguments and Values:

action ::= a function name or function which specifies the action to be evaluated

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:module** *module-value* |
:destination *destination-value* |
:params *params-value* |
:maintenance *maintenance-value* |
:details *details-value* |
:output *output-value* |
:delay *delay-value* |
:include-maintenance *include-maintenance-value* |
:dynamic *dynamic-value*]

module-value ::= a symbol which names the module which is scheduling the event (default **:none**)

destination-value ::= a symbol which names a module (default **nil**)

params-value ::= a list of values to pass to the action (default **nil**)

maintenance-value ::= generalized boolean indicating whether this is a maintenance event (default **nil**)

details-value ::= a string to output in the trace or **nil** (default **nil**)

output-value ::= [**t** | **high** | **medium** | **low** | **nil**] (default **t**)

delay-value ::= [**t** | **nil** | **:abort**] (default **t**)

include-maintenance-value ::= a generalized boolean indicating whether to consider maintenance events when determining when to schedule this event (default **nil**)
dynamic-value ::= generalized boolean indicating whether to allow rescheduling under real time mode (default **nil**)
actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-event-after-change creates a new event using the supplied parameters for its corresponding attributes and the current model for its model.

If there is any event currently in the event queue with the same model as the current model and either include-maintenance-value is **t** or the event is not a maintenance event, then this new event is placed into the event queue at the time of the next such matching event (lowest time) with a priority of **:min**. If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If there is no event in the event queue that matches on model and is of the appropriate event type (model or maintenance), then the value of delay-value determines what happens to the new event.

If delay-value is **t** then the new event is placed into the set of waiting events to be scheduled after an event which matches the conditions necessary to schedule this new event.

If delay-value is **nil** then the new event is added to the event queue for immediate execution. Its time will be set to the current time and its priority will be **:max**.

If delay-value is **:abort** then the new event is discarded without being scheduled or placed onto the waiting queue.

The setting of dynamic-value does not matter under normal circumstances, but may be useful if a custom clock is provided for real time operations. See the [Configuring Real Time Operations](#) section for details of how it works.

Schedule-event-after-change returns two values. If there is no current model or current meta-process or any of the parameters are invalid, then no event is scheduled and both values are **nil**. If an event is scheduled then the first value will be the event and the second value will be **t** if the event is in the waiting queue or **nil** if it is in the event queue. If the event is aborted, the first value will be **nil** and the second value will be **:abort**.

Examples:

see [schedule-even-after-module](#) for related examples

schedule-periodic-event

Syntax:

```
schedule-periodic-event period action {event-descriptors} -> [ actr-event | nil ]
```

Arguments and Values:

period ::= a number indicating how many seconds or milliseconds after which this action should be evaluated again

action ::= a function name or function which specifies the action to be evaluated

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:module** *module-value* |
:destination *destination-value* |
:priority *priority-value* |
:params *params-value* |
:time-in-ms *time-units* |
:maintenance *maintenance-value* |
:details *details-value* |
:output *output-value* |
:initial-delay *initial-delay-value*]

module-value ::= a symbol which names the module which is scheduling the event (default **:none**)

destination-value ::= a symbol which names a module (default **nil**)

priority-value ::= [**:max** | **:min** | a number] (default 0)

params-value ::= a list of values to pass to the action (default **nil**)

time-units ::= a generalized boolean indicating whether the time is set in milliseconds
(default is **nil**)

maintenance-value ::= generalized boolean indicating whether this is a maintenance event (default **nil**)

details-value ::= a string to output in the trace or **nil** (default **nil**)

output-value ::= [**t** | **high** | **medium** | **low** | **nil**] (default **t**)

initial-delay-value ::= a number indicating how many seconds or milliseconds before the first such event (default 0)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-periodic-event creates a new event with a time that is equal to the current time plus initial-delay and using the other supplied parameters for its corresponding attributes and the current model for its model which is then added to the event queue of the current meta-process. After that event occurs a new event will automatically be scheduled to occur *period* seconds (or milliseconds if *:time-in-ms* is specified as true) after that time with the same parameters as the initial one. That

rescheduling will continue every period seconds (or milliseconds) until the event this function returned is deleted.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled, and every time that it is rescheduled there will be a check of the waiting events.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

The scheduled event is returned when successfully created and scheduled.

Note that there are actually two events generated for each occurrence of the event described. The first is a maintenance event with the priority provided. It schedules the actual event described with the parameters specified with a priority of **:max** (so that it should be the next event to execute) and also schedules the next periodic event at the appropriate delay.

Examples:

```
1> (schedule-periodic-event 1 (lambda ()
                               (model-output "Periodic event")))
                               :initial-delay .5)
#S(ACT-R-PERIODIC-EVENT :TIME 0.5 ...)
```

```
2> (mp-show-queue)
Events in the queue:
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.500  NONE                Periodic-Action Unnamed function 1
2
```

```
3> (run 3)
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
Periodic event
  0.500  PROCEDURAL          CONFLICT-RESOLUTION
Periodic event
  1.500  PROCEDURAL          CONFLICT-RESOLUTION
Periodic event
  2.500  PROCEDURAL          CONFLICT-RESOLUTION
  3.000  -----            Stopped because time limit reached
3
10
NIL
```

```
E> (schedule-periodic-event 'a (lambda ()))
#|Warning: period must be greater than 0.|#
NIL
```

schedule-break

Syntax:

schedule-break *time* {*event-descriptors*} -> [*actr-event* | **nil**]

Arguments and Values:

time ::= a number representing an absolute time for the event in seconds or milliseconds
event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair
event-key-value-pair ::= [**:details** *details-value* | **:priority** *priority-value* | **:time-in-ms** *time-units*]
details-value ::= a string to output in the trace or **nil** (defaults to **nil**)
priority-value ::= [**:max** | **:min** | a number] (defaults to **:max**)
time-units ::= a generalized boolean indicating whether the time is set in milliseconds
(default is **nil**)
actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-break creates a new break event at the specified time with the priority-value and details-value provided. The model of the event will be **nil** (a break event does not need to exist within a specific model), the module is set to **:none**, and the output for that event is set to low. A break event does not have an action and is only used to stop the scheduler. That new event is then added to the event queue of the current meta-process.

If any of the parameters are invalid or there is no current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

The scheduled event is returned when successfully created and scheduled.

Examples:

```
1> (schedule-break 12.5 :details "Stop the system now")
#S(ACT-R-BREAK-EVENT ...)

2> (mp-show-queue)
Events in the queue:
   0.000  PROCEDURAL          CONFLICT-RESOLUTION
  12.500  -----          BREAK-EVENT Stop the system now
2

E> (schedule-break 'bad-time)
#|Warning: Time must be non-negative number.|#
NIL

E> (schedule-break 10 :priority 'bad-priority)
#|Warning: Priority must be a number or :min or :max.|#
NIL

E> (schedule-break 10)
#|Warning: schedule-break called with no current meta-process.|#
NIL
```

schedule-break-relative

Syntax:

schedule-break-relative *delta-time* {*event-descriptors*} -> [*actr-event* | **nil**]

Arguments and Values:

delta-time ::= a number representing a delay in seconds or milliseconds before executing the event

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:details** *details-value* | **:priority** *priority-value* | **:time-in-ms** *time-units*]

details-value ::= a string to output in the trace or **nil** (defaults to **nil**)

priority-value ::= [**:max** | **:min** | a number] (defaults to **:max**)

time-units ::= a generalized boolean indicating whether the time is set in milliseconds
(default is **nil**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-break-relative creates a new break event *delta-time* seconds or milliseconds from the current time with the *priority-value* and *details-value* provided. The model of the event will be **nil** (a break event does not need to exist within a specific model), the module is set to **:none**, and the output for that event is set to low. A break event does not have an action and is only used to stop the scheduler. That new event is then added to the event queue of the current meta-process.

If any of the parameters are invalid or there is no current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

The scheduled event is returned when successfully created and scheduled.

Examples:

see [schedule-break](#) for related examples

schedule-break-after-module

Syntax:

schedule-break-after-module *after-module* {*event-descriptors*} -> [*actr-event* | **nil**]

Arguments and Values:

after-module ::= a symbol which names a module

event-descriptors ::= 0 or 1 instance of each possible event-key-value-pair

event-key-value-pair ::= [**:details** *details-value* | **:delay** *delay-value* | **:dynamic** *dynamic-value*]

details-value ::= a string to output in the trace or **nil** (defaults to **nil**)

delay-value ::= [**t** | **nil** | **:abort**] (defaults to **t**)

dynamic-value ::= generalized boolean indicating whether to allow rescheduling under real time mode
(defaults to **nil**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-break-after-module creates a new event using the supplied parameters for its corresponding attributes and the current model as its model. That event will be scheduled to occur after the next event of the specified module in the same model as the current one.

If there is an event currently in the event queue with the module name of after-module for the current model then this new break event is placed into the event queue at the time of the next such matching event (lowest time) with a priority of **:min**. If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If there is no event in the event queue that matches the model and module, then the value of delay-value determines what happens to the new break event.

If delay-value is **t** then the new break event is placed into the set of waiting events to be scheduled after an event which matches the conditions necessary to schedule this new event.

If delay-value is **nil** then the new break event is added to the event queue for immediate execution. Its time will be set to the current time and its priority will be **:max**.

If delay-value is **:abort** then the new break event is discarded without being scheduled or placed onto the waiting queue.

The setting of dynamic-value does not matter under normal circumstances, but may be useful if a custom clock is provided for real time operations. See the [Configuring Real Time Operations](#) section for details of how it works.

Schedule-break-after-module returns two values. If there is no current meta-process, no current model, or any of the parameters are invalid, then no event is scheduled and both values are **nil**. If an event is scheduled then the first value will be the event and the second value will be **t** if the event is in the waiting queue or **nil** if it is in the event queue. If the event is aborted, the first value will be **nil** and the second value will be **:abort**.

Examples:

```

1> (mp-show-queue)
Events in the queue:
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
1

2> (schedule-break-after-module 'procedural :details "after procedural")
#S(ACT-R-BREAK-EVENT ...)
NIL

3> (mp-show-queue)
Events in the queue:
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.000  -----            BREAK-EVENT after procedural
2

4> (schedule-break-after-module :vision :details "waiting for vision")
#S(ACT-R-BREAK-EVENT ...)
T

5> (mp-show-queue)
Events in the queue:
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.000  -----            BREAK-EVENT after procedural
2

6> (mp-show-waiting)
Events waiting to be scheduled:
  NIL     -----            BREAK-EVENT waiting for vision Waiting for: (MODULE VISION T)
1

7> (schedule-break-after-module :vision :details "not waiting for vision" :delay nil)
#S(ACT-R-BREAK-EVENT ...)
NIL

8> (mp-show-queue)
Events in the queue:
  0.000  -----            BREAK-EVENT not waiting for vision
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.000  -----            BREAK-EVENT after procedural
3

9> (mp-show-waiting)
Events waiting to be scheduled:
  NIL     -----            BREAK-EVENT waiting for vision Waiting for: (MODULE VISION T)
1

10> (schedule-break-after-module :vision :delay :abort :details "aborted")
NIL
:ABORT

11> (mp-show-queue)
Events in the queue:
  0.000  -----            BREAK-EVENT not waiting for vision
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.000  -----            BREAK-EVENT after procedural
3

12> (mp-show-waiting )
Events waiting to be scheduled:
  NIL     -----            BREAK-EVENT waiting for vision Waiting for: (MODULE VISION T)
1

13> (schedule-event 3 (lambda ()) :module :vision)
#S(ACT-R-EVENT ...)

14> (mp-show-queue)
Events in the queue:
  0.000  -----            BREAK-EVENT not waiting for vision
  0.000  PROCEDURAL          CONFLICT-RESOLUTION

```

```

    0.000  -----          BREAK-EVENT after procedural
    3.000  VISION          RUN
    3.000  -----          BREAK-EVENT waiting for vision
5
15> (mp-show-waiting)
Events waiting to be scheduled:
0

```

schedule-break-after-all

Syntax:

```
schedule-break-after-all {:details details-value} -> [ actr-event | nil ]
```

Arguments and Values:

details-value ::= a string to output in the trace or **nil** (defaults to **nil**)

actr-event ::= the actual event which was generated and recorded by the meta-process event queue

Description:

Schedule-break-after-all creates a new break event with the provided details. The time for this new event is the greatest time of any event currently in the event queue of the current meta-process and its priority is :min. It will be inserted into the event queue such that it will occur after all of the events currently scheduled.

If there is no current meta-process then a warning is printed, no event is scheduled, and **nil** is returned.

The scheduled event is returned when successfully created and scheduled.

Examples:

```

1> (mp-show-queue)
Events in the queue:
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    12.000  NONE              a future action
2

2> (schedule-break-after-all :details "at the end")
#S(ACT-R-BREAK-EVENT ...)

3> (mp-show-queue)
Events in the queue:
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    12.000  NONE              a future action
    12.000  -----          BREAK-EVENT at the end
3

E> (schedule-break-after-all)
#|warning: schedule-break called with no current meta-process.|#
NIL

```

delete-event

Syntax:

`delete-event` *actr-event* -> [**t** | **nil**]

Arguments and Values:

actr-event ::= an actual ACT-R event as returned by one of the scheduling functions

Description:

If *actr-event* is an event which is currently in either the main event queue or the waiting queue of the current meta-process then `delete-event` removes that event from the queue that it is in and returns **t**.

If there is no current meta-process or the item is not in either event queue no action is taken and **nil** is returned.

Examples:

```
1> (schedule-break 10)
#S(ACT-R-BREAK-EVENT ...)

2> (defvar *event* *)
*EVENT*

3> (mp-show-queue)
Events in the queue:
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    10.000  -----          BREAK-EVENT
2

4> (delete-event *event*)
T

5> (mp-show-queue)
Events in the queue:
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
1

6> (delete-event *event*)
NIL

E> (delete-event 'not-an-event)
#|Warning: NOT-AN-EVENT is not a valid event.|#
NIL
```

Event Hooks

In addition to being able to schedule events it is possible to add functions which can monitor the events as they are executed. One can add what is called an “event hook” function which will be passed each event either before or after it is executed. The event hook can be used for recording information about what has happened in the model (for instance if one wanted to add an alternate tracing mechanism) or for checking for particular events to occur for data collection or other purposes. The hook function should not modify the event that is passed to it, and as noted above, the API does not provide any mechanism for doing so. When created, the hook functions are added to the current meta-process and persist across a reset. They are only removed if they are explicitly deleted with the [delete-event-hook](#) command, the meta-process in which they were created is itself deleted, or when a call to [clear-all](#) happens.

add-pre-event-hook

Syntax:

```
add-pre-event-hook hook-fn {warn-for-duplicate} -> [ hook-id | nil ]
```

Arguments and Values:

hook-fn ::= a function or the name of a function which takes one parameter

hook-id ::= a number which is the reference for the hook function that was added

warn-for-duplicate ::= a generalized boolean which indicates whether or not to show a warning if the same function is attempted to be put on the event hook again

Description:

If *hook-fn* is a function or the name of a function which is not already in the set of pre-event hooks for the current meta-process then it will be added to that set. That function will be called before each event is evaluated and it will be passed that event as its only parameter. The hook function will remain in the pre-event hook set for that meta-process until it is explicitly removed or until a clear-all occurs.

If the hook function is added to the set, then a unique *hook-id* is returned which can be used to explicitly remove that function from the set of pre-event hook functions.

If *hook-fn* is invalid or there is no current meta-process then a warning is printed and **nil** is returned. If *hook-fn* is already in the set of pre-event hook functions then **nil** is also returned and a warning is printed unless *warn-for-duplicate* is provided as **nil**.

Examples:

This example assumes that the count model from unit 1 of the tutorial is loaded.

```
1> (defun show-event (event)
      (format t "Hook sees event with module: ~S~%" (evt-module event)))
SHOW-EVENT

2> (add-pre-event-hook 'show-event)
0

3> (run .05)
Hook sees event with module: GOAL
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
Hook sees event with module: PROCEDURAL
0.000 PROCEDURAL CONFLICT-RESOLUTION
Hook sees event with module: PROCEDURAL
0.000 PROCEDURAL PRODUCTION-SELECTED START
Hook sees event with module: PROCEDURAL
Hook sees event with module: PROCEDURAL
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
Hook sees event with module: GOAL
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL PRODUCTION-FIRED START
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
Hook sees event with module: DECLARATIVE
0.050 DECLARATIVE START-RETRIEVAL
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
Hook sees event with module: NIL
0.050 ----- Stopped because time limit reached
0.05
12
NIL

4E> (add-pre-event-hook 'show-event)
#|Warning: SHOW-EVENT is already on the pre-event-hook list not added again|#
NIL

5> (add-pre-event-hook 'show-event nil)
NIL

E> (add-pre-event-hook 'show-event)
#|Warning: add-pre-event-hook called with no current meta-process|#
NIL

E> (add-pre-event-hook 'bad-function-name)
#|Warning: parameter BAD-FUNCTION-NAME to add-pre-event-hook is not a function|#
NIL
```

add-post-event-hook

Syntax:

```
add-post-event-hook hook-fn {warn-for-duplicate} -> [ hook-id | nil ]
```

Arguments and Values:

hook-fn ::= a function or the name of a function which takes one parameter
hook-id ::= a number which is the reference for the hook function that was added
warn-for-duplicate ::= a generalized boolean which indicates whether or not to show a warning if
the same function is attempted to be put on the event hook again

Description:

If hook-fn is a function or the name of a function which is not already in the set of post-event hooks for the current meta-process then it will be added to that set. That function will be called after each event is evaluated and it will be passed that event as its only parameter. The hook function will remain in the post-event hook set for that meta-process until it is either explicitly removed or until a clear-all occurs.

If the hook function is added to the set, then a unique hook-id is returned which can be used to explicitly remove that function from the set of post-event hook functions.

If hook-fn is invalid or there is no current meta-process then a warning is printed and **nil** is returned. If hook-fn is already in the set of post-event hook functions then **nil** is also returned and a warning is printed unless warn-for-duplicate is provided as **nil**.

Examples:

This example assumes that the count model from unit 1 of the tutorial is loaded.

```
1> (defun show-event (event)
      (format t "Hook sees event with module: ~S~%" (evt-module event)))
SHOW-EVENT

2> (add-post-event-hook 'show-event)
1

3> (run .05)
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
Hook sees event with module: GOAL
0.000 PROCEDURAL CONFLICT-RESOLUTION
Hook sees event with module: PROCEDURAL
0.000 PROCEDURAL PRODUCTION-SELECTED START
Hook sees event with module: PROCEDURAL
Hook sees event with module: PROCEDURAL
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
Hook sees event with module: PROCEDURAL
Hook sees event with module: GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
Hook sees event with module: PROCEDURAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
Hook sees event with module: PROCEDURAL
0.050 DECLARATIVE START-RETRIEVAL
```

```

Hook sees event with module: DECLARATIVE
  0.050   PROCEDURAL           CONFLICT-RESOLUTION
Hook sees event with module: PROCEDURAL
Hook sees event with module: NIL
  0.050   -----           Stopped because time limit reached
0.05
12
NIL

4E> (add-post-event-hook 'show-event)
#|Warning: SHOW-EVENT is already on the post-event-hook list not added again |#
NIL

5> (add-post-event-hook 'show-event nil)
NIL

E> (add-post-event-hook 'not-a-function)
#|Warning: parameter NOT-A-FUNCTION to add-post-event-hook is not a function |#
NIL

E> (add-post-event-hook 'show-event)
#|Warning: add-post-event-hook called with no current meta-process |#
NIL

```

delete-event-hook

Syntax:

delete-event-hook *hook-id* -> [hook-fn | **nil**]

Arguments and Values:

hook-id ::= a hook function id returned by one of the add event hook functions
hook-fn ::= the function or function name that was removed from the event hook

Description:

If the event hook function associated with *hook-id* is still a member of the set of event hooks in the current meta-process then it is removed from the set of hook functions and the function or function name that was used to create the event hook is returned.

If *hook-id* does not correspond to the id of an event hook or the event has already been removed from the set of event hooks then **nil** is returned. If there is no current meta-process, then a warning is printed and **nil** is returned.

Examples:

This example assumes that the event hook shown in the example for [add-pre-event-hook](#) exists.

```

1> (delete-event-hook 0)
SHOW-EVENT

```

```
2> (delete-event-hook 0)  
NIL
```

```
E> (delete-event-hook 'hook)  
NIL
```

```
E> (delete-event-hook 0)  
#|Warning: delete-event-hook called with no current meta-process |#  
NIL
```

About the Included Modules

The modules that are included with ACT-R fall into three general categories. The first is system or control modules which are not based on the theory and only serve to provide functionality to the software. The second and third categories are the cognitive modules which represent the theory of ACT-R, and those are subdivided into modules which provide the perceptual and motor actions which allow models to interact with an environment and modules which represent cognitive capabilities completely within the model. The only reason for distinguishing between the two types of cognitive modules is that the perceptual and motor modules have an additional interface for interacting with the world whereas the purely cognitive modules do not.

The following sections will describe the details of the modules that are included with the current system. For each module the operation of the module will be described, any parameters that the module has will be shown, if the module has any buffers their uses will be covered, and any commands which the module provides will be documented.

The basic system modules will be described first. Then the “internal” cognitive modules will be described, and after some description of the interface to the world the perceptual and motor modules will be described.

Printing module

The printing module controls what the system prints and where it goes. This module provides several parameters to configure the output. The commands available for outputting information are described in the [Printing and Output](#) section. Because it is a system module it has no buffer or impact on modeling results.

Parameters

:cbct

The copy buffer chunk trace parameter. This parameter controls whether or not an event will be shown in the trace indicating that a buffer has made a copy of a chunk. It can take a value of **t** or **nil**.

The default is **nil**.

If it is set to **t** then an event like this will be shown in the trace each time a buffer makes a copy of a chunk:

```
0.135  BUFFER      Buffer VISUAL copied chunk TEXT0 to TEXT0-0
```

It indicates which buffer made the copy along with the name of the original chunk and the name of the copy. Those events will be shown with the high and medium trace detail settings.

:cmdt

The command trace parameter controls where the [command output](#) is displayed.

The possible values for **:cmdt** are:

- **nil** – this turns off all command output for this model
- **t** – send command output to the ***standard-output*** stream
- a stream – command output is sent to that stream
- a pathname – the specified file is opened and output is appended to it
- a string – if the string denotes a valid pathname, then it is used as a pathname above

The default value is **t**.

If a file is used, it will be opened when the parameter is set and closed when either the parameter is changed or the model is deleted. Note that for file output the actual output to the file may be buffered in Lisp before being written. Thus, that output file should not be opened or read until the model is done with its output. Resetting or deleting the model will signal that it is done as will setting the `:cmdt` parameter to some other value.

:model-warnings

The `model-warnings` parameter controls whether or not [model warnings](#) are displayed. It can be set to **t** or **nil**.

The default value is **t**.

If it is set to **t**, then the model warnings are shown and if it is set to **nil** then they are not.

:trace-detail

The trace detail parameter controls which events are shown in the model's trace. It can be set to one of the values: **high**, **medium**, or **low**. The default value is **medium**.

If it is set to **high**, then all events which have a non-**nil** output setting are displayed. If it is set to **medium** then only those events with a **medium** or **low** output setting are shown, and if it is set to **low**, then only those events with a **low** output setting are shown.

:trace-filter

The trace filter parameter allows one more detailed control over which events are displayed in the trace. It can be set to a function, function name, or **nil**.

The default is **nil**.

If it is set to a function, then that function should take one parameter. For each event that occurs that function will be called with the event as its parameter. If the function returns **nil** then that event will not be displayed in the trace. Otherwise, the trace-detail level will be used to determine whether or not to display the event.

There is one filter function available with the system called `production-firing-only`. If that is set as the value of `:trace-filter` it will restrict the printed trace to only the production-fired events.

:v

The verbose parameter. The :v parameter controls where the [model output](#) is displayed. The trace of the model is included in model output.

The possible values for :v are:

- **nil** – this turns off all model output for this model
- **t** – send model output to the ***standard-output*** stream
- a stream – model output is sent to that stream
- a pathname – the specified file is opened and output is appended to it
- a string – if the string denotes a valid pathname, then it is used as a pathname above

The default value is **t**.

If a file is used, it will be opened when the parameter is set and closed when either the parameter is changed or the model is deleted. Note that for file output the actual output to the file may be buffered in Lisp before being written. Thus, that output file should not be opened or read until the model is done with its output. Resetting or deleting the model will signal that it is done as will setting the :v parameter to some other value.

Naming Module

The naming module provides the system with all of the symbols that it generates for the names of things. It guarantees that the system does not duplicate names within a model and that the name returned does not already name a chunk (chunks are the predominant item for which names are created by the system). It also allows for the uninterning of those symbols that were generated when the model is reset or deleted if they are not also in use in other models. Having a module to handle this is done instead of just using `gentemp` or `gensym` because those do not have a means of automatically cleaning up the symbol once it is no longer needed.

The names are generated by appending an increasing counter to the end of the provided name prefix (each prefix has its own counter). An additional benefit of the naming module is that the counters are reset to 0 when the model is reset. Thus, a deterministic model will always result in the same sequence of names being created when it is run after being reset, which can be very useful for debugging a model. Because the [random module](#) allows one to set the seed for the pseudo-random numbers the model generates, one can also temporarily make a stochastic model deterministic for debugging.

Parameters

:ncnar

The normalize chunk names after run parameter. The `:ncnar` parameter controls whether the model will call [normalize-chunk-names](#) after every call to one of the model running functions. If it is set to `t` then `normalize-chunk-names` will be called (without specifying the `unintern` parameter). If it is set to the value **delete** then `normalize-chunk-names` will be called with the `unintern` parameter true. If it is set to **nil** then `normalize-chunk-names` will not be called.

Having `normalize-chunk-names` called can be useful for debugging, but if the model generates a lot of chunks it may take a significant amount of time to complete. Thus for models that generate a lot of chunks, or for which there are several calls to model running functions setting this parameter to **nil** may improve performance.

The value of **delete** is provided as an option for extreme cases where the model is exhausting the computer memory and unable to run. Setting it to **delete** is not recommended for general use because some modules may have internal references to the chunk names which will be made invalid when the

uninterning option is used. Extra caution should therefore be used when specifying a value of **delete** for `:ncnar`.

The default value is **t**.

:dcnn

The dynamic chunk name normalizing parameter. The `:dcnn` parameter works in conjunction with the `:ncnar` parameter to normalize chunk names. If `:ncnar` is set to **t** or **delete** then `:dcnn` controls whether the model will normalize the chunk names while the model runs in addition to normalizing at the end of the run. If `:dcnn` is set to **t** then the chunk names stored in slots of chunks will be automatically updated whenever chunks are merged such that all chunks hold only the true chunk names of chunks in their slots. It essentially spreads the normalizing out during the run instead of performing it all at the end, but it will not delete any chunks until the end of the run if `:ncnar` is set to do so. This may be more efficient for some models and may also make debugging easier when a model breaks or while stepping through a run. If `:ncnar` is set to **nil** then the setting of the `:dcnn` parameter has no effect on the system.

The default value is **t**.

:dcsc-hook

The dynamic chunk slot change hook parameter. The `:dcsc-hook` parameter provides a way for modules or other code to be notified if a chunk is dynamically changed as a result of chunk normalizing. This parameter can be set with a function which takes one parameter and any number of such functions may be set. The reported value of this parameter is a list of all functions which have been set. Whenever a chunk is modified by normalizing, after the normalizing has changed a slot value, each of the functions set for this parameter will be called with the name of the chunk that had a slot modified through normalizing. The hook functions will be called every time a chunk changes due to normalizing regardless of how the normalizing occurred.

If the parameter is set to **nil** then all functions are removed from the `dcsc-hook`.

The default value is **nil**.

:short-copy-names

The `:short-copy-names` parameter controls how chunk names are created when a chunk is copied. The most common place for this to occur is when a chunk is placed into a buffer and it gets copied automatically. If the parameter is set to **nil** then the copy will have a hyphen and a number appended to it. The number is typically 0, but if that would conflict with another name it will be incremented until it is “safe” (see the [new-name](#) command). If this parameter is set to **t** then instead of adding a new hyphen and number when copying a chunk which is itself a copy only the number will be incremented as needed.

Assuming there are no conflicts, with this parameter set to **nil** if a chunk named “chunk” is copied it will be named chunk-0 and if chunk-0 is copied that new chunk will be named chunk-0-0. If this parameter is set to **t** then copying chunk will still result in chunk-0, but copying chunk-0 will result in the name chunk-1. In either case, if a chunk named chunk-0 is created explicitly by the model (it is not a copy of a chunk named chunk) then a copy of that chunk will be named chunk-0-0.

This parameter only really matters if copies of chunks are being made from copies – either directly or through the actions of modules like declarative or vision which may reuse copies of chunks. Its setting should not affect how the model operates with the standard modules because they do not rely on specific chunk names.

The default value is **nil**.

Commands

new-name

Syntax:

```
new-name {prefix} -> [ name-symbol | nil ]  
new-name-fct {prefix} -> [ name-symbol | nil ]
```

Arguments and Values:

`prefix ::=` if provided should be a string or symbol (defaults to “CHUNK” if not given)
`name-symbol ::=` a symbol created by appending a number onto the prefix

Description:

New-name is used to generate unique name symbols (which have been interned) within a model, similar to the Lisp function `gentemp`. Unlike `gentemp`, it does not guarantee that the symbol does not

already exist. What it does guarantee is that it was not previously returned by new-name within the current run of the model and that it is not currently used to name a chunk in the current model.

When the module is deleted or reset it clears its name space and uninterns any symbols it has generated which are no longer "necessary". By necessary, it means that no instance of the naming module has generated such a name, nor was that symbol interned prior to new-name generating it for the first time.

If there is no current model or current meta-process or a parameter other than a symbol or string is provided then a warning is printed and **nil** is returned.

Anywhere a modeler would consider using gentemp, new-name should probably be used instead to guarantee the automatic cleanup upon reset or model deletion. Note that it is typically not necessary to generate a name for a new chunk because omitting a name in the call to define-chunks results in the chunk getting a name generated by new-name automatically.

Examples:

```
> (new-name)
CHUNK0

> (new-name temp)
TEMP0

1> (new-name-fct 'fact)
FACT0

2> (new-name-fct "FACT")
FACT1

1> (new-name "temp")
TEMP1

2> (define-chunks (temp2 isa chunk))
(TEMP2)

3> (new-name "temp")
TEMP3

E> (new-name 10)
#|Warning: Invalid parameter passed to new-name. Must be a string or symbol.|#
NIL

E> (new-name)
#|Warning: get-module called with no current model.|#
#|Warning: No naming module available cannot create new name.|#
NIL

E> (new-name "A")
#|Warning: get-module called with no current meta-process.|#
#|Warning: No naming module available cannot create new name.|#
NIL
```

release-name

Syntax:

release-name *name* -> [**t** | **nil**]

release-name-fct *name* -> [**t** | **nil**]

Arguments and Values:

name ::= a symbol which was generated by *new-name*

Description:

Release-name can be used to possibly unintern symbols which have been generated by *new-name*. This is the same process which occurs on a reset or deletion of the naming module for symbols generated by *new-name*, but in some circumstances one may want to perform such a cleanup without resetting. Thus, if the symbol given in *name* was generated by the *new-name* command and no instance of the naming module other than the one in the current model has generated such a name and that symbol was not interned prior to *new-name* generating it for the first time then it is uninterned.

If the symbol is uninterned, then **t** is returned otherwise **nil** is returned.

If there is no current model or meta-process then a warning is printed and **nil** is returned.

Generally, this command is not necessary because a reset or clear-all will automatically clear out the symbols. However, if one is generating an extremely large number of temporary names with *new-name* it can lead to issues with the size of the symbol table in Lisp and explicitly removing names prior to a reset may be useful.

Examples:

```
1> (find-symbol "CHUNK0")
NIL
NIL

2> (new-name)
CHUNK0

3> 'chunk1
CHUNK1

4> (find-symbol "CHUNK1")
CHUNK1
:INTERNAL

5> (new-name)
CHUNK1

6> (release-name-fct 'chunk0)
```

T

```
7> (release-name chunk1)
NIL
```

```
8> (release-name-fct 'chunk0)
NIL
```

```
E> (release-name chunk0)
#|Warning: get-module called with no current model. |#
#|Warning: No naming module available cannot release name CHUNK0. |#
NIL
```

```
E> (release-name chunk0)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No naming module available cannot release name CHUNK0. |#
NIL
```

new-symbol

Syntax:

```
new-symbol {prefix} -> [ new-symbol | nil ]
new-symbol-fct {prefix} -> [ new-symbol | nil ]
```

Arguments and Values:

prefix ::= if provided should be a string or symbol (defaults to “CHUNK” if not given)
new-symbol ::= a newly interned symbol created by appending a number onto the *prefix*

Description:

New-symbol is used to generate and intern a unique symbol, similar to the Lisp function gentemp and the new-name command. Like gentemp, it guarantees that the symbol does not already exist. Unlike the command new-name, it does not guarantee that the numbering is reset with the model.

When the module is deleted or reset all new-symbols created are uninterned.

If there is no current model or meta-process then a warning is printed and **nil** is returned.

New-symbol should only be used when one needs a completely new symbol and the name of that symbol is not meaningful to the model (for instance something that might show up in the trace should use new-name).

Examples:

```
> (new-symbol)
CHUNK-0
```

```
1> (new-symbol "temp")
TEMP-0
```

```
2> (new-symbol-fct 'temp)
TEMP-1
```

```
3> (let (temp-2 temp-3))
NIL
```

```
4> (new-symbol temp)
TEMP-4
```

```
E> (new-symbol)
#|Warning: get-module called with no current model. |#
#|Warning: No naming module available cannot create new symbol. |#
NIL
```

```
E> (new-symbol)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No naming module available cannot create new symbol. |#
NIL
```

Random module

The random module provides a consistent pseudorandom number generator for the system. It is not dependent on the Lisp random function or the Lisp `*random-state*` global variable. This makes it consistent across all instances of ACT-R regardless of the Lisp application or OS being used. This is very useful for teaching because it guarantees the output of the tutorial material will be the same for all users. It also makes models and modules easier to debug and verify because the random state can be set with an ACT-R parameter for testing. It also serves to protect the model from any potential weaknesses in the random function of a particular Lisp.

The particular pseudorandom number generator chosen is the Mersenne Twister (as implemented by the `mt19937ar.c` code) because it is designed for Monte-Carlo simulations. Details can be found at:

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

That algorithm is used by some Lisps for the random function already. However, because the internal representations of state differ among Lisps it is still necessary to use the ACT-R random module implementation for consistency of ACT-R across systems.

Parameters

:randomize-time

This parameter can be set to **t**, **nil**, or an integer. It is used by the [randomize-time](#) command to determine the range in which a specified time will be randomized. The `randomize-time` command is used mainly by the perceptual and motor modules to add noise to the action times.

The default value is **nil** which means not to randomize the times. A value of **t** is the same as setting the value to 3.

:seed

This is the current seed for the pseudorandom number generator. It must be a list of two numbers. The first is used to initialize the array used by the Mersenne Twister algorithm and the second is an offset from that starting point where the model should start (or where it currently is if the value is read after the model has been run). Thus, if one is specifying a seed explicitly, the second number

should probably be kept “small” because that many pseudorandom numbers will be generated when the parameter is set to get to the offset point.

There is no default value for the seed parameter. When the module is created in a model it generates a new seed based on the current result of `get-internal-real-time`. Resetting the model does not return the seed to that point – the random module will continue generating new pseudorandom numbers from the point where it last left off.

Commands

act-r-random

Syntax:

act-r-random *limit* -> [value | **nil**]

Arguments and Values:

limit ::= a positive number (either an integer or floating point)

value ::= a pseudorandom number which is non-negative and less than *limit*

Description:

Act-r-random will operate like `random` as defined in the ANSI Lisp specification, except without the optional parameter for a random-state. It uses the seed value from the current model to generate the next random number. Thus unless models explicitly set the same seed value each model will have a different sequence of pseudorandom numbers returned by `act-r-random`.

It returns a pseudorandom number that is a non-negative number less than *limit* and of the same type as *limit*. An approximately uniform choice distribution is used. If *limit* is an integer, then each of the possible results occurs with (approximate) probability $1/\text{limit}$.

If an invalid value is passed to `act-r-random` then a warning is printed and **nil** is returned.

If there is no current model a warning will be printed and a pseudorandom number generated from a special instance of the random module will be returned.

Examples:

```
1> (sgp :seed)
:SEED (94875970549 0) (default NO-DEFAULT) : Current seed of the random number generator
((94875970549 0))
```

```

2> (act-r-random 100)
46

3> (act-r-random 34.5)
11.947622

4> (act-r-random 1000)
679

5> (act-r-random 0.5)
0.31684825

6> (sgp :seed (94875970549 1))
((94875970549 1))

7> (act-r-random 34.5)
11.947622

8> (act-r-random 1000)
679

9> (act-r-random 0.5)
0.31684825

E> (act-r-random 'a)
#|Warning: Act-r-random called with an invalid value A|#
NIL

E> (act-r-random 2)
#|Warning: get-module called with no current model.|#
0

```

act-r-noise

Syntax:

act-r-noise s -> [value | **nil**]

Arguments and Values:

s ::= a non-negative number

value ::= a pseudorandom number generated as described below

Description:

Act-r-noise generates a value from a logistic distribution (approximation of a normal distribution) with a mean of 0 and an s value as given. It does this using the act-r-random function. The s value is related to the variance of the distribution, σ^2 , by this equation:

$$\sigma^2 = \frac{\pi^2}{3} s^2$$

If s is invalid a warning is printed and **nil** is returned.

If there is no current model or meta-process a warning will be printed and the noise will be based on a pseudorandom number generated from a special instance of the random module (as indicated by act-r-random).

Examples:

```
> (act-r-noise 0)
0.0

> (act-r-noise .5)
0.15807568

> (act-r-noise .5)
1.4271247

E> (act-r-noise 'a)
#|Warning: Act-r-noise called with an invalid s A |#
NIL

E> (act-r-noise .5)
#|Warning: get-module called with no current model. |#
-1.131652
```

randomize-time

Syntax:

randomize-time *time* -> [time | rand-time]

Arguments and Values:

time ::= should be a number
rand-time ::= a randomized time value

Description:

Randomize-time is used to return a number randomly chosen from a uniform distribution around the provided time. It depends on the setting of the [:randomize-time](#) parameter in the current model. If the parameter is set to **nil** then no randomization is done by the randomize-time command and time is returned.

If the :randomize-time parameter is a number or **t** (which means use the default number 3) then a number randomly chosen from the uniform distribution in the range of:

$$\left[time * \frac{n-1}{n}, time * \frac{n+1}{n} \right)$$

where n is the value of :randomize-time, will be returned.

If time is not a number or there is no current model or meta-process, then a warning is printed and time is returned.

Examples:

```
1> (sgp :randomize-time nil)
(NIL)

2> (randomize-time 10)
10

1> (sgp :randomize-time t)
(T)

2> (randomize-time 50)
65.277435

3> (randomize-time 50.0)
58.443718

1> (sgp :randomize-time 100)
(100)

2> (randomize-time 100)
100.1965

3> (randomize-time 100)
99.349

E> (randomize-time 'a)
#|Warning: Invalid value passed to randomize-time: A|#
A

E> (randomize-time 10)
#|Warning: get-module called with no current model.|#
10

E> (randomize-time 10)
#|Warning: get-module called with no current meta-process.|#
10
```

randomize-time-ms

Syntax:

```
randomize-time-ms time -> [ time | rand-time ]
```

Arguments and Values:

time ::= should be an integer
rand-time ::= a randomized time value

Description:

Randomize-time-ms is used to return a number randomly chosen from a uniform distribution around the provided time. The difference between randomize-time-ms and [randomize-time](#) is that

randomize-time-ms returns integer results because it is for use with times in milliseconds instead of seconds and the precision of the scheduling is in milliseconds (an attempt to schedule a time specified in milliseconds which is not an integer will result in a warning and failure to schedule the event). It depends on the setting of the `:randomize-time` parameter in the current model. If the parameter is set to `nil` then no randomization is done by the randomize-time command and time is returned.

If the `:randomize-time` parameter is a number or `t` (which means use the default number 3) then a number randomly chosen from the uniform distribution in the range of:

$$\left[\text{time} - \left\| \frac{1}{n} \right\|, \text{time} + \left\| \frac{1}{n} \right\| \right]$$

where `n` is the value of `:randomize-time`, will be returned.

If `time` is not an integer or there is no current model or meta-process, then a warning is printed and `time` is returned.

Examples:

```
1> (sgp :randomize-time nil)
(NIL)
```

```
2> (randomize-time-ms 10)
10
```

```
3> (sgp :randomize-time t)
(T)
```

```
4> (randomize-time-ms 50)
37
```

```
1> (sgp :randomize-time 100)
(100)
```

```
2> (randomize-time-ms 100)
99
```

```
3> (randomize-time-ms 100)
101
```

```
4> (randomize-time-ms 100)
101
```

```
E> (randomize-time-ms 'a)
#|Warning: Invalid value passed to randomize-time-ms: A. Value must be an integer.|#
A
```

```
E> (randomize-time-ms 1.0)
#|Warning: Invalid value passed to randomize-time-ms: 1.0. Value must be an integer.|#
1.0
```

```
E> (randomize-time-ms 1)
#|Warning: get-module called with no current model.|#
```

1

```
E> (randomize-time-ms 1)
#|warning: get-module called with no current meta-process. |#
1
```

Buffer trace module

The buffer trace module records the actions which occur through the buffers while a model runs and can report that information in a text display (an alternative to the standard trace) or return a list of Lisp structures which encode the actions that took place.

The module has no buffer of its own. It has five parameters that control the tracing and a command for retrieving the trace information.

Here is an example of the buffer trace using the demo2 model from the tutorial:

```

> (do-demo2)
0.000 | PRODUCTION | GOAL | VISUAL-LOCATION | VISUAL | MANUAL |
0.025 | +FIND-UNATTEN+ | . | .VISUAL-LOCATIO | . | . |
0.050 | ***** | . | . | . | . |
0.075 | +ATTEND-LETTE+ | . | ===== | .VISUAL-LOCATIO | . |
0.100 | ***** | . | . | . | . |
0.125 | . | . | . | . | +CMD MOVE-ATT+ |
0.150 | . | . | . | . | ***** |
0.175 | . | . | . | . | ***** |
0.185 | +ENCODE-LETTE+ | . | . | . | TEXT0 |
0.210 | ***** | . | . | . | . |
0.235 | ***** | . | ===== | . | . |
0.260 | . | . | . | . | . |
0.285 | . | . | . | . | . |
0.310 | . | . | . | . | . |
0.335 | . | . | . | . | . |
0.360 | . | . | . | . | . |
0.385 | . | . | . | . | . |
0.410 | . | . | . | . | . |
0.435 | + RESPOND + | . | . | . | . |
0.460 | ***** | . | . | . | . |
0.485 | ***** | . | ===== | . | . |
0.510 | . | . | . | . | +CMD PRESS-KE+ |
0.535 | . | . | . | . | ***** |
0.560 | . | . | . | . | ***** |
0.585 | . | . | . | . | ***** |
0.610 | . | . | . | . | ***** |
0.635 | . | . | . | . | ***** |
0.660 | . | . | . | . | ***** |
0.685 | . | . | . | . | ***** |
0.710 | . | . | . | . | ***** |
0.735 | . | . | . | . | ***** |
0.760 | . | . | . | . | ***** |
0.785 | . | . | . | . | ***** |
0.810 | . | . | . | . | ***** |
0.835 | . | . | . | . | ***** |
0.860 | . | . | . | . | ***** |
0.885 | . | . | . | . | ***** |
0.910 | . | . | . | . | ***** |
0.935 | . | . | . | . | ***** |
0.960 | . | . | . | . | ***** |
0.970 | . | . | . | . | ***** |
0.995 | . | . | . | . | ***** |
1.020 | . | . | . | . | ***** |
1.035 | . | . | . | . | ***** |
1.060 | . | . | . | . | ***** |
1.060 | ----- | Stopped because no events left to process
"v"

```

These are the parameters that were set to achieve that:

```
(sgp :buffer-trace t :buffer-trace-step .025 :traced-buffers (production goal visual-  
location visual manual))
```

The parameters are described below as are the details about what is shown in that trace.

The following information is recorded at each event of the model and then aggregated over all events at a given time on a buffer by buffer basis:

- Whether the buffer's module is busy
- Whether the buffer's module is in an error state
- Whether the buffer is full
- If the buffer is cleared
- If the chunk in the buffer is modified
- If a request is sent to the module
- If a new chunk is set in the buffer

For the first 5, if the stated condition is true during any event at the current time the buffer record will indicate it as t. For requests, only the last request at the given time is recorded and the information that is recorded is either the slots and values of the request or the details string of the event if it has one. If a chunk is set in the buffer, then the name of that chunk is recorded, and as with requests, only the last setting at a specific time is recorded.

The buffer trace attempts to show all of that information in a textual format. At each time step of the model i.e. each time that would be shown in the regular trace, and at extra time steps if needed to meet the :buffer-trace-step setting, there will be one line of trace printed. At the start of the line will be the time of the summary and for each buffer traced there will be a column of information in the trace (the columns are separated by the vertical bar character '|'). In a column for a buffer, the first character will be "E" if the module is in an error state or a space otherwise. The second character will be a "." if there is currently a chunk in the buffer or a space if it is empty. The rest of the column will show one of the following things in their order of priority (truncated to maintain the column width):

- If there is a new chunk set in the buffer the name of that chunk
- If there is a request the request is shown between two "+" characters
- If the buffer is modified it will show a series of "=" characters
- If the buffer is cleared it will show a series of "-" characters
- If the module is busy it will show a series of "*" characters
- Otherwise it will be filled with spaces

The graphic tracing tools of the ACT-R Environment rely on the buffer trace module to generate the data it uses as does the [BOLD computation](#) module.

Parameters

:buffer-trace

If the `:buffer-trace` parameter is set to **t**, then the normal event trace is disabled and the buffer trace is printed instead. The default value for this parameter is **nil**.

:buffer-trace-hook

This parameter allows the modeler to have access to the buffer trace summaries “on the fly”. It defaults to **nil**, but if it is set to a function which takes one parameter then that function will be called with every buffer-record structure (as described in the [get-current-buffer-trace](#) command) at the time they are available. They are made available when the clock changes, when an extra time step is inserted because of the `:buffer-trace-step` setting, or when the run terminates.

:buffer-trace-step

If `:buffer-trace-step` is set to a number it specifies the maximum amount of time that is allowed to elapse before creating a new buffer summary. Note however that there may be smaller time steps that correspond to model actions. The default value is **nil** which means to only create the records when there are events i.e. there is no minimum or maximum time guaranteed between the buffer summaries.

:save-buffer-trace

This parameter controls whether the buffer trace information is recorded for later recovery by the [get-current-buffer-trace](#) command. It defaults to **nil**, which means do not record the information. If it is set to **t**, then the buffer trace module will record the summary data so that it can be retrieved for later use. This parameter does not alter the printed trace i.e. if `:buffer-trace` is **nil** and `:save-buffer-trace` is **t** then the standard event trace will be printed even though the buffer trace data is being recorded.

:traced-buffers

This can be set to a list of buffers which are to be traced. It has a default value of **t**, which means to trace all buffers. Only those buffers specified on this list will have their data recorded. The order of

the buffers in this list is the order they will be printed in the output when the buffer-trace is displayed. If it is set to **t** all buffers will be displayed in alphabetical order.

Commands

get-current-buffer-trace

Syntax:

get-current-buffer-trace {*clear*} -> (buffer-record*)

Arguments and Values:

clear ::= a generalized boolean indicating whether or not to clear the recorded trace information

buffer-record ::= a structure which is defined like this:

```
(defstruct buffer-record ms-time buffers)
```

In the buffer-record structure the ms-time slot is a number indicating the model time at which the summary was recorded in milliseconds and the buffers slot is a list of buffer-summary structures.

buffer-summary ::= a structure which is defined like this:

```
(defstruct buffer-summary name cleared busy busy->free error  
error->clear full modified request chunk-name notes)
```

In the buffer-summary structure the name slot holds the name of the buffer for which this is a record. The cleared, busy, busy->free, error, error->clear, full, and modified slots are flags which will be either **t** or **nil** to indicate if the named condition was true during that time. The request slot will hold either a string of the slot-value pairs in the request or the document string of the event (if it had one) for the last request made at the recorded time or **nil** if there was no request made through the buffer at that time. The chunk-name slot will hold a string of the name of the last chunk which was placed into the buffer at the specified time, or **nil** if there was no chunk placed into the buffer at that time. The notes slot will contain the last note added to the buffer using the [add-buffer-trace-notes](#) command, or **nil** if no notes have been added to the buffer.

Description:

The `get-current-buffer-trace` command has one optional parameter. It returns a list of the buffer-record structures which have been collected since the `:save-buffer-trace` parameter was set to **t** in the current model. If the `:save-buffer-trace` parameter was not set it will return **nil**. If there is no current model or current meta-process then the command will print a warning and return **nil**.

If the optional parameter is provided with a true value then the buffer record stored in the module will be deleted after generating the summary which is returned. If it is **nil** or not provided then the record will be retained.

This command, along with the `:buffer-trace-hook` parameter, are provided as a mechanism for modelers to collect buffer/module activity without needing to engineer special purpose hooks or make any module modifications for such purpose. This data is used by the graphic tracing tools in the ACT-R Environment as well by the code which produces [BOLD predictions](#) from model runs. It is also used by the “[module demand](#)” functions which summarize the results and may be more useful than parsing the raw data.

Examples:

This example uses the `demo2` model as was shown in the buffer trace above with the `:save-buffer-trace` parameter also set to **t**.

```
> (pprint (get-current-buffer-trace))
(#S(BUFFER-RECORD :MS-TIME 0
  :BUFFERS (#S(BUFFER-SUMMARY :NAME PRODUCTION
    :CLEARED NIL
    :BUSY T
    :BUSY->FREE NIL
    :ERROR NIL
    :ERROR->CLEAR NIL
    :FULL NIL
    :MODIFIED NIL
    :REQUEST "FIND-UNATTENDED-LETTER"
    :CHUNK-NAME NIL
    :NOTES NIL)
    #S(BUFFER-SUMMARY :NAME GOAL
    :CLEARED NIL
    :BUSY NIL
    :BUSY->FREE NIL
    :ERROR NIL
    :ERROR->CLEAR NIL
    :FULL T
    :MODIFIED NIL
    :REQUEST NIL
    :CHUNK-NAME "GOAL"
    :NOTES NIL)
    #S(BUFFER-SUMMARY :NAME VISUAL-LOCATION
    :CLEARED NIL
    :BUSY NIL
    :BUSY->FREE NIL
    :ERROR NIL
    :ERROR->CLEAR NIL
    :FULL T
    :MODIFIED NIL
    :REQUEST NIL
    :CHUNK-NAME "LOC0"
    :NOTES NIL)
    #S(BUFFER-SUMMARY :NAME VISUAL
```

```

:CLEARED NIL
:BUSY NIL
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL NIL
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)
#S(BUFFER-SUMMARY :NAME MANUAL
:CLEARED NIL
:BUSY NIL
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL NIL
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)))
#S(BUFFER-RECORD :MS-TIME 25
:BUFFERS (#S(BUFFER-SUMMARY :NAME PRODUCTION
:CLEARED NIL
:BUSY T
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL NIL
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)
#S(BUFFER-SUMMARY :NAME GOAL
:CLEARED NIL
:BUSY NIL
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL T
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)
#S(BUFFER-SUMMARY :NAME VISUAL-LOCATION
:CLEARED NIL
:BUSY NIL
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL T
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)
#S(BUFFER-SUMMARY :NAME VISUAL
:CLEARED NIL
:BUSY NIL
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL NIL
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)
#S(BUFFER-SUMMARY :NAME MANUAL

```

```
:CLEARED NIL
:BUSY NIL
:BUSY->FREE NIL
:ERROR NIL
:ERROR->CLEAR NIL
:FULL NIL
:MODIFIED NIL
:REQUEST NIL
:CHUNK-NAME NIL
:NOTES NIL)))
```

...

```
E> (get-current-buffer-trace)
#|Warning: get-module called with no current model.|#
NIL
```

```
E> (get-current-buffer-trace)
#|Warning: get-module called with no current meta-process.|#
NIL
```

add-buffer-trace-notes

Syntax:

add-buffer-trace-notes *buffer notes* -> [notes | nil]

Arguments and Values:

buffer ::= a symbol which should be the name of a buffer

notes ::= any data which one wants to have stored in the buffer trace record

Description:

The `add-buffer-trace-notes` command takes two parameters. The first is a symbol which should name a buffer and the second is some data to store in the buffer trace of the current model in the current meta-process at the current time. If *buffer* is the name of a buffer in the current model then *notes* will be recorded in the buffer-summary for the named buffer at the current time and *notes* will be returned. If there is no current model or *buffer* does not name a valid buffer then the command will print a warning and return **nil**.

Any notes which are added will also be displayed in the graphic tracing tools of the ACT-R Environment when one places the cursor over an event box in the image.

Examples:

```
> (add-buffer-trace-notes 'goal "save this string")
"save this string"
```

```
E> (add-buffer-trace-notes 'not-a-buffer 10)
#|Warning: NOT-A-BUFFER does not name a buffer in the current model no notes added.|#
```

NIL

```
E> (add-buffer-trace-notes 'goal "no model")  
#|warning: No current model so cannot add notes. |#  
NIL
```

Central Parameters Module

This module maintains three parameters of the system and provides no other model relevant components. These parameters are used by more than one of the cognitive modules to control how they operate. Thus, they need to exist outside of any one of them in particular, and may also be referred to by new modules as well. It also provides a system parameter which can be used to effectively overwrite the default value for any parameter.

Technically, this module is not automatically loaded by ACT-R because it is provided in the support directory of the distribution and thus only loaded when needed. Therefore, any module or other file which uses the parameters in this module should ensure that it is available by making a require-compiled call to be safe:

```
(require-compiled "CENTRAL-PARAMETERS" "ACT-R-support:central-parameters")
```

Since the declarative and procedural modules also currently require it that is not actually necessary, but still recommended if one is using these parameters when defining a new module.

This module provides a way for other modules to register that they are using the :esc parameter and note which of their parameters rely on :esc being set to **t**. It will output a warning if :esc is **nil** at the start of a model run when any of those registered parameters have been changed from their default values – the assumption is that for a parameter which uses :esc that parameter’s default value is valid when :esc is set to **nil** and any other setting would require :esc to be **t**.

This section will provide the basic details of these parameters. The specific modules which use them will describe the details of how these parameters modify their operations.

This module has no buffer.

Parameters

:er

This is the enable randomness parameter. It specifies how deterministically modules should operate. It can be set to **t** which means act non-deterministically or **nil** which means act deterministically. The default value is **nil**. Generally, the setting of this parameter is used to determine how to break “ties” in a module’s processing and it has more impact when :esc is **nil** because when the subsymbolic

parameters are enabled there are not often ties that need to be broken because of noise in the computations.

:esc

This is the enable subsymbolic computations parameter. It specifies whether modules should work in a purely symbolic fashion or whether they should use their full subsymbolic processing. The default value is **nil** which means that modules should be purely symbolic. If it is set to **t**, then modules should use whatever subsymbolic computations they provide.

:ol

This is the optimized learning parameter. It specifies whether modules should use their full computational forms of subsymbolic quantities or use some simplified approximation. Currently, it is only used by the declarative system to control the base-level learning equation, but other modules could also check it as a guide. It can be set to **t**, which means use the optimized (or simplified) form of the computation, **nil**, which means use the full computation, or a positive number, which can be used as a parameter specifying how much optimization to apply. The default value is **t**.

System Parameters

:starting-parameters

The `:starting-parameters` parameter allows one to set parameters which will be applied at the start of all models that are defined. It can be set to a list of parameter values which are valid for passing to `sgp-fct` and those parameter values will be set before the code in a model definition is evaluated. If this parameter is set in a `.lisp` file that is placed into the `user-loads` directory then those settings will always be made, and that is the recommended use for this parameter.

To be completely safe, one should also make sure that the `central-parameters` module is loaded using this call before setting the parameter:

```
(require-compiled "CENTRAL-PARAMETERS" "ACT-R-support:central-parameters")
```

particularly if it is being done within the context of a module or extension that is loaded prior to the `user-loads` directory being processed. However, since the declarative and procedural modules currently also use the `central-parameters` module it should always be available even without explicitly requiring it.

Commands

register-subsymbolic-parameters

Syntax:

```
register-subsymbolic-parameters { param*} -> nil
```

Arguments and Values:

param ::= a keyword which should name a valid parameter

Description:

The register-subsymbolic-parameters command is used to indicate when a module's parameters depend on the :esc parameter being set to **t**. When a model first starts running (time 0) if :esc is set to **nil** and any of the parameters which have been registered in this way have a value other than their default value a warning will be printed like this:

```
#|Warning: Subsymbolic parameters have been set but :esc is currently nil. |#
```

This command only needs to be called once for a given parameter. It does not need to go into the creation or reset functions of a module, and should be called directly after the module definition.

It always returns **nil**.

Examples:

```
> (register-subsymbolic-parameters :ul :alpha)
NIL
```

The Procedural System

The procedural system implements the procedural cognitive module of the theory. The procedural system is implemented as three separate modules in the code. Those three modules are the [procedural module](#), the [utility module](#), and the [production-compilation module](#). The procedural module handles the productions' specification and matching at the symbolic level and the conflict resolution among productions which relies on the utility module. The utility module handles the computation of the subsymbolic quantity Utility for the productions and maintains the parameters and history information necessary to do so. Finally, the production-compilation module is responsible for the learning of new productions by the model when it is enabled.

Procedural Module

The procedural module implements the procedural memory system. It provides the commands for specifying productions, a pattern matcher that works in conjunction with the utility module to choose which production to fire, and tools for inspecting and debugging the productions of a model. This module holds a central role in the system because the productions coordinate the interaction between all of the other modules of the theory.

A production specifies a set of conditions to match against the current contents of the buffers and the states of the buffers and modules along with a set of actions to perform. When the conditions of a production are satisfied it may be selected to fire, and when it fires its actions are executed. See the ACT-R tutorial for more details on specifying and using productions.

Conflict Resolution

Only one production can be selected and fired at any time. The process by which the next production to fire is chosen is called conflict resolution. When conflict resolution occurs all productions have their conditions checked to determine which ones match the current state. The conditions specify a conjunction of tests which must all be true for the production to match (see the [p command](#) and ACT-R tutorial unit 1 for more information on how the matches occur). Among those that match, the one that has the highest utility value will be chosen (see the [utility module](#) for details on the utility calculations). If there is a tie for the highest utility value then the setting of the [:er parameter](#) determines how that tie is broken. If :er is **t** then the tie is broken randomly. If :er is **nil** then a deterministic process is used such that the same production will be chosen for that model each time the same tie condition occurs. However, that deterministic process is not specified as part of the procedural module's definition because it is not intended to be a process which one relies on for production ordering or model control.

The procedural module will automatically schedule conflict-resolution events to perform the conflict resolution process. The first one is scheduled at time 0 and a new one is scheduled after each production fires. If no production is selected during a conflict-resolution event then a new conflict-resolution event is scheduled to occur after the next change occurs. A change in this context is any other [non-maintenance event](#). The module also schedules production-selected and production-fired events as a result of conflict resolution. Those events will indicate the specific production which was selected and fired and will look like this in the trace:

0.000	PROCEDURAL	CONFLICT-RESOLUTION
0.000	PROCEDURAL	PRODUCTION-SELECTED START
0.050	PROCEDURAL	PRODUCTION-FIRED START

There is also a procedural module request event scheduled after the production-selected event. That event will not be shown in the trace and serves to indicate that the procedural module has started an action for purposes of the buffer trace module (the event itself performs no actions). Several other events are scheduled as a result of the production selection and production firing processes. Those will be described under the production creation command [p](#).

The production-fired event is scheduled based on the production's specified action time. That defaults to 50ms, but is controlled by parameters (see the [utility module](#)). In addition, if one sets the [:vpft parameter](#) to **t** then that time has noise added to it using the [randomize-time](#) command.

Parameters

:conflict-set-hook

This parameter allows one to specify functions which can intervene in the production selection process. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). During conflict-resolution, after all productions have been matched and have had their utilities calculated, a list of the productions which matched in order of utility (highest first – thus the first production on the list is the one which will be selected unless something intervenes) will be passed to each of the functions on the conflict-set-hook list. The return value of the functions on the conflict-set-hook list is used as follows:

- If it is the name of a production in the conflict set then that production will be the one selected regardless of the normal conflict resolution mechanism.
- If it is a string, no production will be selected and a warning will be output to the trace indicating that conflict resolution was canceled and the string returned will be provided as a reason in that warning. The next conflict resolution event will be scheduled to occur after the default action time (see the [:dat parameter](#)).
- If it is **nil** then the normal conflict resolution mechanism is used.
- If it is anything else, a warning will be output and the normal mechanism will be used.

If multiple hook functions are set and more than one returns a non-nil value a warning will be displayed and the result of one of those non-nil values will be used. Which one gets used is picked

by an unspecified mechanism. No assumptions should be made as to which will be applied and it may vary from run to run. In general one should not have more than one conflict-set-hook function returning non-nil.

If the parameter is set to **nil** then all functions are removed from the conflict-set-hook list.

:crt

The conflict resolution trace parameter can be used to include the details of the conflict resolution process in the trace. If it is set to **t** then after each conflict-resolution event, for each production, it will print out either that the production matches the current state or that it fails to match along with the first condition which failed to match as would be shown by the [whynot command](#).

:cst

The conflict set trace parameter can be used to print the details of the productions in the conflict set during conflict resolution. If it is set to **t** then after each conflict-resolution event the current instantiation of each production that matches is printed in the command trace. The instantiation of a production is the production text with the variables replaced by the specific values they have based on the current buffer contents.

:cycle-hook

This parameter allows one to specify functions to be called automatically when productions fire. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). During the production-fired event each of the functions on the cycle-hook list will be called with the name of the production that is firing as the parameter. The return values of those functions are ignored. If the parameter is set to **nil** then all functions are removed from the cycle-hook list. A function should only be set once. If a specific function is specified more than once as a value for the **:cycle-hook** parameter a warning will be displayed and a value of **nil** will be returned as the current value in that call to `sgp`.

:dat

The default action time parameter specifies the default time that it takes to fire a production in seconds. That is the amount of time that passes between the production's selection and fired events. The default value is .05 (50ms) and generally that value is not changed.

:do-not-harvest

This parameter controls the strict harvesting mechanism of the productions. By default all buffers are subjected to the strict harvesting practice, but buffers can be exempted from that by specifying them with the `:do-not-harvest` parameter. The parameter is set one buffer at a time. The reported value for the parameter is a list of all the buffers which have been set using the parameter. If it is set to **nil** then all buffers will be subjected to strict harvesting. A particular buffer should only be set once and if it is specified again a warning will be displayed and **nil** will be returned as the current value.

:lhst

The left hand side trace parameter controls whether the matching conditions from the selected production are shown in the trace. If this parameter is set to **t** (which is the default) and the [:trace-detail parameter](#) is set to high then the matching conditions are displayed. If this parameter is set to **nil** or the `:trace-detail` is medium or low, then such events are not shown.

:ppm

The procedural partial matching parameter controls whether productions are allowed to be selected and fired even if they are not a perfect match to the buffers' current contents. If this parameter is set to a number then production matching is allowed to occur for buffer tests which are not a perfect match to the chunk in the buffer. The default value is **nil** which means productions will only match when the buffer tests are exact matches.

If the procedural partial matching is enabled, then slots tested for equality (slots without a modifier and those with an explicit `=` modifier) in buffer tests of productions may be considered a match even if the value is not [chunk-slot-equal](#) to the current value in the slot of the chunk in the buffer. The partial match will occur if there is a similarity value between the value specified for the slot in the production and the value currently in that slot of the buffer and that similarity is greater than the maximum similarity difference as returned from the declarative module's [similarity command](#).

A production which is not a perfect match will have its utility value decremented for purposes of determining which production to fire, but the production's utility parameters and the learning of utility are not affected by that decrementing. The default decrement will be that for each slot which is not a perfect match, the matching utility of the production will be adjusted by adding the similarity difference (a negative value) between the specification and the current buffer chunk's slot value multiplied by the value of the `:ppm` parameter. Thus, when the procedural partial matching is

enabled the utility used to determine whether production i should be selected among those that match would be:

$$U'_i(t) = U_i(t) + \epsilon + \sum_j ppm * similarity(s_j, v_j)$$

$U_i(t)$ is the production's current true utility value

ϵ is the noise which may be added to the utility

j is the set of slots for which production i had a partial match

s_j is the specification for the slot j in production i

v_j is the value in the slot j of the chunk in the buffer

Alternatively, one can use the `:ppm-hook` parameter to specify a function for computing a custom penalty for mismatched production tests. In that case the expression used in the summation above is overridden by a user specified function.

:ppm-hook

The procedural partial matching hook parameter allows one to specify a function that will compute the utility offset added to a production which does not match the current state exactly when procedural partial matching is enabled. The default value for the parameter is **nil** which results in the offset being computed as described for the `:ppm` parameter. However, if the parameter is set to a function then that function will be passed a production name and a list of mismatch lists, one for each mismatch which occurred while testing the named production. A mismatch list will be a 5 element list consisting of: a buffer name, a slot name, the specified slot value, the actual value in the slot of the chunk in the buffer, and the reported similarity between those two items. If the hook function returns a number that will be added to the production's utility, any other return value will result in the default calculation being added.

:rhst

The right hand side trace parameter controls whether the actions from the fired production are shown in the trace. If this parameter is set to **t** (which is the default) and the [:trace-detail parameter](#) is set to high then the production's actions are displayed. If this parameter is set to **nil** or the `:trace-detail` is medium or low, then such events are not shown. Note that this only controls the direct actions made by the procedural module. Whether any events generated by other modules as a result of those actions are displayed is controlled by those modules.

:style-warnings

The style warnings parameter controls whether the procedural module reports warnings about possible production issues which do not prevent the production from being created, like requests to a buffer without querying it in the conditions or multiple requests to the same buffer. Style warnings also indicate possible issues among the productions, like requests to a buffer which is not tested in the conditions of other productions or setting/modifying slots which are never tested in any production. If the parameter is **t** (the default value) then additional warnings may be displayed when a model is defined and reset (they will be output using the [model-warning command](#)). If it is set to **nil** then those additional warnings will not be displayed.

:use-tree

The use tree parameter controls whether a decision tree is created to use during production matching. If the parameter is **nil** (the default value) then each production is tested individually to determine if it matches. If `:use-tree` is set to **t**, then when the model is loaded a decision tree is created based on the conditions among the productions and that tree is consulted first in production matching to potentially reduce the set of productions that need to be tested further. This should result in a faster model run time, but does have an initial tree creation cost and requires additional memory to store the tree. That additional time and space used are typically insignificant relative to what it takes to run a model, but in some situations may become a factor.

:vpft

The variable production firing time parameter controls whether the time of a production's firing is constant or variable. If the parameter is **nil** (the default value) then each production takes its specified action time exactly each time it fires. If `:vpft` is **t**, then the [randomize-time command](#) is used to randomize the production's action time. Note that randomize-time depends on the setting of the [:randomize-time parameter](#) and if it is **nil** then there will be no randomization.

Production buffer

The procedural module has a buffer called production. It exists for the purpose of allowing the module to have its state tracked. It never has any chunks placed into it and practically speaking it does not accept any requests. There is no reason to use the production buffer other than for tracking the state of the procedural module (typically through the [buffer trace module](#)).

Activation spread parameter: :production-activation
Default value: 0.0

Queries

The production buffer only responds to the default queries.

‘State busy’ will be **t** when a production is firing (the time between the production-selected and production-fired events). It will be **nil** at all other times.

‘State free’ will be **nil** when a production is firing and **t** at all other times.

‘State error’ will always be **nil**.

Requests

{*slot value*}*

The production buffer will accept any request without a warning or error, but the request is completely ignored – no actions are performed regardless of the slots specified.

Commands

p/define-p/p*/define-p*

Syntax:

p production-definition -> [p-name | **nil**]
p-fct (production-definition) -> [p-name | **nil**]
define-p production-definition -> [p-name | **nil**]
define-p-fct (production-definition) -> [p-name | **nil**]
p* production-definition -> [p-name | **nil**]
p*-fct (production-definition) -> [p-name | **nil**]
define-p* production-definition -> [p-name | **nil**]
define-p*-fct (production-definition) -> [p-name | **nil**]

Arguments and Values:

production-definition ::= p-name { doc-string } condition* ==> action*
p-name ::= a symbol that serves as the name of the production for reference
doc-string ::= a string which can be used to document the production
condition ::= [buffer-test | query | eval | binding | multiple-value-binding]
action ::= [buffer-modification | request | buffer-clearing | modification-request | buffer-overwrite | eval | binding | multiple-value-binding | output | **!stop!**]
buffer-test ::= =buffer-name> {isa chunk-type} slot-test*


```

(p example-production "a production showing most syntactic elements"
=goal>
  isa goal-type
  state start
  test =test
  < value 4
  value =value
  - slot1 =test
  slot1 =last-loc
  buffer =check

?visual>
  state =check
  buffer empty

=visual-location>
  isa visual-location
  >= screen-x =value
  <= screen-x 100
  > screen-y =value
  < screen-y =max-value
  =value =current-value

=imaginal>

!eval! (check-value =value)
!bind! =max-value (+ =value 100)
!mv-bind! (=quotient =remainder) (floor =max-value)
==>
!safe-bind! =new-state (get-a-state)

=goal>
  =check =new-state
  value =quotient
  step continue

*visual-location>
  isa visual-location
  screen-x =max-value

+retrieval>
  :recently-retrieved nil
  < screen-x =max-value
  - value =current-value
  color blue

+visual-location> =last-loc

+imaginal>
  isa goal-type
  state done

@visual> =imaginal

!output! (Moving to state =new-state with max-value of =max-value)
!safe-eval! (record-results =check 75 =quotient =remainder)
!stop!
)

```

Description:

The p family of commands is used to create the productions for a model. The p* commands exist to provide some backward compatibility with older versions of ACT-R and should not be used since the corresponding p command has the same functionality now. The “define-” commands are provided as a convenience for those using Lisp editors which provide special operations for use with Lisp forms that start with “define-”, and do the same as the corresponding command without the “define-” on the front.

A production must be given a name and can be given an optional documentation string. Then it contains a set of conditions to be tested and a set of actions to be executed when the production is fired. If the specification of the production is syntactically correct, then that production is entered into the procedural memory of the current model, as maintained by the procedural module, and the production’s name is returned.

If the name given for a production is already used by a production in the procedural memory of the current model then a warning is printed, the old production is removed, and it is replaced with the newly defined production.

If there is an error in parsing the production then one or more warnings will be output indicating what was wrong, no new production is entered into the procedural memory, and **nil** is returned.

Within a production there are many possible components and each will be described in detail below.

Variables

Productions contain variables to allow for more general matching and actions. The variables are symbols which start with an “=” character e.g. =slot, =answer, =goal. The variables are only relevant within the context of a single production and essentially serve three purposes. The first is to compare two or more values. The second is to copy a value from a condition into an action or specific query. The last is a process referred to as dynamic pattern matching and it allows a variable to be used in the specification of a slot for a buffer test, modification, modification request, or request. A single variable may be used for any/all of those purposes within a production i.e. it could compare two slots to determine that they are the same, copy that value into a request action, and modify a slot using that variable in a different action.

With respect to the dynamic pattern matching there are some restrictions on how the variables can be used. First, there is no search performed in the matching and all variable slot names must be

“grounded” by being used as a slot value in an explicitly named slot condition. This means the dynamic pattern matching cannot be used to “find a slot which has a specific value” like this:

```
(p invalid
  =buffer>
    =slot 150
==>
  ...
)
```

Also, there is only one level of indirection allowed in the use of variablized slot names. Thus it is not possible to do something like this:

```
(p also-invalid
  =buffer>
    slot1 =val
    =val =val2
    =val2 300
==>
  ...
)
```

However, one may go one level deep across multiple buffers:

```
(p valid
  =buffer1>
    slot1 =s1
    =s2 =value
  =buffer2>
    slot2 =s2
    =s1 =value
==>
  ...
)
```

Constants

Any symbols used in the production for values which are not variables are assumed to be the names of chunks. If there is not a chunk with such a name at the time the production is created then a new chunk with no slots will automatically be created with that name. Other non-symbol values (strings, numbers, etc) are treated as the corresponding Lisp value.

Modifiers

When testing slot values in conditions and checking queries in a production there are several modifiers which can be used: =, -, <, >, <=, and >=. The = modifier is used to check that the value in the buffer chunk’s slot and the value given in the production are equal as determined by the [chunk-slot-equal command](#). If no modifier is provided, then the = modifier is assumed (one usually never

sees the = modifier in a production). The – modifier means to negate the test. In a buffer test that means that the buffer chunk’s slot does not equal the value specified in the production and for a query it means that the match should succeed if the specified query returns false. The inequality tests (<, >, <=, and =>) can only be used when the values are numbers (the test fails if either of the elements being tested is not a number). If the values are numbers then the test is true if the inequality holds between the value in the specified slot of the chunk in the buffer and the value specified in the production in that order i.e. if the test is < slot1 10 then the condition matches if the value in the slot1 chunk of the buffer is less than 10.

isa

In many of the conditions and actions of a production it is possible to specify an optional chunk-type using the symbol *isa*. That chunk-type specification is only used during the definition of the production and does not directly affect the condition or action in which it occurs. It is effectively a declaration for the slots to be used, and unless the chunk-type has default slot-value pairs the same production would result without the chunk-type being specified.

When a chunk-type is specified in that way it allows the production to perform additional syntax checking to determine if the slots specified for that condition or action conform to the slots available for the chunk-type indicated. It also adds any default slot-value pairs specified for that chunk-type into the corresponding condition or action if such a slot is not specified explicitly within the condition or action. If a slot which is not valid for a specified chunk-type is used in a condition or action that will result in a warning being output, but it does not prevent the production from being specified.

Conditions

The conditions of the production are also referred to as the production’s left hand side (LHS). They are a conjunction of tests which must all be true for the production to be selected. The order in which the conditions are specified does not matter – there are no ordering constraints and the order in which the tests are performed will not necessarily be the same as they are specified in the production. The only ordering which will be maintained is that eval and binding conditions will be performed in the order specified if possible, and if they have to be reordered a warning will be displayed when the production is defined.

Here is the general description of the conditions that can be tested in a production. When a production is selected during conflict resolution it will generate an event to indicate each buffer

match and buffer query condition that it contains and are shown in the trace under the conditions indicated with the [:lhst parameter](#).

buffer test

The buffer-test is the primary condition used in productions. It is comparing the chunk currently in a specific buffer to a pattern provided in the production. Each buffer may have one test specified per production. The slot values in the buffer's chunk are compared to the values specified in the condition using the [chunk-slot-equal command](#), and all of the slots must satisfy the conditions specified for the production to match (except when the [:ppm parameter](#) is set which allows imperfect matches). If a value of nil is specified for a slot in a buffer test that indicates a test for the absence of the slot – a slot cannot actually have a value of nil.

In a production which is selected, a buffer test is also referred to as “harvesting” the chunk in the buffer. Here is an example of a buffer-test:

```
=goal>
  slot1 =value
  state  start
- slot2 =value
```

Every buffer test starts with a variable that names the buffer followed by the ‘>’ character. Thus, this is testing the chunk in the model's goal buffer. That variable will be bound to the chunk currently in the buffer and can be used like any other variable in the production. This condition requires that the chunk have slots named slot1 and state. The state slot must have the chunk start as its value and the slot1 slot can hold any value which will be bound to the =value variable. The other constraint here is that if the chunk has a slot named slot2 it must not have the same value as the slot1 slot.

In the trace a buffer test will show up as a buffer-read-action event like this:

```
0.000   PROCEDURAL           BUFFER-READ-ACTION GOAL
```

indicating which buffer was tested by the production.

query

A query is one or more tests of a buffer and/or its owning module. There are several default queries which may be made for every buffer and a module may provide many more queries to which it will respond. Each query will produce either a true or false result. The production will only match if the result of each query is true, or if the result is false and the negative test modifier, '-', is used. The queries that can be specified for every buffer are:

```
buffer empty
buffer full
buffer failure
buffer requested
buffer unrequested
state free
state busy
state error
error t
error nil
```

The first five are tests of the buffer itself and the module is not contacted to determine the status. The rest, and any others which a module provides, are tests which are relayed to the module to get its response. For the first five, the semantics of the queries are the same for all buffers and are as follows:

- buffer full: is true if there is currently a chunk in the buffer
- buffer failure: is true if the failure flag has been set for the buffer and it does not hold a chunk
- buffer empty: is true if there is not a chunk currently in the buffer and the failure flag is clear
- buffer requested: is true only if there currently is a chunk in the buffer and the module has indicated that it was put there as a result of a request to the module
- buffer unrequested: is true only if there currently is a chunk in the buffer and it has been marked as not having been put there as a result of a request to the module

The other queries are dependent on the how the module responds to them and thus one needs to check the particular module description to determine how they are used. Generally, they have the following semantics, but some modules may not follow this convention:

- state free: is true if the module is ready for new requests
- state busy: is true if the module is currently handling a request
- state error: is true if the last request resulted in some sort of error
- error t: this is the same as a test of state error (which is the query which will actually be sent to the module) and is provided as a shorthand notation for production syntax

- error nil: this is the same as a query for “– state error” thus checking that the module is not currently reporting that “state error” is true and again is a shorthand notation in the production syntax

Here is an example query to the goal buffer:

```
?goal>  
  state free  
  buffer full  
- state error
```

A query starts with a symbol composed of a ‘?’, the name of the buffer being queried, and the symbol ‘>’. This query is testing that the goal module is currently reporting that its state is free, there is currently a chunk in the goal buffer, and that the goal module is not currently reporting an error.

In the trace each buffer queried by the selected production will show up as a query-buffer-action:

```
0.450 PROCEDURAL QUERY-BUFFER-ACTION GOAL
```

eval

The !eval! condition is provided to allow the modeler to add any arbitrary conditions to the LHS of a production or to perform some side effects (like data collection or model tracking information). In the testing of the production’s conditions the form provided to evaluate will be called during conflict resolution. If the result of the evaluation is **nil**, then the production cannot be selected, but any other return value will allow the production to continue with the pattern matching of the LHS. Using !eval! is something that should be considered carefully when modeling. Generally, they should be used for abstracting away components of the model or task which are unnecessary for the current modeling or for performing non-model related actions.

Here is an example of a call to !eval!:

```
!eval! (special-test =var1 =var2 start)
```

This would pass three parameters to the function called special-test. Those parameters will be the current bindings for the =var1 and =var2 variables in the production and the symbol start. If that function returns a non-**nil** value then this production can continue in conflict resolution, but if it returns **nil** then it will be removed from the current conflict set.

There are actually two forms of the eval condition `!eval!` and `!safe-eval!`. Both do the same thing, and the difference is only meaningful for the [production compilation mechanism](#).

One note about using `!eval!` in conditions is that their evaluation may or may not occur during every conflict-resolution event. They will only be evaluated as needed to determine if a production matches. Thus not every `!eval!` within a production may be evaluated during a given conflict-resolution event and possibly none of the `!eval!` actions in any of the productions may be evaluated during a conflict-resolution event. Of the `!eval!` actions which are evaluated the only guarantee on the order of their evaluation is that those provided within a production will be evaluated in the order specified, but there is no guarantee on the order in which the `!eval!` actions from different productions will be evaluated. Because of the uncertainty in when and whether or not `!eval!` conditions will be evaluated it is recommended that the functions called have no side effects.

binding

The `!bind!` condition is very similar to the `!eval!` condition. However, with `!bind!` the return value of the evaluation is saved in a variable of the production which can then be used just as any other variable in the production. As with `!eval!` the return value must be non-**nil** for the production to successfully match. Here is an example of a `!bind!`:

```
!bind! =test-value (convert-value =var)
```

This will pass the current binding of the `=var` variable to the function `convert-value` and then bind the result to the `=test-value` variable in the production if it is non-**nil**.

Just like `!eval!`, there is also a second binding condition `!safe-bind!` which operates exactly like the `!bind!` condition for conflict resolution purposes, but has a difference with respect to how [production compilation](#) occurs.

It is also possible to bind multiple return values in a single test with the `!mv-bind!` condition. This works the same as `!bind!` except a list of variables is specified and each is bound to the corresponding return value from the evaluation. If there are fewer return values than variables to be bound the production will not match, and if any of the return values which are to be bound to a variable is **nil** then the production will also not match. Here is an example of `!mv-bind!`:

```
!mv-bind! (=value1 =value2) (split-values =var)
```

The same constraints on ordering and evaluation as described for !eval! conditions applies to !bind! conditions as well and again the recommendation is to not have any side effects in the functions used.

Actions

When a production fires it executes all of its actions, which are also referred to as its right hand side (RHS). Those actions are executed in a specific order regardless of how they are specified within the production definition. Except for the eval, output, and bind actions, the actions are processed through individual events scheduled to occur at the same time as the production's firing. The ordering of those events is determined using the priority of the scheduling of the events. Whether those events will be shown in the trace is determined by the [:rhst parameter](#). Those actions may result in other events being scheduled by the modules or otherwise with output not controlled by :rhst.

The ordering of the actions, along with their specific priorities, is as follows:

1. All eval, output, and bind calls occur during the production-fired event
2. The production-fired event schedules all of the other events
3. buffer modification actions [priority 100]
4. buffer overwrite actions [priority 90]
5. modification requests [priority 60]
6. module requests [priority 50]
7. buffer clearings [priority 10]
8. a !stop! action generates a break event [priority :min]

In general the production actions fall into three categories: cognitive actions performed directly by the production (those which begin with an =, @, or -), actions that are passed off to a module to handle (those that begin with a + or *), and debugging/modeler extension actions (those that begin with an !). The different actions possible are described in the following sections.

buffer modification

A buffer modification action is used to change the slot values of a chunk in a buffer. This is done directly by the production and works the same for every buffer (the buffer's module is not involved in the process). It is essentially the same as using the [mod-chunk command](#) on the chunk in the buffer. Here is an example of a buffer modification:

```
=goal>  
  state next-step
```

```
slot1 =value
```

The buffer modification must first name the buffer to be modified by using a symbol composed of the '=' character, the name of the buffer and the '>' character. That is followed by pairs of slot names and values. Thus, this example will change the state slot of the chunk in the goal buffer to now contain the chunk next-step and the slot1 slot will now hold whatever the variable =value is bound to in the production.

Alternatively, one can also specify what is referred to as an indirect modification. Instead of specifying the slots and values to modify, a chunk can be specified instead (either as a constant or through a variable). Specifying a chunk is equivalent to specifying all the slots and values of that chunk in the modification. Thus, if one has a chunk defined like this:

```
(define-chunks (test-chunk slot1 10 state finish))
```

then this modification action:

```
=goal> test-chunk
```

will do the same thing as this one:

```
=goal>
  slot1 10
  state finish
```

In order to perform a buffer modification action that buffer must have also been tested with a buffer-test in the conditions of the production to guarantee that it contains a chunk to be modified.

The buffer modification actions will show up in the trace as mod-buffer-chunk events indicating the buffer that is modified:

```
0.050    PROCEDURAL          MOD-BUFFER-CHUNK GOAL
```

buffer clearing

A buffer clearing action is used to remove a chunk from a buffer. As with buffer modification actions, this is done directly by the production without consulting the buffer's module. However, modules may be monitoring for buffer clearing events and perform some action whenever a buffer is cleared (their own buffer or any other). Once the action completes the buffer will be empty. Here is an example that would clear the goal buffer:

-goal>

The action shows up in the trace as a clear-buffer event indicating which buffer was cleared:

```
0.150    PROCEDURAL          CLEAR-BUFFER GOAL
```

A buffer can be cleared regardless of whether or not it was tested on the production's LHS. Note that often one does not need to explicitly clear a buffer because there are two situations which result in buffers being cleared implicitly as described below in [Implicit Production Actions](#).

buffer overwrite

A buffer overwrite action is used to replace the chunk in a buffer with a different chunk. As with the buffer modification and buffer clearing actions, this is performed directly by the production without notifying the buffer's module. This action does not clear the buffer first. Thus, the chunk which was in the buffer is essentially lost when this action occurs. Here is an example of a buffer overwrite action:

```
@goal> =value
```

This will replace the chunk in the goal buffer with the chunk that is bound to the =value variable (a constant chunk name could also be used in place of the variable). It will show up in the trace as an overwrite-buffer-chunk event indicating the buffer and the chunk being copied into it:

```
0.600    PROCEDURAL          OVERWRITE-BUFFER-CHUNK GOAL CHUNK-10
```

If the item provided is not a valid chunk name then a warning will be printed at run time and no change will be made to the buffer:

```
#|warning: overwrite-buffer-chunk called with an invalid chunk name BAD-NAME |#
```

module request

A module request (usually just referred to as a request) is how the production asks a module to perform some action. It is indicated by specifying a buffer name with a "+" on the front and a ">" on the end. Syntactically, a request can be specified using any slots and values for any buffer. Semantically, what a request actually does is specific to the module. Some modules may not even

accept requests through their buffers, or may have a more restrictive syntax than the general syntax available in the production (for example only allowing a slot to be specified once in the request). Thus, to know what can be requested, how it needs to be specified, and how that will then be processed one needs to know the details of the modules. A syntactically correct production may make semantically invalid requests which would typically generate warnings at run time from the module receiving the request.

In the production a module request can be specified in the same ways as a modification action. It may directly include all of the details of the request (which may include modifiers unlike the modification actions) or create the request indirectly by specifying a chunk. As far as the module which receives the request is concerned, there is no difference between the different specifications in the production i.e. the module does not know whether explicit slot and value specifications were provided or a single chunk which resulted in the slot value pairs of the request.

In addition to specifying slots in the request one may also include additional labels called request parameters. A request parameter is used like a slot in the request, but it is not actually the name of a slot. It is a Lisp keyword which has been created by a particular module to allow additional information to be provided in a request without having to use slots in chunks.

Here is an example of specifying the details of a request:

```
+retrieval>
  slot1 =value
- slot1 10
  slot2 start
  <= count =count
  :recently-retrieved nil
```

That request would be sent off to the retrieval buffer's module for handling, and what happens based on that request depends on the module which gets the request.

Here is an example of an indirect request:

```
+retrieval> =value
```

If =value is bound to a chunk name in the instantiation of the production being fired, then that chunk is essentially expanded to its slot value pairs to make the request. Thus, if =value were bound to the chunk A and the chunk A were defined like this:

```
(define-chunks (a state start slot2 10))
```

Then that would be equivalent to specifying this in the production:

```
+retrieval>
  state start
  slot2 10
```

It is also possible to include request parameters in an indirect request by specifying a list which has the chunk as the first element and the request parameters and values as the remaining items:

```
+retrieval> (=value :recently-retrieved t)
```

If a variable used in an indirect request is not bound to the name of a chunk in the production's instantiation, then warnings are printed and no request is made (note that the [implicit clearing](#) described below is still performed even if such a warning is encountered). The warnings will look like this in the trace:

```
#|Warning: define-chunk-spec's 1 parameter doesn't name a chunk: (not-a-chunk) |#
#|Warning: schedule-module-request called with an invalid chunk-spec NIL |#
```

A valid request action will show up in the trace as a module-request event specifying the buffer to which the request was sent:

```
0.800 PROCEDURAL MODULE-REQUEST GOAL
```

Typically, that will be followed by events from the module to which the request was made performing the requested action.

modification request

A modification request is similar to both the module request and buffer modification actions, and it is another way for a production to ask a module to perform some action. The differences between a modification request and a module request are that a modification request does not include an implicit clearing of the buffer and it is restricted to the same specification style as a buffer modification (no slot modifiers are allowed, no request parameters are accepted, and the buffer to which this request is made must have been used in a buffer test on the production's LHS). As with a request, what the module does in response to a modification request is entirely up to the module.

To create a modification request the buffer name is preceded by the "*" character. Here is an example of a modification request:

```
*imaginal>  
  slot1 =value  
  slot2 start
```

This would send those slot and value pairs off to the imaginal buffer's module to process. It is also possible to make an indirect modification request just like an indirect buffer modification:

```
*imaginal> =chunk
```

A modification request will show up in the trace as a module-mod-request event specifying the buffer to which the request is made:

```
1.000  PROCEDURAL          MODULE-MOD-REQUEST IMAGINAL
```

If a module does not accept modification requests then a warning like this will be displayed when the production is fired:

```
#!/warning: Module XXXXX does not support buffer modification requests. |#
```

eval

The eval action works just like the eval condition except that the return value does not matter.

binding

The binding action works just like the binding condition except that a returned value of **nil** can be bound to the variable or variables in the action.

output

The !output! action allows one to embed additional text in the model's trace. The output will be shown when the [:v parameter](#) is non-**nil** under any of the [:trace-detail](#) setting.

There are three ways to use the output action. It can be used to print a single value like this:

```
!output! =value  
!output! started
```

In that case the item specified will be output followed by a newline in the trace. If the item is a variable then it is the binding of the variable which is output.

It can also output several items if they are placed into a list:

```
!output! (the value is =value)
```

In that case all of the items will be printed on one line followed by a newline. Again, variables will be replaced with their current bindings before outputting.

Finally, it can use the Lisp format control mechanisms to output the text. If the first item given in a list to an !output! action is a string then that string is assumed to be a format specification. It is used to generate the output text using the remaining arguments in the same way that the Lisp format command would:

```
!output! ("The count is -6,3f.-%The value is ~a~%" =count =value)
```

stop

The stop action is used to force the model to stop after firing that production. A stop action is created with this:

```
!stop!
```

A break event will be generated by the stop action that will cause the current run to terminate.

Implicit production clearing actions

In addition to the events specified in the production there are two situations where a buffer clearing action will be implicitly executed when the production fires. They are referred to as strict harvesting and implicit clearing.

strict harvesting

Strict harvesting means that when a production tests a buffer on its LHS (harvests the chunk) it will automatically clear that buffer in the actions of the production. That will occur unless either: a modification is performed on that buffer (either a buffer modification with an = or a modification request with a *) or the buffer has been specified as one which should not be strict harvested using the [:do-not-harvest parameter](#).

request clearing

For each buffer which has a module request on the RHS of a production there is an implicit buffer clearing action performed on that buffer. There is no mechanism provided for suppressing this clearing action.

Examples:

For examples of productions in actual models see the example models in the ACT-R tutorial. These examples will show some of the warnings which may result when trying to define productions. Only those examples which result in no production being defined are indicated as errors.

For these examples the following chunk-type is assumed to have been defined:

```
(chunk-type goal-type slot slot2 state)
```

and the [:style-warnings parameter](#) has been turned off to avoid the warnings which may result because of the productions being defined individually.

```
1> (p test
    "Automatically creating a chunk which is referenced"
    =goal>
      isa goal-type
      state start
    ==>
  )
#|Warning: Creating chunk START with no slots |#
TEST

2> (p test
    "Redefining the production test"
    =goal>
      isa goal-type
      state start
    ==>
  )
#|Warning: Production TEST already exists and it is being redefined. |#
TEST

E> (p test
    "No current model"
    =goal>
      isa goal-type
      state start
    ==>
  )
#|Warning: get-module called with no current model. |#
#|Warning: No procedural modulue found cannot create production. |#
NIL

> (p test2
    "Slot name doesn't match chunk-type specified"
    =goal>
      isa goal-type
      bad-slot t
    ==>
  )
```

```

)
#|Warning: Production TEST2 uses previously undefined slots (BAD-SLOT).|#
TEST2

E> (p test2
  "Invalid buffer name in condition"
  =buffer>
  isa goal-type
  state start
  ==>
)

#|Warning: No production defined for (TEST2 "Invalid buffer name in condition" =BUFFER>
ISA GOAL-TYPE STATE START ==>).|#
#|Warning: First item on LHS is not a valid command|#
#|Warning: --- end of warnings for undefined production TEST2 ---|#
NIL

E> (p test2
  "Invalid buffer in query after a valid buffer test"
  =goal>
  isa goal-type
  ?buffer>
  buffer empty
  ==>
)

#|Warning: No production defined for (TEST2 "Invalid buffer in query after a valid buffer
test" =GOAL> ISA GOAL-TYPE ?BUFFER> BUFFER EMPTY ==>).|#
#|Warning: Invalid syntax in (=GOAL> ISA GOAL-TYPE ?BUFFER> BUFFER EMPTY) condition.|#
#|Warning: Invalid slot-name ?BUFFER> in call to define-chunk-spec.|#
#|Warning: --- end of warnings for undefined production TEST2 ---|#
NIL

E> (p test2
  "Buffer not tested on LHS for modification action"
  ==>
  =goal>
  state start
)

#|Warning: No production defined for (TEST2 "Buffer not tested on LHS for modification
action" ==> =GOAL> STATE START).|#
#|Warning: Buffer modification action for untested buffer (=GOAL> STATE START).|#
#|Warning: --- end of warnings for undefined production TEST2 ---|#
NIL

```

all-productions

Syntax:

all-productions -> (production-name*)

Arguments and Values:

production-name ::= a symbol which names a production

Description:

The `all-productions` command takes no parameters. It returns a list of the names of all the productions defined in the current model of the current meta-process. If there is no current model it prints a warning and returns **nil**.

Examples:

This example assumes the `count` model from unit 1 of the tutorial has been loaded.

```
> (all-productions)
(START INCREMENT STOP)
```

```
E> (all-productions)
#|Warning: get-module called with no current model.|#
NIL
```

```
E> (all-productions)
#|Warning: get-module called with no current meta-process.|#
NIL
```

pp

Syntax:

```
pp production-name* -> (production-name*)
pp-fct (production-name*) -> (production-name*)
```

Arguments and Values:

production-name ::= a symbol which names a production

Description:

The `pp` command is used to print the procedural module's representation of a production. It takes any number of production names and for each one prints out the representation of the named production in the current model of the current meta-process. If no names are provided it prints out all of the productions in the current model (which can be useful when production compilation is enabled to see what productions the model has learned). It returns a list of the names of the productions that were printed.

Note that the representation printed for a production may not exactly match the text which was used to define the production. In particular, any `chunk-types` specified in the original definition will not be shown in the printed representation since the `chunk-type` information is only used for the definition and does not represent an actual component of the production.

If there is no current model or current meta-process a warning is printed and **nil** is returned. If an invalid production name is given a warning is printed.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
> (pp)
(P START
  =GOAL>
    START =NUM1
    COUNT NIL
  ==>
  =GOAL>
    COUNT =NUM1
  +RETRIEVAL>
    FIRST =NUM1
)
(P INCREMENT
  =GOAL>
    COUNT =NUM1
  - END =NUM1
  =RETRIEVAL>
    FIRST =NUM1
    SECOND =NUM2
  ==>
  =GOAL>
    COUNT =NUM2
  +RETRIEVAL>
    FIRST =NUM2
  !OUTPUT! (=NUM1)
)
(P STOP
  =GOAL>
    COUNT =NUM
    END =NUM
  ==>
  -GOAL>
  !OUTPUT! (=NUM)
)
(START INCREMENT STOP)

E> (pp-fct (list 'not-production 'start))
#|warning: No production named NOT-PRODUCTION is defined |#
(P START
  =GOAL>
    START =NUM1
    COUNT NIL
  ==>
  =GOAL>
    COUNT =NUM1
  +RETRIEVAL>
    FIRST =NUM1
)
(START)

E> (pp)
#|warning: get-module called with no current model. |#
#|warning: No procedural module found |#
NIL

E> (pp)
#|warning: get-module called with no current meta-process. |#
```

```
#|Warning: No procedural module found |#
NIL
```

pbreak/punbreak

Syntax:

```
pbreak production-name* -> (break-production*)
pbreak-fct (production-name*) -> (break-production*)
punbreak production-name* -> (break-production*)
punbreak-fct (production-name*) -> (break-production*)
```

Arguments and Values:

production-name ::= a symbol that names a production

break-production ::= a symbol that names a production which is currently set to generate a break event

Description:

The `pbreak` command can be used to force a break event to occur when particular productions are selected during conflict resolution. Each production which is specified in a call to `pbreak` will force a break event if it is selected. Before the break, the current instantiation of the production will be output to the trace.

The `punbreak` command is used to remove the productions from the break condition. Each production passed to `punbreak` will no longer force a break event upon its selection. If no productions are specified for `punbreak` then all productions have their break status cleared.

Both commands return a list of all productions which currently have a break status set in the current model of the current meta-process.

If there is no current model or current meta-process or a production name provided is invalid a warning is printed.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
1> (pbreak)
NIL
```

```
2> (pbreak start)
(START)
```

```
3> (run 10)
    0.000    GOAL                SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
```

```

0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
(P START
=GOAL>
  START 2
  COUNT NIL
==>
=GOAL>
  COUNT 2
+RETRIEVAL>
  FIRST 2
)
0.000 ----- BREAK-EVENT PRODUCTION START

4> (pbreak-fct '(start increment))
(INCREMENT START)

5> (punbreak start)
(INCREMENT)

6> (pbreak start stop)
(STOP INCREMENT START)

7> (punbreak-fct '(increment stop))
(START)

8E> (pbreak bad-name)
#|Warning: BAD-NAME is not the name of a production |#
(START)

9E> (punbreak-fct '(not-a-production))
#|Warning: NOT-A-PRODUCTION is not the name of a production |#
(START)

10> (punbreak)
NIL

11> (pbreak)
NIL

E> (pbreak)
#|Warning: There is no current model - pbreak cannot be used. |#
NIL

E> (punbreak start)
#|Warning: There is no current model - punbreak cannot be used. |#
NIL

```

pdisable/penable

Syntax:

```

pdisable production-name* -> (disabled-production*)
pdisable-fct (production-name*) -> (disabled-production*)
penable production-name* -> (disabled-production*)
penable-fct (production-name*) -> (disabled-production*)

```

Arguments and Values:

production-name ::= a symbol that names a production

disabled-production ::= a symbol that names a production which is currently disabled

Description:

The `pdisable` command can be used to disable productions. A production which is disabled will not participate in conflict resolution. Each production which is specified in a call to `pdisable` will be disabled.

The `penable` command is used to enable productions which have been disabled. Each production passed to `penable` will no longer be disabled. If no productions are specified for `penable` then all productions will be enabled.

Both commands return a list of all productions which currently have been disabled in the current model of the current meta-process.

If there is no current model or meta-process or a production name is invalid a warning is printed.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
1> (pdisable)
NIL

2> (pdisable start stop)
(STOP START)

3> (penable-fct '(start))
(STOP)

4> (pdisable-fct '(start increment))
(STOP INCREMENT START)

5> (penable)
NIL

E> (penable)
#|Warning: There is no current model - penable cannot be used.|#
NIL

E> (pdisable bad-name)
#|Warning: BAD-NAME is not the name of a production|#
NIL

E> (penable-fct '(not-a-production))
#|Warning: NOT-A-PRODUCTION is not the name of a production|#
NIL
```

whynot

Syntax:

```
whynot production-name* -> (matching-production*)
whynot-fct (production-name*) -> (matching-production*)
```

Arguments and Values:

production-name ::= a symbol that names a production

matching-production ::= a symbol that names a production which matches at the current time

Description:

The `whynot` command is a useful model debugging tool. For each of the named productions passed to it (or all productions if no names are provided) it will print out whether the production matches the current state or not. If it does match, then the instantiation of the production is printed and if it does not match then the production is printed and the first condition that is unsatisfied at the current time is provided.

If the `:ppm parameter` is enabled to allow for imperfect matching then the instantiation of a production which is a partial match will indicate the current slot value, the imperfectly matching buffer slot value and the similarity between those values.

It returns a list of all the productions which do match the current state in the current model of the current meta-process (regardless of whether they were passed into `whynot` for display).

If there is no current model or meta-process then a warning is printed and `nil` is returned. If an invalid production-name is provided a warning will be displayed.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
1> (reset)
DEFAULT

2> (whynot)

Production START does NOT match.
(P START
  =GOAL>
    START =NUM1
    COUNT NIL
  ==>
  =GOAL>
    COUNT =NUM1
  +RETRIEVAL>
    FIRST =NUM1
)
It fails because:
The GOAL buffer is empty.

Production INCREMENT does NOT match.
(P INCREMENT
  =GOAL>
```

```

    COUNT =NUM1
  - END =NUM1
  =RETRIEVAL>
    FIRST =NUM1
    SECOND =NUM2
==>
  =GOAL>
    COUNT =NUM2
  +RETRIEVAL>
    FIRST =NUM2
  !OUTPUT! (=NUM1)
)

```

It fails because:
The GOAL buffer is empty.

Production STOP does NOT match.

```

(P STOP
  =GOAL>
    COUNT =NUM
    END =NUM
==>
  -GOAL>
  !OUTPUT! (=NUM)
)

```

It fails because:
The GOAL buffer is empty.
NIL

3> (run-n-events 2)

```

  0.000 GOAL
  0.000 -----

```

SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
Stopped because event limit reached

0.0

2

NIL

4> (whynot-fct '(increment))

```

(P INCREMENT
  =GOAL>
    COUNT =NUM1
  - END =NUM1
  =RETRIEVAL>
    FIRST =NUM1
    SECOND =NUM2
==>
  =GOAL>
    COUNT =NUM2
  +RETRIEVAL>
    FIRST =NUM2
  !OUTPUT! (=NUM1)
)

```

It fails because:
The chunk in the GOAL buffer does not have slot COUNT.
(START)

5> (whynot start)

Production START matches:
Production START matches:

```

(P START
  =GOAL>
    START 2
    COUNT NIL
==>
  =GOAL>
    COUNT 2
  +RETRIEVAL>

```

```
        FIRST 2
    )
  (START)
```

This example uses a simple model definition to show a production which is a partial match with :ppm enabled:

```
1> (define-model test
    (sgp :esc t :ppm 1)
    (chunk-type test slot)
    (define-chunks (buffer-value) (production-test))
    (set-similarities (buffer-value production-test -.5))
    (set-buffer-chunk 'goal (car (define-chunks (slot buffer-value))))

    (p test
      =goal>
      isa test
      slot production-test
      ==>))
TEST
```

```
2> (whynot)
```

```
Production TEST partially matches the current state:
Production TEST partially matches the current state:
(P TEST
 =GOAL>
  SLOT [PRODUCTION-TEST, BUFFER-VALUE, -0.5]
 ==>
)
```

```
(TEST)

E> (whynot)
#|Warning: Whynot called with no current model. |#
NIL
```

```
E> (whynot bad-name)
```

```
BAD-NAME does not name a production.
(START)
```

production-firing-only

Syntax:

production-firing-only *event* -> production-firing-event?

Arguments and Values:

event ::= an ACT-R event

production-firing-event? ::= a generalized boolean that is true if *event* has an action of production-fired and is false otherwise

Description:

This is not a command which would be called by the modeler directly. It is provided as a possible value for the [:trace-filter parameter](#) to restrict the trace to only the production-fired events.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
> (with-parameters (:trace-filter production-firing-only)
  (run 10))
  0.050 PROCEDURAL PRODUCTION-FIRED START
  0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT
2
  0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT
3
  0.300 PROCEDURAL PRODUCTION-FIRED STOP
4
  0.300 ----- Stopped because no events left to process
0.3
50
NIL
```

un-delay-conflict-resolution

Syntax:

```
un-delay-conflict-resolution -> nil
```

Arguments and Values:

Description:

The `un-delay-conflict-resolution` command takes no parameters and will cause a new conflict-resolution event to be scheduled in the current model of the current meta-process if the procedural module is currently waiting for some change in the system to schedule the next conflict-resolution event. If there is no current model or current meta-process then this command has no effect and a warning is output.

This is not a command which one will typically need to use. Right now, it is only used by the procedural system modules (`procedural`, `utility`, and `production-compilation`) to ensure that conflict resolution gets rescheduled if necessary when parameters are changed or new productions are created. However, in rare circumstances other modules could put the system in such situations and thus may need to use this command (though typically the other module should just be able to schedule an event which would allow conflict resolution to occur normally).

Examples:

```
> (un-delay-conflict-resolution)
NIL

E> (un-delay-conflict-resolution)
#|Warning: get-module called with no current model.|#
NIL
```

```
E> (un-delay-conflict-resolution)
#|Warning: get-module called with no current meta-process. |#
NIL
```

clear-productions

Syntax:

clear-productions -> nil

Arguments and Values:

Description:

The `clear-productions` command will delete all of the productions from the current model of the current meta-process. It is not recommended, but there may be times where one finds doing so necessary. It will also print out a warning indicating that it is not recommended. If there is no current model or current meta-process then a warning is printed.

It always returns **nil**.

Examples:

```
> (clear-productions)
#|Warning: Clearing the productions is not recommended |#
NIL
```

```
E> (clear-productions)
#|Warning: get-module called with no current model. |#
#|Warning: No procedural module was found. |#
NIL
```

```
E> (clear-productions)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No procedural module was found. |#
NIL
```

declare-buffer-usage

Syntax:

```
declare-buffer-usage buffer type-name { [:all | slot*] } -> [ t | nil ]
declare-buffer-usage-fct buffer type-name { [:all | (slot*) ] } -> [ t | nil ]
```

Arguments and Values:

buffer ::= a symbol that is the name of a buffer

type-name ::= a symbol that is the name of a chunk-type

slot ::= a symbol that names a slot of the chunk-type *type-name*

Description:

The `declare-buffer-usage` command is used to indicate a chunk-type and any slots from that chunk-type that are going to be used with a particular buffer but which are not created by the productions nor set initially in the model definition. By declaring slots used it will prevent style warnings from the production definitions for the slots provided. This will typically be needed when chunks are being placed into buffers from code. In that situation using this command in the model definition to indicate the items which are being set outside of the model definition will avoid possible style warnings.

If `buffer` names a valid buffer, `type-name` names a valid chunk-type, and all slots specified are valid for the chunk-type given then style warnings related to those items will be avoided and a value of `t` will be returned. If the keyword `:all` is provided instead of slot names then that will be equivalent to specifying all the slots of the chunk-type given.

If any of the parameters are invalid or there is no current model or current meta-process then a warning will be printed and `nil` will be returned.

Examples:

```
> (declare-buffer-usage goal chunk)
T

> (declare-buffer-usage imaginal text screen-pos value)
T

> (declare-buffer-usage-fct 'imaginal 'visual-object '(value color))
T

> (declare-buffer-usage retrieval text :all)
T

> (declare-buffer-usage-fct 'imaginal 'visual-object :all)
T

E> (declare-buffer-usage not-a-buffer visual-location)
#|Warning: Cannot declare usage for NOT-A-BUFFER because it does not name a buffer in the model.|#
NIL

E> (declare-buffer-usage-fct 'goal 'not-a-chunk-type)
#|Warning: Cannot declare usage for buffer GOAL because NOT-A-CHUNK-TYPE does not name a chunk-type in the model.|#
NIL

E> (declare-buffer-usage goal chunk not-a-slot)
#|Warning: Cannot declare usage for buffer GOAL because the slots (NOT-A-SLOT) are not valid for chunk-type CHUNK.|#
NIL

E> (declare-buffer-usage goal type)
#|Warning: get-module called with no current model.|#
#|Warning: No procedural module found. Cannot declare buffer usage.|#
NIL
```

```
E> (declare-buffer-usage goal type)
#|warning: get-module called with no current meta-process. |#
#|warning: No procedural module found.  Cannot declare buffer usage. |#
NIL
```

Utility module

The utility module provides the support for the productions' subsymbolic utility value which is used in conflict resolution. This is a numeric quantity associated with each production that can be learned while the model runs or specified in advance for each production.

Only an overview of how utility works is provided here. The details of the utility calculation are described in the additional documentation file named "new-utility" as well as tutorial units 6 and 7. The variable noise mechanism discussed in the new-utility document is not implemented at this time.

Each production has a utility value associated with it, which we will call U . Of the productions in the conflict set (those which match the current state) the one with the highest current U will be the one selected. When [utility learning is enabled](#), the U value is based on the rewards that a production receives and will change as the model runs. The learning of utilities is controlled by the following equation for a production i after its n th usage:

$$U_i(n) = U_i(n-1) + \alpha[R_i(n) - U_i(n-1)]$$

α is the learning rate set by the [:alpha parameter](#)

$R_i(n)$ is the effective reward value given to production i for its n th usage

$U_i(0)$ is determined based on initial parameter settings

The learning occurs when a reward is triggered, at which time productions which have been used since the last reward may have their utility values updated (see the [trigger-reward](#) command for details on which productions will be updated). The effective reward for a production i is the reward value received after its n th usage minus the time since that n th selection of production i .

When the [:esc parameter](#) is enabled the utility values may also have a noise component added to them (regardless of whether the learning mechanism is enabled). If `:esc` is enabled and the [:egs parameter](#) is set then each time a production's utility is calculated it will also have a noise component added to it. That noise is generated using the [act-r-noise command](#) with the `s` value being the current setting of the `:egs` parameter.

Parameters

:alpha

The α parameter in the utility learning equation. The default value is .2.

:egs

This is the expected gain s parameter. It specifies the s parameter for the noise added to the utility values. It defaults to 0 which means there is no noise in utilities.

:iu

The initial utility value for a user defined production. This is the $U(0)$ value for a production if utility learning is enabled and the default utility if learning is not enabled. The default value is 0.

:nu

This is the starting utility for a newly learned production (those created by the [production compilation](#) mechanism). This is the $U(0)$ value for such a production if utility learning is enabled and the default utility if learning is not enabled. The default value is 0.

:reward-hook

The reward-hook parameter allows the modeler to override the default calculation for effective reward, $R_i(n)$. It can be set to a function which must take three parameters. If the :reward-hook parameter is not **nil** (which is the default value) then each time a reward is propagated back to a production the reward-hook function will be called. It will be passed the name of the production as the first parameter, the reward value being propagated as the second, and the time since the production was selected (in seconds) as the third. If that function returns a number then that number is used as the $R_i(n)$ value in updating the production's utility instead of the normal calculation (which is the reward minus the time since the production's selection). If any other value is returned, then the standard calculation for $R_i(n)$ is used. Only one reward-hook function may be specified at a time and if the parameter is changed from one function to another a warning will be output.

:reward-notify-hook

This parameter allows one to specify functions which will be called whenever there is a reward provided to the model. This parameter can be set to a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). Whenever [trigger-reward](#) is called the functions set with this parameter will be called during the propagate-reward event that gets generated as a result of that trigger-reward. Each

of the functions that has been set for this parameter will be called with one parameter which is the reward value passed to trigger-reward. The return value from a function called through this hook is ignored. If the parameter is set to **nil** then all functions are removed from the reward-notify-hook list. A function should only be set once. If a function is specified more than once as a value for the reward-notify-hook a warning will be displayed and a value of **nil** will be returned as the current value.

:ul

This is the utility learning flag. If it is set to **t** or **complete** then the utility learning equation shown above will be used to learn the utilities as the model runs. If it is set to **nil** then the explicitly set utility values for the productions are used (though the noise will still be added if :egs is non-zero). The default value is **nil**.

:ult

This is the utility learning trace flag. If it is set to **t** then when a reward is received and utilities are updated the corresponding changes will be output in the model trace. If it is set to **nil** then there will be no additional trace output from utility updating. The default value is **nil**.

:ut

This is the utility threshold. If it is set to a number then that is the minimum utility value that a production must have to compete in conflict resolution. Productions with a lower utility value than that will not be selected. The default value is **nil** which means that there is no threshold value and all productions will be considered.

:utility-hook

The utility-hook parameter allows the modeler to override or bypass the default utility calculation. It can be set to a function which must take one parameter. If the :utility-hook parameter is not **nil** (which is the default value) then each time a production's utility is to be calculated, first this hook function is called with the name of the production as the parameter. If that function returns a number then that number is used as the production's utility instead of using the normal mechanisms. Only one utility-hook function may be specified at a time and if the parameter is changed from one function to another a warning will be output.

:utility-offsets

This parameter allows one to specify functions which can extend the utility equation with new terms. This parameter can be set to a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). Whenever a production's utility is computed each of the functions that has been set for this parameter will be called with one parameter which is the name of the production. If a function called returns a number then that value will be added to the utility of the production. If a function returns any other value then no change is made to the utility of the production. If the parameter is set to **nil** then all functions are removed from the utility-offsets list. A function should only be set once. If a function is specified more than once as a value for the utility-offsets a warning will be displayed and a value of **nil** will be returned as the current value.

Commands

trigger-reward

Syntax:

trigger-reward reward {maintenance} -> [**t** | **nil**]

Arguments and Values:

reward ::= [reward-value | **nil**]

reward-value ::= a number which indicates the amount of reward to apply

maintenance ::= a generalized boolean which indicates whether the propagate-reward event should be marked as a maintenance event or not

Description:

The trigger-reward command allows the modeler to present rewards to the current model in the current meta-process for the purposes of utility learning. If the reward specified is a number then that value is used in computing the updated utility for the productions which need to be updated. If the reward is **nil** then no utilities are updated but this still indicates when the last reward was given. This function can be called at any time to introduce a reward to the model – it does not need to be called synchronously with a production's firing. If there is a current model in the current meta-process and the reward value is valid, then a propagate-reward event will be scheduled at the current time to perform the new computations and **t** will be returned. The event will show the value of the reward being used like this:

If utility learning is not enabled, then that will be followed by a warning indicating that no change has occurred:

```
##Warning: Trigger-reward can only be used if utility learning is enabled. |#
```

That event will be marked as a maintenance event if a non-**nil** maintenance value is provided otherwise it will be a regular event. One may want to have the reward be a maintenance event if a model is being tested in both learning and non-learning to avoid the introduction of the rewards for the learning case adding additional changes relative to the non-learning situation.

The productions which will have their utilities updated depends upon the setting of the [:ul parameter](#). If it is set to **t** then all productions which have been selected since the last reward was provided will receive an update. If it is set to **complete** then only productions which have “completed” since the last reward will be updated. For a production to be completed it must have fired (as opposed to only being selected) and all the requests and modification requests which it has made must also have been marked as completed by the module which received the request. The details on when a particular module signals that its requests are completed are described in that module’s section, and information about how one would handle that when creating a new module can be found in the sections on [using buffers](#) and [creating new modules](#).

If there is no current model, no current meta-process, or the parameter provided is invalid then a warning is printed, no utilities are changed, and **nil** is returned.

Examples:

```
> (trigger-reward 10)
T
```

```
> (trigger-reward nil)
T
```

```
> (trigger-reward 3 t)
T
```

```
E> (trigger-reward "value")
##Warning: Trigger-reward must be called with a number or nil. |#
NIL
```

```
E> (trigger-reward 10)
##Warning: No current model. Trigger-reward has no effect. |#
NIL
```

spp

Syntax:

```
spp [ { [ production-name | (production-name*) } { [ param-name* | param-value-pair* ] } |  
      (production-name [ param-name* | param-value-pair* ])* ] -> (param-values*)  
spp-fct ( [ { [ production-name | (production-name*) } { [ param-name* | param-value-pair* ] } |  
           (production-name [ param-name* | param-value-pair* ])* ] ) -> (param-values*)
```

Arguments and Values:

production-name ::= a symbol which is the name of a production in the current model

param-name ::= a keyword which names a production parameter

param-value-pair ::= param-name new-param-value

new-param-value ::= a Lisp value to which the preceding param-name is to be set

param-values ::= [(param-value*) | (production-name*) | :error]

param-value ::= the current value of a requested production parameter or :error

Description:

Spp is used to set or get the value of the parameters of the productions in the current model of the current meta-process. It is similar to the [sgp command](#) which is used to set and get the module parameter.

Each production has five parameters associated with it. Two of them are read only, but the other three can be adjusted by the modeler. Those parameters are:

:at

The action time of the production. This is how much time passes (in seconds) between the production's selection and when it fires. This can be set explicitly for each production and defaults to the value of the [:dat parameter](#) at the time the production was created.

name

The name parameter returns the name of the production. This cannot be changed. Requesting this parameter is useful for annotating the results that are returned as shown in the examples.

:u

The u parameter returns the current U(n) value for the production. This may be set directly only when the [utility learning mechanism is not enabled](#). It defaults to the value of the [:iu parameter](#) at the

time the production is created unless it is created through [production compilation](#) in which case it defaults to the value of the [:nu parameter](#).

:utility

The last computed utility value of the production during conflict resolution (including any noise which was added). This cannot be changed by the modeler. If the production has not yet been a member of the conflict set, then the value will be **nil**.

:reward

This is a reward value to apply when this production fires. The default is **nil** which means the production does not provide a reward. If it is set to a number then after this production fires a [trigger-reward](#) call will be made using that reward value. If it is set to **t** then after the production fires a trigger-reward call will be made with a value of **nil** (a null reward which clears the history without adjusting any utilities).

If no parameters are provided to `spp`, then all of the current model's productions' parameters are printed and a list of all the production names is returned. For each production it prints the production's name followed by the parameters which are currently appropriate based on the settings of [:esc](#) and [:ul](#). If [:esc](#) is **nil**, then the [:u](#) and [:at](#) parameters are printed. If [:esc](#) is **t** and [:ul](#) is **nil** [:utility](#), [:u](#) and [:at](#) are printed, and if [:esc](#) is **t** and [:ul](#) is **t**, then [:utility](#), [:u](#), [:at](#), and [:reward](#) are printed.

If a production or list of productions is specified as the first parameter to `spp` then the following parameters are set or retrieved from only those productions. If no production names are provided then the settings are applied to or retrieved from all productions that exist at the time of the call to `spp`.

If production names are specified but no specific parameters are specified then the parameters for those productions are printed and the list of those production names is returned. If any of the production names provided are invalid a warning will be printed and the corresponding element of the return list will be **:error**.

If all of the parameters passed to `spp` (after any production names) are keywords, then it is a request for the current values of the parameters named. Those parameters are printed for the productions

specified and a list containing a list for each production specified is returned. Each sub-list contains the values of the parameters requested in the order requested and the sub-lists are in the order of the productions which were requested. If an invalid parameter is requested, then a warning is printed and the value returned in that position will be the keyword **:error**.

If there are any non-keyword parameters in the call to `spp` and the number of parameters (not counting the production names) is even, then they are assumed to be pairs of a parameter name and a parameter value. For all of the specified productions (or all productions if none are specified) those parameters will be set to the provided values. The return value will be a list containing a list for each production specified. Each sub-list contains the values of the parameters set in the order they were set and the sub-lists are in the order of the productions which were specified. If a particular parameter value was not of the appropriate type, then a warning is printed and the value returned in that position will be the keyword **:error**.

It is also possible to pass lists of production-name and parameter settings to `spp`. Essentially, each list provided must be formatted as something that could be passed to `spp` on its own and they will each be processed as appropriate.

If there is no current model or current meta-process at the time of the call, then a warning is displayed and **nil** is returned.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
1> (sgp :esc nil :ul nil)
(NIL NIL)

2> (spp)
Parameters for production START:
 :u  0.000
 :at 0.050
Parameters for production INCREMENT:
 :u  0.000
 :at 0.050
Parameters for production STOP:
 :u  0.000
 :at 0.050
(START INCREMENT STOP)

3> (sgp :esc t)
(T)

4> (spp)
Parameters for production START:
 :utility  NIL
 :u  0.000
 :at  0.050
```

```

Parameters for production INCREMENT:
:utility  NIL
:u  0.000
:at  0.050
Parameters for production STOP:
:utility  NIL
:u  0.000
:at  0.050
(START INCREMENT STOP)

5> (sgp :ul t)
(T)

6> (spp)
Parameters for production START:
:utility  NIL
:u  0.000
:at  0.050
:reward  NIL
Parameters for production INCREMENT:
:utility  NIL
:u  0.000
:at  0.050
:reward  NIL
Parameters for production STOP:
:utility  NIL
:u  0.000
:at  0.050
:reward  NIL
(START INCREMENT STOP)

7> (spp start)
Parameters for production START:
:utility  NIL
:u  0.000
:at  0.050
:reward  NIL
(START)

> (spp-fct '((increment stop) :name :u))
Parameters for production INCREMENT:
:NAME INCREMENT
:U  0.000
Parameters for production STOP:
:NAME STOP
:U  0.000
((INCREMENT 0) (STOP 0))

1> (spp :at 10)
((10) (10) (10))

2> (spp-fct nil)
Parameters for production START:
:utility  NIL
:u  0.000
:at  10.000
:reward  NIL
Parameters for production INCREMENT:
:utility  NIL
:u  0.000
:at  10.000
:reward  NIL
Parameters for production STOP:
:utility  NIL
:u  0.000
:at  10.000
:reward  NIL

```

```

(START INCREMENT STOP)

1> (sgp :esc t :egs 3)
(T 3)

2> (run 10)
...
0.3
48
NIL

4> (spp-fct '(:name :utility :u))
Parameters for production START:
:NAME START
:UTILITY -1.806
:U 0.000
Parameters for production INCREMENT:
:NAME INCREMENT
:UTILITY -1.594
:U 0.000
Parameters for production STOP:
:NAME STOP
:UTILITY -3.252
:U 0.000
((START -1.80585 0) (INCREMENT -1.5941277 0) (STOP -3.2521996 0))

5> (spp (start) (stop :name :utility) (increment :at 10))
Parameters for production START:
:utility -1.806
:u 0.000
:at 0.050
Parameters for production STOP:
:NAME STOP
:UTILITY -3.252
(START (STOP -3.2521996) (10))

E> (spp (start end))
Parameters for production START:
:utility NIL
:u 0.000
:at 10.000
:reward NIL
#|Warning: Spp cannot adjust parameters because production END does not exist|#
(START :ERROR)

E> (spp start :u :ve)
Parameters for production START:
:U 0.000
#|Warning: NO PARAMETER VE DEFINED FOR PRODUCTIONS.|#
:VE ERROR
((0 :ERROR))

E> (spp)
#|Warning: get-module called with no current model.|#
NIL

```

Production Compilation Module

The production compilation module implements the process of learning new productions. More details of the production compilation process can be found in the additional document named “compilation” and tutorial unit 7. Only the basic mechanisms of the module will be described here.

Production compilation works by combining two productions that fire in sequence into one new production. When enabled, it will attempt to create a new production for each pair of productions that fire. To determine if two productions can be combined into one production all of the buffers that are referenced in those productions (in any condition or action) are checked to see if they have a compatible usage between the two productions. If any buffer does not have a compatible usage between the two productions then the productions cannot be combined.

Compatible usage is determined by the “compilation type” of the buffer. The compilation type also controls how the usage of that buffer in the productions gets combined into the new production. The provided module specifies five different compilation types, and each of the buffers in the system is considered to be of one of those types. The five types are goal, imaginal, retrieval, perceptual, and motor. The details of what constitutes valid usage for those compilation types are described in the excel spreadsheet compilation.xls and the mechanism used to create the new production based on the compilation type is described in the compilation document. It is possible to change the compilation type of a buffer using the [specify-compilation-buffer-type command](#) and it is also possible to [add new compilation types to the system](#). For the buffers provided in the default system this is the assignment of buffer to compilation type:

Buffer Name	Compilation Type
goal	Goal
imaginal	Imaginal
retrieval	Retrieval
aural	Perceptual
aural-location	Perceptual
visual	Perceptual
visual-location	Perceptual
temporal	Perceptual
imaginal-action	Motor
production	Motor
manual	Motor
vocal	Motor

Any buffer added with a new module will be of the motor compilation type by default, but may be changed with the [specify-compilation-buffer-type command](#).

If all of the buffers have a compatible usage between two successive productions which fire then the following situations are checked. If any of these are true the productions cannot be combined:

- is the time between the productions greater than the threshold time?
- does either production have a !eval! condition or action?
- does either production have a !bind! or !safe-bind! condition?
- does either production have a !bind! action?
- does either production use !mv-bind! in either the conditions or actions?
- does either production have a buffer overwrite action?
- does either production use any indirect actions?
- does either production use slot modifiers other than = in its conditions?
- does the first production make multiple requests using the same buffer?
- does the first production have a RHS !stop! action?

If none of those situations are true, then the two productions are combined into one new production.

After creating the new production, it is compared against the other productions in the procedural memory of the model. If the new production is not semantically equivalent to an existing production then the new production is also added to the procedural memory of the model. The parameters for the new production are set as follows:

- The :utility is the value of the :nu parameter.
- The :at parameter is set to the max of the :at parameters from its two parent productions.
- If both of the parent productions have a numeric :reward the new production gets the max of those two as its :reward. If only one of the parents has a numeric :reward the new production gets that value as its :reward. If neither parent has a numeric :reward, but at least one has a value of **t** for the :reward then the new production will get a value of **t**. If both parents have a **nil** :reward value the new production also gets a **nil** :reward value.

If the newly formed production is semantically equivalent to an existing production then what happens depends on whether the production to which it is equivalent was also created by production compilation or whether it was one of the explicitly specified productions of the model and also whether utility learning is enabled.

If the production is equivalent to one of the explicitly specified productions or utility learning is not enabled, then nothing happens and the newly created production is ignored.

If the production is equivalent to one which was previously created by production compilation and utility learning is enabled then that production receives an update to its utility. The effective reward which it receives is the current :u value of the first of the two productions which fired to create it.

Parameters

:epl

The enable production learning parameter controls whether the production compilation process is enabled or disabled. If :epl is set to **t** then the process is enabled and if it is set to **nil** then it is disabled. The default value is **nil**.

:pct

The production compilation trace parameter controls whether information about the production compilation process is output to the trace. If it is set to **t** then after each production fires a notice about the production compilation process will be displayed. If a new production was created then that production will be printed along with its parameter values. If a new production was not created then information indicating why not will be displayed. If the parameter is set to **nil** then no output will be given for production compilation. The default value is **nil**.

:tt

The threshold time parameter specifies the maximum amount of time in seconds that is allowed to pass between the firing of two productions and still allow them to be combined through production compilation. The default value is 2 seconds.

Commands

show-compilation-buffer-types

Syntax:

show-compilation-buffer-types -> nil

Description:

The `show-compilation-buffer-types` command can be used to print out the current assignments of all the buffers' compilation types in the current model of the current meta-process. If there is no current model or current meta-process a warning is printed.

Examples:

```
> (show-compilation-buffer-types)
  Buffer          Type
VISUAL          PERCEPTUAL
TEMPORAL        PERCEPTUAL
AURAL-LOCATION    PERCEPTUAL
VISUAL-LOCATION   PERCEPTUAL
PRODUCTION      MOTOR
AURAL           PERCEPTUAL
VOCAL           MOTOR
IMAGINAL-ACTION MOTOR
MANUAL          MOTOR
GOAL            GOAL
IMAGINAL        IMAGINAL
RETRIEVAL       RETRIEVALNIL
```

```
E> (show-compilation-buffer-types)
#|warning: get-module called with no current model. |#
#|warning: No production compilation module found |#
NIL
```

```
E> (show-compilation-buffer-types )
#|warning: get-module called with no current meta-process. |#
#|warning: No production compilation module found |#
NIL
```

compilation-buffer-type

Syntax:

```
compilation-buffer-type buffer-name -> [buffer-type | nil ]
compilation-buffer-type-fct buffer-name -> [buffer-type | nil ]
```

Arguments and Values:

buffer-name ::= should be a symbol which names a buffer
buffer-type ::= the compilation type of *buffer-name*

Description:

`compilation-buffer-type` will return the compilation type for a buffer in the current model of the current meta-process. If there is no current model, no current meta-process, or the buffer name is invalid then **nil** is returned.

Examples:

```
> (compilation-buffer-type-fct 'manual)
MOTOR

> (compilation-buffer-type goal)
GOAL

> (compilation-buffer-type not-a-buffer)
NIL

E> (compilation-buffer-type goal)
#|Warning: get-module called with no current model. |#
#|Warning: No production compilation module found |#
NIL

E> (compilation-buffer-type manual)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No production compilation module found |#
NIL
```

specify-compilation-buffer-type

Syntax:

```
specify-compilation-buffer-type buffer-name buffer-type -> [ t | nil ]
specify-compilation-buffer-type-fct buffer-name buffer-type -> [ t | nil ]
```

Arguments and Values:

buffer-name ::= should be a symbol which names a buffer
buffer-type ::= a symbol which names a valid compilation type

Description:

`specify-compilation-buffer-type` allows one to change the compilation type for the buffers of the current model. If *buffer-name* and *buffer-type* are both valid, then that buffer will now be treated as a *buffer-type* buffer for compilation purposes, and **t** will be returned. If either parameter is invalid or there is no current model or current meta-process, then a warning is printed and **nil** is returned.

Note that this setting is only valid until the model is reset because the production compilation module will return all buffers to their default types when it is reset. For that reason, if one wants to make a new module's buffer's default type something other than motor without needing to specify this call in each model definition that uses the buffer then this setting must be placed into the secondary reset function for that module to ensure that the setting is not overwritten by the resetting of the production compilation module (it uses the primary reset function to set the default values).

Examples:

```

1> (specify-compilation-buffer-type goal motor)
T

2> (specify-compilation-buffer-type-fct 'visual-location 'retrieval)
T

3> (show-compilation-buffer-types)
  Buffer                Type
VISUAL                PERCEPTUAL
TEMPORAL              PERCEPTUAL
AURAL-LOCATION         PERCEPTUAL
VISUAL-LOCATION        RETRIEVAL
PRODUCTION           MOTOR
AURAL                PERCEPTUAL
VOCAL                MOTOR
IMAGINAL-ACTION     MOTOR
MANUAL               MOTOR
GOAL                 GOAL
IMAGINAL             IMAGINAL
RETRIEVAL           RETRIEVAL
NIL

E> (specify-compilation-buffer-type visual bad-type)
#|Warning: Invalid compilation buffer type BAD-TYPE.|#
NIL

E> (specify-compilation-buffer-type-fct 'bad-buffer 'motor)
#|Warning: No buffer named BAD-BUFFER found.|#
NIL

E> (specify-compilation-buffer-type goal perceptual)
#|Warning: get-module called with no current model.|#
#|Warning: No production compilation module found|#
NIL

E> (specify-compilation-buffer-type goal goal)
#|Warning: get-module called with no current meta-process.|#
#|Warning: No production compilation module found|#
NIL

```

Goal Module

The goal module provides the system with a goal buffer which is typically used to maintain the current task state of a model and to hold relevant information for the current task. The goal buffer also serves as a source of activation for declarative memory retrievals by default. The only action which the goal module provides a model is the creation of new chunks.

Goal buffer

The goal module sets the goal buffer to **not** be strict harvested.

Activation spread parameter: :ga
Default value: 0.0

Queries

The goal buffer only responds to the default queries.

‘State busy’ will always be **nil**.

‘State free’ will always be **t**.

‘State error’ will always be **nil**.

Requests

All

*{slot value}**

Each slot in the request should be specified at most once.

The request is used to create a new chunk which is placed into the goal buffer immediately. It will result in two events which look like this in the trace:

```
0.050 GOAL CREATE-NEW-BUFFER-CHUNK GOAL
0.050 GOAL SET-BUFFER-CHUNK GOAL CHUNK0
```

There is also a third event generated which will not show up in the trace. It is a maintenance event with the action clean-up-goal-chunk which performs some system clean up and will occur after the set-buffer-chunk event.

All requests are marked as completed at the time they are made.

Modification Requests

All

*{slot value}**

The goal buffer accepts modification requests and those specified buffer modifications are passed to [mod-buffer-chunk](#) for the goal buffer at the time of the request. It is assumed that the chunk will still be there at that time. If there are slots specified which are not valid slot names they will be added as [extended slots](#) prior to sending the modification request.

The event below will show up in the trace as a result of such an action being made by a production (always at the same time as the procedural request):

```
0.050    GOAL                MOD-BUFFER-CHUNK GOAL
```

If there are extended slots added then there will also be an event which looks like this indicating that the procedural module extended the possible slots:

```
0.050    PROCEDURAL          EXTEND-BUFFER-CHUNK GOAL
```

A modification request for the goal buffer is equivalent to performing the same buffer modification action in the production. The only difference is that by using a modification request to perform the change to the chunk the effort is attributed to the goal module instead of the procedural module which may be important for purposes of [predicting the BOLD response](#) or for tracing purposes.

All modification requests are marked as completed at the time they are made.

Commands

goal-focus

Syntax:

```
goal-focus {chunk-name} -> [ goal-chunk | nil ]  
goal-focus-fct {chunk-name} -> [ goal-chunk | nil ]
```

Arguments and Values:

chunk-name ::= should be a symbol which names a chunk
goal-chunk ::= a symbol which names the chunk in the goal buffer or the chunk which will be in the goal buffer

Description:

If chunk-name is provided, then goal-focus will schedule an event to put that chunk into the goal buffer of the current model at the current time with a priority of :max. This will result in the unrequested query being true for the buffer because this chunk was not placed into the buffer as a result of a module request. There will also be a maintenance event scheduled with an action of clear-delayed-goal following the set-buffer-chunk event which updates some internal information for the goal module but which will not show up in the trace. If an event is created to place this chunk into the buffer then chunk-name is returned. If chunk-name is not a valid chunk or there is no current model or meta-process then a warning is printed and **nil** is returned.

If chunk-name is not provided, then the chunk currently in the goal buffer is printed if there is one and that chunk's name is returned. If the buffer is empty, then a message stating that is printed and **nil** is returned. If there is a pending change to the chunk in the goal buffer (an event generated by goal-focus has been scheduled but not yet executed), then a notice of that is printed along with any chunk which may be in the buffer currently and the name of the chunk which will be in the buffer is returned.

This command typically occurs in a model to set the initial chunk in the goal buffer. If declarative learning of past goals is something that the model will be doing, then one should consider using [define-chunks](#) to create that initial goal instead of [add-dm](#) so that it is not in declarative memory prior to the start of the task.

Examples:

```
> (goal-focus)  
Goal buffer is empty  
NIL  
  
1> (goal-focus black)
```

BLACK

2> (goal-focus)

Will be a copy of BLACK when the model runs

BLACK

COLOR BLACK

BLACK

3> (run-n-events 1)

0.000 GOAL
0.000 -----

SET-BUFFER-CHUNK GOAL BLACK REQUESTED NIL
Stopped because event limit reached

0.0

1

NIL

4> (goal-focus)

BLACK-0

COLOR BLACK

BLACK-0

5> (goal-focus-fct 'free)

FREE

6> (goal-focus-fct)

Will be a copy of FREE when the model runs

Currently holds:

BLACK-0

COLOR BLACK

FREE

7> (run-n-events 1)

0.000 GOAL
0.000 -----

SET-BUFFER-CHUNK GOAL FREE REQUESTED NIL
Stopped because event limit reached

0.0

1

NIL

8> (goal-focus)

FREE-0

NAME FREE

FREE-0

E> (goal-focus notchunk)

#|warning: NOTCHUNK is not the name of a chunk in the current model - goal-focus failed |#
NIL

E> (goal-focus black)

#|warning: get-module called with no current model. |#

#|warning: get-chunk called with no current model. |#

#|warning: BLACK is not the name of a chunk in the current model - goal-focus failed |#

NIL

mod-focus

Syntax:

mod-focus {*slot-name slot-value* }* -> [chunk-name | **nil**]

mod-focus-fct ({*slot-name slot-value* }*) -> [chunk-name | **nil**]

Arguments and Values:

slot-name ::= a symbol that should be the name of a slot

slot-value ::= a Lisp value to set for the value of the corresponding slot-name

chunk-name ::= a symbol which will be the name of the chunk in the goal buffer

Description:

Mod-focus is used to modify the chunk currently in the goal buffer. The slot-name and slot-value pairs provided are passed to `mod-chunk` along with the name of the chunk which is in the goal buffer of the current model. In addition, an event is scheduled with the action goal-modification and priority :max so that the model can detect that the buffer has been changed outside of its actions.

If there is a chunk in the buffer and the modifications are valid for that chunk, then that chunk's name is returned. If there is no current model or current meta-process, no chunk in the goal buffer, or the modifications are invalid a warning is printed and **nil** is returned.

Examples:

```
1> (define-model test
    (chunk-type test color state)
    (goal-focus-fct (car (define-chunks (state start) (start)))))
TEST

2> (run 1)
    0.000 GOAL          SET-BUFFER-CHUNK GOAL CHUNK0 REQUESTED NIL
    0.000 PROCEDURAL   CONFLICT-RESOLUTION
    0.000 -----     Stopped because no events left to process
0.0
4
NIL

3> (mod-focus color green)
CHUNK0-0

4> (run 1)
    0.000 GOAL          GOAL-MODIFICATION
    0.000 PROCEDURAL   CONFLICT-RESOLUTION
    0.000 -----     Stopped because no events left to process
0.0
2
NIL

5> (mod-focus-fct '(state finished))
#|Warning: Creating chunk FINISHED with no slots|#
CHUNK0-0

6> (goal-focus)
CHUNK0-0
    COLOR GREEN
    STATE FINISHED

CHUNK0-0
```

```
E> (mod-focus slot)
#|Warning: Odd length modifications list in call to mod-chunk. |#
NIL
```

```
E> (mod-focus slot value)
#|Warning: No chunk in the goal buffer to modify |#
NIL
```

```
E> (mod-focus bad-slot green)
#|Warning: Invalid slot name BAD-SLOT specified for mod-chunk. |#
NIL
```

```
E> (mod-focus slot value)
#|Warning: buffer-read called with no current model. |#
#|Warning: No chunk in the goal buffer to modify |#
NIL
```

Imaginal Module

The imaginal module provides the system with an imaginal buffer which is typically used to maintain context relevant to the current task and a buffer called imaginal-action which can be used to call user functions for manipulating the contents of the imaginal buffer. The imaginal buffer's requests operates like the goal buffer's request in creating new chunks. However, unlike the goal buffer, requests to create chunks using the imaginal module take time. It also accepts modification requests which it also handles the same way the goal buffer does, except again there is a cost for doing that with the imaginal module.

Parameters

:imaginal-delay

The imaginal-delay parameter controls how long it takes a request or modification request to the imaginal buffer to complete. It can be set to a non-negative time (in seconds) and defaults to .2.

:vidt

The variable imaginal delay time parameter controls whether the actions of the imaginal buffer take exactly the amount of time specified by :imaginal-delay or if they are randomized with the [randomize-time command](#). If it is set to **t**, then randomize-time is used to adjust the delay time, and if it is set to **nil**, which is the default, then the delay times are fixed at the :imaginal-delay time.

Chunk-types & Chunks

The imaginal module defines the following two chunk-type:

```
(chunk-type generic-action action slots result)
(chunk-type simple-action action slots (simple t)))
```

It creates no initial chunks.

Imaginal buffer

The imaginal buffer is typically used to create and hold task relevant information. It operates similar to the goal buffer except that there is a time cost associated with creating and manipulating the chunks.

Activation spread parameter: :imaginal-activation

Default value: 1.0

Queries

The imaginal buffer only responds to the default queries.

‘State busy’ will be **t** after a new request or modification request is received (through either of the module’s buffers). If the request came through the imaginal buffer the state will return to **nil** once the request is completed. If the request came through the imaginal-action buffer then the busy state will remain **t** until the [set-imaginal-free command](#) is used to clear the busy state back to **nil**.

‘State free’ will be **t** when the ‘state busy’ flag is **nil** i.e. when the module is not currently handling a request, and it will be **nil** when the module is handling a request.

‘State error’ will be **nil** unless set by the user with the [set-imaginal-error command](#) which will cause it to be **t**. If set with the command, it will stay **t** until the next request is made to either of the module’s buffers. When the error state begins the imaginal buffer’s failure flag will also be set.

Requests

All

*{slot value}**

Each slot should be specified at most once and no modifiers are allowed.

The request is used to create a new chunk which is placed into the imaginal buffer after [:imaginal-delay](#) seconds. That will result in two events showing up in the trace:

```
0.750  IMAGINAL          CREATE-NEW-BUFFER-CHUNK IMAGINAL
0.750  IMAGINAL          SET-BUFFER-CHUNK IMAGINAL VISUAL-OBJECT1
```

There are three other maintenance events generated which will occur at the same time as the two actions shown but will not show up in the trace. The first occurs before the create-new-buffer-chunk event and will have an action of goal-style-request. The second has an action of clean-up-goal-chunk which performs some system cleanup as happens for the goal module and will occur after the set-buffer-chunk event. The final event, which occurs after the clean-up-goal-chunk event, has the action

set-imaginal-free and serves to set the module back to the state free after it has created the new chunk.

The module can only handle one request at a time. If a request is made while the module is busy processing a previous request it will print a warning and ignore the newer request.

```
#|Warning: Imaginal request made to the IMAGINAL buffer while the imaginal module was busy. New request ignored. |#
```

The module marks a request as completed when the set-imaginal-free action occurs.

Modification requests

All

*{slot value}**

The imaginal buffer accepts modification requests and those specified buffer modifications are passed to [mod-buffer-chunk](#) for the imaginal buffer after [:imaginal-delay](#) seconds have passed. It is assumed that the chunk will still be there at that time. If there are slots specified which are not valid slot names they will be added as [extended slots](#) prior to sending the modification request to the module.

This event will show up in the trace as a result of such an action:

```
1.000  IMAGINAL          MOD-BUFFER-CHUNK IMAGINAL
```

If there are any extended slots created then there will also be an event which looks like this indicating that at the time of the request:

```
0.800  PROCEDURAL      EXTEND-BUFFER-CHUNK IMAGINAL
```

There will also be a maintenance event generated which will not output to the trace. It will have the action of set-imaginal-free and occur after the mod-buffer-chunk event. It serves to set the module back to the free state after the buffer-modification is complete.

The module marks a modification request as completed when the set-imaginal-free action occurs.

Imaginal-action buffer

The imaginal-action buffer is available for users to extend the capabilities of the imaginal module. It does not perform any actions on its own nor does the module place any chunks into the buffer. It essentially provides a hook for the modeler to perform actions on the imaginal buffer's chunk which are attributed to the imaginal module.

Activation spread parameter: `:imaginal-action-activation`
Default value: 0.0

Queries

The imaginal-action buffer only responds to the default queries.

'State busy' will be **t** after a new request or modification request is received (through either buffer). If the request came through the imaginal buffer the state will return to **nil** once the request is completed. If the request came through the imaginal-action buffer then the busy state will remain **t** until the [set-imaginal-free command](#) is used to clear the busy state back to **nil**.

'State free' will be **t** when the 'state busy' flag is **nil** i.e. when the module is not currently handling a request and it will be **nil** when the module is handling a request.

'State error' will be **nil** unless set by the user with the [set-imaginal-error command](#) which will cause it to be **t**. If set with the command, it will stay **t** until the next request is made to either of the module's buffers. When the error state begins the imaginal-action buffer's failure flag will also be set.

Requests

Generic action

```
action function-name  
{slots (slot-name*)}
```

This request to the imaginal-action buffer requires the action slot be specified and its value must name a function. An optional list of slot names may be provided as the value of the slots slot in the request. Every slot-name provided must be the name of a slot of the chunk currently in the imaginal

buffer. When the request is received by the module the state busy query for the module will be set to **t**, the state error query will be set to **nil**, and the function specified by function-name will be called. If there is a list of slot names provided the function will be applied to that list, otherwise the function will be called with no parameters. No chunks will be placed into the buffer by default and the function which is called must return the module to the state free using the [set-imaginal-free command](#) either directly or by scheduling an event to do so at a later time. The function may also call the [set-imaginal-error command](#) to set the state error to **t** for the module if that is needed. The request will be marked as completed by either set-imaginal-free or set-imaginal-error.

This action essentially allows users to manipulate the chunk which is in the imaginal buffer via arbitrary functions and have the action attributed to the imaginal module i.e. it is essentially the same as a call to !eval! in a production except that the request is marked as going to the imaginal module and the module indicates “state busy” during that time. This allows modelers to add commands which they feel should be attributed to the imaginal module (for example rotations, translations or other manipulations of the representation) without having to change the code for the module.

No events are explicitly generated by this request, but the function which is called should generate events for any changes it makes to the buffer and to schedule the call to set-imaginal-free.

Simple action

```
action function-name  
simple t  
{slots (slot-name*)}
```

This request to the imaginal-action buffer is very similar to the generic action described above. The distinction between the two is that the simple action requires that an additional slot named simple be specified with the value **t** in the request. The difference between the two is that the simple action handles some of the event scheduling automatically.

When the request is received by the module the state busy query for the module will be set to **t**, the state error query will be set to **nil**, and the function specified by function-name will be called. If there is a list of slot names provided the function will be applied to that list, otherwise the function will be called with no parameters. That function should return either the name of a chunk or **nil**. If it returns the name of a chunk then after the current [:imaginal-delay](#) time passes that chunk will be copied into the imaginal buffer, the module will be returned to the free state, and the request will be completed. If the function returns **nil** then after the imaginal-delay time passes the module will be set to the free

state, it will be marked as also being in the error state, and the request will be completed. This request provides an easy way to have code create arbitrary chunks for the imaginal buffer without having to schedule any events or manage the internal imaginal state flags directly.

A valid request of this type will generate two events. One will be an event with the action of `set-imaginal-free`. The other depends on whether the action function returned a chunk or **nil**. If it returns a chunk then an event with the action `set-buffer-chunk` will be generated, but if **nil** was returned an event with the action `set-imaginal-error` will be created.

Commands

set-imaginal-free

Syntax:

set-imaginal-free -> [t | nil]

Description:

`Set-imaginal-free` is used to clear the busy state of the imaginal module in the current model of the current meta-process and indicate that a request has been completed. It should only be used by functions that are called as a result of a generic action request to the `imaginal-action` buffer.

If there is a current model then it returns **t**. If there is no current model then it returns **nil** and no change is made.

Examples:

```
> (set-imaginal-free)
T
```

```
E> (set-imaginal-free)
#|Warning: get-module called with no current model.|#
#|Warning: Call to set-imaginal-free failed|#
NIL
```

set-imaginal-error

Syntax:

set-imaginal-error -> [t | nil]

Description:

Set-imaginal-error is used to set the error state of the imaginal module in the current model of the current meta-process and indicate that a request has been completed. It should only be used by functions that are called as a result of a generic action request to the imaginal-action buffer. It causes queries of either of the imaginal module's buffers for 'state error' to return **t** and also sets the failure flags for both buffers as requested failures using [set-buffer-failure](#). The module's error state will remain until another request is made to either of the module's buffers.

If there is a current model then it returns **t**. If there is no current model then it prints a warning, returns **nil**, and no change is made.

Examples:

```
> (set-imaginal-error)
T
```

```
E> (set-imaginal-error)
#|Warning: get-module called with no current model. |#
#|Warning: Call to set-imaginal-error failed |#
NIL
```

Declarative module

The declarative module provides the model with a declarative memory which stores the chunks that are generated by the model and it provides a mechanism for retrieving those chunks through its retrieval buffer. The retrieval of chunks from the declarative memory depends on many factors that affect the accuracy and speed with which a chunk can be retrieved which is based on research of human memory performance.

The model's declarative memory (DM) consists of all the chunks which are explicitly added to it by the modeler as well as all of the chunks which are cleared from the buffers. Whenever a chunk is cleared from a buffer the declarative module merges that chunk into the model's DM. The merging process compares the chunk being added to the chunks already in DM. If the chunk being added is equivalent to a chunk which is already in DM, then the new chunk is merged with the existing chunk using the [merge-chunks command](#) and that chunk is strengthened by giving it another reference at the time of the merging. If the chunk being added is not equivalent to any chunk in DM then that chunk is added to DM.

Retrieval of chunks from DM is done through requests to the module. A request specifies a description of a chunk which is desired and if there are chunks in DM which match the specification request, then one of those chunks is placed into the retrieval buffer as a response. If no such chunk is found then it signals that a failure to retrieve occurred by signaling the state error, setting the buffer's failure flag, and leaving the buffer empty.

The time it takes to complete the request and how a single chunk is chosen among possibly many which match the request are controlled by several parameters. First, the setting of the [:esc parameter](#) determines very coarsely what process is used.

If the `:esc` parameter is **nil** then only the symbolic matching is considered. The chunks are always retrieved immediately and if there is more than one chunk which matches the request the setting of [:er](#) determines how a chunk is chosen. If `:er` is **t** then the choice is made randomly. If `:er` is **nil** then a deterministic process is used such that the same chunk will be chosen for that model each time the same set of possible chunks could be retrieved. However, that process is not specified as part of the declarative module's definition because it is not intended to be a process which one relies on for chunk preferences.

If the `:esc` parameter is `t` then the selection of which chunk is retrieved and how long it takes is controlled by a quantity called activation. Each chunk in DM has an activation value associated with it and among the chunks which match the request the one with the highest activation value, above a parameterized threshold, is the one that will be retrieved. If no matching chunk has an activation above the threshold then a failure to retrieve occurs. The activation of that chunk also determines how long the request takes to complete. If multiple matching chunks have the same highest activation, then the `:er` parameter determines how one of those chunks is chosen in the same way it happens when `:esc` is `nil`.

Activation

How the activation of a chunk is computed is based on the setting of several parameters which determine which mechanisms are to be used and those will be discussed in the specific sections which follow. Here is the general equation for the activation (A) of a chunk i :

$$A_i = B_i + S_i + P_i + \varepsilon_i$$

B_i : This is the [base-level activation](#) and reflects the recency and frequency of use of the chunk.

S_i : This is the [spreading activation](#) value computed for the chunk which reflects the effect that the contents of the buffers have on the retrieval process.

P_i : This is the [partial matching](#) value computed for the chunk which reflects the degree to which the chunk matches the specification requested.

ε_i : A [noise](#) value with both a transient and permanent component.

Each of those components will be described in more detail below. Note that for each of those components it is possible for the modeler to replace the mechanism as described with their own mechanism using the hook function parameters available in the declarative module.

Base-level

The base-level component, B_i , is computed differently based on the setting of the [:bll](#) and [:ol](#) parameters.

If :bll is **nil** then the setting of :ol does not matter and the base-level is a constant value determined by the [:blc parameter](#) or specific user settings for the chunk.

$$B_i = \beta_i$$

β_i : A constant offset which is determined by the :blc parameter or the chunk's :base-level parameter.

If :bll is set to a number, then the setting of :ol determines how the base-level is computed.

If :ol is **nil** then this equation is used:

$$B_i = \ln\left(\sum_{j=1}^n t_j^{-d}\right) + \beta_i$$

n : The number of presentations for chunk i .

t_j : The time since the j th presentation. A presentation is either the chunk's initial entry into DM or when another chunk is merged with a chunk which is in DM (these are also called the chunk's references).

d : The decay parameter which is set using the :bll parameter.

β_i : A constant offset determined by the :blc parameter.

If :ol is **t** then this approximation to that equation is used which does not require recording the complete history of the chunk:

$$B_i = \ln(n/(1-d)) - d * \ln(L) + \beta_i$$

n : The number of presentations of chunk i .

L : The lifetime of chunk i (the time since its creation).

d : The decay parameter (the value of :bll).

β_i : A constant offset determined by the :blc parameter.

If :ol is set to a number, then a hybrid of those is used such that the specified number of true references are used and the approximation is used for any remaining references (if there are not more total references than the parameter setting for :ol ($n \leq k$) then the full equation is used and the extra term in this equation is not computed):

$$B_i = \ln\left(\sum_{j=1}^k t_j^{-d} + \frac{(n-k) * (t_n^{1-d} - t_k^{1-d})}{(1-d) * (t_n - t_k)}\right) + \beta_i$$

k : is the value of the :ol parameter.

- t_j : The time since the j th presentation (for this equation t_1 is the time since the most recent presentation and t_n the time since the first presentation)
- n : The total number of presentations of chunk i .
- d : The decay parameter (the value of :bll).
- β_i : A constant offset determined by the :blc parameter.

Spreading Activation

Whether spreading activation is used is determined by the setting of the [:mas parameter](#). If it is **nil**, which is the default value, then the value of S_i is 0 in the activation equation. If :mas is set to a number, then this equation determines the spreading activation component of chunk i 's activation:

$$S_i = \sum_k \sum_j W_{kj} S_{ji}$$

The elements k being summed over are all of the buffers in the model which currently contain a chunk with a slot that has a chunk as its value (a source of activation) and which has a non-zero source activation.

The elements j being summed over are the chunks which are in the slots of the chunk in buffer k (these are referred to as the sources of activation).

W_{kj} : This is the amount of activation from source j in buffer k . It is the source activation of buffer k divided by the number of sources j in that buffer by default.

S_{ji} : This is the strength of association from source j to chunk i .

strength of association

The strength of association, S_{ji} , between two chunks is computed using the following equations by default, but can also be set explicitly by the modeler using the [add-sji command](#) or through the [sji-hook parameter](#).

If chunks j and i are not the same chunk and j is not in a slot of chunk i :

$$S_{ji} = 0$$

If chunks j and i are the same chunk or chunk j is in a slot of chunk i :

$$S_{ji} = S - \ln(\text{fan}_{ji})$$

S: The maximum associative strength set with the `:mas` parameter.

fan_{ji}: a measure of how many chunks are associated with chunk *j*.

The fan is typically thought of as the number of chunks in which *j* is the value of a slot plus one for chunk *j* being associated with itself. However, because *j* may appear in more than one slot of the chunk *i*, this is the specific calculation which is used to compute the fan:

$$\text{fan}_{ji} = \frac{1 + \text{slots}_j}{\text{slotsof}_{ji}}$$

slots_j: the number of slots in which *j* is the value across all chunks in DM.

slotsof_{ji}: the number of slots in chunk *i* which have *j* as the value (plus 1 when chunk *i* is chunk *j*).

The S_{ji} value can become negative as the fan_{ji} value grows, but that is generally an undesirable situation. By default the declarative module will print a warning if S_{ji} becomes negative due to that calculation and use 0.0 instead of the negative value, but that can be changed using the [:nsji](#) parameter.

Partial Matching

When the partial matching process is enabled it is possible for a chunk that is not a perfect match to the retrieval specification to be the one that is retrieved. To enable the partial matching one needs to set the [:mp](#) parameter to a number instead of its default of **nil**. When enabled, the similarity of the values requested to those in the slots of the chunks in DM are computed to determine the activation value. If partial matching is disabled then P_i is 0, but if it is enabled it is computed with this equation:

$$P_i = \sum_k PM_{ki}$$

The elements *k* being summed over are the slot values of the retrieval specification for slots with an = or – modifier only.

P: This is a match scale parameter (set with [:mp](#)) that reflects the amount of weighting given to the similarity.

M_{ki} : The similarity between the value k in the retrieval specification and the value in the corresponding slot of chunk i .

similarities

The possible range of default similarity values is configurable using the maximum similarity parameter ([:ms](#)) and the maximum difference parameter ([:md](#)). The default range is from 0 to -1 with 0 being the most similar and -1 being the largest difference. In general, the similarity can be thought of more as a difference penalty because the concept is not to boost the similar items, but to penalize the different ones. Since it gets added to the activation value, the values should be negative for items that are not the same and using positive similarities is not recommended (though not prohibited).

By default, a chunk has a maximum similarity to itself and a maximum difference to all other chunks. Any other similarities must be set explicitly by the modeler either using the [set-similarities command](#) or the [sim-hook parameter](#). For non-chunk slot values in a retrieval request, the similarity is the maximum similarity if the values are equal using the [chunk-slot-equal](#) function and the maximum difference if they are not. The only way to set non-default similarity values for items which are not chunks is through the sim-hook capability. Similarities set explicitly with the declarative commands or through the sim-hook function are not constrained by the [:ms](#) and [:md](#) parameters.

One final note on the computation of the M_{ki} values. If the retrieval specification is requesting the value using the negation modifier, then the M_{ki} value for that slot test k is not the specific similarity value between the items involved. If the similarity between those items is equal to the maximum similarity then M_{ki} is set to the maximum difference. Otherwise, M_{ki} is set to 0 for that slot. Essentially, if they are the same (or perfectly similar items), then the chunk is given the maximum penalty since the request specified that it not have that value and if they are different (any similarity other than a perfect match) then no penalty is applied.

Noise

The noise calculation has two components. There is a transient component which is computed each time a retrieval request is made and there is a permanent component which is associated specifically with the chunk that is generated once when the chunk is entered into DM. The total noise added to the activation is the sum of the two components. Often, the transient component is sufficient for modeling and the permanent noise is left disabled.

Each one is generated using the [act-r-noise command](#). Thus they are generated from a logistic distribution with a mean of 0 and an s as specified by the corresponding parameter of the declarative module. The parameter for the transient noise s value is [:ans](#) and the parameter for the permanent noise s value is [:pas](#). The default value for each parameter is **nil**. For the transient noise that means to not generate any transient noise and for the permanent noise it means to leave the chunk's permanent noise set to 0. The permanent noise is always added to the activation of the chunk and can be set by the modeler using the [sdp command](#) for creating specific offset values when needed.

Retrieval time

The time that it takes the declarative module to respond to a request for a chunk, i , is determined by the activation that the chunk has using this equation when the subsymbolic computations are enabled:

$$RT = Fe^{-(f * A_i)}$$

RT: The time to retrieve the chunk in seconds

A_i: The activation of the chunk i which is being retrieved

F: The [latency factor parameter](#)

f: The [latency exponent parameter](#)

If there is no chunk found in response to a request or the chunk with the highest activation is below the retrieval threshold then the time required to indicate a failure to retrieve any chunk is determined by this equation when subsymbolic computations are enabled:

$$RT = Fe^{-(f * \tau)}$$

RT: The time until the failure is noted in seconds.

τ : The [retrieval threshold parameter](#)

F: The [latency factor parameter](#)

f: The [latency exponent parameter](#)

Declarative finsts

The declarative module maintains a record of the chunks which have been retrieved and provides a mechanism which allows one to indicate whether a retrieval request should search only those which have or have not been so marked. This is done through the use of a set of finsts (fingers of instantiation) which mark those chunks. The finsts are limited in the number that are available and how long they persist specified by the parameters [:declarative-num-finsts](#) and [:declarative-finst-span](#)

respectively. If more finsts are required than are available, then the oldest one (the one marking the chunk retrieved at the earliest time) is removed and used to mark the most recently retrieved chunk. The details on how to use the finsts are found in the description of the [retrieval buffer requests](#).

Parameters

The declarative module has a lot of parameters which can be set and they fall into four general categories. The first category contains the parameters which are used in the activation equations as described above. The next category contains the parameters which control basic functionality of the module. The third category is parameters which allow the user to adjust the chunk activation equation: each of the four primary components may be replaced by the user, the default computations for strengths of association and similarities can be replaced, and additional terms may be added to the equation. The final set of parameters allow the user to install functions to monitor the operations of the module.

Note on parameters for the declarative module. There are some parameters which should not be adjusted once chunks have entered DM. The reason for this is that the module does not attempt to reconcile differences in interpretations which may result due to such changes i.e. chunks which have had their internal parameters set under one set of parameters may no longer be valid under different settings. A warning will be printed if one does change such parameters after there are chunks in DM. That warning does not mean that things will necessarily break (there are a few situations where the warning can be safely ignored), but one should be cautious when changing things in that manner. The critical parameters are the [:bll](#) and [:mas](#) parameters of the declarative module as well as the parameters [:esc](#) and [:ol](#).

:act

The activation trace parameter controls whether or not the declarative module should print the details of a chunk's activation computation when it is computed (whether during a retrieval request or otherwise). If it is set to **t** then all of the components of the equation are output to the model's trace when a chunk's activation is computed, and if it is set to **nil** then no extra trace is generated. The default value is **nil**.

There are also two lesser output levels available for the activation trace if one specifies a setting of **medium** or **low** for the parameter value. The medium output level does not print the information

about chunks which did not match a retrieval request and the low level only prints the final activation values computed for the chunks.

:activation-offsets

This parameter allows one to specify functions which can extend the activation equation with new terms. This parameter can be set to a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). Whenever a chunk's activation is computed each of the functions that has been set for this parameter will be called with one parameter which is the name of the chunk. If a function called returns a number then that value will be added to the activation of the chunk and if the activation trace is enabled a line will be shown indicating the name of the function and the offset added. If a function returns any other value then no change is made to the activation of the chunk. If the parameter is set to **nil** then all functions are removed from the activation-offsets list. A function should only be set once. If a function is specified more than once as a value for the activation-offsets a warning will be displayed and a value of **nil** will be returned as the current value.

:ans

The activation noise s parameter specifies the s value used to generate the instantaneous noise added to the activation equation if it is set to a positive number. If it is set to **nil**, which is the default, then no instantaneous noise is added to the activations.

:bl-hook

This parameter allows one to override the base-level calculation. If it is set to a function then that function will be passed one parameter which is the name of the chunk for which a base-level is needed. If the function returns a number then that will be the B_i value used in the activation equation, otherwise the standard base-level calculation will be used.

:blc

The base-level constant parameter specifies the default value for the β_i component of the base-level equations. If base-level learning is disabled (:bll is **nil**) and the :base-level parameter for the chunk is set through the [sdp command](#) then that overrides the :blc setting. The default value is 0.0.

:bll

The base-level learning parameter controls whether base-level learning is enabled, and if so what the value of the decay parameter, *d*, is set to. It can be set to any positive number or the value **nil**. The value **nil** means do not use base-level learning and is the default value, a number means that base-level is enabled. The recommended value for *:bll* is .5, and it is one of the few parameters which have a strong recommended value.

:chunk-add-hook

This parameter allows one to specify functions to be called automatically when chunks are added to the current model's DM. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). Whenever a chunk is added into DM each of the functions that has been set will be called with one parameter which is the name of the chunk that was added into DM. This call is made after the chunk has been added to DM and its declarative parameters updated appropriately. The return values of those functions are ignored. If the parameter is set to **nil** then all functions are removed from the chunk-add-hook. A function should only be set once. If a specific function is specified more than once as a value for the chunk-add-hook a warning will be displayed and a value of **nil** will be returned as the current value.

:chunk-merge-hook

This parameter allows one to specify functions to be called automatically when chunks are merged into to the current model's DM. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). Whenever a chunk is merged into DM each of the functions that has been set will be called with one parameter which is the name of the chunk that was merged into DM. This call is made after the chunk has been merged into DM and its declarative parameters updated appropriately. The return values of those functions are ignored. If the parameter is set to **nil** then all functions are removed from the chunk-merge-hook. A function should only be set once. If a specific function is specified more than once as a value for the chunk-merge-hook a warning will be displayed and a value of **nil** will be returned as the current value.

:declarative-finst-span

This parameter controls how long a finst can mark a chunk as having been recently retrieved. After a finst has been on a chunk for this amount of time the finst is removed and the chunk is no longer marked as recently retrieved. It can be set to any positive number and defaults to 3.0.

:declarative-num-finsts

This parameter controls how many finsts are available in the declarative module. It can be set to any positive number and the default is 4.

:declarative-stuffing

This parameter controls whether or not the declarative module will place chunks into the retrieval buffer without a corresponding request i.e. stuff the buffer. The default value is **nil** which means to disable the mechanism. If it is set to a number that enables the mechanism and sets the period of time in seconds for the potential stuffing actions to occur.

When it is enabled, at time 0 and again whenever the module is not busy and either the buffer empties or the time period has passed a new stuffing attempt will occur. The chunk that will be stuffed into the buffer is the chunk with the current highest activation. That stuffing will not interrupt an ongoing retrieval request nor overwrite a requested chunk in the buffer. This mechanism is a very speculative bottom-up approach which needs further refinement based on use – if you use this parameter please let me know how it worked out and any recommendations or comments you have about it.

:le

The latency exponent value, f , in the equation for retrieval times. It can be set to any non-negative value and defaults to 1.0.

:lf

The latency factor value, F , in the equation for retrieval times. It can be set to any non-negative value and defaults to 1.0.

:mas

The maximum associative strength parameter controls whether the spreading activation calculation is used, and if so, what the S value in the S_{ji} calculations will be. It can be set to any number or the value **nil**. The value **nil** means do not use spreading activation and is the default value, any number means that spreading activation is enabled.

:md

The maximum difference. This is the default similarity value between two items which are not [chunk-slot-equal](#). It can be set to any number and defaults to -1.0.

:mp

The mismatch penalty parameter controls whether the partial matching system is enabled, and if so, what the value of the penalty parameter, P, in the activation equation is set to. It can be set to any number or the value **nil**. The value **nil** means do not use partial matching (only exact matches can be retrieved), and is the default. If it is set to a number then partial matching is enabled.

:ms

The maximum similarity. This is the default similarity value between items which are [chunk-slot-equal](#). It can be set to any number and defaults to 0.0.

:noise-hook

This parameter allows one to override the noise calculation. If it is set to a function then that function will be passed one parameter which is the name of the chunk for which a noise value is needed. If the function returns a number then that will be the noise value used in the activation equation.

:nsji

Whether or not to allow negative S_{ji} values from the S-log(fan) calculation, Can be set to **t** which means that they are allowed, **warn** which means that negative values will be treated as 0 activation spread and a warning will be displayed, or **nil** which means that negative values will be treated as 0 activation spread and no warning will be displayed. A value of **warn** or **nil** does not prevent the user from setting explicit negative values if desired. The default is **warn**.

:partial-matching-hook

This parameter allows one to override the partial matching calculation. If it is set to a function then that function will be passed two parameters. The first will be the name of the chunk for which a partial matching value is needed and the second will be the [chunk-spec](#) of the request. If the function returns a number then that will be the P_i value used in the activation equation.

:pas

The permanent activation noise `s` parameter specifies the `s` value used to generate the permanent noise added to the activation equation of a chunk if it is set to a positive number. The permanent noise is only generated once when the chunk is added to DM. If `:pas` is set to **nil**, which is the default, then no permanent noise is automatically generated for the chunks, but it may still have such a value set explicitly using the [sdp command](#).

:retrieval-request-hook

This parameter allows one to specify functions to be called automatically when a retrieval request is made. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). When there is a start-retrieval event these functions will be called with the [chunk-spec](#) of the request as the parameter. The return values of those functions are ignored. If the parameter is set to **nil** then all functions are removed from the retrieval-request-hook. A function should only be set once. If a specific function is specified more than once as a value for the retrieval-request-hook a warning will be displayed and a value of **nil** will be returned as the current value.

:retrieval-set-hook

This parameter allows one to specify functions to be called during the retrieval process. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). After the set of chunks which match have been determined and their activations are calculated this function will be called with the list of the chunk names that match the request. The first chunk on the list is the one that will be retrieved (if its activation is above the threshold). This function can be used to override the choice of which chunk to retrieve. If this function returns a cons of a chunk name and a number, then that chunk will be the one placed in the retrieval buffer after that many seconds pass. If the function returns a number, then the declarative module will signal a retrieval failure after that many seconds pass. If the function returns anything else then the normal retrieval process will occur. If more than one function on the list for this parameter return a value, then none of those values will be used and the default mechanisms will be applied. If the parameter is set to **nil** then all functions are removed from the retrieval-set-hook. A function should only be set once. If a specific function is specified more than once as a value for the retrieval-set-hook a warning will be displayed and a value of **nil** will be returned as the current value.

:retrieved-chunk-hook

This parameter allows one to specify functions to be called automatically when chunks are retrieved or a failure to retrieve occurs. This parameter can be set with a function which takes one parameter and any number of such functions may be set (the reported value of this parameter is a list of all functions which have been set). When there is a retrieved-chunk event these functions will be called with the name of the chunk which has been retrieved, and when there is a retrieval-failure event these functions will be called with **nil** as the parameter. The return values of those functions are ignored. If the parameter is set to **nil** then all functions are removed from the retrieved-chunk-hook list. A function should only be set once. If a specific function is specified more than once as a value for the retrieved-chunk-hook a warning will be displayed and a value of **nil** will be returned as the current value.

:rt

The retrieval threshold. This is the minimum activation a chunk must have to be able to be retrieved, τ , in the retrieval failure time equation. It can be set to any number and defaults to 0.0.

:sact

The save activation trace parameter controls whether or not the declarative module should save the details of the activation computation when it is computed during a retrieval request. If it is set to a non-nil value then all of the components of the activation calculations are saved and a trace like the one displayed when using the :act parameter can be printed out after a run using the [print-activation-trace](#) and [print-chunk-activation-trace](#) commands. The default value is **nil**.

The parameter can be set to values like :act i.e. **t**, **medium**, **low**, or **nil**. The output of the print-activation-trace and print-chunk-activation-trace commands will use the specific setting of this parameter to display the requested amount of detail in the traces shown.

:sim-hook

This parameter allows one to override the similarity calculation. If it is set to a function then that function will be passed two parameters. The first will be the slot contents of the request specification, the k from the partial matching equation, and the second will be the considered chunk's value for that slot. If the function returns a number that will be the M_{ki} value used for that term in the partial matching equation. If the function returns **nil** or a non-numeric value the default M_{ki} value will be used.

:sji-hook

This parameter allows one to override the strength of association calculation. If it is set to a function then that function will be passed two parameters. The first will be the chunk j from a slot in a buffer chunk and the second will be the considered chunk i . If the function returns a number that will be the S_{ji} value used in the equation of S_i for the chunk i . If the function returns **nil** or a non-numeric value the default S_{ji} value will be used.

:spreading-hook

This parameter allows one to override the spreading activation calculation. If it is set to a function then that function will be passed one parameter which is the name of the chunk for which a spreading activation value is needed. If the function returns a number then that will be the S_i value used in the activation equation. If the function returns **nil** or a non-numeric value the default spreading activation calculation will be used.

:w-hook

This parameter allows one to override the default values for W_{kj} in the strength of association calculation. If it is set to a function then that function will be passed two parameters. The first will be the name of the buffer, the k , and the second will be the name of a slot for which the source of activation, the j , is being spread. If the function returns a number that will be the W_{kj} value used in the equation of S_i at that time. If the function returns **nil** or a non-numeric value the default W_{kj} value will be used.

Retrieval buffer

The retrieval buffer is used to retrieve chunks from the model's declarative memory using the mechanisms described above.

Activation spread parameter: :retrieval-activation
Default value: 0.0

Queries

In addition to the default queries the retrieval buffer can be queried with recently-retrieved which can be checked for the values of **t** or **nil**.

‘State busy’ will be **t** while a retrieval request is being processed – the time between the start-retrieval event and either the retrieved-chunk or retrieval-failure event. It will be **nil** at all other times.

‘State free’ will be **nil** while a retrieval request is being processed – the time between the start-retrieval event and either the retrieved-chunk or retrieval-failure event. It will be **t** at all other times.

‘State error’ will be **t** if no chunk matching the most recent request was found (a retrieval-failure event has occurred) and **nil** otherwise. Once it becomes **t** it will not change back to **nil** until the next retrieval request is made.

‘Recently-retrieved t’ will be **t** if there is a chunk in the retrieval buffer and there is a chunk in DM from which that chunk was copied which is currently marked with a declarative finst. Otherwise this query will be **nil**.

‘Recently-retrieved nil’ will be **t** if there is a chunk in the retrieval buffer and there is a chunk in DM from which that chunk was copied which is currently not marked with a declarative finst. Otherwise this query will be **nil**.

Requests

All

```
{{modifier} slot value}*  
{:recently-retrieved [t | nil | reset]}  
{:mp-value [ nil | temp-value]}
```

A request to the retrieval buffer is a description of a chunk which the declarative module will try to find in DM and place into the retrieval buffer. For the declarative module to consider a chunk in DM as a possible candidate it must have all of the slots specified with non-**nil** values and not have any slots specified with a value of **nil**.

There are two request parameters which can be used to modify how that request is handled. The `:recently-retrieved` request parameter can be used to test the declarative finsts associated with the chunks in addition to the chunks' contents. If `:recently-retrieved` is specified as **t** then the request will only match to chunks which have a finst set for them at the time of the request. If `:recently-retrieved` is specified as **nil** then the request will only match to chunks which do not have a finst set for them at the time of the request, and if `:recently-retrieved` is specified as **reset** then all of the declarative finsts are removed before the request is processed. The `:mp-value` request parameter allows one to

temporarily change the setting of the declarative module's `:mp_parameter` while this request is processed. That can only be used if the `:mp` parameter has been enabled (set to a number) for the model, and the value provided can be anything that is valid for `:mp` (a number or **nil**).

If a chunk which matches the request is found it will be placed into the retrieval buffer. If no chunk is found, or no chunk which matches has an activation which is above the retrieval threshold when the `:esc` parameter is **t**, then the buffer is left empty, the module signals that its state is error, and the retrieval buffer's failure flag is set.

The declarative module will only process one request at a time. If a new request comes in prior to the completion of a previous request the older request is terminated immediately – no chunk will be placed into the buffer or error signaled as a result of that request. A warning will be output to the trace indicating the early termination of the previous request and the module will remain busy while processing the new request.

A successful request to the declarative module will generate the events start-retrieval, retrieved-chunk, set-buffer-chunk like this:

```
0.050  DECLARATIVE          START-RETRIEVAL
...
0.100  DECLARATIVE          RETRIEVED-CHUNK C
0.100  DECLARATIVE          SET-BUFFER-CHUNK RETRIEVAL C
```

A failed request will generate the events start-retrieval and retrieval-failure:

```
10.250  DECLARATIVE          START-RETRIEVAL
...
10.300  DECLARATIVE          RETRIEVAL-FAILURE
```

If a request is terminated prematurely by a new request this is the warning that will show in the trace:

```
10.350  DECLARATIVE          START-RETRIEVAL
...
#|Warning: A retrieval event has been aborted by a new request |#
```

The request is considered completed when it succeeds and places a chunk into the buffer, when the retrieval-failure action occurs, or when it is aborted by a new request.

Commands

add-dm

Syntax:

```
add-dm ({chunk-name}[doc-string isa chunk-type | isa chunk-type]} {slot value}*)* -> (chunk*)
```

add-dm-fct (*chunk-name* *doc-string isa chunk-type | isa chunk-type* *slot value**)* -> (chunk*)

Arguments and Values:

chunk-name ::= a symbol that will be the name of the chunk

doc-string ::= a string that will be the documentation for the chunk

chunk-type ::= a symbol that names a chunk-type in the model

slot ::= a symbol that names a slot for the chunk

value ::= any Lisp value which will be the contents of the correspondingly named slot for this chunk

chunk ::= a symbol which names a chunk that was created

Description:

The `add-dm` command functions exactly like the [define-chunks command](#) to create new chunks for the model. In addition, `add-dm` places those chunks into the current model's declarative memory. It returns a list of the names of the chunks that were created.

If the syntax is incorrect or any of the components are invalid for a list describing a chunk then a warning is displayed and no chunk is created for that chunk description, but any other valid chunks defined will still be created.

If there is no current model or meta-process then a warning is displayed, no chunks are created and **nil** is returned.

`Add-dm` is used to provide the model with a set of initial memories. It should not be used for the creation of general chunks. In particular, it should not be used by other modules to create the chunks to place into their buffers because those chunks will be added to DM automatically when the buffer is cleared and should not be placed there prior to that.

Examples:

```
1> (chunk-type number value)
NUMBER
```

```
2> (add-dm (isa number value 1)
          (two value 2))
(NUMBER0 TWO)
```

```
3> (add-dm-fct (list '(three "the number 3" isa number value 3)
                    '(value 4)))
(THREE CHUNK0)
```

```
E> (add-dm (bad-chunk isa invalid-type)
          (bad-slot 10))
```

```
#|Warning: Invalid chunk definition: (BAD-CHUNK ISA INVALID-TYPE) chunk-type specified
does not exist. |#
```

```
#|Warning: Extending chunks with slot named BAD-SLOT because of chunk definition (BAD-SLOT
10) |#
```

(CHUNK1)

```
E> (add-dm (a))
#|warning: get-module called with no current model. |#
#|warning: Could not create chunks because no declarative module was found |#
NIL
```

```
E> (add-dm (a))
#|warning: get-module called with no current meta-process. |#
#|warning: Could not create chunks because no declarative module was found |#
NIL
```

dm

Syntax:

```
dm chunk-name* -> (chunk-name*)
dm-fct (chunk-name*) -> (chunk-name*)
```

Arguments and Values:

chunk-name ::= a symbol which names a chunk

Description:

The `dm` command is used to print out chunks which are in the declarative memory of the current model in the current meta-process. For each chunk name provided that chunk will be printed to the current model's command output stream. If no chunk names are provided then all of the chunks in DM will be printed. A list of the names of the chunks which are printed will be returned.

If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
> (dm)
FIRST-GOAL
  START 2
  END 4

F
  FIRST 5
  SECOND 6

E
  FIRST 4
  SECOND 5

D
  FIRST 3
  SECOND 4
```

```

C
  FIRST 2
  SECOND 3

B
  FIRST 1
  SECOND 2

(FIRST-GOAL F E D C B)

> (dm b d)
B
  FIRST 1
  SECOND 2

D
  FIRST 3
  SECOND 4

(B D)

> (dm-fct '(first-goal c))
FIRST-GOAL
  START 2
  END 4

C
  FIRST 2
  SECOND 3

(FIRST-GOAL C)

E> (dm bad-name)
NIL

E> (dm)
#|Warning: get-module called with no current model.|#
#|Warning: No declarative memory module found|#
NIL

E> (dm)
#|Warning: get-module called with no current meta-process.|#
#|Warning: No declarative memory module found|#
NIL

```

sdm

Syntax:

```

sdm {isa chunk-type} slot-test* -> (chunk-name*)
sdm-fct ( {isa chunk-type} slot-test* ) -> (chunk-name*)

```

Arguments and Values:

```

chunk-type ::= a symbol which names a chunk-type
slot-test ::= {slot-modifier} slot value
slot-modifier ::= [= | - | < | > | <= | >=]
slot ::= a symbol which names a slot
value ::= any Lisp value

```

chunk-name ::= a symbol which names a chunk

Description:

The `sdm` command is used to search the declarative memory of the current model and print out chunks which match the search specification. If no parameters are provided then all chunks in DM are printed. If parameters are provided then all chunks from DM which match the specification provided are printed. If the `chunk-type` is specified then the slots given must be valid for that `chunk-type` otherwise a warning will be printed and no chunks returned. As is the case elsewhere, the `chunk-type` itself is not a constraint in the search, but if it contains default slot values those will be included in the specification unless other values are provided. If no `chunk-type` is provided then any slots and values may be provided. A list of the names of the chunks which are printed is returned.

If there is an error in the specification or there is no current model a warning is printed and **nil** is returned.

Examples:

```
1> (chunk-type test (slot1 t))
TEST

2> (add-dm (t1 isa test slot1 t) (t2 isa test slot1 nil))
(T1 T2)

3> (sdm isa test)
T1
  SLOT1 T

(T1)

1> (chunk-type type1 slot1 slot2)
TYPE1

2> (chunk-type number name value)
NUMBER

3> (chunk-type type2 name slot1)
TYPE2

4> (add-dm (one isa chunk)
          (a isa type1 slot1 1 slot2 a)
          (b isa type1 slot1 2 slot2 a)
          (c isa number name one value 1)
          (d isa type2 name one slot1 3))
(ONE A B C D)

5> (sdm isa type1)
D
  NAME ONE
  SLOT1 3

C
  NAME ONE
  VALUE 1
```

```

A
  SLOT1 1
  SLOT2 A

B
  SLOT1 2
  SLOT2 A

ONE

(D C A B ONE)

6> (sdm name one)
D
  NAME ONE
  SLOT1 3

C
  NAME ONE
  VALUE 1

(D C)
7> (sdm-fct '(< slot1 2))
A
  SLOT1 1
  SLOT2 A

(A)

8> (sdm - slot2 a)
D
  NAME ONE
  SLOT1 3

C
  NAME ONE
  VALUE 1

ONE

(D C ONE)

E> (sdm isa bad-type)
#|warning: Element after isa in define-chunk-spec isn't a chunk-type. (ISA BAD-TYPE) |#
#|warning: Invalid chunk specification (ISA BAD-TYPE) passed to sdm |#
NIL

E> (sdm isa type1 value 3)
#|warning: Invalid slot-name VALUE in call to define-chunk-spec. |#
#|warning: Invalid chunk specification (ISA TYPE1 VALUE 3) passed to sdm |#
NIL

E> (sdm)
#|warning: get-module called with no current model. |#
#|warning: No declarative memory module found |#
NIL

E> (sdm)
#|warning: get-module called with no current meta-process. |#
#|warning: No declarative memory module found |#
NIL

```

print-dm-finsts

Syntax:

`print-dm-finsts -> (<chunk-name time-stamp>*)`

Arguments and Values:

`chunk-name ::=` a symbol which names a chunk in DM

`time-stamp ::=` a number indicating the time the finst was applied to the chunk `chunk-name` in seconds

Description:

Print-dm-finsts can be used to see which chunks in the current model of the current meta-process have declarative finst markers. It takes no parameters and will print out a table of the chunks with finsts on them showing the time at which the finst was set in seconds. It returns a list of cons cells where each cell has the name of a chunk with a finst on it as the car and the creation time of its finst in seconds as the cdr.

If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

This example assumes that the count model from tutorial unit 1 has been loaded.

```
1> (print-dm-finsts)
```

```
Chunk name      Time Stamp
```

```
-----
```

```
NIL
```

```
CG-USER(839): (run 10)
```

```
0.000 GOAL SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED START
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK C
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL C
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 PROCEDURAL PRODUCTION-SELECTED INCREMENT
0.100 PROCEDURAL BUFFER-READ-ACTION GOAL
0.100 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT
```

```
2
```

```
0.150 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.150 PROCEDURAL MODULE-REQUEST RETRIEVAL
```

```

0.150 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.150 DECLARATIVE START-RETRIEVAL
0.150 PROCEDURAL CONFLICT-RESOLUTION
0.200 DECLARATIVE RETRIEVED-CHUNK D
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL D
0.200 PROCEDURAL CONFLICT-RESOLUTION
0.200 PROCEDURAL PRODUCTION-SELECTED INCREMENT
0.200 PROCEDURAL BUFFER-READ-ACTION GOAL
0.200 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT
3
0.250 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.250 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.250 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.250 DECLARATIVE START-RETRIEVAL
0.250 PROCEDURAL CONFLICT-RESOLUTION
0.250 PROCEDURAL PRODUCTION-SELECTED STOP
0.250 PROCEDURAL BUFFER-READ-ACTION GOAL
0.300 DECLARATIVE RETRIEVED-CHUNK E
0.300 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL E
0.300 PROCEDURAL PRODUCTION-FIRED STOP
4
0.300 PROCEDURAL CLEAR-BUFFER GOAL
0.300 PROCEDURAL CONFLICT-RESOLUTION
0.300 ----- Stopped because no events left to process
0.3
50
NIL
3> (print-dm-finsts)

```

```

Chunk name      Time Stamp
-----
E                0.300
D                0.200
C                0.100
((E . 0.3) (D . 0.2) (C . 0.1))

```

```

E> (print-dm-finsts)
#|warning: get-module called with no current model.|#
NIL

```

```

E> (print-dm-finsts)
#|warning: get-module called with no current meta-process.|#
NIL

```

sdp

Syntax:

```

sdp [{ [ chunk-name | (chunk-name*) ] } { [ param-name* | param-value-pair* ] } |
(chunk-name [ param-name* | param-value-pair* ])* -> (param-values*)
sdp-fct ( [ { [ chunk-name | (chunk-name*) ] } { [ param-name* | param-value-pair* ] } |
(chunk-name [ param-name* | param-value-pair* ])* ) -> (param-values*)

```

Arguments and Values:

chunk-name ::= a symbol which is the name of a chunk in declarative memory
param-name ::= a keyword which names a declarative parameter
param-value-pair ::= *param-name* *new-param-value*
new-param-value ::= a Lisp value to which the preceding *param-name* is to be set
param-values ::= [(*param-value**) | (*chunk-name**)] **:error**

param-value ::= the current value of a requested declarative parameter or **:error**.

Description:

Sdp is used to set or get the declarative parameters of the chunks in the current model. It is similar to the [sgp command](#) which is used to set and get module parameters.

Each chunk has several declarative parameters associated with it, and most of those are relate to the computation of the activation equation. The declarative parameters are only relevant when the [:esc parameter](#) is enabled, and even then, some of the parameters are only available when specific options of the declarative module are also enabled.

The declarative parameters available through sdp are listed below. An important thing to note is that while these declarative parameters are maintained using general [chunk parameters](#) not all of the general chunk parameters are available through sdp. Only the chunk parameters which are relevant to the declarative module are accessible through the sdp command, and the mapping of declarative parameters onto chunk parameters may not be one-to-one. Thus, the declarative parameters should only be accessed through the sdp command and not be read or changed through the general chunk parameter accessors since their internal representations are not part of the declarative module's API.

- name := the name of the chunk. Cannot be changed.
- activation := the chunk's current activation value. Cannot be changed through sdp.
- permanent-noise := the permanent noise is a value which is added to the activation of the chunk each time it is computed. It defaults to 0.0. If :pas is set to a number, then when a chunk is initially added to DM a random noise value generated using the :pas value as the s parameter to [act-r-noise](#) will be generated and set as the chunk's permanent noise. It can be set by the modeler to any number, which will override the default and automatically generated values.
- base-level := the chunk's current base-level value. Cannot be changed directly through sdp when base-level learning is enabled because in that case it is controlled by the chunk's creation-time, reference-count, and/or reference-list. If base-level learning is disabled then it can be set to a number which will be the β_i value for the chunk.
- creation-time := the time (in seconds) when the chunk was first added to DM. Used by the base-level equations to determine the t_j values and the chunk's life time. Can be set to any time (including negative values) which is less than or equal to the current time. This is only applicable when :bll is set to a non-**nil** value.

- `reference-count` := the number of references which the chunk has received. Can be set to any positive value. It is applicable when `:bll` is non-**nil** and `:ol` is either **t** or a number. It is the `n` value used in the optimized base-level equations. If the `reference-count` is set by the user then the `reference-list` for the chunk will be adjusted to contain an appropriate number of references as follows. If the `reference-list` is as long as the `reference-count` specified or `:ol` is **t** then the `reference-list` will be unchanged. If the `reference-list` is currently longer than the specified `reference-count` it will be truncated to that many of the most recent entries. If the `reference-list` is shorter than the specified `reference-count` but has as many items as the current setting of `:ol` it will be unchanged. Otherwise, the `reference-list` will be set to an evenly distributed set of references as would be done by the [set-base-levels command](#).
- `reference-list` := the list of times at which the chunk's references have occurred (most recent first). Can be set to a list of times. It is applicable when `:bll` is non-**nil** and `:ol` is either **nil** or a number. If `:ol` is **nil**, then the list will contain all of the reference times of the chunk. If `:ol` is set to a number then it will only hold that many references (the most recent). If more references are provided than are needed, the list is truncated to the necessary length. When the `reference-list` is set by the user the `reference-count` may be adjusted automatically. If `:ol` is **nil**, then the `reference-count` will be updated to the length of the `reference-list` (even though the `reference-count` is not actually used when `:ol` is **nil**). If `:ol` is a number and the `reference-count` is less than the length of the (possibly truncated) `reference-list` it will be set to the length of the `reference-list`. Otherwise, the `reference-count` will be unchanged.
- `references` := This parameter is deprecated and should not be used, but is still available for compatibility with older models. This parameter will not be shown for a chunk and modelers should use `reference-count` and `reference-list` instead to get/set the relevant values.
- `source` := the chunk's current activation spread from the current buffer contents. Cannot be changed directly through `sdp`. Only applicable when the `:mas` parameter is set to a number.
- `sjis` := this reports the S_{ji} value for chunks `j` to the chunk `i` (where `i` is the chunk for which the value is being reported). The reported value is a list of cons cells where the car of a cons is the name of a chunk `j` and the cdr is the S_{ji} between that chunk `j` and the chunk `i`. Only chunks `j` which have a connection with `i` are reported – all other S_{ji} values to chunk `i` will be 0.0 (unless a provided `:sji-hook` overrides that). This parameter is only relevant when the `:mas` parameter is set to a number. It is possible to set this parameter using a list of cons cells (or two element lists). Each value specified is effectively added to the set of S_{ji} values for the chunk as if the [add-sji command](#) was called (possibly replacing a value which was previously

set or overriding a default value). The `add-sji` command is the recommended way to set S_{ji} values instead of using this parameter through `sdp`.

- `similarities :=` this reports the similarities (the M_{ki} values) between other chunks k and this chunk, i . The reported value is a list of cons cells where the car of the cell is the name of the chunk k and the cdr is the similarity value M_{ki} . Only chunks k for which a similarity value has been set and the similarity of the chunk with itself are reported. All other M_{ki} values will be the maximum difference (the `:md` parameter setting). This parameter is only relevant when the `:mp` parameter is set to a number. It is possible to set this parameter using a list of cons cells (or two element lists). Each value specified is effectively added to the similarity values for the chunk (possibly replacing a value which was previously set or overriding a default value). The [set-similarities command](#) is the recommended means of setting similarities, but if one wants asymmetric similarities between chunks they must be set explicitly with `sdp` per chunk (`set-similarities` always sets the values symmetrically).
- `last-retrieval-activation :=` the activation that the chunk had the last time that it was attempted to be retrieved. This value is computed during the start-retrieval event and will be updated for all chunks which match the retrieval request. Cannot be changed through `sdp`.
- `last-retrieval-time :=` the time at which the last attempt to retrieve this chunk occurred i.e. the time at which the last-retrieval-activation value was set. Cannot be changed through `sdp`.

Note: because parameter settings are applied in the order provided and there are dependences between the creation-time, reference-count, and reference-list the order in which they are given may affect the resulting values. The ordering to achieve what is typically wanted would be to set creation-time, then reference-count, and then reference-list. That ensures that the creation-time has been updated before any automatic references are generated and that the specified reference-list is not overwritten by one automatically created by the reference-count. It is not required that they be provided in that order however, and one may use other orderings to achieve different resultant values as desired.

If no parameters are provided to `sdp`, then all of the current model's declarative memory chunks' parameters are printed and a list of all the chunk names is returned.

If a chunk or list of chunks is specified as the first parameter to `sdp` then the following parameters are set or retrieved from only those chunks. If no chunk names are provided then the settings are applied to or retrieved from all chunks in DM at the time of the call to `sdp`.

If chunk names are specified but no specific parameters are specified then the parameters for those chunks are printed and the list of those chunk names is returned.

When `sdp` prints the parameters for a chunk its name is printed followed by the parameters which are currently appropriate based on the declarative module's parameter settings to the command output stream.

If any of the chunk names provided are invalid a warning will be printed and the corresponding element of the return list will be **:error**.

If all of the parameters passed to `sdp` (after any chunk names) are keywords, then it is a request for the current values of the parameters named. Those parameters are printed for the chunks specified and a list containing a list for each chunk specified is returned. Each sub-list contains the values of the parameters requested in the order requested and the sub-lists are in the order of the chunks which were requested. If an invalid parameter is requested, then a warning is printed and the value returned in that position will be the keyword **:error**.

If there are any non-keyword parameters in the call to `sdp` and the number of parameters (not counting the chunk names) is even, then they are assumed to be pairs of a parameter name and a parameter value. For all of the specified chunks (or all chunks in DM if none are specified) those parameters will be set to the provided values. The return value will be a list containing a list for each chunk specified. Each sub-list contains the values of the parameters set in the order they were set and the sub-lists are in the order of the chunks which were specified. If a particular parameter value was not of the appropriate type, then a warning is printed and the value returned in that position will be the keyword **:error**.

It is also possible to pass lists of a chunk name and parameter settings to `sdp`. Essentially, each list provided could be formatted as something that could be passed to `sdp` and they will each be processed as appropriate.

If there is no current model or current meta-process at the time of the call, then a warning is displayed and **nil** is returned.

There is one small issue worth noting about using `sdp`. If the `:activation` or `:base-level` value is returned (either because it is explicitly specified or because no parameters were specified and thus it gets returned automatically) that will cause the chunk's activation to be recomputed at the current time if it is not currently at the time of the chunk's last retrieval attempt (the value of the `:last-`

retrieval-time parameter). That activation computation will include the activation noise if there is any. There are two consequences of that. First, multiple calls to `sdp` will likely return different :activation values for a given chunk, even if those calls occur at the same model time. The other consequence is that if there is noise in the activations then if `sdp` has to recompute the chunk activations it will affect the random sequence which will likely change how a model with a set [:seed parameter](#) runs from that point on relative to how it would have run had `sdp` not been called.

Examples:

The examples assume that this model has been defined:

```
(define-model test
  (sgp :esc t :bll .5 :mas 2 :mp 1)
  (chunk-type test slot1 slot2)
  (add-dm (a isa test)
    (b isa test slot1 a slot2 c)
    (c isa test slot1 a slot2 c)
    (d isa test slot2 b)))

1> (sdp)
Declarative parameters for chunk D:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Creation-Time 0.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((D . 2.0) (B . 1.3068528))
:Similarities ((D . 0.0))
Declarative parameters for chunk C:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Creation-Time 0.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((A . 0.9013877) (C . 1.5945349))
:Similarities ((C . 0.0))
Declarative parameters for chunk B:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Creation-Time 0.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((B . 1.3068528) (A . 0.9013877) (C . 0.9013877))
:Similarities ((B . 0.0))
Declarative parameters for chunk A:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Creation-Time 0.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((A . 0.9013877))
:Similarities ((A . 0.0))
(D C B A)

2> (sdp a :permanent-noise)
```

```

Declarative parameters for chunk A:
:PERMANENT-NOISE 0.000
((0.0))

3> (sdp a :permanent-noise .3)
((0.3))

4> (sdp a)
Declarative parameters for chunk A:
:Activation 2.491
:Permanent-Noise 0.300
:Base-Level 2.191
:Creation-Time 0.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((A . 0.9013877))
:Similarities ((A . 0.0))
(A)

5> (sdp (a b) :name :activation)
Declarative parameters for chunk A:
:NAME A
:ACTIVATION 2.491
Declarative parameters for chunk B:
:NAME B
:ACTIVATION 2.191
((A 2.4910133) (B 2.1910133))

6> (sdp-fct '(c))
Declarative parameters for chunk C:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Creation-Time 0.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((A . 0.9013877) (C . 1.5945349))
:Similarities ((C . 0.0))
(C)

7> (sdp-fct '((d b) :sjis))
Declarative parameters for chunk D:
:SJIS ((D . 2.0) (B . 1.3068528))
Declarative parameters for chunk B:
:SJIS ((B . 1.3068528) (A . 0.9013877) (C . 0.9013877))
(((D . 2.0) (B . 1.3068528))) (((B . 1.3068528) (A . 0.9013877) (C . 0.9013877))))

8> (sdp (a :base-level) (b :creation-time -1.0))
Declarative parameters for chunk A:
:BASE-LEVEL 2.191
((2.1910133) (-1.0))

9> (sdp b)
Declarative parameters for chunk B:
:Activation 0.693
:Permanent-Noise 0.000
:Base-Level 0.693
:Creation-Time -1.000
:Reference-Count 1
:Source-Spread 0.000
:Sjis ((B . 1.3068528) (A . 0.9013877) (C . 0.9013877))
:Similarities ((B . 0.0))
(B)

10> (sdp-fct '(:reference-count 3))
((3) (3) (3) (3))

```

```

11> (sdp-fct '((a b)))
Declarative parameters for chunk A:
:Activation 3.590
:Permanent-Noise 0.300
:Base-Level 3.290
:Creation-Time 0.000
:Reference-Count 3
:Source-Spread 0.000
:Sjis ((A . 0.9013877))
:Similarities ((A . 0.0))
Declarative parameters for chunk B:
:Activation 1.792
:Permanent-Noise 0.000
:Base-Level 1.792
:Creation-Time -1.000
:Reference-Count 3
:Source-Spread 0.000
:Sjis ((B . 1.3068528) (A . 0.9013877) (C . 0.9013877))
:Similarities ((B . 0.0))
(A B)

E> (sdp bad-chunk)
#|Warning: BAD-CHUNK does not name a chunk in DM. |#
(:ERROR)

E> (sdp a :bad-parameter)
Declarative parameters for chunk A:
#|Warning: BAD-PARAMETER is not a declarative parameter for chunks. |#
((:ERROR))

E> (sdp)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL

E> (sdp)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module found |#
NIL

```

sjj/add-sji

Syntax:

```

sjj chunk-name-j chunk-name-i -> [ sjj | nil ]
sjj-fct chunk-name-j chunk-name-i -> [ sjj | nil ]
add-sji (chunk-name-j chunk-name-i sjj)* -> ([sjj | :error])*
add-sji-fct ((chunk-name-j chunk-name-i sjj)* -> ([sjj | :error])*

```

Arguments and Values:

chunk-name-j ::= a symbol which names a chunk
chunk-name-i ::= a symbol which names a chunk
sjj ::= a number which is the associative strength (S_{ji} value) from *chunk-name-j* to *chunk-name-i*

Description:

The **sjj** command is used to get the S_{ji} value between two chunks in the current model. It takes two parameters which are the names of the chunk *j* and the chunk *i* respectively, and it returns the S_{ji}

value between them. The S_{ji} value will be either the value determined by the [standard equation](#), the explicit value which has been set using the add-sji command, or the result returned by the [sji-hook](#) if it is specified. The sji-hook function overrides an explicit setting and the default calculation, and an explicitly set value will override the default calculation. If either of the chunk names is invalid then a warning is printed and an S_{ji} of 0.0 is returned. If there is no current model or current meta-process then a warning is printed and **nil** is returned.

The add-sji command is used to specify explicit S_{ji} values between chunks. It can be used to set any number of S_{ji} values at a time. Each parameter to add-sji (or element of the list passed to add-sji-fct) should be a list of three items. Those items are the chunk j, the chunk i, and the S_{ji} value between them respectively. It applies the S_{ji} values in order left to right. Thus, if any pair of items is specified more than once it will be the right most setting for the pair that will be their S_{ji} . It returns a list of the S_{ji} values set in the order they were specified. If any of the lists are not three elements long, have bad chunk names, or an invalid S_{ji} value then that item is ignored for purposes of setting an S_{ji} , a warning is printed, and the corresponding element of the return list will be **:error**.

Examples:

The example assumes this initial model is defined.

```
(define-model sji-demo
  (sgp :esc t :mas 2)
  (chunk-type item slot)
  (add-dm (a isa item slot nil)
    (b isa item slot a)
    (c isa item slot d)
    (d isa item slot c)
    (e isa item slot c)
    (f isa item slot f)
    (g isa item slot f)))

> (sji a b)
1.3068528

> (sji b a)
0.0

> (sji c d)
0.9013877

> (sji d c)
1.3068528

> (sji-fct 'a 'a)
1.3068528

> (sji-fct 'f 'f)
1.5945349

E> (sji-fct 'bad-name 'a)
#|Warning: BAD-NAME does not name a chunk in the current model.|#
0.0
```

```

E> (sji a a)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL

E> (sji a a)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module found |#
NIL

1> (add-sji (a b 2.5) (b a 10))
(2.5 10)

2> (sji a b)
2.5

3> (sji b a)
10

4> (add-sji-fct '((f f 1) (d c 0)))
(1 0)

5> (sji f f)
1

6> (sji d c)
0

7> (add-sji (c d 1.0) (c d 2.0))
(1.0 2.0)

8> (sji c d)
2.0

E> (add-sji a b 1.0)
#|Warning: Bad Sji setting in A |#
#|Warning: Bad Sji setting in B |#
#|Warning: Bad Sji setting in 1.0 |#
(:ERROR :ERROR :ERROR)

E> (add-sji (a bad 1.0))
#|Warning: Bad Sji setting in (A BAD 1.0) |#
(:ERROR)

E> (add-sji-fct '(a b 2))
#|Warning: Bad Sji setting in A |#
#|Warning: Bad Sji setting in B |#
#|Warning: Bad Sji setting in 2 |#
(:ERROR :ERROR :ERROR)

E> (add-sji (a a 1))
#|Warning: get-chunk called with no current model. |#
#|Warning: Bad Sji setting in (A A 1) |#
(:ERROR)

E> (add-sji (a a 1))
#|Warning: get-chunk called with no current meta-process. |#
#|Warning: Bad Sji setting in (A A 1) |#
(:ERROR)

```

similarity/set-similarities

Syntax:

```
similarity item1 item2 -> [ sim | nil ]
similarity-fct item1 item2-> [ sim | nil ]
set-similarities (chunk-name-k chunk-name-i sim)* -> ([sim | :error]*)
set-similarities-fct ((chunk-name-k chunk-name-i sim)* ) -> ([sim | :error]*)
```

Arguments and Values:

item1 ::= any value
item2 ::= any value
chunk-name-k ::= a symbol which names a chunk
chunk-name-i ::= a symbol which names a chunk
sim ::= a number which is the similarity between the specified items

Description:

The similarity command is used to get the similarity value between two items in the current model (the items do not have to be chunks). It takes two parameters which are the items, and it returns the similarity value between them (the M_{ki} from the partial matching equation). The value will be determined either by the [default calculation](#), explicit user settings, or the hook function. The sim-hook function overrides an explicit setting and the default calculation, and an explicitly set value will override the default calculation. If there is no current model then a warning is printed and **nil** is returned.

The set-similarities command is used to specify explicit similarity values between chunks (to use values other than the defaults between non-chunk items one must use the hook function). It can be used to set any number of similarity values at a time. Each parameter to set-similarities (or element of the list passed to set-similarities-fct) should be a list of three items. Those items are the chunk k, the chunk i, and the similarity value between them respectively. It applies the similarity values in order left to right. Thus, if any pair of items is specified more than once it will be the right most setting for the pair that will be their similarity. Note that similarities are set reciprocally with this command, and thus setting the similarity for k,i also sets the same similarity for i,k. It returns a list of the similarity values set in the order they were specified. If any of the lists are not three elements long, have bad chunk names, or an invalid similarity value then that item is ignored for purposes of setting a similarity, a warning is printed, and the corresponding element of the return list will be **:error**. If there is no current model or meta-process then a warning will be printed and all elements in the list will be **:error**.

Examples:

This example assumes that this initial model is defined.

```
(define-model sim-demo
```

```

(sgp :esc t :mp 1)
(add-dm (a isa chunk)
        (b isa chunk)
        (c isa chunk)))

1> (sgp :ms :md)
:MS 0.0 (default 0.0) : Maximum Similarity
:MD -1.0 (default -1.0) : Maximum Difference
(0.0 -1.0)

2> (similarity a b)
-1.0

3> (similarity a a)
0.0

4> (similarity-fct 'a 'c)
-1.0

5> (similarity-fct "STRING" "string")
0.0

6> (similarity-fct "STRING" 'string)
-1.0

7> (similarity-fct 1.5 1.5)
0.0

E> (similarity a b)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL

E> (similarity a b)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module found |#
NIL

1> (set-similarities (a b -.5) (b c -2))
(-0.5 -2)

2> (similarity a b)
-0.5

3> (similarity b a)
-0.5

4> (similarity b c)
-2

5> (set-similarities-fct '((a a .5) (a c -1) (a c -2)))
(0.5 -1 -2)

6> (similarity a c)
-2

7> (similarity a a)
0.5

E> (set-similarities a b .5)
#|Warning: Bad similarity setting in A |#
#|Warning: Bad similarity setting in B |#
#|Warning: Bad similarity setting in 0.5 |#
(:ERROR :ERROR :ERROR)

E> (set-similarities (d e .5))
#|Warning: Bad similarity setting in (D E 0.5) |#

```

```
(:ERROR)

E> (set-similarities (a b 1) (a d 1) (b c 1))
#|Warning: Bad similarity setting in (A D 1) |#
(1 :ERROR 1)

E> (set-similarities (a a 1))
#|Warning: get-chunk called with no current model. |#
#|Warning: Bad similarity setting in (A A 1) |#
(:ERROR)

E> (set-similarities (a a 1))
#|Warning: get-chunk called with no current meta-process. |#
#|Warning: Bad similarity setting in (A A 1) |#
(:ERROR)
```

get-base-level/set-base-levels/set-all-base-levels

Syntax:

```
get-base-level {chunk-name*} -> ([base-level | :error]*)
get-base-level-fct (chunk-name*) -> ([base-level | :error]*)
set-base-levels (chunk-name level {creation-time})* -> ([base-level | :error]*)
set-base-levels-fct ((chunk-name level {creation-time})*) -> ([base-level | :error]*)
set-all-base-levels level {creation-time} -> [t | nil]
```

Arguments and Values:

chunk-name ::= a symbol which names a chunk

base-level ::= a number representing the base-level activation of a chunk

level ::= a number which is the setting for the base-level of *chunk-name*

creation-time ::= a number which represents the time at which *chunk-name* was added to DM

Description:

Get-base-level will return a list of the current base-level activations in the current model for the chunks provided in the same order as they are given. This will result in the base-level being recomputed for those chunks. If a *chunk-name* given does not name a chunk which is in the DM of the current model then the corresponding base-level value will be **:error**. If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Set-base-levels is used to set the base-level activation for chunks in the DM of the current model. For each chunk specified, its base-level is set as described below and if a new creation time is specified that is also set for the chunk. The list of current base-level activations is returned for the chunks specified in the same order as they were given. If a *chunk-name* is invalid, the level is not a number, or the creation time is specified and is not a number then a warning is printed, no change is made to the chunk's parameters and the corresponding base-level returned will be **:error**.

The setting of the chunk's base-level depends on the settings of the `:bll` and `:ol` parameters. If `:bll` is **nil** then the level provided is used directly as the chunk's base-level. If `:bll` is non-**nil** then the setting of the `:ol` parameter determines how the level is used. If `:ol` is **t** then the level is the number of references for the chunk (n in the optimized learning equation). If `:ol` is **nil** then the level specifies how many references the chunk has and a history of the chunk is generated which evenly spaces those references between the current time and the chunk's creation time (which will be the new value if provided). If `:ol` is a number, then the level specifies the number of references for the chunk (the n in the hybrid optimized equation) and a history list is generated which evenly spaces either the value of `:ol` or level (whichever is lesser) references between the current time and the chunk's creation-time.

The `set-all-base-levels` command works like the `set-base-levels` command except that it applies the level and creation-time (if provided) to all chunks in DM of the current model at the time it is called. If it was successful it returns **t**. If there was a problem then a warning is printed and **nil** is returned.

Examples:

```
1> (define-model test-base-levels
    (sgp :esc t :bll nil)
    (add-dm (a isa chunk)
            (b isa chunk)
            (c isa chunk)))

TEST-BASE-LEVELS

2> (get-base-level a b)
(0.0 0.0)

3> (set-all-base-levels 1.5)
T

4> (set-base-levels (c -1))
(-1)

5> (get-base-level a b c)
(1.5 1.5 -1)

1> (define-model test-base-levels-2
    (sgp :esc t :bll .5 :ol t)
    (add-dm (a isa chunk)
            (b isa chunk)
            (c isa chunk)))

TEST-BASE-LEVELS-2

2> (get-base-level-fct '(a b c))
(2.1910133 2.1910133 2.1910133)

3> (set-all-base-levels 4 -1)
T

4> (sdp)
Declarative parameters for chunk C:
:Activation 2.079
```

```

:Permanent-Noise 0.000
:Base-Level 2.079
:Reference-Count 4.000
:Creation-Time -1.000
Declarative parameters for chunk B:
:Activation 2.079
:Permanent-Noise 0.000
:Base-Level 2.079
:Reference-Count 4.000
:Creation-Time -1.000
Declarative parameters for chunk A:
:Activation 2.079
:Permanent-Noise 0.000
:Base-Level 2.079
:Reference-Count 4.000
:Creation-Time -1.000
(C B A)

5> (set-base-levels-fct '((a 2 -10)))
(0.2350018)

6>(get-base-level-fct '(a b))
(0.2350018 2.0794415)

E> (get-base-level bad-name)
(:ERROR)

E> (set-all-base-levels :not-a-number)
#|Warning: Invalid level :NOT-A-NUMBER |#
NIL

E> (set-all-base-levels 1.5 :not-a-number)
#|Warning: Invalid creation-time :NOT-A-NUMBER |#
NIL

E> (set-base-levels (a))
#|Warning: Invalid level in setting (A) |#
(:ERROR)

E> (set-base-levels (a 1.5) (:not-a-chunk 1.5))
#|Warning: :NOT-A-CHUNK does not name a chunk in DM. |#
(1.5 :ERROR)

E> (get-base-level a)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL

E> (set-base-levels (b 3))
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL

E> (set-all-base-levels 10)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL

E> (get-base-level a)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module found |#
NIL

E> (set-base-levels (b 3))
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module found |#
NIL

```

```
E> (set-all-base-levels 0)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module found |#
NIL
```

clear-dm

Syntax:

```
clear-dm -> [ t | nil ]
```

Arguments and Values:

Description:

The `clear-dm` command can be used to remove all chunks from the declarative memory of the current model. It is not recommended for general use, but there may be rare situations where it would be needed. The command returns **t** if the current model's DM was cleared and **nil** if there was no current model, no current meta-process or some other problem was encountered. It will always print a warning that states either that all chunks were cleared or that a problem occurred.

Examples:

```
> (clear-dm)
#|Warning: All the chunks cleared from DM. |#
T
```

```
E> (clear-dm)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module found |#
NIL
```

reset-declarative-finsts

Syntax:

```
reset-declarative-finsts -> nil
```

Arguments and Values:

Description:

The `reset-declarative-finsts` command can be used to remove all of the finst markers from the declarative module of the current model. It takes no parameters and always returns **nil**. If there is no current model or current meta-process then it will print a warning.

This command is not recommended for typical modeling use because the “:recently-retrieved reset” request parameter setting can be used in the procedural requests to accomplish the same thing in a more model-driven manner. However, sometimes it may be necessary or more convenient to do that through the Lisp code accompanying the model.

Examples:

```
> (reset-declarative-finsts)
NIL
```

```
E> (reset-declarative-finsts)
#|Warning: get-module called with no current model.|#
#|Warning: No declarative module found - cannot reset the finsts.|#
NIL
```

merge-dm

Syntax:

```
merge-dm ({{chunk-name}}[doc-string isa chunk-type | isa chunk-type] {slot value}* -> (chunk*))
merge-dm-fct ({{chunk-name}}[doc-string isa chunk-type | isa chunk-type] {slot value}* -> (chunk*))
```

Arguments and Values:

chunk-name ::= a symbol that will be the name of the chunk

doc-string ::= a string that will be the documentation for the chunk

chunk-type ::= a symbol that names a chunk-type in the model

slot ::= a symbol that names a slot for the chunk

value ::= any Lisp value which will be the contents of the correspondingly named slot for this chunk

chunk ::= a symbol which names a chunk that was created

Description:

The merge-dm command functions exactly like the [add-dm command](#) to create new chunks for the model. However, unlike add-dm merge-dm will merge those chunks into the current model’s declarative memory in the same way they would be merged if they had been cleared from a buffer. Thus, if any of the chunks created by merge-dm are equal using the [chunk-equal](#) test with a chunk already in declarative memory that existing declarative memory chunk is strengthened with a new reference at the current time and those two chunks are merged. If a chunk created by merge-dm is not equal to an existing chunk in the current model’s declarative memory then that new chunk is added to declarative memory as if it had been created using add-dm.

If there are dependencies among the chunks created with merge-dm then those chunks will be merged into declarative memory in an order that allows for proper merging of all chunks if such an order

exists. If there are dependencies and no safe order exists a warning will be displayed to indicate that and the chunks will be merged into declarative memory in the order that they are provided. Merge-dm returns a list of the names of the chunks that were created in the order in which they were merged into declarative memory (first chunk returned was the first merged).

If the syntax is incorrect or any of the components are an invalid list describing a chunk then a warning is displayed and no chunk is created for that chunk description, but all valid chunks defined will still be created.

If there is no current model or current meta-process then a warning is displayed, no chunks are created and **nil** is returned.

Examples:

These examples assume that the base-level learning is enabled so that there is a strengthening of activations when additional references to a chunk occur.

```
1> (chunk-type node slot1 slot2)
NODE

2> (add-dm (a slot1 b slot2 c)
          (b slot1 10)
          (c slot2 20))
(A B C)

3> (sdp)
Declarative parameters for chunk C:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Reference-Count 1.000
:Creation-Time 0.000
Declarative parameters for chunk B:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Reference-Count 1.000
:Creation-Time 0.000
Declarative parameters for chunk A:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Reference-Count 1.000
:Creation-Time 0.000
(C B A)

4> (merge-dm (d slot1 e slot2 f)
            (e slot1 10)
            (f slot2 20)
            (x slot1 30))
(E F D X)

5> (sdp)
Declarative parameters for chunk C:
:Activation 2.884
:Permanent-Noise 0.000
```

```

:Base-Level 2.884
:Reference-Count 2.000
:Creation-Time 0.000
Declarative parameters for chunk X:
:Activation 2.191
:Permanent-Noise 0.000
:Base-Level 2.191
:Reference-Count 1.000
:Creation-Time 0.000
Declarative parameters for chunk B:
:Activation 2.884
:Permanent-Noise 0.000
:Base-Level 2.884
:Reference-Count 2.000
:Creation-Time 0.000
Declarative parameters for chunk A:
:Activation 2.884
:Permanent-Noise 0.000
:Base-Level 2.884
:Reference-Count 2.000
:Creation-Time 0.000
(C X B A)

6> (merge-dm-fct '((y)))
(Y)

7E> (merge-dm (g slot1 h)
              (h slot2 g))
#|Warning: Chunks in call to merge-dm have circular references.|#
#|Warning: Because of that there is no safe order for merging and they will be merged in
the order provided.|#
(G H)

E> (merge-dm (isa bad-type-name))
#|Warning: Invalid chunk definition: (ISA BAD-TYPE-NAME) chunk-type specified does not
exist.|#
NIL

E> (merge-dm (a))
#|Warning: get-module called with no current model.|#
#|Warning: Could not create chunks because no declarative module was found|#
NIL

E> (merge-dm (a))
#|Warning: get-module called with no current meta-process.|#
#|Warning: Could not create chunks because no declarative module was found|#
NIL

```

print-activation-trace

Syntax:

```
print-activation-trace time {time-in-ms} -> nil
```

Arguments and Values:

time ::= a number which is the time of a start-retrieval event
time-in-ms ::= a generalized boolean indicating the units for time

Description:

The `print-activation-trace` command works in conjunction with the `:sact parameter` to allow one to print the activation trace information for retrieval requests that occurred during a model run after the model has stopped. If the `:sact` parameter is non-**nil**, then this command will print out the activation trace for the retrieval request which started at the time provided, measured in milliseconds if the `time-in-ms` parameter is true or seconds if it is **nil** or not specified, from the current model in the current meta-process as it would have appeared in the model trace if the `:act parameter` had been set, except that this trace information will be printed to the command output instead of the model output. If the `:sact` parameter is **nil**, the time provided does not correspond to the time of a start-retrieval event, or there is no current model or meta-process then a warning will be printed instead of an activation trace.

Examples:

These examples assume that the fan model from unit 5 of the ACT-R tutorial has been loaded and modified to set the `:sact` parameter to `t`.

```
1> (fan-sentence-model "hippie" "park" t 'person)
...
1.444 ----- Stopped because no events left to process
(1.354 T)

2> (print-activation-trace 0.485)
Chunk PARK matches
Computing activation for chunk PARK
Computing base-level
User provided chunk base-level: 10.0
Total base-level: 10.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (HIPPIE)
      Spreading activation 0.0 from source HIPPIE level 1.0 times Sji 0.0
Total spreading activation: 0.0
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk PARK has an activation of: 10.0
Chunk PARK with activation 10.0 is the best
NIL

3> (print-activation-trace 585 t)
Chunk P3 matches
Chunk P2 matches
Chunk P1 matches
Computing activation for chunk P3
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (PARK HIPPIE)
      Spreading activation 0.0 from source PARK level 0.5 times Sji 0.0
        Spreading activation 0.10685283 from source HIPPIE level 0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
```

```

Adding permanent noise 0.0
Chunk P3 has an activation of: 0.10685283
Computing activation for chunk P2
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (PARK HIPPIE)
    Spreading activation 0.0 from source PARK level 0.5 times Sji 0.0
    Spreading activation 0.10685283 from source HIPPIE level 0.5 times Sji 0.21370566
Total spreading activation: 0.10685283
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P2 has an activation of: 0.10685283
Computing activation for chunk P1
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (PARK HIPPIE)
    Spreading activation 0.10685283 from source PARK level 0.5 times Sji 0.21370566
    Spreading activation 0.10685283 from source HIPPIE level 0.5 times Sji 0.21370566
Total spreading activation: 0.21370566
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P1 has an activation of: 0.21370566
Chunk P1 with activation 0.21370566 is the best
NIL

4E> (print-activation-trace 0.050)
#|Warning: No activation trace information available for time 0.05|#
NIL

E> (print-activation-trace 0.0)
#|Warning: get-module called with no current model.|#
#|Warning: No declarative module available for reporting activation trace.|#
NIL

E> (print-activation-trace 0.0)
#|Warning: get-module called with no current meta-process.|#
#|Warning: No declarative module available for reporting activation trace.|#
NIL

```

print-chunk-activation-trace

Syntax:

```

print-chunk-activation-trace chunk-name time {time-in-ms} ->
                                                                    [nil | total base-level spreading similarity noise]
print-chunk-activation-trace-fct chunk-name time {time-in-ms} ->
                                                                    [nil | total base-level spreading similarity noise]

```

Arguments and Values:

chunk-name ::= a symbol which should name a chunk in the model's declarative memory
time ::= a number which is the time of a start-retrieval event
time-in-ms ::= a generalized boolean indicating the units for time
total ::= a number which is the total activation the chunk had for the retrieval

base-level ::= a number which is the total value of the base-level component of the chunk's activation
spreading ::= a number which is the total spreading activation component of the chunk's activation or **nil** if spreading activation is not enabled
similarity ::= a number which is the total partial matching component of the chunk's activation or **nil** if partial matching is not enabled
noise ::= a number which is the total amount of noise added to the chunk's activation

Description:

The `print-chunk-activation-trace` command works in conjunction with the `:sact parameter` to allow one to print the activation trace information for the retrieval requests that occurred during a model run after the model has stopped. If the `:sact` parameter is non-**nil**, then this command will print out the activation trace for the specified chunk at the time provided, measured in milliseconds if the `time-in-ms` parameter is true or seconds if it is **nil** or not specified, from the current model in the current meta-process similar to how it would have appeared in the model trace if the `:act parameter` had been set, except that this trace information will be printed to the command output instead of the model output. If the `:sact` parameter is **nil**, the time provided does not correspond to the time of a start-retrieval event, or there is no current model or meta-process then a warning will be printed instead of an activation trace. If the provided chunk-name doesn't name a chunk or wasn't an element in declarative memory then the output will indicate it doesn't have any activation information to display.

If there was chunk activation information displayed then this command will return five values. The first value will be the total activation for the chunk. The remaining four values are the primary components of that activation value: base-level activation, spreading activation, partial matching penalty, and noise. The spreading activation and partial matching penalty values will be **nil** if the corresponding mechanism is not enabled for the model. If there is no chunk activation information displayed then the return value of the command is **nil**.

Examples:

These examples assume that the fan model from unit 5 of the ACT-R tutorial has been loaded and modified to set the `:sact` parameter to `t`.

```
1> (fan-sentence-model "hippie" "park" t 'person)
...
1.444 ----- Stopped because no events left to process
(1.354 T)

2> (print-chunk-activation-trace park .485)
Computing activation for chunk PARK
Computing base-level
```

```

User provided chunk base-level: 10.0
Total base-level: 10.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (HIPPIE)
      Spreading activation 0.0 from source HIPPIE level 1.0 times Sji 0.0
Total spreading activation: 0.0
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk PARK has an activation of: 10.0
10.0
10
0.0
NIL
0.0

3> (print-chunk-activation-trace-fct 'p1 585 t)
Computing activation for chunk P1
Computing base-level
Starting with blc: 0.0
Total base-level: 0.0
Computing activation spreading from buffers
  Spreading 1.0 from buffer IMAGINAL chunk CHUNK0-0
    sources of activation are: (PARK HIPPIE)
      Spreading activation 0.10685283 from source PARK level 0.5 times Sji 0.21370566
      Spreading activation 0.10685283 from source HIPPIE level 0.5 times Sji 0.21370566
Total spreading activation: 0.21370566
Adding transient noise 0.0
Adding permanent noise 0.0
Chunk P1 has an activation of: 0.21370566
0.21370566
0.0
0.21370566
NIL
0.0

4> (print-chunk-activation-trace-fct 'p5 0.585)
Chunk P5 did not match the request.
NIL

5> (print-chunk-activation-trace park 0.585)
Chunk PARK was not considered.
NIL

6> (print-chunk-activation-trace not-a-chunk 0.585)
Chunk NOT-A-CHUNK was not considered.
NIL

7E> (print-chunk-activation-trace park 0.0)
#|Warning: No activation trace information available for time 0.0 |#
NIL

8E> (print-chunk-activation-trace chunk 0.0)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative module available for reporting activation trace. |#
NIL

9E> (print-chunk-activation-trace chunk 0.0)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative module available for reporting activation trace. |#
NIL

```

saved-activation-history

Syntax:

saved-activation-history -> (activation-history*)

Arguments and Values:

activation-history ::= (time chunk-name*)

time ::= a number which is the time of a retrieval request for which a history has been saved

chunk-name ::= a symbol which names a chunk for which activation information was stored at time

Description:

The `saved-activation-history` command returns a list which indicates what retrieval information has been saved as a result of the `:sact parameter` being enabled for the current model in the current meta-process. If the `:sact` parameter is enabled then this command will return a list of lists where each sub-list consists of a time in milliseconds and the chunks which were attempted to be retrieved at that time for which the activation information was recorded. There will be a separate sub-list for each time for which activation details are recorded and those lists will be in order based on the times (lowest time first). If the `:sact` parameter is not enable or there is no current model or meta-process then a warning will be printed and the return value will be **nil**.

Examples:

This first example assume that the fan model from unit 5 of the ACT-R tutorial has been loaded and modified to set the `:sact` parameter to `t`.

```
1> (fan-sentence-model "hippie" "park" t 'person)
...
  1.444  -----          Stopped because no events left to process
(1.354 T)
```

```
2> (saved-activation-history)
((235 HIPPIE) (485 PARK) (585 P3 P2 P1))
```

```
E> (saved-activation-history)
#|Warning: No activation trace information available |#
NIL
```

```
E> (saved-activation-history)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative module available for reporting activation trace. |#
NIL
```

```
E> (saved-activation-history)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative module available for reporting activation trace. |#
NIL
```

whynot-dm

Syntax:

whynot-dm *chunk-name** -> (matching-chunk*)

whynot-dm-fct (*chunk-name**) -> (matching-chunk*)

Arguments and Values:

chunk-name ::= a symbol that names a chunk in the current model

matching-chunk ::= a symbol that names a chunk which matched the last retrieval request

Description:

The `whynot-dm` command can be used to determine which chunks in the current model's declarative memory matched the last retrieval request which the declarative module has received and to indicate reasons why a chunk did not get retrieved. If there has been a retrieval request made to the declarative module then the `whynot-dm` command will display the time at which the most recent request occurred along with the display of the request's specification. Then, for each of the chunk names passed to it (or all chunks in the model's declarative memory if no names are provided) it will print out the chunk, its appropriate parameters if subsymbolic computations are enabled, and then whether the chunk matched that request or not.

It returns a list of all the chunks which did match that request at the time it was made (regardless of whether they were passed into `whynot-dm` for display). The list is sorted by the chunks' activations at the time of the request (highest activation first), and if there was a chunk retrieved it will be the first element of the list.

If there is no current model then a warning is printed and **nil** is returned. If an invalid chunk-name is provided it will indicate that in the output.

Examples:

This example assumes the count model from unit 1 of the tutorial has been loaded.

```
1> (reset)
```

```
DEFAULT
```

```
2> (whynot-dm)
```

```
No retrieval request has been made.
```

```
NIL
```

```
3> (run .1)
```

```
...
```

```
0.050  DECLARATIVE  
0.050  PROCEDURAL
```

```
START-RETRIEVAL  
CONFLICT-RESOLUTION
```

```

    0.100  DECLARATIVE          RETRIEVED-CHUNK C
...
    0.100  -----             Stopped because time limit reached
0.1
20
NIL

4> (whynot-dm)
Retrieval request made at time 0.05:
    FIRST 2

FIRST-GOAL
    START 2
    END 4

Declarative parameters for chunk FIRST-GOAL:
:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000

FIRST-GOAL did not match the request

F
    FIRST 5
    SECOND 6

Declarative parameters for chunk F:
:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000

F did not match the request

E
    FIRST 4
    SECOND 5

Declarative parameters for chunk E:
:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000

E did not match the request

D
    FIRST 3
    SECOND 4

Declarative parameters for chunk D:
:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000

D did not match the request

C
    FIRST 2
    SECOND 3

Declarative parameters for chunk C:
:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000
:Last-Retrieval-Activation 0.000
:Last-Retrieval-Time 0.050

C matched the request

```

C was the chunk chosen to be retrieved

B

FIRST 1
SECOND 2

Declarative parameters for chunk B:

:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000

B did not match the request
(C)

5> (whynot-dm b)

Retrieval request made at time 0.05:
FIRST 2

B

FIRST 1
SECOND 2

Declarative parameters for chunk B:

:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000

B did not match the request
(C)

6> (whynot-dm-fct (list 'c))

Retrieval request made at time 0.05:
FIRST 2

C

FIRST 2
SECOND 3

Declarative parameters for chunk C:

:Activation 0.000
:Permanent-Noise 0.000
:Base-Level 0.000
:Last-Retrieval-Activation 0.000
:Last-Retrieval-Time 0.050

C matched the request
C was the chunk chosen to be retrieved
(C)

7> (whynot-dm first-goal-0)

Retrieval request made at time 0.05:
FIRST 2

Chunk FIRST-GOAL-0 is not in the model's declarative memory.
(C)

8E> (whynot-dm :bad-name)

Retrieval request made at time 0.05:
FIRST 2

:BAD-NAME does not name a chunk in the current model.
(C)

E> (whynot-dm)

#|Warning: Whynot-dm called with no current model. |#
NIL

simulate-retrieval-request

Syntax:

simulate-retrieval-request *specification* -> (matching-chunk*)
simulate-retrieval-request-fct (*specification*) -> (matching-chunk*)
simulate-retrieval-request-plus-seed-fct (*specification*) -> (matching-chunk*) seed

Arguments and Values:

specification ::= [chunk-name | {isa chunk-type} {{modifier} slot value}*]

chunk-name ::= a symbol which names a chunk in the current model

chunk-type ::= a symbol which names a chunk-type in the current model

modifier ::= [= | - | < | > | <= | >=]

slot ::= [possible-slot | request-param]

possible-slot ::= a symbol which names a valid slot in the specified chunk-type or any valid slot if no chunk-type provided

request-param ::= a Lisp keyword which is a valid retrieval buffer request parameter

value ::= any Lisp value

matching-chunk ::= a symbol that names a chunk in DM which matched *specification*

seed ::= the value of the :seed parameter in the model after the retrieval process was simulated

Description:

The `simulate-retrieval-request` command can be used to simulate the results of making a request to the retrieval buffer under the current model conditions. If there is a current model in the current meta-process then the *specification* provided is used to create a [chunk-spec](#) which is used to simulate a request to the retrieval buffer. That request will perform the matching and activation calculations based on the current buffer contents for spreading activation and report the results of whether chunks match the request or not and show the activation for those which do similar to the low detail activation trace.

The simulated request does not perform many of the side effects which occur with a normal retrieval request. The simulated request does not trigger calls to the functions set with the [:retrieval-request-hook](#), [:retrieval-set-hook](#), or [:retrieved-chunk-hook](#) parameters. The results of the simulated request are not recorded for use with [why-not-dm](#). The activations computed will include noise, but the model's [:seed parameter](#) will be restored to the value it had prior to those computations. It does however set the activation parameter of the chunks that are matched which can be accessed using the function `chunk-activation` (see [extending chunks](#) for information on using chunk parameters).

It returns a list of all the chunks which match the request specified sorted by the chunks' activations (highest activation first) with any chunk which would have been retrieved as the first element of the

list. However, just because there is a first element in the list does not mean that it would necessarily be retrieved because it may have an activation below the current retrieval threshold.

The `simulate-retrieval-request-plus-seed-fct` function returns the value of the `:seed parameter` after the retrieval process completes as its second value. That is done so that one can use this mechanism to implement a second buffer that accesses the model's declarative memory and update the random state appropriately.

If there is no current model or current meta-process then a warning is printed and `nil` is returned. If an invalid specification is provided it will indicate that in the output and return `nil`.

Examples:

These examples assume the grouped model from unit 5 of the tutorial has been loaded.

```
> (simulate-retrieval-request)
Chunk LIST has the current best activation 0.13675894
Chunk FIRST has activation -0.36428827
Chunk SECOND has activation -0.2693139
Chunk THIRD is now the current best with activation 0.15103722
Chunk FOURTH has activation 0.069821134
Chunk GROUP1 has activation 0.14088637
Chunk GROUP2 has activation 0.05243059
Chunk GROUP3 has activation -0.31867743
Chunk ITEM1 has activation -0.010031511
Chunk ITEM2 has activation 0.11004096
Chunk ITEM3 has activation -0.087834366
Chunk ITEM4 has activation -0.27916592
Chunk ITEM5 is now the current best with activation 0.21117316
Chunk ITEM6 has activation -0.2579252
Chunk ITEM7 has activation 0.18635121
Chunk ITEM8 has activation 0.09208258
Chunk ITEM9 has activation -0.2433808
Chunk GOAL has activation -0.044656888
Chunk ITEM5 with activation 0.21117316 is the best
(ITEM5 ITEM7 THIRD GROUP1 LIST ITEM2 ITEM8 FOURTH GROUP2 ITEM1 ...)
```

```
> (simulate-retrieval-request isa item - name nil)
Chunk ITEM1 has the current best activation -0.36428827
Chunk ITEM2 is now the current best with activation -0.2693139
Chunk ITEM3 is now the current best with activation 0.15103722
Chunk ITEM4 has activation 0.069821134
Chunk ITEM5 has activation 0.14088637
Chunk ITEM6 has activation 0.05243059
Chunk ITEM7 has activation -0.31867743
Chunk ITEM8 has activation -0.010031511
Chunk ITEM9 has activation 0.11004096
Chunk ITEM3 with activation 0.15103722 is the best
(ITEM3 ITEM5 ITEM9 ITEM4 ITEM6 ITEM8 ITEM2 ITEM7 ITEM1)
```

```
> (simulate-retrieval-request-fct '(position third))
Chunk GROUP1 has the current best activation -1.3642883
Chunk GROUP2 is now the current best with activation -0.76931393
Chunk GROUP3 is now the current best with activation 0.15103722
Chunk ITEM1 has activation -0.9301789
Chunk ITEM2 has activation -0.35911363
```

```
Chunk ITEM3 has activation 0.05243059
Chunk ITEM4 has activation -1.3186774
Chunk ITEM5 has activation -0.5100315
Chunk ITEM6 has activation 0.11004096
Chunk ITEM7 has activation -1.0878344
Chunk ITEM8 has activation -0.7791659
Chunk ITEM9 is now the current best with activation 0.21117316
Chunk ITEM9 with activation 0.21117316 is the best
(Item9 GROUP3 ITEM6 ITEM3 ITEM2 ITEM5 GROUP2 ITEM8 ITEM1 ITEM7 ...)
```

```
> (simulate-retrieval-request group1)
Chunk GROUP1 has the current best activation -0.36428827
Chunk GROUP2 has activation -1.7693139
Chunk GROUP3 has activation -1.8489628
Chunk GROUP1 with activation -0.36428827 is the best
(GROUP1 GROUP2 GROUP3)
```

```
E> (simulate-retrieval-request)
#|Warning: get-module called with no current model. |#
#|Warning: No declarative memory module available. Simulate-retrieval-request cannot
perform the request. |#
NIL
```

```
E> (simulate-retrieval-request)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No declarative memory module available. Simulate-retrieval-request cannot
perform the request. |#
NIL
```

Perceptual & Motor modules

The perceptual and motor modules provide a model with a way to interact with a world. The provided perceptual modules allow the model to attend to visual and aural stimuli and the given motor modules provide the model with hands and a voice. The perceptual and motor modules provided with ACT-R are essentially updated versions of the modules which comprised those same components in the ACT-R/PM system. ACT-R/PM was developed by Mike Byrne as a combination of ACT-R, the Visual Interface for ACT-R created by Mike Matessa, and the EPIC cognitive architecture created by David Kieras and David Meyer. Providing a model access to an external world was an important development in the advancement of ACT-R modeling, and that integration lead to many of the changes that came about with the introduction of ACT-R 5. Those components are now fully integrated with ACT-R and ACT-R/PM is no longer a separate system because the perceptual and motor modules which it contained are now integral components of the main ACT-R system.

Unlike the cognitive modules, the perceptual and motor modules each primarily work with chunks that are created using the slots of specific chunk-types that the module defines, and the requests which they respond to are more rigidly specified descriptions of actions. The chunk-types a module defines and any initial chunks which it creates will be listed with each module's description.

These modules also have more complicated internal states than the basic state free and state busy which can be queried for all modules. Each of these modules has three separate internal systems: preparation, processor, and execution. Each of those systems can be queried individually for being busy or free. Different requests to these modules may require the use of different internal systems and thus may not require that all internal states be free before being allowed to progress. This is particularly useful in the motor module to request an action before the previous has completed. How the modules respond to the queries and how that affects the requests which can be made to them vary from module to module.

Before describing the perceptual and motor modules themselves, however, the interface that they have to the world will be described.

The Device Module

The world with which the perceptual and motor modules of an ACT-R model interacts is called a *device*. The device defines the operations which the perceptual modules can use for gathering information and the operators available to the motor modules for manipulating that device. The device module provides a model with its interface to the device and provides the commands to the modeler for installing and configuring a device.

The typical device used for modeling is one where the model is operating a computer. Essentially, the model is sitting in front of a monitor, has its hands on a keyboard/mouse, and the computer has speaker outputs and an input microphone. That is the situation generally assumed by the device module, and the one for which many of the parameters are relevant. However it is not the only device with which a model can be interfaced. Models have also been situated within driving and flight simulation systems, in virtual worlds as provided by the game Unreal Tournament or the RoboCup soccer simulation, and have also used devices that allow them to receive the inputs and control the actions of real robots.

In general, a device can be any Lisp object and the control of the device is handled by defining the appropriate methods for that object. For this section and the description of the following modules, the assumption is that the world with which the model is interacting is the basic computer device created using one of the default devices included with ACT-R. Information on construction new devices can be found in the slides titled “extending-actr” in the docs directory of the distribution and in example code found in the “examples/creating devices” directory.

The system includes devices for the basic GUI classes in LispWorks, ACL for Windows, CCL on Macs, as well as a virtual windowing system that works for models in any Lisp whether or not it provides a GUI interface. There is a set of commands called the AGI (ACT-R GUI Interface) which can be used with any Lisp to create interfaces using those devices, and when used with the ACT-R Environment the AGI will create a real GUI with which a person or model can interact even in a command line only Lisp. The AGI and ACT-R Environment each have their own manual included in the docs directory.

Parameters

:mouse-fitts-coeff

This parameter is the b coefficient in the Fitts's Law equation for aimed movements which is used when the model moves the mouse cursor. The default value is .1 and it can be set to any positive value.

:needs-mouse

This parameter controls whether or not the model will take control of the computer's mouse cursor when interacting with a real interface. If it is set to **t**, then the system will attempt to keep the mouse pointer located where the model has placed it within the current device. The default value is **t**. If the model does not need to use the mouse, but is interacting with a real interface then setting this parameter to **nil** will prevent the system from taking over the mouse cursor.

:pixels-per-inch

This is the number of pixels/inch assumed for the device which the model is looking at. It is used in computations of the size of items in terms of degrees of visual angle. The default value is 72 and it can be set to any positive number.

:process-cursor

This parameter controls whether the mouse cursor should be included as a feature for the vision module when processing a display. If it is set to **t**, then a feature for the mouse will be generated. The default value is **nil**.

:show-focus

This parameter controls whether a red circle is drawn in a real device window to indicate where the model's visual attention is located. If it is set to **t**, then a red circle is drawn. The default is **nil**.

:stable-loc-names

When using the virtual device windows, this parameter determines whether the device sorts the items found in the window before generating the visual features. If the model is not using the virtual device windows, then this parameter has no effect. Setting the parameter to **t**, which is the default value, will force the items to be sorted which means that the same interface display will result in the same feature names each time it is run on all systems – it will be deterministic. If it is set to **nil** then the items in the interface will not be sorted and the names could vary among runs or across systems.

Setting it to **nil** should not affect the model's performance and may improve the time it takes to run the simulation at the potential cost of debugging time needed to compare different model runs.

:trace-mouse

This parameter controls whether or not the device module maintains a record of where the model has moved the mouse. If it is set to **t** then one can use the [get-mouse-trace command](#) to access the history of mouse positions. The default value is **nil**.

:viewing-distance

This is the assumed distance between the model's eyes and the display in inches. It is used in determining the size of items in terms of degrees of visual angle. The default is 15 and it can be set to any positive number.

:vwt

The virtual window trace parameter controls whether the included virtual window device reports interactions with the model in the trace. If the parameter is set to **t**, then virtual window interactions will be printed in the trace surrounded by "<<" and ">>" characters, otherwise no additional output will be generated. The default value is **nil**.

Commands

These are the general commands relating to the device itself. Commands which are related to a specific module's interaction with the device will be described with that module.

install-device

Syntax:

install-device {*device*} -> [*device* | **nil**]

Arguments and Values:

device ::= any Lisp value for which the appropriate device methods have been defined

Description:

The `install-device` command takes one parameter which should be an item for which the appropriate ACT-R device methods have been defined. It makes that item the current device for the current model. The device itself is returned if no problems occurred. If there is no current model or no current meta-process then a warning is printed and **nil** is returned.

A model must have a device installed before it issues requests to the perceptual or motor modules.

Examples:

```
> (install-device (make-instance 'my-device))
#<MY-DEVICE @ #x2b481f72>

> (install-device nil)
NIL

E> (install-device (make-instance 'my-device))
#|Warning: install-device called with no current model. |#
NIL

E> (install-device nil)
#|Warning: install-device called with no current meta-process. |#
NIL
```

current-device

current-device -> [device | nil]

Arguments and Values:

device ::= the currently installed device

Description:

The `current-device` command takes no parameters. It returns the device which has been installed for the current model in the current meta-process. If there is no current model or no current meta-process then a warning is printed and **nil** is returned.

Examples:

```
1> (install-device (make-instance 'my-device))
#<MY-DEVICE @ #x2b5a7972>

2> (current-device)
#<MY-DEVICE @ #x2b5a7972>

E> (current-device)
#|Warning: current-device called with no current model. |#
NIL

E> (current-device)
#|Warning: current-device called with no current meta-process. |#
NIL
```

get-mouse-trace

get-mouse-trace -> [trace | :mouse-trace-off | nil]

Arguments and Values:

trace ::= (<time x-y>*)

time ::= a number in seconds at which a model generated mouse movement occurred

x-y ::= a vector of two numbers indicating the position to which the mouse was moved

Description:

The `get-mouse-trace` command takes no parameters. If tracing of the mouse has been enabled by setting the `:trace-mouse parameter` to `t` then it returns a list of cons which record every time the current model of the current meta-process has moved the mouse and where it was moved to. The car of the cons is the time of the action and the cdr is a vector indicating the x and y coordinates (respectively). If the mouse tracing has not been enabled, then the keyword `:mouse-trace-off` is returned. If there is no current model or current meta-process then a warning is printed and `nil` is returned.

Examples:

```
> (get-mouse-trace)
((0.909 . #(130 160)))
```

```
> (get-mouse-trace)
:MOUSE-TRACE-OFF
```

```
E> (get-mouse-trace)
#|Warning: get-module called with no current model.|#
#|Warning: No device interface found for get-mouse-trace.|#
NIL
```

```
E> (get-mouse-trace)
#|Warning: get-module called with no current meta-process.|#
#|Warning: No device interface found for get-mouse-trace.|#
NIL
```

model-generated-action

model-generated-action -> [model-name | nil]

Arguments and Values:

model-name ::= a symbol which names the model that generated the action

Description:

The model-generated-action command takes no parameters. It can be used in writing the interface methods for a device as a way to determine whether the model performed the action which caused the method to be called, and if so, which model it was if there are multiple models defined. If a model performed the action which triggered the method it returns the name of the model which generated the action. If the method was called other than through a model's action then it returns **nil**. Outside of a method called from a device interaction this command will always return **nil**.

That last point is an important one and puts some limits on where this command can be used. It can only be used in the device interface methods or those functions and methods which are called directly by those methods. If a function or method is not called directly by those methods this command should not be used. In particular, the rpm-window-key-event-handler method and button action functions are called indirectly when using the native interface in ACL and LispWorks because those are called by the Lisp in response to the corresponding system event that was generated by the interface methods. However, it can be used in those methods and functions when working with the virtual windows and the visible windows generated through the ACT-R environment because those are all handled directly instead of going out through a system event.

Examples:

```
> (model-generated-action)
NIL
```

Vision module

The vision module is used to provide a model with information about what can be seen in the current device and provides the model with a visual attention system. The vision module consists of two subsystems, a "where" system and a "what" system. The two subsystems work together, but each has its own buffer and accepts specific requests. One important thing to note is what the vision module does not do – it does not model eye movements. It is a model of visual attention abstracted away from what is occurring with the eyes. There has been work done by Dario Salvucci to create a more detailed vision module which takes into account eye movements. That extension makes the time required for attention shifts dependent on the eccentricity between the requested location and the current point of gaze as well as the frequency of items. That work is instantiated in a system called EMMA (eye movements and movements of attention) which was built as a replacement for the default vision module of ACT-R. That module is included in the extras directory of ACT-R, but is not part of the default system and will not be described here.

The model's visual world

The vision module sees the features that are made available by the current device. The device provides a set of features to which a model may attend. That set of features is referred to as the visicon (visual icon). The features in the visicon are the items which can be found with the where system, and contain basic information about items like their general categorization, location, and color. When one of those features is attended by the what system the model gets access to the more detailed representation reported by the device for the item that was attended. The details as to the information provided for the visicon items and the attended objects is controlled by the device. The default devices include representations for text, button, and line features for the visicon and attended objects that will be used in the examples here.

The Where System

The where system takes requests through the visual-location buffer. A request to the visual-location buffer specifies a set of constraints, and the where system searches for a feature in the visicon which matches those constraints and creates a chunk representing the corresponding item's location if one is found. This is often referred to as "finding a location". The constraints are specified as slot-value pairs in a request which represent visual properties of the object ("color red"), the spatial location of the object ("y position greater-than 153"), coarse temporal information such as whether it recently became visible, and tests of whether the model has previously attended to that location. This is akin

to so-called “pre-attentive” visual processing (Triesman & Gelade, 1980) and supports visual pop-out effects. For example, if the display consists of one green object in a field of blue objects, the time to determine the location of the green object is constant regardless of the number of blue objects.

When a visual-location request is made, if there is a feature in the visicon that meets the constraints, then a chunk representing the location of that object is placed in the visual-location buffer. If multiple objects meet the constraints, then the newest one (the one with the most recent onset time) will be returned. If multiple objects meet the constraints and they have the same onset time, then one will be picked randomly. If there are no objects which meet the constraints, then the buffer will be left empty and an error will be indicated.

Finsts

As noted above, one property of the objects in the visicon which can be tested is whether the item has been previously attended by the model. The vision module is able to keep track of a small number of locations to which it has attended. It does so using a set of markers called finsts (fingers of instantiation) which are limited both in number and in duration. When a location is attended by the model a finst marker is placed upon it. The finst marker will remain until the finst’s duration expires, at which time the location will revert to unattended, or until an attention shift requires a finst and there are none available. If all finsts are in use and a new one is needed then the oldest one which was assigned will be removed (thus forcing that location to revert to unattended) and reused for the newly attended location. The visual finsts work the same as those described for the [declarative module](#) (the declarative module actually had the finsts mechanism added recently as a copy of the visual finsts), but the two systems are completely separate with each module having its own set of finsts.

The What System

The what system takes requests through the visual buffer. Its primary use is to attend to locations which have been found using the where system. A request to the what system entails providing a chunk representing a visual location, which will cause the what system to shift visual attention to that location, process the object located there, and place a chunk representing the object into the visual buffer. If there is more than one object at the location specified when the attention shift completes, only one of them will be encoded and placed into the buffer. The vision module chooses among the objects by using the constraints which led to the where system finding that visual location. Thus, if

the location to be attended to was found based on a constraint of having the color red, and there are three objects at that location, one of which is red, then the red one will be encoded.

The what system has a rudimentary tolerance for movement. That is, if the location chunk provided to be attended specifies a location, but the object which created that feature has moved slightly such that it is no longer at that specific location the vision module will still attend that item if the movement is small. Just how far an object can move and still be encoded is configurable with a parameter, and the default tolerance is 0.5 degrees of visual angle. That means the object can move up to 0.5 degrees of visual angle from the location which was found by the where system and still be processed by the what system.

The basic assumption behind the vision module is that the chunks placed into the visual buffer as a result of an attention operation are episodic representations of the objects in the visual scene. Thus, a chunk with the value "3" represents a memory of the character "3" available via the eyes, not the semantic THREE used in arithmetic—a declarative retrieval would be necessary to make that mapping. Note also that there is no "top-down" influence on the creation of these chunks; top-down effects are assumed to be a result of the system's processing of these basic visual chunks, not anything that's done by the vision module. (See Pylyshyn, 1999 for a clear argument about why it should work this way.)

Re-encoding

Once a location has been attended to, if the visual world changes at that location, the module will automatically update the chunk in the visual buffer. The where system will be busy while it re-encodes the new object (or lack of one) at the currently attended location. This behavior is sometimes undesirable because visual attention cannot be shifted to a new location while it is busy re-encoding a change at the current location which may make a response to new stimuli slower than desired. If this is a problem, it is possible for the model to un-allocate visual attention after it has processed the visual chunk. The [clear request](#) to the visual buffer can be used to make the vision module "stop attending" to the visual scene, and then it will no longer re-encode items until a new attention shift is performed.

Scene change

The vision module can detect when there is a significant change to the visual display. When there is a change to the visicon the module computes the proportion of the items which have changed. If that value is greater-than or equal to a threshold (set with the [:scene-change-threshold parameter](#)) then the

module will signal that there has been a scene change. That signal will only last for a short time (controlled with the [:visual-onset-span parameter](#)), and is made available to the model through the [scene-change query](#) of the visual buffer.

This is the calculation which computes the proportion of items which have changed:

$$Change = \frac{d + n}{o + n}$$

o: The number of features in the scene prior to the update

d: The number of features which have been deleted from the original scene

n: The number of features which are newly added to the scene by the update

Note: if both **o** and **n** are 0 then the change value will also be 0.

Tracking

The vision module has a rudimentary ability to track moving objects. The basic pattern is to attend the object, then issue a request to start tracking. While the module is tracking an object the chunks representing that item in the visual-location and visual buffers will be updated as the object moves, and the where system will remain busy. Tracking will continue until a new request is made of the what system, which could be a clear request to stop attending everything.

Parameters

:auto-attend

This parameter controls whether or not a visual-location request results in an automatically generated request to also move attention to the location which is found. This is designed as a modeling shortcut to allow one to skip productions which make attention shift requests when they will always follow a visual-location request in the model. It does not affect the timing of the model because there is a 50ms delay before the visual request is sent to compensate for the skipped production's firing time. It is off by default (**nil**) but setting it to **t** will enable that functionality.

:delete-visicon-chunks

This parameter controls what happens to chunks which the vision module creates internally for the visicon (they are copies of the chunks returned by the device) when they are no longer needed i.e. not still elements of the visicon after an update occurs. The default value is **t** which means that those

chunks are deleted and the names uninterned through the [purge-chunk command](#). Setting this parameter to **nil** will prevent those chunks from being deleted. Most models should leave this parameter at the default value, but if one is working to extend or modify the operation of the vision module itself it may be necessary to disable this functionality.

:optimize-visual

This parameter controls how text is processed with the default devices. If it is set to **t** (the default value), then each word in text items will be parsed into one feature for the visicon (how words are determined is described with the [add-word-characters command](#)). If it is set to **nil**, then each letter is parsed into multiple features.

There are several options for what features will result from carving up the letters; there is no universally agreed-upon way to do this. The default option is to carve the letters into features consisting of a LED-type representation of the characters of the text. Different feature sets are also available, including Gibson's (1968) set and Briggs & Hochevar's (1975) set. See the [set-char-feature-set command](#) for details on changing the feature set used when `:optimize-visual` is set to **nil**.

:overstuff-visual-location

This parameter controls whether the module can “stuff” a new chunk into the buffer when there is already a previously stuffed chunk in the buffer. The default value is **nil** which means that it will not overwrite a previously stuffed chunk, but if it is set to **t** then it will overwrite a chunk which was stuffed into the visual-location buffer with a new one.

:scene-change-threshold

This parameter controls the smallest proportion of change in the visicon which will result in signaling that the scene has changed. It must be set to a number in the range [0.0 – 1.0] and defaults to .25.

:test-feats

This parameter controls how items in the visicon are compared on successive calls to [proc-display](#) to determine which items are the same between the two calls for purposes of maintaining the finsts, tracking, and scene change information. If it is set to **nil** then the only test performed is whether the same chunk names are used. If it is set to **t** then the items are also compared across all of their features (slot values). The default value is **t**.

Setting it to **nil** can allow for a significant performance improvement in proc-display. However, there are only certain circumstances where setting it to **nil** is safe i.e. if it is set to **nil** in other circumstances it may result in incorrect operation of the module. One safe situation is if all proc-display calls also specify “:clear t”. In that case the screen is always considered as all new items and there is no need to perform the checks between the old and new visicon items. The other situation is if the device’s build-vis-locs-for method(s) always return the same chunks for the visicon items (chunks with the same names). For the included devices the virtual windows meet that criteria, but the Lisp specific devices do not. So, this parameter should only be set to **nil** if proc-display is only called with “clear” screens, the model only uses the virtual windows without modifying the object features explicitly, or if a custom device is installed which satisfies the “same chunk name” constraint on visual items.

:unstuff-visual-location

This parameter lets the modeler specify whether the vision module removes the chunks which it “stuffs” into the visual-location buffer. The default value is **t** which means that if a stuffed chunk is still in the visual-location buffer and has not be modified after the [:visual-onset-span](#) time has passed that chunk will be [erased](#) from the buffer by the vision module. If it is set to a number then it works similar to the setting of **t** except that the value of this parameter specifies the time in seconds after which the erasing occurs instead of the [:visual-onset-span](#) parameter. If it is set to **nil** then the module will not automatically remove a chunk which has been stuffed into the visual-location.

:visual-attention-latency

This parameter specifies how long a visual attention shift will take in seconds. The default value is .085.

:visual-finst-span

This parameter controls how long a finst marker will remain on a feature. It is measured in seconds and default to 3.0.

:visual-movement-tolerance

This parameter controls how far an object can move and still being considered the same object by the vision module without being explicitly tracked. It is measured in degrees of visual angle and defaults to 0.5.

:visual-num-finsts

This parameter controls how many finsts are available to the vision module. It can be set to any positive number and defaults to 4.

:visual-onset-span

This parameter specifies how long an item recently added to the visicon will be marked as new and also for how long a scene change notice will be available. It is measured in seconds, and the default value is 0.5.

Visual-location buffer

The visual-location buffer is used to access the where system of the vision module as described above. In addition to taking requests to find locations, the vision module will also place chunks into the visual-location buffer automatically without a model request (a process referred to as “buffer stuffing”). Whenever there is an update to the visual scene (which is indicated to the model by calling [proc-display](#)), if the visual-location buffer is empty or the [:overstuff-visual-location](#) parameter is set to **t** and a previously “stuffed” chunk is in the visual-location buffer, a location chunk of some visual feature may be placed into the visual-location buffer. The feature which gets “stuffed” into the buffer is chosen based on preferences which can be set either by the modeler using the [set-visloc-default command](#) or directly by the model with a [set-visloc-default request](#). The default preference is for the left-most unattended item. If the [:unstuff-visual-location](#) parameter is not **nil** then the module will also automatically remove a stuffed chunk from the buffer if it is there past the indicated delay time.

Activation spread parameter: `:visual-location-activation`
Default value: 0.0

Queries

‘State busy’ will always be **nil**.

‘State free’ will always be **t**.

‘State error’ will be **t** if the last visual-location request failed to find a matching location and it will be **nil** in all other situations. Once it becomes **t** it will remain **t** until a new visual-location request is made or a clear request is made of the *visual* buffer.

The visual-location buffer has an additional query that allows one to check the attended status of the location represented by the chunk currently in the visual-location buffer.

‘Attended t’ will be **t** if there is a chunk in the visual-location buffer and that location currently has a first marker on it. Otherwise it will be **nil**.

‘Attended nil’ will be **t** if there is a chunk in the visual-location buffer and that location does not currently have a first marker on it. Otherwise it will be **nil**.

‘Attended new’ will be **t** if there is a chunk in the visual-location buffer, that location does not currently have a first marker, and that feature was added to the model’s visicon within the [:visual-onset-span](#). Otherwise it will be **nil**.

Requests

Find location

```
{isa location-chunk-type}  
{ {modifier} valid-slot [value | variable] }*  
{:nearest nearest-spec}  
{:attended [t | nil | new] }  
{:center [vis-loc | vis-obj] }
```

location-chunk-type ::= a symbol which names a chunk-type used for visual location features

modifier ::= [= | - | > | < | >= | <=]

valid-slot ::= the name of a slot which is valid for the location-chunk-type if provided or any chunk otherwise

value ::= any Lisp value, but the symbols **lowest**, **highest** and **current** have special meanings.

variable ::= a Lisp symbol which starts with the character &

nearest-spec ::= [*vis-loc* | **current** | **current-x** | **current-y** | **current-distance** | **clockwise** | **counterclockwise**]

vis-loc ::= a chunk which represents a visual location

vis-obj ::= a chunk which represents a visual object

A find-location request to the visual-location buffer is an attempt to find an item in the visicon. All of the items in the visicon are compared against the specification provided in the request and if there is an item which matches that specification a chunk describing that item is placed into the visual-location buffer. The specification given describes the properties which the item must have in order to match. If a property is not specified then its value is not considered for the matching.

Any of the slots may be specified using any of the modifiers (-, <, >, <=, or >=) in much the same way one specifies a [retrieval request](#). Each of the slots may be specified any number of times. In addition, there are some special tests which one can use that will be described below. All of the constraints specified will be used to find a location in the visicon to be placed into the visual-location buffer. If there is no location in the visicon which satisfies all of the constraints then the visual-location buffer will indicate an error state and set the visual-location buffer's failure flag.

When the slot being tested holds a number it is also possible to use the slot modifiers <, <=, >, and >= along with specifying the value. If the value being tested or the value specified is not a number, then those tests will result in warnings and are not considered in the matching.

You can use the values **lowest** and **highest** in the specification of any slot which has a numeric value. Of the features which match the other constraints the one with the numerically lowest or highest value for that slot will then be the one found. There is one note about using **lowest** and **highest** when more than one slot is specified in that way. First, all of the non-relative values are used to determine the set of items to be tested for relative values. Then the relative tests are performed one at a time in the order provided to reduce the matching set.

It is also possible to use the special value **current** in a slot of the request. That means the value of the slot must be the same as the value for the location of the currently attended object (the one attention was last shifted to with a move-attention request to the visual buffer). If the model does not have a currently attended object (it has not yet attended to anything or has cleared its attention) then the tests with a value of **current** are ignored.

An additional component of the find location requests is the ability to use variables to compare the particular values within a feature to each other in the same way that the LHS tests of a production use variables to match chunks. If a value for a slot in a find location request starts with the character **&** then it is considered to be a variable in the request. The request variables can be combined with the modifiers and any of the other values allowed to be used in the requests.

The `:nearest` request parameter can be used to find the items closest to the currently attended location in some dimension, or closest to some other location. If there are constraints other than `:nearest` specified then they are all tested first. The nearest of the locations that matches all of the other constraints is the one that will be placed into the buffer. There are several options available for using the `:nearest` request parameter. To find the location of the object nearest to the currently attended location (computed as the straight line distance based on the screen-x, screen-y, and distance values in the chunk) we can again use the value **current**. Alternatively, one can specify any location chunk for the nearest test, and the location of the object nearest to that location will be the one returned. It is also possible to find the locations of objects nearest to the current object along a particular axis by specifying **current-x**, **current-y**, or **current-distance**. Finally, one can request locations which are nearest in angular distance relative to an arbitrary center point using either **clockwise** or **counterclockwise** as the nearest specification. The center point used for the calculation is the location specified using the `:center` request parameter (or the location of the object if a visual object is provided as the `:center`). If no `:center` is specified in the request then the most recent center specified by the [set-visual-center-point command](#) is used, or the default center point of 0,0 is used if `set-visual-center-point` has not been called.

If the `:attended` request parameter is specified, that is used as a test of the finsts: `:attended t` means that the item is currently marked with a finst, `:attended nil` means that it is not, and `:attended new` means that it is not currently marked and it has recently been added to the visicon (within the time specified by the [:visual-onset-span parameter](#)).

If there is more than one item which is found as a match, then the one which has been added to the visicon most recently will be the one chosen, and if there is more than one with the same recent onset time, then a random one of those will be chosen.

This request takes no time to return the resulting chunk and will show up with the following events when successful:

```
0.050  VISION          Find-location
0.050  VISION          SET-BUFFER-CHUNK VISUAL-LOCATION LOC1
```

When no location in the visicon matches the request a failure will be indicated in the trace with a `find-loc-failure` event:

```
0.755  VISION          FIND-LOC-FAILURE
```

When the `:auto-attend` parameter is set to `t` a move attention request will follow all successful find location requests. The move-attention event will occur 50ms after the find-location event and the vision module will be busy during the entire time from the find-location request until the attention shift is completed:

```

0.050  VISION          Find-location
0.050  VISION          SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION1-0
0.050  VISION          automatically attending VISUAL-LOCATION1-0
...
0.100  VISION          Move-attention VISUAL-LOCATION1-0
0.185  VISION          Encoding-complete VISUAL-LOCATION1-0 NIL

```

A find-location request completes immediately upon being made.

Set-visloc-default

```

{isa set-visloc-default-type}
set-visloc-default t
{type vis-loc-type}
{{modifier} valid-slot [value | variable]}*
{:nearest nearest-spec}
{:attended [t | nil | new]}
{:center [vis-loc | vis-obj]}

```

`set-visloc-default-type ::=` a symbol which names a chunk-type used for setting default visual location feature information (typically a subtype of some visual location type with the added slot `set-visloc-default`)

`vis-loc-type ::=` a symbol which names a chunk-type used for visual location features

`modifier ::=` [= | - | > | < | >= | <=]

`valid-slot ::=` the name of a slot which is valid for the `set-visloc-default-type` if provided or any chunk otherwise

`value ::=` any Lisp value, but the symbols **lowest**, **highest** and **current** have special meanings.

`variable ::=` a Lisp symbol which starts with the character `&`

`nearest-spec ::=` [vis-loc | **current** | **current-x** | **current-y** | **current-distance** | **clockwise** | **counterclockwise**]

`vis-loc ::=` a chunk which has represents a visual location

`vis-obj ::=` a chunk which represents a visual object

A `set-visloc-default` request allows the model to change the constraints that are used when determining which (if any) chunk from the visicon will be stuffed into the visual-location buffer when the screen is updated. It works the same as the [set-visloc-default command](#) described below.

The slot values provided can be specified in the same way that they can for a find location request with two differences. The first is that this request requires the slot `set-visloc-default` be specified with

a value of **t** which is what distinguishes it from a normal find location request. The other is that a slot named **type** may be specified once to indicate a chunk-type to use when constructing the specification to be tested e.g. this request:

```
+visual-location>
  set-visloc-default t
  type block-locations
  color red
```

would result in this test being used to find possible locations to be stuffed into the buffer:

```
isa block-locations
color red
```

This request does not directly place a chunk into the visual-location buffer. It works essentially as a delayed request – for each future screen change this specification will be used to determine if a chunk should be placed into the visual-location buffer. However, it does initiate an immediate check of the vision using the new specification which could result in a chunk being placed into the buffer at the time of the request.

It will generate an event in the trace which looks like this:

```
0.850  VISION      Set-visloc-default
```

If a chunk is stuffed into the buffer either immediately due to this request or upon a future screen change the trace showing that setting of the chunk in the buffer will indicate that it was unrequested:

```
0.850  VISION      SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0 REQUESTED NIL
```

A set-visloc-default request completes immediately upon being made.

Visual buffer

The visual buffer is used to access the what system of the vision module as described above. It takes requests to attend to locations, track features, or to stop attending to items, and it will hold a chunk representing a visual object in response to requests.

Activation spread parameter: :visual-activation
Default value: 0.0

Queries

‘State busy’ will be **t** between the time any visual request is started and the time it completes. It will also be **t** between when an unrequested re-encoding starts and it completes. It will be **nil** otherwise.

‘State free’ will be **nil** between the time any visual request is started and the time it completes. It will also be **nil** between when an unrequested re-encoding starts and it completes. It will be **t** otherwise.

‘State error’ will be **t** if the last visual request failed or **nil** otherwise. It will not change from **t** to **nil** until a successful request is completed -- any of the available visual requests will reset the error state to **nil** if completed.

‘Scene-change t’ will be **t** if there has been a detected scene change which has not been explicitly cleared within the past :visual-onset-span seconds. Otherwise it will be **nil**.

‘Scene-change nil’ will be **t** if there has not been a detected scene change within the past :visual-onset-span seconds or such a scene change has been explicitly cleared. Otherwise it will be **nil**.

‘Scene-change-value *value*’ will be **t** if *value* is a number and the last scene change had a proportion of change which is greater than or equal to *value*. Otherwise it will be **nil**. This query is intended primarily as a debugging aid so that a modeler can see the last change value via the buffer-status command, but there may be circumstances where it would be useful to test that directly within a model.

The visual buffer can be used to query the internal states of the vision module, but this is generally not needed since there is no benefit to doing so since the requests cannot be “pipelined” by checking for particular subsystems being free.

‘Preparation busy’ will be **t** during the execution of a clear request and it will be **nil** otherwise.

‘Preparation free’ will be **nil** during the execution of a clear request and it will be **t** otherwise.

‘Processor busy’ will be **t** between the time a move-attention request is started and the time it completes and it will be **nil** otherwise.

‘Processor free’ will be **nil** between the time a move-attention request is started and the time it completes and it will be **t** otherwise.

‘Execution busy’ will be **t** between the time of a move-attention or start-tracking request and its completion as well as during an unrequested re-encoding. It will be **nil** otherwise.

‘Execution free’ will be **nil** between the time of a move-attention or start-tracking request and its completion as well as during an unrequested re-encoding. It will be **t** otherwise.

‘Last-command *command*’ The query will be **t** if *command* is a symbol which is the name of the last request received by the vision module (through either the visual-location or visual buffer) or the symbol none, otherwise it will be **nil**. The visual-location requests are named find-location and set-visloc-default, and the visual requests are named move-attention, start-tracking, clear-scene-change, and assign-finst. If there has not been a request, or a clear request has occurred then the last command will be recorded as the symbol none.

Requests

move-attention

```
cmd move-attention  
screen-pos location  
{scale [ phrase | word ]}
```

The move-attention request moves the vision module’s attention to the item at the location given, which must be a chunk specifying a visual location (it must have the slots screen-x and screen-y). If there is an item at that location, then a chunk which represents that item is placed into the visual buffer after the [attention shift time](#) passes. If there is no item there, then the buffer is left empty and the error state of the visual buffer is set. The scale value affects how the default devices text items are parsed. If :optimize-visual is set to **t** (the default), then the basic unit is words (a sequence of adjacent letters of the same type as described in [add-word-characters](#)), but specifying a scale of phrase will have the module attend to an entire phrase as a single item (where a phrase is defined as all words on the same line). If :optimize-visual is **nil**, then the default attention shift is to the letter which contains the sub-letter feature which is attended, but the module can be told to look for words, or phrases using the scale.

This results in the following events being displayed in the trace showing the move-attention event with the location to which attention was shifted and the scale used (shown as NIL here because no scale was specified in that request), the encoding-complete occurring after the [:visual-attention-latency](#) time passes (in this case the default .085 seconds) also showing the location and scale from the request, and then the buffer being set to the chunk that encodes the visual information:

```

0.100  VISION          Move-attention LOC1-0 NIL
...
0.185  VISION          Encoding-complete LOC1-0 NIL
0.185  VISION          SET-BUFFER-CHUNK VISUAL TEXT1

```

If no item is found, then the same first two events will be shown, but there will be no setting of a chunk in the buffer and instead an event will show that no object was found:

```

0.185  VISION          No visual-object found

```

and the state error query will report **t** and the visual buffer's failure flag will be set.

There may also be maintenance events generated which have no trace output that have an action of unlock-device when the move-attention request comes from a production. That is to allow the visual scene to be processed after the request has been received – see the [proc-display command](#).

If a move-attention request is received while the module is currently handling another request, then a warning is printed and the current request is ignored:

```

#|Warning: Attention shift requested at 0.8 while one was already in progress. |#

```

The timing of the encoding-complete event for this request uses the [randomize-time command](#). So, if the [:randomize-time parameter](#) is set to non-**nil** the timing on that event will be randomized accordingly.

A move-attention request is completed when the encoding-complete event happens if the request was valid, or completed immediately if the request could not be processed because of invalid information or because there was already a move-attention request ongoing.

start-tracking

cmd start-tracking

The start-tracking request will make the vision module continue to “track” an item in the visicon as it moves and/or changes over time until the tracking is terminated. If there is an item currently attended by the vision module, then the start-tracking request will keep the vision module's attention focused on that item. Both the visual-location and visual buffers will be updated with changes to that item and if the buffers are empty new chunks will be placed into them representing any changes. The execution state will be busy while the model is tracking. This will show up in the trace as a start-tracking event and there will be subsequent set-buffer-chunk and mod-buffer chunk events to update

the visual-location and visual buffers as necessary (note that the setting is marked as requested **nil** because the chunks are not a direct result of the request):

```

0.185  VISION          Start-tracking
...
0.185  VISION          SET-BUFFER-CHUNK VISUAL-LOCATION LOC3 REQUESTED NIL
0.185  VISION          SET-BUFFER-CHUNK VISUAL TEXT1 REQUESTED NIL
...
0.700  VISION          MOD-BUFFER-CHUNK VISUAL-LOCATION
0.700  VISION          MOD-BUFFER-CHUNK VISUAL
...
1.000  VISION          SET-BUFFER-CHUNK VISUAL-LOCATION LOC11 REQUESTED NIL
1.000  VISION          SET-BUFFER-CHUNK VISUAL TEXT5 REQUESTED NIL

```

If a request to start-tracking is made when there is no currently attended item a warning will be displayed and the module will not become busy:

```
#|Warning: Request to track object but nothing is currently being attended.|#
```

There is one minor issue which may be important in the model when using tracking. For the visual-location buffer to be updated it must either be empty when the start-tracking request is made or it must contain the visual-location chunk which was used when the object was first attended. However, buffer stuffing may place a different chunk into that buffer between the attention shift and the starting of tracking. Thus, if one wants to make sure that both buffers will be updated with tracking there should also be a `-visual-location>` action in the production which makes the start-tracking request.

A start-tracking request is completed immediately upon receipt by the vision module.

clear

[**cmd clear** | **clear t**]

A clear request can be used to make the vision module stop attending to any currently attended object which will stop the re-encoding from occurring until a new item is attended. A clear request will also stop the tracking of an item, clear the error flags if set for either of the vision module's buffers, and clear the change scene notice if one is set. A clear request will make the preparation state busy for 50ms.

Here are the events which will show for a clear request.

```

1.455  VISION          CLEAR
...
1.505  VISION          CHANGE-STATE LAST NONE PREP FREE

```

Note, that after the clear request is done the last-command recorded by the module will be none and not clear – the clear effectively clears the history of its own request as well.

When the clear event occurs the request is considered complete, and any other requests which may have been pending for the module will also be considered complete.

clear-scene-change

cmd clear-scene-change

A clear-scene-change request can be used to clear any pending scene change notice from the module. It will not affect any other operation of the module. A clear-scene-change request will take no time and does not make any of the module’s stages busy.

Here is the event which will show in the trace for a clear-scene-change request.

```
0.485  VISION          CLEAR-SCENE-CHANGE
```

A clear-scene-change request is completed immediately upon receipt by the vision module.

assign-finst

cmd assign-finst

[object obj | location loc]

The assign-finst request can be used to explicitly tag a visual item as having been attended with a finst. The item can be specified using one of the chunks previously returned by the vision module through either buffer. If it is an object chunk returned through the visual buffer then it must be specified using the object slot, and if it is a location chunk from the visual-location buffer it must be specified using the location slot. It takes no time to process the request and the module will not be marked as busy. An event like this will be shown in the trace indicating which slot was used to assign the finst:

```
0.100  VISION ASSIGN-FINST #<VISION-MODULE> OBJECT NIL LOCATION VISUAL-LOCATION0-0-0
```

If the chunk provided represents an item in the current visicon then that item will have a finst marker placed on it in the same way as if it had been explicitly attended. If the item does not represent a feature in the visicon then a warning is printed and no action is performed:

```
#|Warning: X does not name an object or feature in the current visicon. No finst created.|  
#
```

This request does not affect the status of any of the module's queries other than last-command.

An assign-finst request is completed immediately upon receipt by the vision module.

clear-all-finsts

cmd clear-all-finsts

A clear-all-finsts request can be used to clear all of the finst markers from the vision module. It will take no time and does not make any of the module's stages busy.

Here is the event which will show in the trace for a clear-all-finsts request.

```
0.485  VISION          CLEAR-ALL-FINSTS
```

A clear-all-finsts request is completed immediately upon receipt by the vision module.

Chunks & Chunk-types

The vision module creates several chunk-types which may be used in creating visual location and visual object chunks as well as chunk-types for the requests which it accepts:

```
(chunk-type visual-location screen-x screen-y distance kind color value height
              width size)
(chunk-type (char-primitive (:include visual-location)) (kind char-primitive)
              left right)
(chunk-type (set-visloc-default (:include visual-location)) (set-visloc-default t) type)

(chunk-type visual-object screen-pos value status color height width)
(chunk-type (text (:include visual-object)) (text t))
(chunk-type (empty-space (:include visual-object)) (empty-space t))
(chunk-type (oval (:include visual-object)) (oval t))
(chunk-type (cursor (:include visual-object)) (cursor t))
(chunk-type (line (:include visual-object)) (line t) end1-x end1-y end2-x end2-y)
(chunk-type (phrase! (:include visual-object)) (phrase! t) objects words colors)

(chunk-type color color)

(chunk-type vision-command cmd)
(chunk-type (move-attention (:include vision-command)) (cmd move-attention)
              screen-pos scale)
(chunk-type (start-tracking (:include vision-command)) (cmd start-tracking))
(chunk-type (assign-finst (:include vision-command)) (cmd assign-finst) object location)
(chunk-type (clear-scene-change (:include vision-command)) (cmd clear-scene-change))
(chunk-type (clear-all-finsts (:include vision-command)) (cmd clear-all-finsts))
```

It creates many chunks which are used in the specification of requests, in the creation of visual objects, and for the names of the requests if they are not already chunks. All of these chunks are marked as immutable if created by the vision module:

```

(define-chunks
  (lowest name lowest)
  (highest name highest)
  (current name current)
  (current-x name current-x)
  (current-distance name current-distance)
  (current-y name current-y)
  (clockwise name clockwise)
  (counterclockwise name counterclockwise)
  (external name external)
  (internal name internal)
  (text name text)
  (box name box)
  (line name line)
  (oval name oval)
  (char-primitive name char-primitive)
  (new name new)
  (find-location name find-location)
  (move-attention isa move-attention)
  (assign-finst isa assign-finst)
  (start-tracking isa start-tracking)
  (clear-scene-change isa clear-scene-change)
  (set-visloc-default isa set-visloc-default)
  (clear-all-finsts isa clear-all-finsts))

```

The module also defines a set of chunks to name colors if chunks with those names do not exist, and it marks the chunks it creates as immutable:

```

(define-chunks
  (black color black)
  (red color red)
  (blue color blue)
  (green color green)
  (white color white)
  (magenta color magenta)
  (yellow color yellow)
  (cyan color cyan)
  (dark-green color dark-green)
  (dark-red color dark-red)
  (dark-cyan color dark-cyan)
  (dark-blue color dark-blue)
  (dark-magenta color dark-magenta)
  (dark-yellow color dark-yellow)
  (light-gray color light-gray)
  (dark-gray color dark-gray)
  (gray color gray)
  (pink color pink)
  (light-blue color light-blue)
  (purple color purple)
  (brown color brown))

```

Commands

proc-display

Syntax:

```
proc-display {:clear clear} -> [visicon-count | nil]
```

Arguments and Values:

`clear` ::= a generalized boolean which specifies whether to consider the visual scene to be all new
`visicon-count` ::= the number of features which are in the visicon

Description:

The `proc-display` command is used to have the vision module of the current model in the current meta-process process the visual display of the currently installed device. This will create a new set of features for the visicon of the current model. By default or if the `clear` parameter is provided as **nil**, then objects which are considered to be the same as those which were in the previous visicon will retain their finsts. If the `clear` parameter is provided as non-**nil**, then all objects are considered to be new items. Calling `proc-display` will trigger the where system to possibly stuff the visual-location buffer. It also triggers the what system to re-encode the currently attended item. It returns the number of items in the visicon unless there is no current model, no current meta-process, or no device installed for the current model in which case a warning is printed and **nil** is returned.

If `proc-display` is called between the selection and firing of a production which is going to make a request to the visual buffer the vision module will delay the processing of the display until after the request has been received. If multiple calls to `proc-display` for a model are made at the same simulation time a model warning will be printed and the processing of all calls after the first one at that time will be postponed until the model has been run to provide an opportunity for the changes from that first call to be handled.

Because the vision module does not account for perception times one can adjust the timing of when the `proc-display` function is called relative to when visual changes occur in the device to account for that if needed. That is done in the sperling model of unit 3 in the tutorial to account for persistence of vision – the screen blanks after 50ms, but the `proc-display` is postponed for approximately a second to allow the vision module to continue attending to items.

Examples:

```
> (proc-display)
3
```

```
> (proc-display :clear t)
3
```

```
1> (proc-display)
1
```

```
2E> (proc-display)
#|Warning: Proc-display should not be called more than once at the same ACT-R time.|#
```

```
NIL
```

```
E> (proc-display)
#|Warning: Cannot process display--no device is installed. |#
NIL
```

```
E> (proc-display)
#|Warning: proc-display called with no current model. |#
NIL
```

```
E> (proc-display)
#|Warning: proc-display called with no current meta-process. |#
NIL
```

print-visicon

Syntax:

```
print-visicon -> nil
```

Description:

The `print-visicon` command will print out a description of the features which are currently in the visicon of the current model in the current meta-process. Each feature will be printed on a separate line showing several of the default parameters used for visual features: `loc` is the x,y position of the item, `att` is whether or not it has been attended, `kind` is a general description of what is there, `value` is a specific value which the visual object will have, `color` is the color of the item, and `id` is the name of the chunk which represents the location of that feature. It will always return **nil**. If there is no current model or current meta-process it will print out a warning instead of the visicon.

Examples:

```
> (print-visicon)
Loc      Att  Kind      Value      Color      ID
-----  ---  -----  -
( 25 26) NEW  TEXT      "a"        BLACK      VISUAL-LOCATION0
( 35 46) NEW  TEXT      "b"        BLACK      VISUAL-LOCATION1
( 45 26) NEW  TEXT      "b"        BLACK      VISUAL-LOCATION2
NIL
```

```
E> (print-visicon)
#|Warning: get-module called with no current model. |#
NIL
```

```
E> (print-visicon)
#|Warning: get-module called with no current meta-process. |#
NIL
```

remove-visual-finsts

Syntax:

```
remove-visual-finsts {:set-new set-new} {:restuff restuff} -> nil
```

Arguments and Values:

set-new ::= a generalized boolean which specifies whether to set all visual features to the new state
restuff ::= a generalized boolean which specifies whether to check for stuffing of the visual-location buffer after removing the finsts

Description:

The remove-visual-finsts command is used to have the vision module of the current model in the current meta-process remove all of the finsts from the current set of visual features in the visicon. If the set-new keyword parameter is true then the features will all be marked as attended new and have their onset times changed to the current time. If the set-new keyword parameter is **nil** (the default value if not provided) then the features will be either attended **new** or attended **nil** based on their original onset relative to the current time. Those which have not been on the display longer than the module's [visual-onset-span time](#) will be marked as attended **new** and the others will be marked as attended **nil**. If the restuff keyword parameter is true, then it will trigger the where system to possibly stuff the visual-location buffer after resetting the finsts. If the restuff keyword parameter is **nil** (the default value if not provided) then it will not attempt to stuff the visual-location buffer. The command always returns **nil**. If there is no current model or no current meta-process a warning is printed.

This command is not recommended as a plausible mechanism of the vision module, but may be useful in creating some experimental situations which would require reprocessing the whole display otherwise.

Examples:

```
1> (print-visicon)
Loc      Att  Kind      Value      Color      ID
-----  ---  -----  -----  -----  -----
( 25 46) T    TEXT      "v"        BLACK      VISUAL-LOCATION0
NIL

2> (remove-visual-finsts)
NIL

3> (print-visicon)
Loc      Att  Kind      Value      Color      ID
-----  ---  -----  -----  -----  -----
( 25 46) NIL  TEXT      "v"        BLACK      VISUAL-LOCATION0
NIL

4> (remove-visual-finsts :set-new t)
NIL

5> (print-visicon)
Loc      Att  Kind      Value      Color      ID
```

```

-----
( 25 46) NEW TEXT "v" BLACK VISUAL-LOCATION0
NIL

6> (remove-visual-finsts :restuff t)
NIL

7> (run .001)
1.000 VISION SET-BUFFER-CHUNK VISUAL-LOCATION VISUAL-LOCATION0-0 REQUESTED NIL
1.000 PROCEDURAL CONFLICT-RESOLUTION
1.001 ----- Stopped because time limit reached
0.001
6
NIL

E> (remove-visual-finsts)
#|Warning: get-module called with no current model.|#
#|Warning: No vision module found. Cannot remove the visual finsts.|#
NIL

E> (remove-visual-finsts)
#|Warning: get-module called with no current meta-process.|#
#|Warning: No vision module found. Cannot remove the visual finsts.|#
NIL

```

set-visloc-default

Syntax:

```

set-visloc-default {isa chunk-type} {{modifier} slot [value | variable]}*
    {:nearest nearest-loc} {:attended [t | nil | new]} {:center [loc | obj]} -> [ t | nil ]
set-visloc-default-fct ( {isa chunk-type} {{modifier} slot [value | variable]}*
    {:nearest nearest-loc} {:attended [t | nil | new]} {:center [loc | obj]} ) -> [ t | nil ]

```

chunk-type ::= a symbol which names a chunk-type

modifier ::= [= | - | > | < | >= | <=]

slot ::= the name of a slot which must be valid for chunk-type if it is specified

value ::= any Lisp value

variable ::= a Lisp symbol which starts with the character &

nearest-loc ::= [vis-loc | **current** | **current-x** | **current-y** | **current-distance** | **clockwise** |
counterclockwise]

vis-loc ::= a chunk which represents a visual location

Description:

Set-visloc-default is used to set the conditions which are tested in the visicon to determine if a chunk should be stuffed into the visual-location buffer for the current model of the current meta-process. The specification is the same as for a [find location request](#) to the visual-location buffer. That specification is used to create a [chunk-spec](#) which is used with [find-matching-chunks](#) to test the visicon chunks at each screen update using “&” as the variable-char and then filtering the results

based on any of the request parameters and/or relative slot tests (lowest or highest) which are included in the spec.

The default specification is given by:

```
(set-visloc-default screen-x lowest :attended new)
```

The command returns **t** if a new specification is set. If the specification is not valid then no change is made and **nil** is returned. If there is no current model or current meta-process, then it prints a warning and returns **nil**.

Examples:

```
> (set-visloc-default isa visual-location screen-x lowest :attended new)
T
```

```
> (set-visloc-default-fct '(isa visual-location color blue :nearest current))
T
```

```
E> (set-visloc-default isa bad-type)
#|Warning: Element after isa in define-chunk-spec isn't a chunk-type. (ISA BAD-TYPE) |#
#|Warning: Invalid chunk specification. Default not changed. |#
NIL
```

```
E> (set-visloc-default)
#|Warning: No current model. Cannot set visloc defaults. |#
NIL
```

```
E> (set-visloc-default)
#|Warning: No current meta-process. Cannot set visloc defaults. |#
NIL
```

add-word-characters

Syntax:

add-word-characters *char** -> [word-char-list | **nil**]

Arguments and Values:

char ::= a character to be added to those used in the construction of words

word-char-list ::= a list of all the characters which have been specified with **add-word-characters**

Description:

The **add-word-characters** command can be used to adjust how the text items are divided into words by the default text processing mechanisms of the provided devices for the current model in the current meta-process. By default, text is broken into words as collections of sequential characters

which are either all alphanumeric or all not alphanumeric. Thus, this text string “one--word” would be broken into three separate visual items “one”, “--”, and “word”. This command allows one to specify additional characters to group with the alphanumerics. Thus, if #_- were added using this command that string would instead be processed as a single word for the visual representation.

Add-word-characters takes any number of characters to add to those grouped with the alphanumerics. If there is a current model it returns a list of all the additional characters which have been added using this command for that model. If there is no current model or current meta-process then a warning is printed and **nil** is returned.

Any characters added with this command are removed when the model is reset. Thus, this command should probably be included in the model’s definition.

Examples:

These examples assume that the model has a currently installed device that was created with the open-exp-window command which is described in the AGI manual along with the add-text-to-exp-window command used here to create text items.

```
1> (add-text-to-exp-window :text "split_on--dash+only")
#<STATIC-TEXT-VDI @ #x2539c3fa>
```

```
2> (proc-display)
7
```

```
3> (print-visicon)
Loc      Att  Kind      Value      Color      ID
-----  ---  -
( 19  6) NEW  TEXT      "split"    BLACK      VISUAL-LOCATION0
( 40  6) NEW  TEXT      "-"        BLACK      VISUAL-LOCATION1
( 50  6) NEW  TEXT      "on"       BLACK      VISUAL-LOCATION2
( 64  6) NEW  TEXT      "--"       BLACK      VISUAL-LOCATION3
( 85  6) NEW  TEXT      "dash"     BLACK      VISUAL-LOCATION4
(103  6) NEW  TEXT      "+"        BLACK      VISUAL-LOCATION5
(120  6) NEW  TEXT      "only"     BLACK      VISUAL-LOCATION6
NIL
```

```
1> (add-word-characters #\_ #\+)
(#\+ #\_)
```

```
2> (add-text-to-exp-window :text "split_on--dash+only")
#<STATIC-TEXT-VDI @ #x2541925a>
```

```
3> (proc-display)
3
```

```
4> (print-visicon)
Loc      Att  Kind      Value      Color      ID
-----  ---  -
( 29  6) NEW  TEXT      "split_on" BLACK      VISUAL-LOCATION0
( 64  6) NEW  TEXT      "--"       BLACK      VISUAL-LOCATION1
(103  6) NEW  TEXT      "dash+only" BLACK      VISUAL-LOCATION2
NIL
```

```
E> (add-word-characters)
#|Warning: get-module called with no current model. |#
#|Warning: No vision module available could not add new word characters. |#
NIL
```

```
E> (add-word-characters)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No vision module available could not add new word characters. |#
NIL
```

set-visual-center-point

Syntax:

```
set-visual-center-point x y -> [ loc | nil]
```

Arguments and Values:

x ::= a number indicating the x coordinate of the screen center

y ::= a number indicating the y coordinate of the screen center

loc ::= a vector of the current center point in the order x y

Description:

The `set-visual-center-point` command is used to specify the center point used for the current model in the current meta-process when determining the nearest location for a request that specifies `:nearest` as **clockwise** or **counterclockwise** and which does not also specify a center with the `:center` request parameter. The x and y values provided must be numbers and they specify the coordinates of the center to use. The default center point is 0,0 if this command is not used. If x and y are both numbers then that point is used as the center and a vector of those values is returned. If either x or y is not a number or if there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

```
> (set-visual-center-point 100 75)
#(100 75)
```

```
E> (set-visual-center-point 10 :not-a-number)
#|Warning: X and Y values must be numbers for set-visual-center-point. |#
NIL
```

```
E> (set-visual-center-point 10 40)
#|Warning: get-module called with no current model. |#
#|Warning: No vision module available so cannot set center point. |#
NIL
```

```
E> (set-visual-center-point 100 100)
#|Warning: get-module called with no current meta-process. |#
#|Warning: No vision module available so cannot set center point. |#
```

NIL

set-char-feature-set

Syntax:

set-char-feature-set *name* -> [t | nil]

Arguments and Values:

name ::= the name of a character feature set

Description:

The `set-char-feature-set` command is used to change the set of sub-letter features which are added to the visicon for text items when the `:optimize-visual` parameter is set to **nil**. It requires one parameter which is the name given to a feature set that has been created. There are four different feature sets available. Others can be constructed, but there is no documentation on that at this time (see the “`framework/vision-categorization.lisp`” file for the code that creates the current ones). The four feature sets which exist are named `:RM-ORIG`, `:RM-CLEAN`, `:GIBSON`, and `:BRIGGS-HOCEVAR`. Those are based on the features from McClelland & Rumelhart (1981), a slightly modified version of those features, Gibson (1969), and Briggs & Hocevar (1975) respectively. The default set is the `:RM-ORIG` set.

If the name provided does not name an existing feature set or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

```
> (set-char-feature-set :rm-orig)
T
```

```
E> (set-char-feature-set :bad)
#|Warning: Feature set :BAD is unknown|#
NIL
```

```
E> (set-char-feature-set :rm-orig)
#|Warning: No current model. Cannot set a char feature set.|#
NIL
```

```
E> (set-char-feature-set :rm-orig)
#|Warning: No current meta-process. Cannot set a char feature set.|#
NIL
```

Audio module

The audio module provides a model with rudimentary audio perception abilities and is very similar to the [vision module](#). It has both a what and where system and two buffers which accept requests. It has a store of audio events called the *audicon*, and those are transformed into chunks which the model can use by requesting an attention shift to a particular event.

Auditory world

Unlike the visual portion of the device, there is no support for hearing real sounds generated by the computer. Instead, all of the sounds for the model must be simulated using the commands described later. There are commands for generating pure tones, spoken digits, spoken words, and a general command which allows one to specify all of the components of the sound directly. In addition, the model will also hear its own speech which it generates using the [speech module](#).

Sound events occur over time and are not immediately available for processing in the audicon. There are several attributes to a sound event which describe it and control the timing of the model's access to the sound.

Here are the attributes of a sound event:

- Onset: The time at which the sound began.
- Duration: The amount of time that the sound is present.
- Content delay: The amount of time between a sound's onset and when the content of the sound is accessible to the auditory system. No information can be extracted before this time has passed.
- Recode time: The amount of time (after the content is available) that it takes for the auditory system to construct a representation of the sound.
- Content: The value which will be made available to the model once the sound is attended.
- Kind: The basic description of the type of sound. The given commands create kinds of **tone**, **digit**, **word**, and **speech**.
- Location: An indication of where the sound originated. The built-in options are **external** for sounds generated by the simulated sound commands, **self** for words the model speaks, and **internal** for words the model subvocalizes. Other values may be provided when creating custom sounds with the commands described below.

The sound events are only available in the audicon for a short time before they decay. After a sound event ends and the [decay time](#) elapses, the sound event is removed from the audicon.

The Where System

The where system takes requests through the aural-location buffer. A request to the aural-location buffer specifies a set of constraints to test against the detectable events in the audicon. If there is a sound event in the audicon that meets those constraints, then a chunk representing that sound event is placed in the aural-location buffer. If multiple objects meet the constraints, then one will be picked randomly. If there are no sound events which meet the constraints, then the buffer will be left empty and the state error query to the aural-location buffer will be true and the buffer's failure flag will be set.

Like the vision and declarative modules, the audio module maintains a set of finsts which mark items that have been attended, and the attended status of an audio event can be a constraint in a request to the aural-location buffer. However, unlike those other modules, because the audicon is already time limited (audio events decay on their own) there is no limit on the number or duration of the auditory finsts.

The What System

The what system takes requests through the aural buffer. Its primary use is to attend to audio events which have been found using the where system. A request to the what system requires a chunk representing an audio event. In response to the request, the what system will shift aural attention to that audio event, process the sound, and place a chunk representing that sound into the aural buffer.

Like the vision module, the assumption behind the aural module is that the sound chunks placed into the aural buffer as a result of an attention operation are episodic representations of the sounds. Thus, a sound chunk with content "3" represents a memory of hearing the number "3" being spoken, not the semantic THREE used in arithmetic—a declarative retrieval would be necessary to make that mapping.

Parameters

:digit-detect-delay

This parameter controls the content delay time given to digit sounds created with the [new-digit-sound](#) command. It is measured in seconds. It can be set to any non-negative value and the default is 0.3.

:digit-duration

This parameter controls the duration given to digit sounds created with the [new-digit-sound](#) command in seconds. It can be set to any non-negative number and the default is 0.6.

:digit-recode-delay

This parameter controls the recode time given to digit sounds created with the [new-digit-sound](#) command in seconds. It is also the time that it takes for a failure to encode to occur when an audio event is no longer available. It can be set to any non-negative number and the default is 0.5.

:overstuff-aural-location

This parameter controls whether the module can “stuff” a new chunk into the buffer when there is already a previously stuffed chunk in the buffer. The default value is **t** which means that it will overwrite a previously stuffed chunk if new information becomes available, but if it is set to **nil** then it will not overwrite a chunk which was stuffed into the aural-location buffer.

:sound-decay-time

This parameter controls how long sound events will stay in the audicon measured in seconds. It can be set to any non-negative number and the default is 3.0.

:tone-detect-delay

This parameter controls the content delay time given to tone sounds created with the [new-tone-sound](#) command in seconds. It can be set to any non-negative number and the default is 0.05.

:tone-recode-delay

This parameter controls the recode time given to tone sounds created with the [new-tone-sound](#) command in seconds. It can be set to any non-negative number and the default is 0.285.

:unstuff-aural-location

This parameter lets the modeler specify whether the audio module removes the chunks which it “stuffs” into the aural-location buffer. The default value is **nil** which means that the module will not automatically remove a chunk which has been stuffed into the aural-location buffer. If it is set to **t** then if a stuffed chunk is still in the aural-location buffer and has not be modified after the [:sound-](#)

[decay-time](#) time has passed that chunk will be [erased](#) from the buffer by the audio module. If it is set to a number then it works similar to the setting of **t** except that the value of this parameter specifies the time in seconds after which the erasing occurs instead of the `:sound-decay-time` parameter.

Aural-location buffer

The aural-location buffer is used to access the where system of the audio module as described above. In addition to taking requests to find sounds, the audio module will also place chunks into the aural-location buffer automatically without a model request (a process referred to as “buffer stuffing”). Whenever there is a new sound available to the model, if the aural-location buffer is empty or the [:overstuff-aural-location](#) parameter is **t** and there is a previously stuffed chunk in the aural-location buffer, an audio-event chunk of a feature from the audicon may be placed into the aural-location buffer. The feature which gets “stuffed” into the buffer is chosen based on preferences which can be set by the modeler using the [set-audloc-default command](#) or by the model with a [set-audloc-default](#) request. The default preference is for any unattended item in the audicon to be stuffed into the buffer. If the [:unstuff-aural-location](#) parameter is not **nil** then the module will also automatically remove a stuffed chunk from the buffer if it is there past the indicated delay time.

Activation spread parameter: `:aural-location-activation`
Default value: 0.0

Queries

‘State busy’ will always be **nil**.

‘State free’ will always be **t**.

‘State error’ will be **t** if the last aural-location request failed to find a matching audio-event and it will be **nil** in all other situations. Once it becomes **t** it will remain **t** until a new aural-location request is made or the explicit clear request is made of the aural buffer.

The aural-location buffer has two additional queries that allows one to check the attended and finished status of the audio-event represented by the chunk in the aural-location buffer.

‘Attended t’ will be **t** if there is a chunk in the aural-location buffer and that audio-event currently has a first marker on it. Otherwise it will be **nil**.

‘Attended nil’ will be **t** if there is a chunk in the aural-location buffer and that location does not currently have a first marker on it. Otherwise it will be **nil**.

‘Finished t’ will be **t** if there is a chunk in the aural-location buffer and that audio-event has finished playing (its offset time has been reached or passed). Otherwise it will be **nil**.

‘Finished nil’ will be **t** if there is a chunk in the aural-location buffer and that audio-event has not finished playing (its offset time has not been reached yet). Otherwise it will be **nil**.

Requests

find-sound

```
{isa audio-event-chunk-type}  
{ {modifier} valid-slot [value | variable]}*  
{:attended [t | nil]}  
{:finished [t | nil]}
```

audio-event-chunk-type ::= a symbol which names a chunk-type used for audio event features

modifier ::= [= | - | > | < | >= | <=]

valid-slot ::= the name of a slot which is valid for the audio-event-chunk-type if provided or any chunk otherwise

value ::= any Lisp value, but the symbols **lowest** and **highest** have special meanings

variable ::= a Lisp symbol which starts with the character &

A find-sound request is an attempt to find an audio event in the audicon. All of the items in the audicon are compared against the values provided in the request and if there is an item which is detectable and matches that specification an audio-event chunk describing that item is placed into the aural-location buffer. The specification given describes the properties which the item must have in order to match. If a property is not specified then its value is not considered for the matching

Any of the slots may be specified using any of the modifiers (-, <, >, <=, or >=) in much the same way one specifies a [retrieval request](#). Each of the slots may be specified any number of times. In addition, there are some special tests which one can use that will be described below. All of the constraints specified will be used to find an audio event in the audicon to be placed into the aural-

location buffer. If there is no audio event in the audicon which satisfies all of the constraints then the aural-location buffer will indicate an error state and its failure flag will be set.

When the slot being tested holds a number it is also possible to use the slot modifiers `<`, `<=`, `>`, and `>=` along with specifying the value. If the value being tested or the value specified is not a number, then those tests will result in warnings and are not considered in the matching.

You can use the values **lowest** and **highest** in the specification of any slot which has a numeric value. Of the features which match the other constraints the one with the numerically lowest or highest value for that slot will then be the one found. There is one note about using **lowest** and **highest** when more than one slot is specified in that way. First, all of the non-relative values are used to determine the set of items to be tested for relative values. Then the relative tests are performed one at a time in the order provided to reduce the matching set.

An additional component of the find-sound requests is the ability to use variables to compare the particular values within a feature to each other in the same way that the LHS tests of a production use variables to match chunks. If a value for a slot in a find-sound request starts with the character **&** then it is considered to be a variable in the request. The request variables can be combined with the modifiers and any of the other values allowed to be used in the requests.

If the attended value is specified, that is used as a test with the auditory finsts: `:attended t` means that the item is currently marked with a finst (has been attended) and `:attended nil` means that it is not marked (has not been attended).

The finished value can be used to test whether or not the sound is still 'playing' or not. That is determined by the offset time of the sound. If the current time is greater-than or equal to the offset time of the sound then it is considered to be finished, but if the current time is less-than the offset time of the sound it is not finished.

If there is more than one item which is found as a match then a random one of those will be chosen. This request takes no time to return the resulting chunk and will show up with the following events when successful:

```
0.100  AUDIO          FIND-SOUND
0.100  AUDIO          SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0
```

If no sound event which matches the request is found then the buffer will be left empty and the state error query of the buffer will be true. That will result in a find-sound-failure event showing up in the trace:

```
0.050 AUDIO find-sound-failure
```

The aural-location buffer may be set to a chunk even without a request being made. If the aural-location buffer is empty and a new sound event becomes detectable an audio-event chunk which represents it may be stuffed into the aural-location buffer. This is the event which will be generated in the trace indicating a chunk being set without being requested:

```
0.060 AUDIO SET-BUFFER-CHUNK AURAL-LOCATION AUDIO-EVENT0 REQUESTED NIL
```

If a sound event which is not yet finished is placed into the aural-location buffer it will not have values for its offset or duration slots since the sound is still playing. If that audio event is attended with a request to the aural buffer and remains in the aural-location buffer until the sound is completed then those slots will be updated to contain the appropriate values with an event that will look like this in the trace:

```
0.500 AUDIO AUDIO-EVENT-ENDED AUDIO-EVENT0
```

A find-sound request completes immediately when the request is made.

set-audloc-default

```
{isa audio-default-chunk-type}  
set-audloc-default t  
{ {modifier} valid-slot [value | variable]}*  
{:attended [t | nil]}  
{:finished [t | nil]}
```

audio-default-chunk-type ::= a symbol which names a chunk-type used for audio event features

modifier ::= [= | - | > | < | >= | <=]

valid-slot ::= the name of a slot which is valid for the audio-default-chunk-type if provided or any chunk otherwise

value ::= any Lisp value, but the symbols **lowest** and **highest** have special meanings

variable ::= a Lisp symbol which starts with the character &

A set-audloc-default request allows the model to change the constraints that are used when determining which (if any) chunk from the audicon will be stuffed into the aural-location buffer when an audio event becomes detectable. It works the same as the [set-audloc-default command](#).

The slot values provided can be specified in the same way that they can for a find-sound request with the only difference being that this request requires the slot set-audloc-default be specified with a value of **t** which is what distinguishes it from a normal find-sound request.

This request does not directly place a chunk into the aural-location buffer. It works essentially as a delayed request – each time a new audio event becomes detectable this specification will be used to determine if a chunk should be placed into the aural-location buffer.

It will generate an event in the trace which looks like this:

```
0.050    AUDIO                SET-AUDLOC-DEFAULT
```

A set-audloc-default request completes immediately when the request is made.

Aural buffer

The aural buffer is used to access the what system of the audio module as described above. It takes requests to attend to audio events. It attends to the event and creates a chunk to encode it which is placed into the aural buffer.

Activation spread parameter: :aural-activation
Default value: 0.0

Queries

‘State busy’ will be **t** between the time any aural request is started and the time it completes. It will be **nil** otherwise.

‘State free’ will be **nil** between the time any aural request is started and the time it completes. It will be **t** otherwise.

‘State error’ will be **t** if the last aural request failed or **nil** otherwise. It will not change from **t** to **nil** until a successful request is completed – either an attend sound or a clear request will reset it.

The aural buffer can be used to query the internal states of the audio module, but this is generally not needed since there is no benefit to doing so since the requests cannot be “pipelined” by checking for particular subsystems being free.

‘Preparation busy’ will be **t** during the time of a clear request and its completion and during an attend sound request if the sound is not yet available (the detect time has not passed since its entry into the audicon) and it will be **nil** otherwise.

‘Preparation free’ will be **nil** during the time of a clear request and its completion and during an attend sound request if the sound is not yet available (the detect time has not passed since its entry into the audicon) and it will be **t** otherwise.

‘Processor busy’ will always be **nil**.

‘Processor free’ will always be **t**.

‘Execution busy’ will be **t** between the start and end of an attend sound request. It will be **nil** otherwise.

‘Execution free’ will be **nil** between the start and end of an attend sound request. It will be **t** otherwise.

‘Last-command *command*’ *command* should be a symbol which corresponds to one of the audio module’s requests for either buffer, or the symbol none. The query will be **t** if that is the name of the last request received by the audio module otherwise it will be **nil**. Note that it is requests to the module in general which are tested, not just requests to the aural buffer. The possible command value are find-sound, set-audloc-default, sound, or none (representing that there has been no requests made since the module was last cleared).

Requests

clear

[**cmd clear** | **clear t**]

A clear request can be sent to clear the error flags of both of the audio module’s buffers. A clear request will make the preparation state busy for 50ms. These events will show in the trace for a clear request to the aural buffer:

```
0.485  AUDIO          CLEAR
...
0.535  AUDIO          CHANGE-STATE LAST NONE PREP FREE
```

Note, that after the clear request completes the last-command recorded by the module will be none and not clear – a clear request effectively clears the history of its own request as well.

When the clear event occurs the request is considered complete, and any other requests which may have been pending for the module will also be considered complete.

attend sound

event *audio-event*

audio-event ::= a chunk which represents an audio event usually provided by the aural-location buffer

An attend sound request moves the audio module's attention to the sound event provided. If that sound event is still available in the audicon then it will be marked as attended and a chunk which represents that sound will be placed into the aural buffer after the sound event's recode time has passed. The resulting chunk will be constructed from the features of the audio event and any additional features specified when it was created. Its kind slot will have the same value as the audio-event that was used to attend to it. Its content slot will be set with the details of the sound. Its event slot will be the name of the original sound event chunk from the audicon, and any additional slots specified when creating the sound action will also be set.

This results in the following events being displayed in the trace showing the attention shift event with the audio-event listed, the audio-encoding-complete occurring after the sound event's recode time passes (in this case the default .285 seconds for a tone sound), and then the buffer being set to the sound chunk that encodes the visual information:

```
0.150  AUDIO          ATTEND-SOUND AUDIO-EVENT0-1
...
0.435  AUDIO          AUDIO-ENCODING-COMPLETE #<TONE-SOUND-EVT>
0.435  AUDIO          SET-BUFFER-CHUNK AURAL TONE0
```

If there is no corresponding sound available in the audicon, then the buffer is left empty, the error state is set to **t**, and the buffer's failure flag is set. This event will show that no object was found after taking the amount of time required to recode a digit (the time to fail is always the same regardless of the type of sound the event represents):

```
0.585  AUDIO          ATTEND-SOUND AUDIO-EVENT0-1
...
1.085  AUDIO          attend-sound-failure
```

If a sound request is received while the module is currently handling another sound request, then a warning is printed and the newer request is ignored:

```
#|Warning: Auditory attention shift requested at 0.6 while one was already in progress. |#
```

The timing of the audio-encoding-complete and attend-sound-failure events for this request use the [randomize-time command](#). Thus, if the [:randomize-time parameter](#) is set to non-nil the timing on those events will be randomized accordingly.

An attend sound request is completed when the audio-encoding-complete event happens if the request was valid, when the attend-sound-failure event happens if the request did not succeed, or immediately if the request could not be processed because of invalid information or because there was already a request ongoing.

Chunks & Chunk-types

The audio module creates chunk-types which are used for creating and requesting audio event and attended sound chunks, and also a type for making a set-audloc-default request:

```
(chunk-type audio-event onset offset duration pitch kind location id)
(chunk-type (set-audloc-default (:include audio-event)) (set-audloc-default t))
(chunk-type sound kind content event)
```

It creates several chunks which are used in the specification of requests, in the creation of audio events, the creation of sound chunks, and for the names of the requests if they are not already chunks. All of these chunks are marked as immutable if created by the audio module:

```
(define-chunks
  (digit isa chunk)
  (speech isa chunk)
  (tone isa chunk)
  (word isa chunk)
  (highest isa chunk)
  (lowest isa chunk)
  (sound isa chunk)
  (find-sound isa chunk)
  (set-audloc-default isa chunk)
  (internal isa chunk)
  (external isa chunk)
  (self isa chunk)
  (high isa chunk)
  (middle isa chunk)
  (low isa chunk))
```

Commands

new-digit-sound/new-tone-sound/new-other-sound/new-word-sound

Syntax:

```
new-digit-sound digit {[onset | onset time-in-ms]} -> [ t | nil ]
new-tone-sound freq duration {[onset | onset time-in-ms]} -> [ t | nil ]
new-word-sound word {[onset | onset location | onset location time-in-ms]} -> [ t | nil ]
new-other-sound content duration delay recode {[onset | onset location | onset location kind |
onset location kind time-in-ms |
onset location kind time-in-ms feature*]} -> [ t | nil ]
```

Arguments and Values:

digit ::= a number representing the digit to be heard
onset ::= a number which is the time at which the sound will begin in seconds or milliseconds
time-in-ms ::= a generalized boolean which indicates the units used for the onset
freq ::= a number representing the frequency of the tone to be heard
duration ::= a number which is the amount of time between the onset and when the sound stops in seconds
word ::= a string which is the word (or words) which will be heard
location ::= any Lisp value which will be the audio event's location slot value
content ::= any Lisp value which is the content for the attended sound chunk
delay ::= a number which is the content delay for the sound in seconds
recode ::= a number which is the recode time for the sound in seconds
kind ::= a symbol which will be used as the value for the kind slot of both chunks
feature ::= ([:**evt** | :**sound** | :**both**] slot value)
slot ::= a symbol which will be used as a slot name in the chunk indicated by the feature
value ::= a symbol which will be used as the value for the corresponding slot

Description:

The new-*-sound commands are used to create new sound events for the audicon of the current model in the current meta-process. A newly created audio event will be placed into the audicon at the onset time of the sound event with the other properties of that audio event being set based on which command was used to create it. If the time-in-ms parameter is specified as non-nil then a provided onset time is measured in milliseconds, and if it is not provided or nil the provided onset time will be measured in seconds. Here is how those values are set based on the command:

new-digit-sound

Onset: as provided or the current time if not provided
Duration: set using [randomize-time](#) on the value of `:digit-duration`
Content delay: set using [randomize-time](#) on the value of `:digit-detect-delay`
Recode time: the value of `:digit-recode-delay`
Content: the provided digit
Kind: **digit**
Location: **external**

new-tone-sound

Onset: as provided or the current time if not provided

Duration: as provided

Content delay: set set using [randomize-time](#) on the value of [:tone-detect-delay](#)

Recode time: the value of [:tone-recode-delay](#)

Content: the provided frequency

Kind: **tone**

Location: **external**

new-word-sound

Onset: as provided or the current time if not provided

Duration: computed using the [get-articulation-time command](#) of the speech module

Content delay: set using [randomize-time](#) on the value of [:digit-detect-delay](#)

Recode time: the maximum between the duration/2 and the duration - .15 seconds

Content: the provided string

Kind: **word**

Location: as provided, or **external** if not provided

new-other-sound

Onset: as provided or the current time if not provided

Duration: as provided

Content delay: as provided

Recode time: as provided

Content: as provided

Kind: as provided, or **speech** if not provided

Location: as provided, or **external** if not provided

Other slots: the event and/or sound chunk will have additional slots set as provided

Each new sound will generate a maintenance event which will not be shown in the trace. It will have an action of `stuff-sound-buffer` and no parameters. It will occur at the time the sound event becomes detectable, and this is where the testing is done to determine whether or not a new sound should be stuffed into the aural-location buffer.

If a sound is successfully created and added to the audicon of the model then **t** is returned. If there is no current model or meta-process or an invalid parameter is provided a warning will be displayed and **nil** will be returned.

Examples:

```
> (new-tone-sound 400 .75)
T
```

```

> (new-tone-sound 400 .5 1000 t)
T

> (new-digit-sound 6 1.0)
T

> (new-word-sound "Hello" .25 'left)
T

> (new-other-sound 'new-content .5 .2 .15 (mp-time) 'external 'other nil '(:both value t)
      '(:evt volume large))
T

E> (new-tone-sound 100 .5 'not-a-time)
#|Warning: Onset must be a number. No new tone sound created.|#
NIL

E> (new-tone-sound 'high .2)
#|Warning: Freq must be a number. No new tone sound created.|#
NIL

E> (new-digit-sound 3)
#|Warning: No current model found. Cannot create a new sound.|#
NIL

E> (new-digit-sound 1)
#|Warning: No meta-process found. Cannot create a new sound.|#
NIL

```

print-audicon

Syntax:

print-audicon -> nil

Description:

The print-audicon command will print out a description of the features which are currently in the audicon of the current model of the current meta-process. Each feature will be printed on a separate line showing several of the default characteristics of the sound event. It will always return **nil**. If there is no current model or current meta-process then it will print out a warning instead.

Examples:

```

1> (new-tone-sound 400 .5)
T

2> (new-word-sound "Hi" 1.0 'external)
T

3> (print-audicon)
Sound event      Att  Detectable  Kind  Content  location  onset  offset  Sound ID
-----
AUDIO-EVENT0    NIL  NIL        TONE  400      EXTERNAL  0      500    TONE0
NIL

```

4> (run-full-time 1.0)

```
1.0
6
NIL
```

```
5> (print-audicon)
Sound event  Att  Detectable  Kind  Content  location  onset  offset  Sound ID
-----
AUDIO-EVENT0  NIL  T          TONE  400      EXTERNAL  0      500    TONE0
AUDIO-EVENT1  NIL  NIL        WORD  "Hi"     EXTERNAL  1000   1100   WORD0
NIL
```

```
E> (print-audicon)
#|warning: get-module called with no current model. |#
#|warning: No audio module found |#
NIL
```

```
E> (print-audicon)
#|warning: get-module called with no current meta-process. |#
#|warning: No audio module found |#
NIL
```

set-audloc-default

Syntax:

```
set-audloc-default {isa chunk-type} {{modifier} slot [value | variable]}*
    {:attended [t | nil]} {:finished [t | nil]} -> [ t | nil ]
set-audloc-default-fct ( {isa chunk-type} {{modifier} slot [value | variable]}*
    {:attended [t | nil]} {:finished [t | nil]} ) -> [ t | nil ]
```

chunk-type ::= a symbol which names a chunk-type

modifier ::= [= | - | > | < | >= | <=]

slot ::= the name of a slot which must be valid for chunk-type if it is specified

value ::= any Lisp value

variable ::= a Lisp symbol which starts with the character &

Description:

The set-audloc-default command is used to set the specification used when testing the items in the audicon to determine if an audio event chunk should be stuffed into the aural-location buffer for the current model of the current meta-process. The parameters provided are used to create a [chunk-spec](#) that will be tested against the set of chunks in the audicon using [find-matching-chunks](#) after it has been filtered for attended and/or finished items. The default specification is given set with:

```
(set-audloc-default :attended nil)
```

The command returns **t** if a new specification is set. If there is no current model or current meta-process then it prints a warning and returns **nil**. If the parameters provided do not properly represent a valid chunk-spec then no changes are made to the current default spec and **nil** is returned.

Examples:

```
> (set-audloc-default isa audio-event onset lowest kind tone)
T
```

```
> (set-audloc-default-fct (list '- 'location 'self))
T
```

```
E> (set-audloc-default isa sound slot)
#|Warning: Chunks extended with slot SLOT during a chunk-spec definition. |#
#|Warning: Invalid specs in call to define-chunk-spec - not enough arguments |#
#|Warning: Invalid chunk specification. Default audloc not changed. |#
NIL
```

```
E> (set-audloc-default :attended t :attended nil)
#|Warning: The :attended and :finished specification for set-audloc-default can only be
specified at most once. |#
#|Warning: Audloc defaults not changed. |#
NIL
```

```
E> (set-audloc-default)
#|Warning: No current model. Cannot set audloc defaults. |#
NIL
```

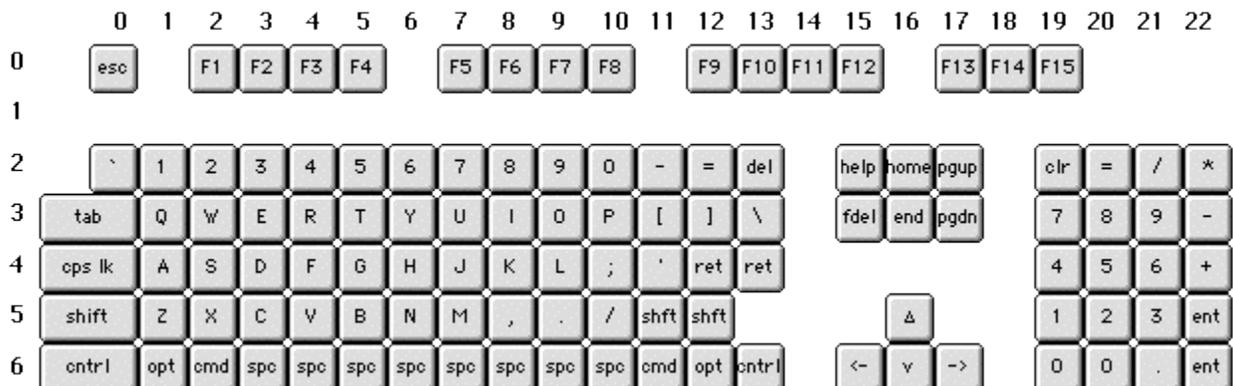
```
E> (set-audloc-default)
#|Warning: No current meta-process. Cannot set audloc defaults. |#
NIL
```

Motor module

The motor module functions as a model's hands. It is conceptually based on EPIC's Manual Motor Processor (Kieras & Meyer, 1996) and is quite similar in many respects. It provides a model with the ability to operate a virtual keyboard and mouse by default, but it is possible to extend the actions and devices with which the model can interact. It has one buffer through which it accepts requests and it does not generate any chunks in response.

Physical world

The model's hands are assumed to be operating a keyboard having this two-dimensional layout:



The mouse has one button and is controlled by the model's right hand when used (considered to be located at location (28 2) relative to the keyboard layout shown above). The default devices accept the input from the model's virtual keyboard and mouse and pass it along as if it was from a real user (including moving the actual mouse cursor for some devices). By default, the model's hands start at the home row positions – left and right index fingers positions over the F and J keys respectively, with the other fingers positioned over the correspondingly adjacent keys and the thumbs placed two rows down and one row “in” from the index fingers. The model's starting hand positions default to the same as the corresponding index fingers. Thus, the starting position for the left hand is (4 4) and for the right hand is (7 4).

Operation

In general, movement requests require specification of a movement type, called a style (which is specified by the cmd slot of the request) and one or more parameters, such as the hand/finger that is to make the movement. The motor module includes several movement styles based on EPIC's Motor Processor. They are:

- Punch. This is a simple downstroke followed by an upstroke of a finger, for pressing a key that is already directly below a given finger.
- Peck. This is a directed movement of a finger to a location followed by a keystroke, all as one continuous movement.
- Peck-recoil. Same as "peck" above, but the finger moves back to the location at which it started its movement.
- Ply. Moves a device (e.g. a mouse) to a given location in space.
- Point-hand. Moves the hand to a new location.

There are also higher level actions which are built out of these basic movement styles. For example, there is a move-cursor style which is translated into a "ply" movement for the right hand, and a press-key style which translates the key to be pressed into either a "punch" or a "peck-recoil" for the appropriate hand and finger with the assumption that the hands are starting from the home position.

The motor module does not place any chunks into its buffer in response to requests. The buffer should always be empty, and it is the state of the module that is most important to the model in this case. As described above for all the perceptual and motor modules, the motor module has three internal states: preparation, processor, and execution. When a request is received by the motor module, it goes through three phases: preparation, initiation, and execution which correspond to those internal states.

In the preparation phase, it builds a list of "features" which guide the actual movement. The amount of time that preparation takes depends on the number of features that need to be prepared - the more that need to be prepared, the longer it takes. The motor module maintains a history of the last set of features that it prepared. The actual number of features that need to be prepared depends upon two things: the complexity of the movement to be made and the difference between that movement and the previous movement. On one end of the scale, the motor module is simply repeating the previous movement, then all the relevant features will already be prepared and do not require preparation. On the other end, a request could specify a movement that requires five features (the most for any of the included styles) which do not overlap with the features in the motor module's history, in which case all five features would need to be prepared.

Movement features are organized into a hierarchy, with the movement style at the top of the hierarchy. If the movement style changes, then all of the features required for a movement must be prepared. If the movement styles are the same but the hand for the movement differs, all features at and below the hand level require preparation. All actions will have a style and a hand, but below that the features vary by style. If the action requires a finger, then it is the next level in the hierarchy. Any other features for an action are considered equal and at the bottom of the hierarchy.

When feature preparation is complete, the motor module makes the specified movement. The first 50 ms (by default) of the movement is movement initiation. During this interval, the preparation state becomes free and the processor and execution states become busy. After initiation ends, the processor state becomes free. The amount of time that a movement takes to execute depends on the type and possibly the distance the movement will traverse. Simple movements have a minimum execution time and more complex movements (such as pointing with the mouse) have a longer execution time based on Fitts's Law.

Fitts's Law

$$T = b * \log_2(D/W + 0.5)$$

T := the time of the movement in seconds

b := a parameter dependent on the type of motor action

D := the distance to the target

W := the width of the target

The motor module can only prepare one movement at a time (though it can be preparing features for one movement while executing another movement). If the motor module is in the process of preparing a movement and another request is sent, the later request will be ignored and the motor module is said to be "jammed." When the module is jammed it will output a warning to indicate when the jamming occurred like this:

```
#|Warning: Module :MOTOR jammed at time 0.68 |#
```

The way to avoid jamming, as with all modules, is to test the state of the module before making a request. Testing that state free is true will avoid jamming the module, but it is possible to issue motor requests faster than that – because the state will not be free until the previous request has been completed. Instead, one only needs to test that the preparation state is free to be able to issue a new request to the motor module. A slightly more conservative approach would be to test the processor state because it is still busy during the initiation time of the last movement request.

Finally, it is possible to prepare movements in advance without executing them, and then execute them later. For instance, let's say the model is a subject doing a simple reaction time task: whenever anything appears on the screen, press the key under the right index finger. The time at which something will appear is unknown, but to minimize the model's response time, the movement to press the key could be prepared in advance and then executed as soon as the visual object is detected.

Parameters

:cursor-noise

If this is set to **nil** (which is the default) then the mouse movements made by the model will be to the exact pixel location requested. If it is set to **t** then there will be noise added to the location to which the cursor is moved.

:default-target-width

This is the effective width, in degrees visual angle, of targets with undefined widths when computing the Fitts's law computation. The default value is 1.0.

:incremental-mouse-moves

When set to **t**, this will update the mouse location approximately every 50 ms when it is in motion. When set to **nil** (which is the default value) it will update the mouse location only at the end of each mouse move. If it is set to a number then that is considered as a time in seconds between the updates and replaces the time of .05 used when it is set to **t**. The incremental positions are calculated along the line between the old and new points using a minimum jerk velocity profile.

:min-fitts-time

The minimum movement time in seconds required to perform an aimed movement (one for which the Fitts's law timing is applied). The default value is 0.1.

:motor-burst-time

This is the minimum time required for the execution of any motor module movement in seconds. the default is .05.

:motor-feature-prep-time

The time in seconds that it takes to prepare each movement feature. The default is .05.

:motor-initiation-time

This is the length of the initiation time for motor actions in seconds. The default is .05.

:peck-fitts-coeff

The b coefficient in the Fitts's law equation for the timing of peck style movements. The default is .075.

Manual Buffer

The motor module does not place any chunks into the manual buffer - it is only used for requests.

Almost all of the timing for the requests to the manual buffer (everything except for a clear request) are randomized using [randomize-time](#).

Activation spread parameter: :manual-activation
Default value: 0.0

Queries:

'State busy' will be **t** when any of the internal states of the module (listed below) also report as being busy. Essentially, it will be **t** while there is any request to the manual buffer that has not yet completed. It will be **nil** otherwise.

'State free' will be **t** when all of the internal states of the module (listed below) also report as being free. Essentially, it will be **t** only when all requests to the manual buffer have completed. It will be **nil** otherwise.

'State error' will always be **nil**.

Unlike the perceptual modules, the internal states of the motor module may be useful to track in a model because it is possible to send new requests while the module is partially busy. Only the preparation stage of the module needs to be free to avoid jamming in most situations. For each request that is received the module will progress through three stages as described above: preparation, initiation, and execution.

‘Preparation busy’ will be **t** after a request has been received and until it completes the preparation of the features needed for that request which depends on how many features there are and whether or not they overlap with the last features prepared. It will be **nil** otherwise.

‘Preparation free’ will be **nil** after a request has been received and until it completes the preparation of the features needed for that request which depends on how many features there are and whether or not they overlap with the last features prepared. It will be **t** otherwise.

‘Processor busy’ will be **t** while preparation is busy and will continue to be **t** after the features have been prepared until the additional initiation time (set with the [:motor-initiation-time parameter](#)) has passed. It will be **nil** otherwise.

‘Processor free’ will be **nil** while preparation is busy and will continue to be **nil** after the features have been prepared until the additional initiation time (set with the [:motor-initiation-time parameter](#)) has passed. It will be **t** otherwise.

‘Execution busy’ will be **t** once the preparation of a request’s features has completed and will remain **t** until the time necessary to complete the action has passed. It will be **nil** otherwise.

‘Execution free’ will be **nil** once the preparation of a request’s features has completed and will remain **nil** until the time necessary to complete the action has passed. It will be **t** otherwise.

‘Last-command *command*’ *command* should be a symbol which corresponds to one of the manual buffer’s requests or the symbol none. The query will be **t** if that is the name of the last request received by the manual buffer otherwise it will be **nil**.

Here is a summary indicating the state transitions for a single movement request assuming that the module is entirely free at the start of the request:

Preparation state	Processor state	Execution state	When
FREE	FREE	FREE	Before event arrives
BUSY	BUSY	FREE	When event is received
FREE	BUSY	BUSY	After preparation of movement
FREE	FREE	BUSY	After initiation movement
FREE	FREE	FREE	When movement is complete

Requests

clear

[**cmd clear** | **clear t**]

A clear request can be sent to clear the history of prepared features. A clear request varies from the other requests in that it will only make the preparation state busy for 50ms and not the processor or execution states. These events will show in the trace for a clear request to the manual buffer:

```
0.050  MOTOR          CLEAR
...
0.100  MOTOR          CHANGE-STATE LAST NONE PREP FREE
```

Note, that after the clear request completes the last-command recorded by the module will be none and not clear – the clear has effectively cleared the history of its own request as well.

When the clear event occurs the request is considered complete, and any other requests which may have been pending for the module will also be considered complete.

punch

cmd punch
hand [*left* | *right*]
finger [*index* | *middle* | *ring* | *pinkie* | *thumb*]

This request will execute a punch action for the specified finger on the specified hand. This will result in pressing the key that is directly under that finger (or the mouse button if it is the right index finger and the hand is currently located on the mouse). The time to execute this action is controlled by the initiation and burst times set for the module. These are the actions which will be shown in the trace for a punch action indicating the request being received, the preparation of the features completing, the initiation time having passed, the actual striking of the key which is under that finger currently (showing the coordinates on the virtual keyboard), and the time to finish the execution of the action (returning the finger to a position where it is ready to act again):

```
0.050  MOTOR          PUNCH HAND LEFT FINGER INDEX
...
0.200  MOTOR          PREPARATION-COMPLETE
...
```

```

0.250  MOTOR          INITIATION-COMplete
...
0.260  MOTOR          OUTPUT-KEY #(4 4)
...
0.350  MOTOR          FINISH-MOVEMENT

```

The punch request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid or the module is jammed.

click-mouse

cmd click-mouse

If the model's right hand is located on the virtual mouse, then this request will result in a punch action (as described above) by the right index finger pressing the primary mouse button. These are the events which one will see for such an action showing the progression through the stages of the motor module and the pressing of the mouse button (which is considered to be located at (28 2) relative to the virtual keyboard):

```

0.050  MOTOR          CLICK-MOUSE
...
0.200  MOTOR          PREPARATION-COMplete
...
0.250  MOTOR          INITIATION-COMplete
...
0.260  MOTOR          OUTPUT-KEY #(28 2)
...
0.350  MOTOR          FINISH-MOVEMENT

```

If the model's right hand is not on the virtual mouse then a warning will be printed and no action taken:

```
#|warning: CLICK-MOUSE requested when hand not at mouse! |#
```

The click-mouse request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid, the hand is not on the mouse, or the module is jammed.

peck

```

cmd peck
hand [ left | right ]
finger [ index | middle | ring | pinkie | thumb ]
r distance
theta direction

```

This request will result in the model making a peck style movement with the indicated finger and hand. That finger will be moved the specified distance and direction from where it currently is and then press the key at that new location. The finger will then remain over that new key until moved elsewhere. The distance is measured in “keys” on the virtual keyboard which is the distance between two adjacent keys in the same row or the same column. The direction is an angle measured in radians with 0 being movement along the x axis to the right and increasing with clockwise rotation. The time taken to execute this action is controlled by [Fitts’s law](#) as described above. These are the events which one will see for such an action showing the progression through the stages of the motor module and the pressing of the corresponding key location on the virtual keyboard (in this case moving one key to the right from the default home location):

```

0.050  MOTOR                PECK HAND LEFT FINGER INDEX R 1 THETA 0
...
0.300  MOTOR                PREPARATION-COMPLETE
...
0.350  MOTOR                INITIATION-COMPLETE
...
0.450  MOTOR                OUTPUT-KEY #(5 4)
...
0.500  MOTOR                FINISH-MOVEMENT

```

There are no constraints on how far or in which direction a model’s finger can move relative to the other fingers. Thus, at this time, the model may be able to make finger movements which would be impossible for a real person to make and it is up to the modeler to consider such issues.

If the movement takes the model’s finger to an invalid location for a key then it effectively presses a key with the value **nil** and a warning will be printed:

```
#|Warning: Invalid key location pressed #(4 7) |#
```

The peck request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid or the module is jammed.

peck-recoil

```

cmd peck-recoil
hand [ left | right ]
finger [ index | middle | ring | pinkie | thumb ]
r distance
theta direction

```

A peck-recoil is effectively the same as a [peck](#), except that the finger is returned to where it started after the key has been pressed. The indicated finger on the specified hand is moved the specified

distance and direction from where it currently is and then it presses the key at that new location. The finger will then return to where it was prior to the peck-recoil request. The distance is measured in “keys” on the virtual keyboard which is the distance between two adjacent keys in the same row or the same column. The direction is an angle measured in radians with 0 being movement along the x axis to the right and increasing with clockwise rotation. The time taken to move the finger is controlled by [Fitts’s law](#) as described above. These are the events which one will see for such an action showing the progression through the stages of the motor module and the pressing of the corresponding key location on the virtual keyboard (in this case moving one key up from the default home location):

```
0.050  MOTOR                PECK-RECOIL HAND LEFT FINGER INDEX R 1 THETA -1.57
...
0.300  MOTOR                PREPARATION-COMPLETE
...
0.350  MOTOR                INITIATION-COMPLETE
...
0.450  MOTOR                OUTPUT-KEY #(4 3)
...
0.600  MOTOR                FINISH-MOVEMENT
```

There are no constraints on how far or in which direction a model’s finger can move relative to the other fingers. Thus, at this time, the model may be able to make finger movements which would be impossible for a real person to make and it is up to the modeler to consider such issues.

If the movement takes the model’s finger to an invalid location for a key then it effectively presses a key with the value **nil** and a warning will be printed:

```
#|Warning: Invalid key location pressed #(4 7) |#
```

The peck-recoil request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid or the module is jammed.

press-key

cmd **press-key**
key *key*

The press-key request is essentially a programming convenience for modeling typing. It assumes that the model's hands are in the home position or that the right hand is on the keypad and translates the specified key into either a punch or a peck-recoil request as needed to press that key. The key can be specified using either a symbol or string that specifies the name of a key to press. It should be the single character for the alpha-numeric keys or one of these symbols for the punctuation and special keys: space, backquote, tab, comma, period (or dot), semicolon, slash, hyphen, quote, return, backslash, or delete. For the keypad keys, the valid key names are: clear, keypad=, keypad-/, keypad-*, keypad-7, keypad-8, keypad-9, keypad-minus, keypad-4, keypad-5, keypad-6, keypad-plus, keypad-1, keypad-2, keypad-3, keypad-0, keypad-period, and enter.

If the model's hands are not in the appropriate location for the request (either the home row or the keypad depending on the key), then the model will strike the wrong key. This request does not represent an ACT-R theory of typing. An ideal theory of typing would be based on a model learning how to type - the model would have to look at the keyboard to determine where each key is, then acquire chunks that encode which keys map to which locations, through practice it would learn specific productions that handle these, and would also involve learning in the motor system itself. Until such time as a mechanism like that is implemented, press-key should be able to handle basic typing actions for modeling a moderately skilled touch typist.

Here are the events which will show in the trace for a press-key request indicating the key that was specified, the progression through the stages of the action, and indicating the location of the key which was pressed:

```
0.050  MOTOR                PRESS-KEY COMMA
...
0.300  MOTOR                PREPARATION-COMPLETE
...
0.350  MOTOR                INITIATION-COMPLETE
...
0.450  MOTOR                OUTPUT-KEY #(8 5)
...
0.600  MOTOR                FINISH-MOVEMENT
```

If an invalid key is specified then a warning is printed and no action is taken:

```
0.050  MOTOR                PRESS-KEY BAD-KEY
#|Warning: No press-key mapping available for key BAD-KEY. |#
```

The press-key request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid or the module is jammed.

move-cursor

cmd move-cursor

[**object** *object* | **loc** *location*]

{**device** [**mouse** | **joystick-1** | **joystick-2**] }

This request will result in a ply style movement of the model's right hand if it is currently located on the mouse. That ply will move the cursor to either the object (which must be a chunk which represents a visual object) or location (which must be a chunk which represents a visual location) specified taking time based on [Fitts's Law](#). If device is not specified then the assumption is that the cursor is positioned with a mouse, but if one of the joystick values is specified that indicates the pointing is done with either a first or second order joystick respectively which increases the Fitts's coefficient for the movement. The coefficient is multiplied by 2 for a first-order joystick and by 4 for a second-order joystick.

If the [:cursor-noise parameter](#) is **nil** then the cursor will be positioned exactly at the coordinates provided by the location given or the coordinates of the location of the object provided. If [:cursor-noise](#) is **t** then a very simplified noise component is added to the target point to determine where the cursor is moved. An offset distance is computed using the effective width of the target – it's width along the vector of approach as used in the Fitts's Law calculation. The offset is computed using the [act-r-noise command](#) with an s value that results in it being on the object (along the approach vector) 96% of the time. That offset is applied in a random direction from the target chosen from a uniform distribution.

Here are the events which show for a move-cursor action showing the module progressing through the internal states and indicating the final position of the cursor:

```
0.050  MOTOR          MOVE-CURSOR OBJECT NIL LOC LOC1
...
0.250  MOTOR          PREPARATION-COMPLETE
...
0.300  MOTOR          INITIATION-COMPLETE
...
0.661  MOTOR          MOVE-CURSOR-ABSOLUTE #(100 200)
...
0.711  MOTOR          FINISH-MOVEMENT
```

If the [:incremental-mouse-moves parameter](#) is specified as **t**, then there may be multiple smaller mouse movements leading up to the final position which are spaced approximately 50 ms apart:

```
0.350  MOTOR          MOVE-CURSOR LOC START
...
0.550  MOTOR          PREPARATION-COMPLETE
...
0.600  MOTOR          INITIATION-COMPLETE
```

```

0.600  MOTOR          MOVE-CURSOR-ABSOLUTE #(2 2)
...
0.650  MOTOR          MOVE-CURSOR-ABSOLUTE #(14 14)
...
0.700  MOTOR          MOVE-CURSOR-ABSOLUTE #(37 37)
...
0.750  MOTOR          MOVE-CURSOR-ABSOLUTE #(63 63)
...
0.800  MOTOR          MOVE-CURSOR-ABSOLUTE #(86 86)
...
0.850  MOTOR          MOVE-CURSOR-ABSOLUTE #(98 98)
...
0.900  MOTOR          MOVE-CURSOR-ABSOLUTE #(100 100)
...
0.950  MOTOR          FINISH-MOVEMENT

```

If the model's right hand is not on the mouse a warning is printed and no action is taken:

```
#|warning: MOVE-CURSOR requested when hand not at mouse!|#
```

The move-cursor request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid, the hand is not on the mouse, or the module is jammed.

hand-to-mouse

cmd hand-to-mouse

The hand-to-mouse request will move the model's right hand from where ever it is to the virtual mouse's location using a ply style movement (the target location for the mouse is (28 2)). These are the events which will show the action progressing through the stages of a motor action along with the final movement indicating the distance and direction the right hand moved:

```

0.050  MOTOR          HAND-TO-MOUSE
...
0.250  MOTOR          PREPARATION-COMPLETE
...
0.300  MOTOR          INITIATION-COMPLETE
...
0.553  MOTOR          MOVE-A-HAND RIGHT 21.095022 -0.094951704
...
0.603  MOTOR          FINISH-MOVEMENT

```

If the model's hand is already on the virtual mouse it will acknowledge the request, but no actions are taken and the module does not become busy. There will only be the one acknowledgement event in the trace:

```
0.050  MOTOR          HAND-TO-MOUSE
```

The hand-to-mouse request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the hand is already on the mouse or the module is jammed.

hand-to-home

cmd hand-to-home

The hand-to-home request will move the model's right hand from where ever it is to the home position on the virtual keyboard using a ply style movement (the target location for the home position is (7 4)) and position all of the fingers over the appropriate keys. These are the events which will show the action progressing through the stages of a motor action along with the final movement indicating the distance and direction the hand moved:

```
0.653  MOTOR          HAND-TO-HOME
...
0.703  MOTOR          PREPARATION-COMPLETE
...
0.753  MOTOR          INITIATION-COMPLETE
...
1.006  MOTOR          MOVE-A-HAND RIGHT 21.095022 3.0466409
...
1.056  MOTOR          FINISH-MOVEMENT
```

Even if the model's hand is already at the home location it will still progress through all of the stages of the movement to move the hand:

```
0.000  MOTOR          HAND-TO-HOME
...
0.200  MOTOR          PREPARATION-COMPLETE
...
0.250  MOTOR          INITIATION-COMPLETE
...
0.350  MOTOR          MOVE-A-HAND RIGHT 0.0 0.0
...
0.400  MOTOR          FINISH-MOVEMENT
```

The hand-to-home request is considered complete when the finish-movement event occurs or immediately upon request by the module if the module is jammed.

hand-to-keypad

cmd hand-to-keypad

The hand-to-keypad request will move the model's right hand from where ever it is to the keypad on the virtual keyboard using a ply style movement (the target location for the keypad position is (19 4)) and position the fingers over the keys 4, 5, 6, and enter with the thumb over 0. These are the events which will show the action progressing through the stages of a motor action along with the final movement indicating the distance and direction the hand moved:

```

3.281  MOTOR          HAND-TO-KEYPAD
...
3.481  MOTOR          PREPARATION-COMPLETE
...
3.531  MOTOR          INITIATION-COMPLETE
...
3.712  MOTOR          MOVE-A-HAND RIGHT 12.0 0.0
...
3.762  MOTOR          FINISH-MOVEMENT

```

Even if the model's hand is already at the keypad location it will still progress through all of the stages of the movement to move the hand:

```

13.583  MOTOR          HAND-TO-KEYPAD
...
13.783  MOTOR          PREPARATION-COMPLETE
...
13.833  MOTOR          INITIATION-COMPLETE
...
13.933  MOTOR          MOVE-A-HAND RIGHT 0.0 0.0
...
13.983  MOTOR          FINISH-MOVEMENT

```

The hand-to-keypad request is considered complete when the finish-movement event occurs or immediately upon request by the module if the module is jammed.

point-hand-at-key

```

cmd point-hand-at-key
hand [ left | right ]
to-key key
{offsets [ standard | (finger-offset*)]}

```

key ::= either a symbol or string that specifies the key to which the hand should move

finger-offset ::= (*finger-name* *x y*)

finger-name ::= [**index** | **middle** | **ring** | **pinkie** | **thumb**]

x ::= an integer indicating the x coordinate offset for the finger from the position of the hand

y ::= an integer indicating the y coordinate offset for the finger from the position of the hand

This request will move the hand on the virtual keyboard causing all of the fingers to be repositioned using a ply style movement. The hand will be positioned at the coordinate of the specified key. If the offsets slot is not specified then the fingers will have the same offset from the hand position as

they had before it was moved (any previous peck actions will still be reflected in the new positions). If the offsets value is specified as standard then the index finger of the specified hand will be positioned over the indicated key and the other fingers being positioned on the keys in the same relative positions as they would be on the home row. If specific finger offsets are provided then those fingers will be over the keys at the specified offset from the position of the hand and fingers not specified will have the same offset they did before the move.

The key can be specified using either a symbol or string that specifies the name of a key as the target location for the hand. It should be the single character for the alpha-numeric keys or one of these symbols for the punctuation and special keys: space, backquote, tab, comma, period (or dot), semicolon, slash, hyphen, quote, return, backslash, or delete.

Here are the events which will show in the trace for a point-hand-at-key request indicating the hand and key that were specified, the progression through the stages of the action, and indicating the distance and direction the hand was moved:

```
0.050  MOTOR          POINT-HAND-AT-KEY HAND LEFT TO-KEY a
...
0.250  MOTOR          PREPARATION-COMPLETE
...
0.300  MOTOR          INITIATION-COMPLETE
...
0.481  MOTOR          MOVE-A-HAND LEFT 3.0 3.1415927
...
0.531  MOTOR          FINISH-MOVEMENT
```

As with the hand-to-home request, even if the hand is already located over the requested key the module will still progress through the movement stages:

```
0.600  MOTOR          POINT-HAND-AT-KEY HAND RIGHT TO-KEY J
0.600  MOTOR          PREPARATION-COMPLETE
...
0.650  MOTOR          INITIATION-COMPLETE
...
0.750  MOTOR          MOVE-A-HAND RIGHT 0.0 0.0
...
0.800  MOTOR          FINISH-MOVEMENT
```

If an invalid key is specified then a warning is printed and no action is taken:

```
0.050  MOTOR          POINT-HAND-AT-KEY HAND LEFT TO-KEY BAD-KEY
#|Warning: No key mapping available for key BAD-KEY |#
```

The point-hand-at-key request is considered complete when the finish-movement event occurs or immediately upon request by the module if the request is invalid or the module is jammed.

prepare

```
cmd prepare  
style [punch | peck | peck-recoil | ply | hand-ply]  
{hand [ left | right ] }  
{finger [ index | middle | ring | pinkie | thumb ] }  
{r distance}  
{theta direction}
```

The prepare request can be used to have the motor module prepare but not immediately execute a set of features. Those features are stored the same way that the features for a previously executed action are, and thus will decrease the preparation time of a subsequent action which shares those features. Using the execute request (described next) it is also possible to perform the action specified by the history of features, so a prepare could be used to set up an action which one may execute later. For instance, if you know the next motor action is going to be a punch of the right index finger, but you're not sure when, you can prepare the movement in advance and then execute it later.

You must specify a style to prepare, and then you may specify any of the parameters that are relevant for that style.

Here is the trace showing a preparation for a ply style action (movement of the left index finger two keys to the right) which shows that only the preparation step occurs for a prepare request:

```
0.050  MOTOR                PREPARE PLY HAND LEFT FINGER INDEX R 2 THETA 0  
...  
0.300  MOTOR                PREPARATION-COMPLETE
```

If that is followed by an execute action there is no preparation step at that time, and the previously prepared action is executed. Here is a continuation of that trace showing an execute request occurring next:

```
0.350  MOTOR                EXECUTE  
...  
0.400  MOTOR                INITIATION-COMPLETE  
...  
0.500  MOTOR                MOVE-A-FINGER LEFT INDEX 2 0  
...  
0.550  MOTOR                FINISH-MOVEMENT
```

The prepare request is considered complete when the preparation-complete event occurs or immediately upon request by the module if the request is invalid or the module is jammed.

execute

cmd execute

The execute request causes the model to execute the last movement which was prepared. That will either be a movement which was specified by an explicit prepare request or the last action which was requested. Because all of the features are already prepared there is no preparation step required as shown in the sequence of events (in this case the last prepared features were for a click-mouse action):

```
0.400  MOTOR          EXECUTE
...
0.450  MOTOR          INITIATION-COMPLETE
...
0.460  MOTOR          OUTPUT-KEY #(28 2)
...
0.550  MOTOR          FINISH-MOVEMENT
```

If there are no currently prepared features then the execute request prints a warning and no action is taken:

```
#|Warning: Motor Module has no movement to EXECUTE. |#
```

The execute request is considered complete when the finish-movement event occurs or immediately upon request by the module if there is no action execute or the module is jammed.

Chunks & Chunk-types

The motor module creates a chunk-type for each of the requests it accepts with the same name as the request and which has a cmd slot with a default value set to a chunk which also has that same name and is defined using that same chunk-type. Here are the chunk-type definitions that are executed by the motor module:

```
(chunk-type motor-command (cmd "motor action"))
(chunk-type (punch (:include motor-command)) (cmd punch) hand finger)
(chunk-type (click-mouse (:include motor-command)) (cmd click-mouse))
(chunk-type (peck (:include motor-command)) (cmd peck) hand finger r theta)
(chunk-type (peck-recoil (:include motor-command)) (cmd peck-recoil) hand finger r theta)
(chunk-type (press-key (:include motor-command)) (cmd press-key) key)
(chunk-type (move-cursor (:include motor-command)) (cmd move-cursor) object loc device)
(chunk-type (hand-to-mouse (:include motor-command)) (cmd hand-to-mouse))
(chunk-type (hand-to-home (:include motor-command)) (cmd hand-to-home))
(chunk-type (hand-to-keypad (:include motor-command)) (cmd hand-to-keypad))
(chunk-type (point-hand-at-key (:include motor-command)) (cmd point-hand-at-key)
  hand to-key offsets)
(chunk-type (prepare (:include motor-command)) (cmd prepare) style hand finger r theta)
(chunk-type (execute (:include motor-command)) (cmd execute))
```

These are the chunks which are created if not already defined. These chunks are marked as immutable if defined by the motor module:

```
(define-chunks
  (punch isa punch)
  (click-mouse isa click-mouse)
  (peck isa peck)
  (peck-recoil isa peck-recoil)
  (press-key isa press-key)
  (move-cursor isa move-cursor)
  (hand-to-mouse isa hand-to-mouse)
  (hand-to-home isa hand-to-home)
  (hand-to-keypad isa hand-to-keypad)
  (point-hand-at-key isa point-hand-at-key)
  (prepare isa prepare)
  (execute isa execute)
  (left name left)
  (right name right)
  (index name index)
  (middle name middle)
  (ring name ring)
  (pinkie name pinkie)
  (thumb name thumb))
```

Commands

start-hand-at-mouse

Syntax:

```
start-hand-at-mouse -> [ t | nil ]
```

Description:

The start-hand-at-mouse command is used to position the right hand of the current model on the virtual mouse instead of on the virtual keyboard which is the default location for the hand. It should be called before running the model, and generally it would be placed into the model definition. It is not intended for moving the model's hand, only specifying its initial location. If the model's hand is successfully placed on the mouse, then **t** is returned. If there is no current model or current meta-process, then no change is made, a warning is printed, and **nil** is returned.

Examples:

```
> (start-hand-at-mouse)
T
```

```
E> (start-hand-at-mouse)
#|Warning: No current model. Cannot set hand at mouse.|#
NIL
```

```
E> (start-hand-at-mouse)
#|Warning: No current meta-process. Cannot set hand at mouse.|#
NIL
```

start-hand-at-keypad

Syntax:

start-hand-at-keypad -> [**t** | **nil**]

Description:

The `start-hand-at-keypad` command is used to position the right hand of the current model on the virtual keyboard's keypad with the fingers over the 4, 5, 6, and enter keys and the thumb over 0. It should be called before running the model, and generally it would be placed into the model definition. It is not intended for moving the model's hand, only specifying its initial location. If the model's hand is successfully placed on the keypad, then **t** is returned. If there is no current model or current meta-process, then no change is made, a warning is printed, and **nil** is returned.

Examples:

```
> (start-hand-at-keypad)
T
```

```
E> (start-hand-at-keypad)
#|Warning: No current model. Cannot set hand at keypad.|#
NIL
```

```
E> (start-hand-at-keypad)
#|Warning: No current meta-process. Cannot set hand at keypad.|#
NIL
```

set-cursor-position

Syntax:

set-cursor-position *x y* -> [*xy-loc* | **nil**]
set-cursor-position-fct *new-loc* -> [*xy-loc* | **nil**]

Arguments and Values:

x ::= a number specifying the x coordinate for the mouse cursor
y ::= a number specifying the y coordinate for the mouse cursor
new-loc ::= a vector of two values representing a new location for the mouse cursor
xy-loc ::= a vector of two values representing the current location of the mouse cursor

Description:

The `set-cursor-position` command can be used to set the position of the mouse cursor for the current device of the current model. It should not be called while the model is running to avoid conflicts with any model generated mouse movements, but can be called before running the model or when it

is stopped. If the position of the mouse cursor is set for the device, then a vector with the current mouse coordinates is returned. If there is no current model or no current meta-process then no change is made, a warning is printed, and **nil** is returned.

Examples:

```
> (set-cursor-position 10 20)
#(10 20)

> (set-cursor-position-fct #(20 30))
#(20 30)

E> (set-cursor-position 10 20)
#|Warning: No current model. Cannot set cursor position.|#
NIL

E> (set-cursor-position 0 0)
#|Warning: No current meta-process. Cannot set cursor position.|#
NIL
```

set-hand-location

Syntax:

```
set-hand-location [ left | right ] x y -> [ t | nil ]
set-hand-location-fct [ left | right ] xy-loc -> [ t | nil ]
```

Arguments and Values:

x ::= a number specifying the starting x coordinate on the virtual keyboard for the specified hand
y ::= a number specifying the starting y coordinate on the virtual keyboard for the specified hand
xy-loc ::= a vector or list of two numbers specifying the starting x and y coordinates respectively on the virtual keyboard for the specified hand

Description:

The `set-hand-location` command can be used to set the position of the specified hand of the current model on the virtual keyboard. It should not be called while the model is running to avoid conflicts with any model generated hand movements, but can be called before running the model or when it is stopped. The *x* and *y* coordinates specify the key over which the specified hand and its index finger are placed, and the other fingers will be located relative to that location as they would for the home row. If the position of the hand is set for the model then **t** is returned. If there is no current model or current meta-process then no change is made, a warning is printed, and **nil** is returned.

Examples:

```
> (set-hand-location left 0 3)
T
```

```

> (set-hand-location-fct 'right '(8 2))
T

E> (set-hand-location left 5 1)
#|Warning: No current model. Cannot set hand location.|#
NIL

E> (set-hand-location right 5 4)
#|Warning: No current meta-process. Cannot set hand location.|#
NIL

```

extend-manual-requests

Syntax:

```

extend-manual-requests chunk-type-def request-function -> [ t | nil ]
extend-manual-requests-fct chunk-type-def request-function -> [ t | nil ]

```

Arguments and Values:

chunk-type-def ::= a list with a valid definition for a new chunk-type
request-function ::= a symbol which is the name of a function to call to handle the new request

Description:

The extend-manual-requests command allows one to add new requests to those which are accepted by the manual buffer. Extend-manual-requests only needs to be called once for each new request being added, and it does not have to occur within the context of a model. The chunk-type-def parameter must be a list which is valid for passing to chunk-type-fct. That chunk-type definition will have an additional slot called cmd specified with a default value that matches the name of the chunk-type from the definition and then be passed to chunk-type-fct. If that chunk-type name is not also defined as a chunk then one will be defined using that same name as the chunk-type and it will be marked as immutable. Thus, if one were to provide this chunk-type-def value:

```
(new-motor-action hand finger pressure)
```

This chunk-type would be defined:

```
(chunk-type new-motor-action hand finger pressure (cmd new-motor-action))
```

and this chunk would be created and marked as immutable:

```
(define-chunks (new-motor-action isa new-motor-action))
```

When a request to the manual buffer is made with the cmd slot specifying the chunk-type-name indicated then the function specified by request-function will be called with the current model's motor module as the first parameter and the [chunk-spec](#) of the request as the second parameter.

There are no restrictions on what the new request may do, nor are there any default operations performed – it is entirely up to the extension to handle all scheduling of events as necessary to change the state of the motor module and perform the actions necessary.

Once a new request has been added it cannot be overwritten by a new request with the same name. However, it can be removed using the [remove-manual-request command](#) and then be defined again.

If chunk-type-def is not a valid list, request-function does not name a currently defined function or the chunk-type being specified is already used for an extension then extend-manual-requests will print a warning, no new request extension will be made, and **nil** will be returned.

If successfully created, the new request will be available to all models from that point on and **t** will be returned.

The recommended use of this command is to place a file which has the necessary request function and any support code needed along with a call to extend-manual-requests into one of the directories for which the files are loaded automatically. That way the request function is compiled with the rest of the sources and is made available to all models right from the start. If it is called while there are models already defined the new request will not be available to those models until they are reset.

Most of the default requests which the motor module accepts are created using extend-manual-requests.

Examples:

These examples assume that there are functions named handle-right-click and handle-hold already defined.

```
> (extend-manual-requests (right-click) handle-right-click)
T
```

```
1> (extend-manual-requests-fct '((hold-key (:include motor-command)) key) 'handle-hold)
T
```

```
2E> (extend-manual-requests-fct '((hold-key (:include motor-command) key)) 'handle-hold)
#|Warning: Request HOLD-KEY is already an extension of the manual buffer. To redefine you
must remove it first with remove-manual-request.|#
NIL
```

```
E> (extend-manual-requests bad-type handle-right-click)
#|Warning: Invalid chunk-type specification BAD-TYPE. Manual requests not extended.|#
NIL
```

```
E> (extend-manual-requests-fct '(chord finger1 finger2) 'not-a-function)
#|Warning: NOT-A-FUNCTION does not name a function. Manual requests not extended.|#
NIL
```

remove-manual-request

Syntax:

```
remove-manual-request chunk-type-name -> [ t | nil ]  
remove-manual-request-fct chunk-type-name -> [ t | nil ]
```

Arguments and Values:

chunk-type-name ::= a symbol which is the name of a chunk-type used to extend the manual requests

Description:

Remove-manual-request is used to remove a request that has been added to the manual buffer through the use of [extend-manual-requests](#). The *chunk-type-name* parameter should be the name of a chunk-type which was defined with a call to `extend-manual-requests`. The request of that chunk-type will be removed from the set of requests that the manual buffer will process for all models and that chunk-type will no longer be defined in any new models (its definition will still remain in any existing models until they are reset).

If *chunk-type-name* does name an extended request, then after removing the request this command will return **t**. If *chunk-type-name* is not the name of a previously extended request then a warning will be printed and **nil** will be returned.

This command is typically only needed when one is developing some extensions to the manual module and needs to change some of the requests under development. It is not recommended for use in other situations.

Examples:

These examples assume that there are functions named `handle-right-click` and `handle-hold` already defined.

```
1> (extend-manual-requests (right-click) handle-right-click)  
T  
2> (extend-manual-requests-fct '((hold-key (:include motor-command)) key) 'handle-hold)  
T  
3> (remove-manual-request-fct 'right-click)  
T  
4> (remove-manual-request hold-key)  
T  
5E> (remove-manual-request hold-key)  
#|Warning: HOLD-KEY is not a previously extended request for the manual module. |#  
NIL
```

```
E> (remove-manual-request-fct 'bad-chunk-name)
#|Warning: BAD-CHUNK-NAME is not a previously extended request for the manual module.|#
NIL
```

Speech module

The speech module gives a model a rudimentary ability to speak. This system is not designed to provide a sophisticated simulation of human speech production, but to allow a model to speak words and short phrases for simulating verbal responses in experiments and for subvocalizing text internally. The module has one buffer, vocal, and works in much the same way as the [motor module](#).

The vocal world

The model's speech output is fairly limited. When it speaks or subvocalizes it will hear its own words through the [audio module](#). That is the only result of subvocalizing. When the model speaks, then the device will detect the output as well. Essentially, the device has a microphone which detects the onset of speech and can record the content of what was said.

Operation

Like the motor module, the speech module takes requests that specify a style of action and the appropriate attributes for that style. There are only two styles available for the speech module:

- Speak to produce normal speech output.
- Subvocalize to produce internal speech – the model speaking to itself with no external output.

The only attribute for each of those styles is a string of text to speak.

The speech module does not place any chunks into its buffer in response to the requests. The vocal buffer should always be empty, and it is the state of the module that is most important to the model in this case. As described above for all the perceptual and motor modules, the speech module has three internal states: preparation, processor, and execution. How a request progresses through those states is described in detail below.

An important thing to note is that while the vocal buffer does not receive a chunk in response to a speech action the [aural-location buffer](#) might. Because the model hears its own speech acts the automatic buffer stuffing of the aural-location buffer may occur as a result of a vocal request.

When a request is received by the speech module, it goes through three phases: preparation, initiation, and execution. The amount of time that speech output preparation takes depends on the history of previous speech acts. The speech module records the last string which it has output. If there is no previous string which has been spoken, then the module takes .15 seconds in preparation

(three 50ms features). If there is a previously spoken string, then the preparation time is 0 seconds if the same string is being output again (no features to prepare) and .1 seconds if it is a different string (two 50ms features).

After preparation is complete, the speech module makes the specified vocal output. The first 50ms of that output is initiation. During this interval, the preparation state becomes free and the processor and execution states become busy. After initiation ends, the processor state becomes free. It is at that time that speech output is made available to the device and the auditory system. Note that the typical detection delay and recoding times still apply for the encoding of sounds as described in the audio module, but speech output through subvocalizing has a separate parameter for the detection delay and a recoding time of 0 seconds. The amount of time that the speech takes to finish after initiation depends on the articulation time of the string which defaults to .15 seconds per assumed syllable, but can be changed in various ways (see the [get-articulation-time command](#) below for details).

The speech module can only prepare one output at a time. If the speech module is in the process of preparing a movement and another request is received, the later request will be ignored and the speech module is said to be "jammed." When the module is jammed it will output a warning indicating when the jamming occurred like this:

```
#|Warning: Module :SPEECH jammed at time 0.1 |#
```

The way to avoid jamming, as with all modules, is to test the state of the module before making a request. Testing that state free is true will avoid jamming the module, but it is possible to issue speech requests faster than that – because the state will not be free until the previous output has been completed. Instead, one usually only needs to test that the preparation state is free to be able to issue a new request to the speech module, but a more conservative approach would be to test the processor state because it is still busy during the initiation time of the last output request.

Parameters

:char-per-syllable

This parameter controls how the model breaks a text string into syllables and is measured in a number of characters (see [get-articulation-time](#) for how it applies). It can be set to any positive number and defaults to 3.

:subvocalize-detect-delay

This parameter sets the detect delay timing for the sound events generated by a subvocalize action (see the [audio module](#) for details on the detect delay). It is measured in seconds and can be set to any non-negative value. It defaults to .3 (which is the same value that normal text has).

:syllable-rate

This parameter controls the time it takes the model to articulate each syllable in a text string and is measured in seconds (see [get-articulation-time](#) below for how it applies). It can be set to any non-negative number and defaults to 0.15.

Vocal buffer

The speech module does not place any chunks into the vocal buffer. The preparation timing for the actions will be set using the [randomize-time function](#), but all other action times will be fixed regardless of the [:randomize-time parameter](#) setting.

Activation spread parameter: :vocal-activation
Default value: 0.0

Queries

‘State busy’ will be **t** when any of the internal states of the module (listed below) also report as being busy. Essentially, it will be **t** while there is any request to the vocal buffer that has not yet completed. It will be **nil** otherwise.

‘State free’ will be **t** when all of the internal states of the module (listed below) also report as being free. Essentially, it will be **t** only when all requests to the vocal buffer have completed. It will be **nil** otherwise.

‘State error’ will always be **nil**.

Unlike the perceptual modules, the internal states of the speech module may be useful to track in a model because it is possible to send new requests while the module is partially busy. Only the preparation stage of the module needs to be free to avoid jamming. For each request that is received the module will progress through three stages as described above: preparation, initiation, and execution.

‘Preparation busy’ will be **t** after a request has been received and until it completes the preparation of the features needed for that request which depends on how many features there are and whether or not they overlap with the last features prepared. It will be **nil** otherwise.

‘Preparation free’ will be **nil** after a request has been received and until it completes the preparation of the features needed for that request which depends on how many features there are and whether or not they overlap with the last features prepared. It will be **t** otherwise.

‘Processor busy’ will be **t** while preparation is busy and will continue to be **t** after the features have been prepared until the additional initiation time (fixed at .05 seconds) has passed. It will be **nil** otherwise.

‘Processor free’ will be **nil** while preparation is busy and will continue to be **nil** after the features have been prepared until the additional initiation time (fixed at .05 seconds) has passed. It will be **t** otherwise.

‘Execution busy’ will be **t** once the preparation of a request’s features has completed and will remain **t** until the time necessary to complete the action has passed. It will be **nil** otherwise.

‘Execution free’ will be **nil** once the preparation of a request’s features has completed and will remain **nil** until the time necessary to complete the action has passed. It will be **t** otherwise.

‘Last-command *command*’ *command* should be a symbol which corresponds to one of the vocal buffer’s requests or the symbol none. The query will be **t** if that is the name of the last request received by the vocal buffer otherwise it will be **nil**.

Here is a summary indicating the state transitions for a single vocal request assuming that the module is entirely free at the start of the request:

Preparation state	Processor state	Execution state	When
FREE	FREE	FREE	Before event arrives
BUSY	BUSY	FREE	When event is received
FREE	BUSY	BUSY	After preparation time
FREE	FREE	BUSY	After initiation (.05 seconds)
FREE	FREE	FREE	When speech completes (articulation time)

Requests

clear

[**cmd clear** | **clear t**]

A clear request can be sent to clear the history of the last string vocalized. A clear request will make the preparation state busy for 50ms. These events will show in the trace for a clear request to the vocal buffer:

```
0.050  SPEECH          CLEAR
...
0.100  SPEECH          CHANGE-STATE LAST NONE PREP FREE
```

Note, that after the clear request completes the last-command recorded by the module will be none and not clear – the clear request has effectively cleared the history of its own request as well.

When the clear event occurs the request is considered complete, and any other requests which may have been pending for the module will also be considered complete.

speak

cmd speak
string text

A speak request causes the model to generate vocal output. The output that is made is the Lisp string specified by text. That output is sent to the current device and thus may be heard/recorded externally. It is also made available to the model's [audio module](#) as a word sound with a location of self. That may trigger buffer stuffing of the aural-location buffer.

Here are the events which will show in the trace for a speak request showing the text that is to be output and the progression through the stages of the action:

```
0.050  SPEECH          SPEAK TEXT Hello
...
0.200  SPEECH          PREPARATION-COMPLETE
...
0.250  SPEECH          INITIATION-COMPLETE
0.250  SPEECH          OUTPUT-SPEECH Hello
...
0.500  SPEECH          FINISH-MOVEMENT
```

If an invalid text value is given this warning will show in the trace and no action will be taken by the module:

```
#|Warning: String slot in a speak request must be a Lisp string. |#
```

The speak request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid or the module is jammed.

subvocalize

cmd subvocalize

string *text*

A subvocalize request causes the current model to generate internal speech (the model is talking to itself). That output is not made available to the device, but it is made available to the model's [audio module](#) as a word sound with an internal location. That may trigger buffer stuffing of the aural-location buffer.

Here are the events which will show in the trace for a subvocalize request showing the text that is to be subvocalized and the progression through the stages of the action:

```
0.050  SPEECH          SUBVOCALIZE TEXT hello
...
0.200  SPEECH          PREPARATION-COMPLETE
...
0.250  SPEECH          INITIATION-COMPLETE
...
0.500  SPEECH          FINISH-MOVEMENT
```

If an invalid text value is given this warning will show in the trace and no action will be taken by the module:

```
#|Warning: String slot in a subvocalize request must be a Lisp string. |#
```

The subvocalize request is considered complete when the finish-movement event occurs, or immediately upon request by the module if the parameters are invalid or the module is jammed.

Chunks & Chunk-types

The speech module creates a chunk-type for each of the requests it accepts with the same name as the request and which has a cmd slot with a default value set to a chunk which also has that same name and is defined using that same chunk-type. Here are the chunk-type definitions that are executed by the speech module:

```
(chunk-type speech-command (cmd "speech command"))
(chunk-type (speak (:include speech-command)) (cmd speak) string speak)
(chunk-type (subvocalize (:include speech-command)) (cmd subvocalize) string subvocalize)
```

These are the chunks which are created if not already defined. These chunks are marked as immutable if defined by the speech module:

```
(define-chunks
 (speak isa speak)
 (subvocalize isa subvocalize)
 (internal name internal)
 (self name self))
```

Commands

get-articulation-time/register-articulation-time

Syntax:

```
get-articulation-time string {time-in-ms} -> [ time | nil ]
register-articulation-time string time -> [ time | nil ]
```

Arguments and Values:

string ::= a Lisp string for which a specific articulation time is required

time-in-ms ::= a generalized boolean indicating the units to use for the returned time

time ::= a non-negative number indicating the time it takes to articulate the string specified

Description:

These commands are used to get the articulation time for a text string and to set an explicit time to articulate a string in the current model. That time will be used by the audio module as the duration of such a string if it is heard by the model and will be the length of time that the model requires to speak such a string using the speech module.

Register-articulation-time sets the articulation time for the provided string in the current model in seconds. If such a value is set then time is returned. If there is no current model, no current meta-process, or one of the parameters is invalid, then a warning is printed and **nil** is returned.

Get-articulation-time returns the time it takes to articulate a string of text for the current model. That time will be either the explicit time that was set using register-articulation-time if one was set, or computed based on the length of the string and the values of the parameters [:syllable-rate](#) and [:character-per-syllable](#) using this equation:

$$AT = r * L / c$$

AT := articulation time for the string

r := value of the :syllable-rate parameter

L := the length of the string

c := value of the :char-per-syllable parameter

If the optional parameter time-in-ms is provided as non-nil then the return value will be measured in milliseconds, otherwise it will be measured in seconds.

If the string is invalid or there is no current model then a warning is printed and **nil** is returned instead.

Examples:

```
1> (sgp :char-per-syllable :syllable-rate)
:CHAR-PER-SYLLABLE 3 (default 3) : Characters per syllable.
:SYLLABLE-RATE 0.15 (default 0.15) : Seconds per syllable.
(3 0.15)

2> (get-articulation-time "Hello")
0.25

3> (get-articulation-time "Hello" t)
250

4> (get-articulation-time "Goodbye")
0.35

5> (get-articulation-time "A")
0.05

6> (register-articulation-time "Hello" .185)
0.185

7> (get-articulation-time "Hello")
0.185

E> (register-articulation-time "Hello" .25)
#|Warning: No current model. Cannot set articulation time.|#
NIL

E> (get-articulation-time "Hello")
#|Warning: No current model. Cannot get articulation time.|#
NIL

E> (get-articulation-time 'hello)
#|Warning: Must specify a string for which to get the articulation time.|#
NIL

E> (register-articulation-time 'hello .2)
#|Warning: Must specify a string for which the articulation time is to be set.|#
NIL

E> (register-articulation-time "Hello" -1)
#|Warning: Articulation time must be a non-negative number.|#
NIL

E> (get-articulation-time "hello")
#|Warning: No current meta-process. Cannot get articulation time.|#
NIL

E> (register-articulation-time "hello" 1.0)
#|Warning: No current meta-process. Cannot set articulation time.|#
NIL
```

Temporal Module

The temporal module provides a model with a means of determining time intervals and is based on research by Taatgen, van Rijn, & Anderson (2007). It does so by providing a timer which counts the number of “ticks” which have passed since the timer was started. The tick lengths are noisy and also increase in duration as time progresses. Thus it is more accurate for timing shorter intervals than longer ones. The current tick count is available to the model via a chunk in the temporal buffer which has a slot called ticks. Once the timer is started the module will continue to update the ticks slot of the chunk in the buffer with the current count automatically. The timer can be explicitly reset to start a new count or stopped by the model at any time, and if the chunk is removed from the buffer it will implicitly stop the count from incrementing.

The tick lengths are generated based on the following equations for the n th tick (the tick count is equal to n , and the first tick happens at t_1):

$$t_0 = start + \epsilon_1$$
$$t_n = a \cdot t_{n-1} + \epsilon_2$$

start := value of the [:time-master-start-increment parameter](#) (default 0.011 seconds)

a := value of the [:time-mult parameter](#) (default 1.1)

b := value of the [:time-noise parameter](#) (default 0.015)

ϵ_1 := noise generated with the [act-r-noise command](#) with an s of $b * 5 * start$

ϵ_2 := noise generated with the [act-r-noise command](#) with an s of $b * a * t_{n-1}$

Parameters

:record-ticks

This parameter controls whether or not the time incrementing event generates an action which the [buffer tracing tools](#) will detect. If it is set to **t** then there will be an extra event in the trace for each clock increment which will be associated with activity in the temporal buffer. It can be set to **t** or **nil** and the default value is **t**.

:time-master-start-increment

This parameter controls the length of the 0th tick in seconds. It can be set to any positive number and defaults to 0.011.

:time-mult

This parameter sets the multiplier for increasing the tick length. It can be set to any positive number and defaults to 1.1.

:time-noise

This parameter scales the s value of the noise added to the tick lengths. It can be set to any positive number and defaults to 0.015.

Temporal buffer

The temporal module sets the temporal buffer to **not** be strict harvested.

Activation spread parameter: :temporal-activation
Default value: 0.0

Queries

The temporal buffer only responds to the default queries and like the [goal module](#) is always available and never in an error state.

‘State busy’ will always be **nil**.

‘State free’ will always be **t**.

‘State error’ will always be **nil**.

Requests

time

ticks value

A time request requires that the ticks slot be specified with any value. A time request will reset the tick counter to 0, put a chunk into the temporal buffer with a ticks slot value of 0, and start the timer which will increment the value in the ticks slot of that chunk as described above. These actions will happen at the time of the request indicating the creation of the new chunk and setting it into the buffer:

```

0.050    TEMPORAL          create-new-buffer-chunk isa time
0.050    TEMPORAL          SET-BUFFER-CHUNK TEMPORAL TIME0

```

The updating of the timer will happen at the appropriate times in the future and will generate multiple events in the trace for each update. If the `:record-ticks parameter` is set to `t` then there will be three events for each timer update:

```

0.063    TEMPORAL          Incrementing time ticks to 1
0.063    TEMPORAL          MODULE-MOD-REQUEST TEMPORAL
0.063    TEMPORAL          MOD-BUFFER-CHUNK TEMPORAL

```

whereas if `:record-ticks` is set to `nil` there will only be two events for each update:

```

0.063    TEMPORAL          Incrementing time ticks to 1
0.063    TEMPORAL          MOD-BUFFER-CHUNK TEMPORAL

```

A time request is considered complete immediately at the time of the request.

clear

[**cmd clear** | **clear t**]

The clear request will stop the temporal counter if it is incrementing. This is often done in a model when a time estimation finishes i.e. the ticks count is no longer needed. Stopping the counter will eliminate the incrementing and modification events from being scheduled. The reason one may want to stop the counter is because as long as those events are being scheduled there will be events in the queue to keep the model running and the procedural module will continue to attempt conflict resolution when those events occur. Typically, that will just be wasted effort by the software if the model is not intended to be doing anything during that time, thus it can improve the software performance without affecting the model's operation. It may also be useful if one is recording the temporal module's activity for comparison to BOLD or other neurophysiological data because otherwise the module will show continuous activity instead of only during a meaningful time estimation period. The clear action will occur at the time of the request and will generate an event like this in the trace:

```

15.559    TEMPORAL          Clear

```

Note: clearing the chunk from the temporal buffer will have the same effect as issuing an explicit clear request to the temporal module because that will also stop the incrementing, but it will not result in the event being generated by the temporal module if that is important for buffer tracing purposes.

A clear request is considered complete immediately at the time of the request.

Modification requests

Technically, the temporal module allows modification requests because they are used internally to update the ticks count when the [:record-ticks parameter](#) is set to t. However, that capability is not intended for general use and modification requests should not be made to the temporal buffer.

Chunks & Chunk-types

The temporal module defines one chunk-type and no chunks:

```
(chunk-type time (ticks 0))
```

Commands

The temporal module provides no user commands. All of its operation is controlled through the requests that it accepts.

Advanced Topics

The previous sections of the manual have covered the basic operation of the system and the operation of most of the modules that are provided. That is the information which should be of general use to those using ACT-R to build models. However, there are other capabilities which one can use in the system in addition to being able to extend the system itself. The following sections will describe more of the low level subsystems that are available for working with ACT-R and will cover more advanced topics in using the system like running multiple models, defining new modules for the model to use, and adjusting how the model operates in real-time mode. These sections will assume that one has a good grasp of the concepts from the earlier sections and some experience with Lisp programming.

One note about the advanced topics is that the purpose is to describe the components that the system makes available. Because the system is provided as Lisp code it is entirely possible for the user to modify or change the definition of the system itself. That is not the sort of thing that will be covered in this, or any, section of the reference manual.

Extending Possible Chunk Slots

As has been mentioned in other places in the manual it is possible for a model to add new slots to chunks as it runs. The mechanism which does that is also available to the user for use in new modules or other extensions of the system. In ACT-R 6.0 this was referred to as extending the chunk-types, but now that chunks no longer have a specific type the mechanism is more appropriately referred to as extending the possible slots of chunks.

Extending the possible chunk slots creates a new possible slot name for use in chunks. It will happen automatically when chunks or productions are defined which specify names for slots which do not currently exist. In those situations a warning will be displayed to indicate that the extension has occurred. It will also happen through dynamic chunk modifications which use slot names that are not currently valid, and when that happens there will be an event in the trace to indicate the extension occurred. It is also possible to explicitly extend the set of possible slot names in code to avoid the warnings which would occur for defining chunks or productions that use new slot names

Note that the command which did this in ACT-R 6.0 (`extend-chunk-type-slots`) has been deprecated and the new command should be used instead since the extended slots are no longer associated with any particular chunk-type.

Commands

extend-possible-slots

Syntax:

extend-possible-slots *new-slot* {*warn?*} -> [*new-slot* | **nil**]

Arguments and Values:

new-slot ::= a symbol which specifies the name of a slot to add to chunks

warn? ::= a generalized boolean which indicates whether or not to print a warning for an existing slot

Description:

The `extend-possible-slots` command is used to add a new slot name for use in chunks. It requires one parameter which should be a non-nil, non-keyword symbol which starts with an alphanumeric character and names the new slot to add for chunks. A second optional parameter may be provided and if it is non-nil then a warning will be output if the new slot name specified already names a valid

slot in chunks (it is **t** by default). If the new slot name provided is a valid symbol and not already the name of a possible slot for chunks then it is added to the possible slot names for the current model in the current meta-process and that slot name is returned.

If the slot name is not valid, there is no current model, or no current meta-process then a warning is printed and **nil** is returned.

Examples:

```
1> (extend-possible-slots 'new-slot-name)
NEW-SLOT-NAME
```

```
2E> (extend-possible-slots 'new-slot-name)
#|Warning: NEW-SLOT-NAME already names a possible slot for chunks.|#
NIL
```

```
3> (extend-possible-slots 'new-slot-name nil)
NIL
```

```
E> (extend-possible-slots nil)
#|Warning: Nil is not a valid slot name when trying to extend slots.|#
NIL
```

```
E> (extend-possible-slots :keyword)
#|Warning: :KEYWORD cannot be used as a slot name because it is a keyword.|#
NIL
```

```
E> (extend-possible-slots '&bad-slot-name)
#|Warning: &BAD-SLOT-NAME cannot be used as a slot name because it does not start with an
alphanumeric character.|#
NIL
```

```
E> (extend-possible-slots 'new-slot-name)
#|Warning: Chunk-type info not available so the slot NEW-SLOT-NAME cannot be added.|#
NIL
```

Chunk-Specs

A chunk-spec is an internal representation for the specification of a chunk (*chunk-specification*). Chunk-specs consist of a set of constraints for the slots of a chunk along with specifications for request parameters. An individual slot or request parameter specification in a chunk-spec is referred to as a slot-spec. Chunk-specs can be used for a variety of purposes and are a key component in the communication between modules. They were used implicitly in several of the commands of the modules described above and occasionally showed up in the text of a warning message. The most obvious use of chunk-specs shown in the modules is in the commands for defining productions. The LHS buffer tests and RHS module requests are both direct applications of chunk-specs. That also describes the two primary uses of chunk-specs: testing or finding chunks that match a specification and making a request to a module.

The description of a slot in a chunk-spec can use modifiers and variables in much the same way that the production pattern matching does. The same modifiers used in productions are available in a chunk-spec: =, -, <, >, <=, and >= for specifying slots. The default behavior is the same as used in productions: = means that the slots must be equal, - means they must differ, and the inequality tests are only valid for numbers. However, one is not restricted to only that usage. How each of those modifiers is used to match or find a chunk with a chunk-spec can be controlled individually in the commands which do so. One example where one might want to change that would be to allow using the inequality tests among some particular set of symbolic qualitative sizes like {tiny, small, normal, big, gigantic} or other non-numeric quantities. Symbols starting with an '=' character are considered as variables for both the slot name and for the slot value positions of a slot-spec.

Unlike a slot's specification, the specification for a request parameter cannot include a modifier other than =. Variables however may still be used for the values in a request parameter specification.

Although the chunk-spec only considers symbols which begin with an '=' character to be variables, symbols which start with other characters may be considered as variables when using the matching and finding commands. An example of using a different character to denote a variable occurs in the [find location requests](#) to the visual-location buffer. For those requests the '&' character is used to denote a variable in a slot value of the request. Note however that when creating a chunk-spec only the '=' character is considered a variable and thus any other symbols specified in the slot name position must be valid for slot names.

The most common place where one will need to use chunk-specs is when developing a new module. All requests to the module will be encoded as chunk-specs. Thus any module which accepts requests will have to process chunk-specs. If the module also holds a set of chunks internally for some purpose, like the vision module does for the features of the visual scene, then one may also find the matching and search commands which are available with chunk-specs to be useful.

Because the semantics of the modifiers are not fixed and the specification of a variable is configurable, a chunk-spec does not really have any particular meaning on its own. Without knowing the context in which the chunk-spec will be used it is impossible to know what it really represents, and thus the chunk-specs are essentially specific to the module or code which uses them.

The internal representation of a chunk-spec is not part of the API for the system. The commands for creating and destructuring them will be described in detail below along with several special purpose commands for testing and reforming a chunk-spec which can make things easier when writing a new module.

Commands

define-chunk-spec

Syntax:

```
define-chunk-spec specification -> [chunk-spec | nil] {(extended*)}  
define-chunk-spec-fct (specification) {extend} {warn} -> [chunk-spec | nil] {(extended*)}
```

Arguments and Values:

specification ::= [chunk-name | {**isa** chunk-type} [{modifier} slot value | {=} request-param value]*]
extend ::= a generalized boolean indicating whether to extend chunks with any invalid slot names
warn ::= a generalized boolean indicating whether to warn if chunks are extended
chunk-name ::= a symbol which names a chunk in the current model
chunk-type ::= a symbol which names a chunk-type in the current model
modifier ::= [= | - | < | > | <= | >=]
slot ::= a symbol which names a valid slot for the chunk-type specified, a symbol which names a slot valid for any chunk if no chunk-type provided and **extend** is **nil**, a variable, or any symbol usable as a slot name if **extend** is true
request-param ::= a Lisp keyword naming a valid request-parameter for the model
variable ::= a symbol which begins with the #\= character
value ::= any Lisp value (a symbol which has a first character of #\= is considered to be a variable)
chunk-spec ::= an ACT-R chunk-spec
extended ::= a symbol which names a slot that was passed to **extend-possible-slots**

Description:

The `define-chunk-spec` command is used to create a chunk-spec. There are two ways to define a chunk-spec. If a chunk-name is given as the only parameter then a chunk-spec will be created that matches that chunk exactly. It will include a slot-spec for each slot in that chunk with the `=` modifier and the current value of that slot in the chunk. Alternatively, an optional chunk-type may be provided and then any number of slot-specs and request parameters may be specified for the chunk-spec.

The two optional parameters for the function allow one to indicate whether the set of possible slots should be extended automatically with newly named slots in the definition of the chunk-spec and whether warnings are displayed when that occurs. If the `extend` parameter is provided as `true` (which is also the default value) then any slots which are specified that are not variables and not currently possible slot names will be extended using the [extend-possible-slots command](#). If the `warn` parameter is provided, then that is passed as the `warn?` parameter to `extend-possible-slots` when a new slot needs to be added for the chunk-spec. The default value of the `warn` parameter is `t` if not provided.

The `define-chunk-spec` macro does not accept the optional parameters for extending and warning. When `define-chunk-spec` is used the default of `t` is used for each of the optional parameters.

A slot specified using a variable (a symbol which begins with an “=” character) is not tested as a possible slot and will not result in extending the possible slots through the definition of the chunk-spec.

If the syntax of the specification is correct and all of the components are valid two values are returned. The first is a chunk-spec and the second is a list of all the slot names which were passed to `extend-possible-slots`.

If the syntax is incorrect, any of the components are invalid, or there is not a current model or current meta-process then a warning is displayed and a single value of `nil` is returned.

As indicated in the general description of a chunk-spec, the chunk-spec representation itself is not part of the API. Thus the specific return value of this command should not be used directly and is only intended to be passed to other chunk-spec processing commands.

Examples:

```
1> (chunk-type goal value state)
GOAL
```

```
2> (define-chunks (g isa goal value 2 state start))
(G)
```

The following three chunk-spec calls all result in the same chunk-spec being returned:

```
3> (define-chunk-spec g)
#S(ACT-R-CHUNK-SPEC ...)
NIL
```

```
4> (define-chunk-spec-fct '(isa goal = value 2 = state start))
#S(ACT-R-CHUNK-SPEC ...)
NIL
```

```
5> (define-chunk-spec value 2 state start)
#S(ACT-R-CHUNK-SPEC ...)
NIL
```

The purpose of the next chunk-spec is ambiguous without knowing a context in which it will be used because the @ may be used to represent a variable, or perhaps the symbol @value has a particular meaning for where it will be used:

```
1> (chunk-type compare val1 val2)
COMPARE
```

```
2> (define-chunk-spec-fct '(isa compare val1 @value <= val2 @value - val2 10))
#S(ACT-R-CHUNK-SPEC ...)
NIL
```

```
> (define-chunk-spec-fct '(new-slot1 start value true))
#|Warning: Chunks extended with slot NEW-SLOT1 during a chunk-spec definition.|#
#S(ACT-R-CHUNK-SPEC ...)
(NEW-SLOT1)
```

```
> (define-chunk-spec-fct '(new-slot1 start value true) t nil)
#S(ACT-R-CHUNK-SPEC ...)
(NEW-SLOT1)
```

```
> (define-chunk-spec =variable-slot =variable-value)
#S(ACT-R-CHUNK-SPEC ...)
NIL
```

```
E> (define-chunk-spec-fct '(new-slot1 start value true) nil)
#|Warning: Invalid slot-name NEW-SLOT1 in call to define-chunk-spec.|#
NIL
```

```
E> (define-chunk-spec not-a-chunk)
#|Warning: define-chunk-spec's 1 parameter doesn't name a chunk: (NOT-A-CHUNK)|#
NIL
```

```
E> (define-chunk-spec isa bad-name)
#|Warning: Element after isa in define-chunk-spec isn't a chunk-type. (ISA BAD-NAME)|#
NIL
```

```
E> (define-chunk-spec )
#|Warning: define-chunk-spec-fct called with no current model.|#
NIL
```

```
E> (define-chunk-spec)
#|Warning: define-chunk-spec-fct called with no current meta-process.|#
NIL
```

chunk-name-to-chunk-spec

Syntax:

chunk-name-to-chunk-spec *name* -> [chunk-spec | nil]

Arguments and Values:

chunk-name ::= a symbol which names a chunk in the current model

chunk-spec ::= an ACT-R chunk-spec

Description:

The chunk-name-to-chunk-spec command is a function which works the same as [define-chunk-spec](#) does when provided a chunk name as its only parameter. A chunk-spec will be created that matches that chunk exactly. It will include a slot-spec for each slot in that chunk with the = modifier and the current value of that slot in the chunk. Chunk-name-to-chunk-spec can be used as a short-cut to calling define-chunk-spec-fct with a list of a chunk name.

If the chunk-name provided names a chunk in the current model of the current meta-process then a chunk-spec which matches that chunk is returned. If the parameter provided does not name a chunk in the current model or there is no current model or current meta-process then a warning will be displayed and **nil** is returned.

As indicated in the general description of a chunk-spec, the chunk-spec representation itself is not part of the API. Thus the specific return value of this command should not be used directly and is only intended to be passed to other chunk-spec processing commands.

Examples:

```
> (chunk-name-to-chunk-spec 'free)
#S(ACT-R-CHUNK-SPEC ...)

E> (chunk-name-to-chunk-spec :not-a-chunk)
#|Warning: Chunk-name-to-chunk-spec called with a non-chunk :NOT-A-CHUNK.|#
NIL

E> (chunk-name-to-chunk-spec 'free)
#|Warning: get-chunk called with no current model.|#
#|Warning: Chunk-name-to-chunk-spec called with a non-chunk FREE.|#
NIL

E> (chunk-name-to-chunk-spec 'free)
#|Warning: get-chunk called with no current meta-process.|#
#|Warning: Chunk-name-to-chunk-spec called with a non-chunk FREE.|#
NIL
```

pprint-chunk-spec

Syntax:

pprint-chunk-spec *chunk-spec* -> nil

Arguments and Values:

chunk-spec ::= a valid ACT-R chunk-spec

Description:

The pprint-chunk-spec command can be used to print a chunk-spec to the model's command output stream. If the chunk-spec provided is valid then a description of that chunk-spec will be output on the model's command output stream. Each slot-spec will be output on a separate line. Note that the = modifier will not be printed in the output, but all other modifiers will be. If the provided parameter is not a valid chunk-spec, then no output will be performed. If there is no current model or current meta-process then a warning will be printed and no output will occur. The command will always return **nil**.

Examples:

```
1> (chunk-type test slot1 slot2 value)
TEST
```

```
2> (defvar *s1* (define-chunk-spec slot1 true - slot2 nil <= slot1 =value :attended new))
*S1*
```

```
3> (pprint-chunk-spec *s1*)
  SLOT1 TRUE
-  SLOT2 NIL
<=  SLOT1 =VALUE
    :ATTENDED NEW
NIL
```

```
E> (pprint-chunk-spec :not-a-spec)
NIL
```

```
E> (pprint-chunk-spec *s1*)
#|Warning: Pprint-chunk-spec called with no current model.|#
NIL
```

```
E> (pprint-chunk-spec *s1*)
#|Warning: Pprint-chunk-spec called with no current meta-process.|#
NIL
```

match-chunk-spec-p

Syntax:

match-chunk-spec-p *chunk-name chunk-spec* {:=test test-fn} {:-test test-fn} {:<=test test-fn} {:>=test test-fn} {:>test test-fn} {:<test test-fn} {:**variable-char** *var*} -> result

Arguments and Values:

chunk-name ::= a symbol which should name a chunk in the current model

chunk-spec ::= a valid chunk-spec

test-fn ::= a function designator which must take two parameters and return a generalized boolean

var ::= a Lisp character

result ::= a generalized boolean indicating whether or not the match was a success

Description:

The `match-chunk-spec-p` command is used to determine if a chunk matches a chunk-spec. If it matches, then a true value is returned and if it does not match **nil** is returned.

A chunk matches the chunk-spec if every slot specified in the chunk-spec with a non-nil value exists in that chunk, every slot specified in the chunk-spec with a nil value does not exist in that chunk, and the test of every slot in the specification with the corresponding slot value in the chunk results in a true value. Any request parameters which are a part of the chunk-spec are ignored for the purposes of matching the chunk with this command. To test the slot values, the test function which corresponds to the modifier on that slot in the chunk-spec will be called with the chunk's slot's value as the first parameter and the chunk-spec's slot-spec value as the second parameter unless the slot value specified is a variable. If the slot-spec's value is a variable, then the value bound to that variable (as described next) will be passed to the testing function. If a slot name is specified as a variable that test will be performed on the slot named by the value to which that variable is bound.

If there are any variables in the chunk-spec they are first tested for consistency and to determine their bindings before performing the slot tests. Each variable in the chunk-spec must appear as a value in at least one slot-spec which uses the = modifier. If there is a variable which does not meet that requirement the match fails. If more than one slot-spec using the = modifier contains the same variable as a value, then all slots where the = test is used with that variable as a value must have the same value (using the function for the = test) or the match will fail. If those conditions are true, then the value in that slot(s) will be bound to the variable for the remainder of the slot tests in the chunk-spec. If a variable is bound in a slot which is named with a variable then that slot name variable will be bound before the slot value can be bound. Unlike productions the specification of variablized slot names is not restricted to a single level of indirection, but if there are dependencies among the variablized slot names such that there is no valid order in which to bind the slots then the matching fails.

If there is not a variable-character provided then the default character of #\= is used to denote variables in the chunk-spec.

If no test functions are specified then the default chunk matching functions are used. Those functions work as follows:

- o The = test is [chunk-slot-equal](#)
- o The – test is the result of negating the = test
- o All of the inequality tests are true if
 - Both values are numbers and
 - The specified inequality holds between them with the chunk’s slot value being the left of the items in the test and the chunk-spec’s slot value the right.

If chunk-name, chunk-spec or one of the testing functions is invalid or there is no current model or current meta-process then a warning is displayed and **nil** is returned.

Examples:

```
1> (chunk-type test slot1 slot2 slot3)
TEST

2> (define-chunks (tiny) (small) (medium) (large)
      (slot1) (slot2)
      (a isa test slot1 10 slot2 5)
      (b isa test slot1 large slot2 small)
      (c isa test slot1 medium slot2 small slot3 tiny)
      (d isa test slot1 10 slot2 slot1))
(TINY SMALL MEDIUM LARGE SLOT1 SLOT2 A B C D E)

3> (defun smaller-size (x y)
      (let ((valid '(tiny small medium large huge)))
        (and (find x valid)
              (find y valid)
              (< (position x valid) (position y valid)))))
SMALLER-SIZE

4> (match-chunk-spec-p 'a (define-chunk-spec - slot1 7 >= slot2 1))
T

5> (match-chunk-spec-p 'a (define-chunk-spec slot1 4))
NIL

6> (match-chunk-spec-p 'a (define-chunk-spec slot1 =v < slot2 =v))
T

7> (match-chunk-spec-p 'a (define-chunk-spec slot1 10 :attended t))
T

8> (match-chunk-spec-p 'a (define-chunk-spec slot3 nil))
T

9> (match-chunk-spec-p 'c (define-chunk-spec slot3 nil))
```

```

NIL

10> (match-chunk-spec-p 'b (define-chunk-spec slot1 =v < slot2 =v))
NIL

11> (match-chunk-spec-p 'b (define-chunk-spec slot1 =v < slot2 =v) :<test 'smaller-size)
T

12> (match-chunk-spec-p 'b (define-chunk-spec slot1 $v < slot2 $v) :<test 'smaller-size)
NIL

13> (match-chunk-spec-p 'b (define-chunk-spec slot1 $v < slot2 $v) :<test 'smaller-size
      :variable-char #\)$)
T

14> (match-chunk-spec-p 'd (define-chunk-spec =slot 10 =slot2 =slot slot3 =slot2))
NIL

15> (match-chunk-spec-p 'e (define-chunk-spec =slot 10 =slot2 =slot slot3 =slot2))
T

E> (match-chunk-spec-p 'bad-chunk (define-chunk-spec))
#|Warning: BAD-CHUNK does not name a chunk in call to match-chunk-spec-p.|#
NIL

E> (match-chunk-spec-p 'a :not-a-spec)
#|Warning: :NOT-A-SPEC is not a valid chunk-spec in call to match-chunk-spec-p.|#
NIL

E> (match-chunk-spec-p 'a (define-chunk-spec slot1 10) :=test 'not-a-function)
#|Warning: Error #<UNDEFINED-FUNCTION @ #x2283eb5a> encountered in matching chunk A.|#
NIL

E> (match-chunk-spec-p nil nil)
#|Warning: Match-chunk-spec-p called with no current model.|#
NIL

E> (match-chunk-spec-p nil nil)
#|Warning: Match-chunk-spec-p called with no current meta-process.|#
NIL

```

find-matching-chunks

Syntax:

```

find-matching-chunks chunk-spec {:chunks [:all | (chunk*)]} {:=test test-fn} {:-test test-fn} {:<=test test-fn}
      {:>=test test-fn} {:>test test-fn} {:<test test-fn} {:variable-char var} -> result

```

Arguments and Values:

chunk-spec ::= a valid chunk-spec

chunk ::= a symbol which should name a chunk in the current model

test-fn ::= a function designator which must take two parameters and return a generalized boolean

var ::= a Lisp character

result ::= (*chunk**)

Description:

The `find-matching-chunks` command tests each chunk provided in the list given as the `:chunks` keyword parameter, or all chunks in the current model if it is omitted or specified as `:all`, against the `chunk-spec` provided. It returns the list of chunk names which match the `chunk-spec` using the same matching as described in `match-chunk-spec-p` which includes the possibility of specifying alternate testing functions and a variable character other than the default of `#\=`.

If the `chunks` list contains items which do not name chunks, then for each such item a warning will be printed and that item will be ignored, but the matching will still occur for the valid chunks.

If `chunk-spec`, `chunks`, or a testing function is invalid or there is no current model then a warning is displayed and `nil` is returned.

Examples:

```
1> (chunk-type test slot1 slot2 slot3)
TEST

2> (define-chunks (tiny) (small) (medium) (large)
      (slot1) (slot2)
      (a isa test slot1 10 slot2 5)
      (b isa test slot1 large slot2 small)
      (c isa test slot1 medium slot2 small slot3 tiny)
      (d isa test slot1 10 slot2 slot1))
      (e isa test slot1 10 slot2 slot1 slot3 slot2))
(TINY SMALL MEDIUM LARGE SLOT1 SLOT2 A B C D E)

3> (defun smaller-size (x y)
      (let ((valid '(tiny small medium large huge)))
        (and (find x valid)
              (find y valid)
              (< (position x valid) (position y valid))))))
SMALLER-SIZE

4> (find-matching-chunks (define-chunk-spec - slot1 nil))
(D B E C A)

5> (find-matching-chunks (define-chunk-spec - slot1 nil) :chunks '(a b free busy))
(B A)

6> (find-matching-chunks (define-chunk-spec < slot1 large) :<test 'smaller-size)
(C)

7E> (find-matching-chunks (define-chunk-spec < slot1 large) :<test 'bad-function-name)
#|Warning: Error #<UNDEFINED-FUNCTION @ #x22999a22> encountered in matching chunk A.|#
#|Warning: Error #<UNDEFINED-FUNCTION @ #x229a58b2> encountered in matching chunk C.|#
#|Warning: Error #<UNDEFINED-FUNCTION @ #x229ad072> encountered in matching chunk E.|#
#|Warning: Error #<UNDEFINED-FUNCTION @ #x229cb872> encountered in matching chunk B.|#
#|Warning: Error #<UNDEFINED-FUNCTION @ #x229cf792> encountered in matching chunk D.|#
NIL

E> (find-matching-chunks :not-a-chunk-spec)
#|Warning: :NOT-A-CHUNK-SPEC is not a valid chunk-spec in call to find-matching-chunks.|#

E> (find-matching-chunks (define-chunk-spec) :chunks '(not-a-chunk))
#|Warning: NOT-A-CHUNK does not name a chunk in call to match-chunk-spec-p.|#
NIL
```

```

E> (find-matching-chunks (define-chunk-spec) :chunks 'not-a-list)
#|Warning: NOT-A-LIST is not a valid value for the :chunks keyword parameter to find-
matching-chunks. |#
NIL

E> (find-matching-chunks :not-a-chunk-spec)
#|Warning: Find-matching-chunks called with no current model. |#
NIL

E> (find-matching-chunks :not-a-chunk-spec)
#|Warning: Find-matching-chunks called with no current meta-process. |#
NIL

```

chunk-spec-slots

Syntax:

chunk-spec-slots *chunk-spec* -> (slot*)

Arguments and Values:

chunk-spec ::= a valid chunk-spec

slot ::= a symbol which names a slot or request parameter specified in chunk-spec

Description:

The chunk-spec-slots command returns a list of the names of the slots and request parameters that are used in the provided chunk-spec. Each slot will occur only once in the return list no matter how many times it may be tested in chunk-spec, and the list is in no particular order. That list will include any variables which are used in the slot name position.

If chunk-spec is not a valid chunk-spec then a warning is displayed and **nil** is returned.

Examples:

```

> (chunk-spec-slots (define-chunk-spec))
NIL

> (chunk-spec-slots (define-chunk-spec color blue size =x))
(COLOR SIZE)

> (chunk-spec-slots (define-chunk-spec - color red - color green))
(COLOR)

> (chunk-spec-slots (define-chunk-spec :attended t =slot t))
(:ATTENDED =SLOT)

E> (chunk-spec-slots 'not-a-spec)
#|Warning: Chunk-spec-slots called with something other than a chunk-spec |#
NIL

```

slot-in-chunk-spec-p

Syntax:

slot-in-chunk-spec-p *chunk-spec slot* -> result

Arguments and Values:

chunk-spec ::= a valid chunk-spec

slot ::= a symbol which names a slot or request parameter

result ::= a generalized boolean indicating whether slot was found in chunk-spec

Description:

The `slot-in-chunk-spec-p` command tests whether the given slot occurs in some slot-spec of the chunk-spec provided. It returns a non-**nil** value if that slot is specified in chunk-spec and **nil** if it is not.

If chunk-spec is not a valid chunk-spec then a warning is printed and **nil** is returned.

Examples:

```
> (slot-in-chunk-spec-p (define-chunk-spec size 10) 'color)
NIL

> (slot-in-chunk-spec-p (define-chunk-spec color blue size =x) 'color)
T

> (slot-in-chunk-spec-p (define-chunk-spec slot1 10 =x 20 slot2 =x) '=x)
T

E> (slot-in-chunk-spec-p 'not-a-spec nil)
#|Warning: Slot-in-chunk-spec-p called with something other than a chunk-spec|#
NIL
```

chunk-spec-slot-spec

Syntax:

chunk-spec-slot-spec *chunk-spec {slot}* -> (spec-list*)

Arguments and Values:

chunk-spec ::= a valid chunk-spec

slot ::= [slot-name | **nil**]

slot-name ::= a symbol which names a slot or request parameter specified in chunk-spec

spec-list ::= (modifier slot-name value)

modifier ::= [= | - | < | > | <= | >=]

value ::= any Lisp value

Description:

The `chunk-spec-slot-spec` command is used to extract the slot specifications from a `chunk-spec`. It returns a list of specification lists. A specification list is a list of three elements. The first item in a specification list is the modifier for the `slot-spec`, the second element is the slot name in the `slot-spec`, and the third element is the value in the `slot-spec`. If no slot is specified, then a specification list is returned for each `slot-spec` in the `chunk-spec` and the specification lists will be in the same order as the `slot-specs` were provided when the `chunk-spec` was defined.

If a slot is specified, then only the specification lists which reference that slot are returned, again in the order that they were provided when the `chunk-spec` was defined. If a slot is specified for `chunk-spec-slot-spec` and there are no `slot-specs` in the `chunk-spec` which use that slot then **nil** will be returned.

If `chunk-spec` is not a valid `chunk-spec` then a warning is printed and **nil** is returned.

Examples:

```
1> (defvar *test-spec* (define-chunk-spec < screen-x 10
                                     > screen-x 0
                                     screen-x =var
                                     color blue
                                     - screen-y =var
                                     :attended t))

*TEST-SPEC*

2> (chunk-spec-slot-spec *test-spec*)
((< SCREEN-X 10) (> SCREEN-X 0) (= SCREEN-X =VAR) (= COLOR BLUE) (- SCREEN-Y =VAR)
 (= :ATTENDED T))

3> (chunk-spec-slot-spec *test-spec* 'color)
(= COLOR BLUE)

4> (chunk-spec-slot-spec *test-spec* 'screen-x)
((< SCREEN-X 10) (> SCREEN-X 0) (= SCREEN-X =VAR))

5> (chunk-spec-slot-spec *test-spec* :attended)
(= :ATTENDED T)

6> (chunk-spec-slot-spec *test-spec* 'size)
NIL

E> (chunk-spec-slot-spec 'not-a-spec)
#|Warning: Chunk-spec-slot-spec called with something other than a chunk-spec|#
NIL
```

slot-spec-modifier/slot-spec-slot/slot-spec-value

Syntax:

```
slot-spec-modifier slot-spec-list -> modifier
slot-spec-slot slot-spec-list -> slot
slot-spec-value slot-spec-list -> value
```

Arguments and Values:

slot-spec-list ::= a three element list describing a slot-spec as returned by chunk-spec-slot-spec

modifier ::= [= | - | < | > | <= | >=]

slot ::= a symbol of the slot, request parameter, or variable in the slot name position of slot-spec-list

value ::= any Lisp value from the value position of slot-spec-list

Description:

The slot-spec-* commands can be used to access the corresponding components of a slot-spec list as returned by [chunk-spec-slot-spec](#). They are not really necessary since a slot-spec list is just a three element list, but it can be cleaner in code to see an accessor name like this than first, second, or third.

If the slot-spec-list isn't a three element list then a warning is printed and **nil** is returned.

Examples:

```
1> (setf *spec-list* (first (chunk-spec-slot-spec (define-chunk-spec color blue))))  
(= COLOR BLUE)
```

```
2> (slot-spec-modifier *spec-list*)  
=
```

```
3> (slot-spec-slot *spec-list*)  
COLOR
```

```
4> (slot-spec-value *spec-list*)  
BLUE
```

```
E> (slot-spec-modifier nil)  
#|Warning: Invalid slot-spec list NIL passed to slot-spec-modifier. |#  
NIL
```

```
E> (slot-spec-slot nil)  
#|Warning: Invalid slot-spec list NIL passed to slot-spec-slot. |#  
NIL
```

```
E> (slot-spec-value nil)  
#|Warning: Invalid slot-spec list NIL passed to slot-spec-value. |#  
NIL
```

chunk-spec-variable-p

Syntax:

chunk-spec-variable-p *value* {*var*} -> result

Arguments and Values:

value ::= any Lisp value

var ::= a Lisp character

result ::= a generalized boolean

Description:

The `chunk-spec-variable-p` command can be used to determine if a value would be considered as a variable in a chunk-spec. If the value is a symbol with a symbol-name longer than one character and the first character of the symbol-name is `var` (or `#\=` if `var` is not specified) then `true` is returned otherwise `nil` is returned.

Examples:

```
> (chunk-spec-variable-p 'value)
NIL

> (chunk-spec-variable-p '=value)
T

> (chunk-spec-variable-p '@value #\@)
T

> (chunk-spec-variable-p '=value #\@)
NIL

> (chunk-spec-variable-p "=value")
NIL
```

chunk-spec-to-chunk-def

Syntax:

chunk-spec-to-chunk-def *chunk-spec* -> [chunk-def | nil]

Arguments and Values:

`chunk-spec` ::= a valid chunk-spec

`chunk-def` ::= a chunk definition list which is valid for passing to [define-chunks](#)

Description:

The `chunk-spec-to-chunk-def` command can be used to convert a chunk-spec into a list which can be passed to [define-chunks](#) for creating a chunk. To be able to create a chunk definition the chunk-spec must meet the following conditions:

- It must specify each slot no more than once
- It must not have any modifiers other than =
- There must not be any variables in the specification

Any request parameters in the chunk-spec are ignored unless they contain variables, in which case they fall under the third case above. If chunk-spec is a valid chunk-spec and meets the three criteria shown then this command returns a list which contains a definition for a chunk based on the slot-specs in the chunk-spec. If chunk-spec is not a valid chunk-spec or one or more of the three conditions specified above is not true then a warning is printed and **nil** is returned. Note that although **nil** would be a valid return value for a chunk-spec that has no slots, instead such a situation will return the list (isa chunk) to provide a non-**nil** value upon success.

This primary use of this command is by modules like [goal](#) and [imaginal](#) which create new chunks based on requests.

Examples:

```
> (chunk-spec-to-chunk-def (define-chunk-spec))
(ISA CHUNK)

> (chunk-spec-to-chunk-def (define-chunk-spec color blue = size 10))
(COLOR BLUE SIZE 10)

> (chunk-spec-to-chunk-def (define-chunk-spec color blue :attended t))
(COLOR BLUE)

E> (chunk-spec-to-chunk-def (define-chunk-spec - color blue))
#|Warning: Chunk-spec may only use the = modifier in a call to chunk-spec-to-chunk-def.|#
NIL

E> (chunk-spec-to-chunk-def (define-chunk-spec color blue color red))
#|Warning: Chunk-spec may only specify a slot once in a call to chunk-spec-to-chunk-def.|#
#
NIL

E> (chunk-spec-to-chunk-def (define-chunk-spec color blue :attended =val))
#|Warning: Chunk-spec has variables in the values in a call to chunk-spec-to-chunk-def.|#
NIL

E> (chunk-spec-to-chunk-def (define-chunk-spec =slot value))
#|Warning: Chunk-spec has variablized slots in a call to chunk-spec-to-chunk-def.|#
NIL

E> (chunk-spec-to-chunk-def 'not-a-chunk-spec)
#|Warning: Chunk-spec-to-chunk-def called with something other than a chunk-spec.|#
NIL
```

verify-single-explicit-value

Syntax:

```
verify-single-explicit-value chunk-spec slot-name module-name cmd { var } -> [ value | nil ] { t }
```

Arguments and Values:

chunk-spec ::= a valid chunk-spec

slot-name ::= a symbol which names a slot or request parameter

module-name ::= a symbol which names a module to display in the warning
cmd ::= a Lisp value to display as a command in a warning
var ::= a Lisp character to consider as the variable indicator
value ::= the value from the specified slot of the chunk-spec

Description:

The `verify-single-explicit-value` command can be used to get the value of a particular slot or request parameter slot-spec in a chunk-spec while also testing for the following conditions:

- the slot occurs exactly once in the chunk-spec
- it uses the = modifier
- it has a non-variable value based on the var character specified or #\= if var not provided

If any of those conditions are violated then a warning is printed using the module-name and cmd value provided to create the warning message. After printing the warning message it returns **nil**.

If all of those conditions are satisfied then the function will return the value of that slot in the chunk-spec as its first return value and **t** as its second return value to indicate success since **nil** could be the value of the slot in the chunk-spec.

This command is used by several of the provided modules in processing their requests and may be of use to those implementing new modules.

Examples:

```
1> (setf *test-spec* (define-chunk-spec color blue - size 10 :attended =x value 3 value 4
                               name &x))
#S(ACT-R-CHUNK-SPEC ...)

2> (verify-single-explicit-value *test-spec* 'color 'test-module 'test-command)
BLUE
T

3E> (verify-single-explicit-value *test-spec* 'size 'test-module 'test-command)
#|Warning: TEST-COMMAND command to TEST-MODULE module requires a value for the SIZE slot.
|#
NIL

4E> (verify-single-explicit-value *test-spec* 'value 'test-module 'test-command)
#|Warning: VALUE slot may only be specified once in a TEST-COMMAND command to the TEST-
MODULE module.
|#
NIL

5E> (verify-single-explicit-value *test-spec* :attended 'test-module 'test-command)
#|Warning: ATTENDED slot must be explicit - not a variable in a TEST-COMMAND command to the
TEST-MODULE module.
|#
NIL

6> (verify-single-explicit-value *test-spec* :attended 'test-module 'test-command #\&)
```

```

=X
T

7> (verify-single-explicit-value *test-spec* 'name 'test-module 'test-command)
&X
T

8E> (verify-single-explicit-value *test-spec* 'name 'test-module 'test-command #\&)
#|Warning: NAME slot must be explicit - not a variable in a TEST-COMMAND command to the
TEST-MODULE module.|#
NIL

9E> (verify-single-explicit-value 'not-a-spec 'name 'test-module 'test-command #\&)
#|Warning: NOT-A-SPEC is not a chunk-spec in TEST-COMMAND command to the TEST-MODULE
module.|#
NIL

```

test-for-clear-request

Syntax:

test-for-clear-request *chunk-spec* -> result

Arguments and Values:

chunk-spec ::= a valid chunk-spec

result ::= a generalized boolean indicating whether the chunk-spec indicates a typical clear request

Description:

The test-for-clear-request command can be used to determine if a chunk-spec has the format of the “clear” requests handled by many modules. It will return true if it is passed a chunk-spec that has only one slot-spec in it and that slot-spec is either: “= **cmd clear**” or “= **clear true**” where *true* is any non-**nil** value. Otherwise it will return **nil**.

This command is used by several of the provided modules in processing their requests and may be of use to those implementing new modules.

Examples:

```

> (test-for-clear-request (define-chunk-spec clear t))
T

> (test-for-clear-request (define-chunk-spec cmd clear))
T

> (test-for-clear-request (define-chunk-spec isa clear))
T

> (test-for-clear-request (define-chunk-spec clear nil))
NIL

> (test-for-clear-request (define-chunk-spec - cmd clear))
NIL

```

```
> (test-for-clear-request (define-chunk-spec clear t cmd clear))
NIL

> (test-for-clear-request 'not-a-chunk-spec)
NIL
```

chunk-difference-to-chunk-spec

Syntax:

chunk-difference-to-chunk-spec *chunk1 chunk2* -> [chunk-spec | **nil**]

Arguments and Values:

chunk1 ::= a symbol which names a chunk
chunk2 ::= a symbol which names a chunk
chunk-spec ::= an ACT-R chunk-spec

Description:

The `chunk-difference-to-chunk-spec` command can be used to create a chunk-spec that contains the modifications necessary to convert *chunk2* into *chunk1*. That chunk-spec could be passed to [mod-buffer-chunk](#) to perform the modifications if *chunk2* were in the desired buffer. If *chunk1* and *chunk2* are the names of chunks in the current model of the current meta-process then a chunk-spec is created which has all of the slots from *chunk1* as would be created from `chunk-name-to-chunk-spec` for that chunk and also includes slot-specs for each of the slots of *chunk2* which must be removed specified with **nil**.

If either *chunk1* or *chunk2* is not the name of a chunk in the current model of the current meta-process or there is no current model or current meta-process then warnings will be displayed and **nil** will be returned.

Examples:

```
1> (define-chunks (a value 1 color blue) (b value 3 color blue size 5))
(A B)
```

```
2> (chunk-difference-to-chunk-spec 'a 'b)
#S(ACT-R-CHUNK-SPEC ...)
```

```
3> (pprint-chunk-spec *)
SIZE NIL
COLOR BLUE
VALUE 1
NIL
```

```
E> (chunk-difference-to-chunk-spec :bad :names)
#|Warning: Chunk-difference-to-chunk-spec called with :BAD and :NAMES which are not both
chunks.|#
```

NIL

```
E> (chunk-difference-to-chunk-spec 'a 'b)
#|warning: get-chunk called with no current model. |#
#|warning: get-chunk called with no current model. |#
#|warning: Chunk-difference-to-chunk-spec called with A and B which are not both chunks. |
#
NIL
```

```
E> (chunk-difference-to-chunk-spec 'a 'b)
#|warning: get-chunk called with no current meta-process. |#
#|warning: get-chunk called with no current meta-process. |#
#|warning: Chunk-difference-to-chunk-spec called with A and B which are not both chunks. |
#
NIL
```

Using Buffers

In the [earlier section on buffers](#) their role in the system was described and the commands which a modeler may use to inspect and query the buffers were presented. This section is going to describe the commands that allow one to manipulate the chunk in a buffer and to use the buffer to interact with modules directly. These commands are generally used when one is creating a new module, but may also be useful for other purposes.

There are several things which one can do with a buffer. They are: read the chunk that it holds, test if the buffer is empty, request the state of the buffer's module, request an action of the buffer's module, modify the chunk in the buffer, clear the chunk from the buffer, place a new chunk into the buffer, set the failure flag of the buffer, and check the completion status of a request which was made to the buffer. The commands for performing those actions will be detailed below. Creation of buffers is not described in this section because buffers are only created when modules are defined – buffers only exist in conjunction with a module. Thus buffer creation will be described in the section on [creating modules](#).

Any buffer may be accessed for any of the operations at any time. With the provided modules, only the [procedural module](#) accesses the buffers belonging to other modules, but there is nothing that prevents other module-to-module communication or interaction. Modules receive the requests sent to them regardless of where they come from or how they are created. However most modules only accept one request at a time and the procedural module's requests from productions are delayed relative to their testing of a module's state i.e. a production queries the module to determine if it is free to accept requests but does not send a request until 50ms later (by default). So, if you plan on using the buffers for requests between modules you may want to take care with how the productions in any models which operate in that situation also issue requests.

Also, while it is possible to write to any buffer at any time, generally only the owning module should be storing chunks into its buffer. Often the module responds to a request by placing a chunk into the buffer, and thus if one were to arbitrarily write a chunk into the buffer while it was processing a request it could lead to problems in a model which made a request and was expecting a particular result. Again, the procedural module's productions are typically the primary source of requests and are typically written to expect a result consistent with the request. Thus, if you write a module or other code which modifies buffers explicitly, care should be taken when using that in conjunction with the normal production processing of a model.

If a buffer accepts requests then it is possible to also have the module indicate when those requests are complete. It is not necessary for a module to do so and the determination of whether it needs to do so or not is specified when the module is created. If a buffer does not “track” its requests then all of the requests will be automatically completed when they are made. If the buffer does track requests, then the module is responsible for indicating when those requests have been completed. That requires using the chunk-spec of the request it receives for indicating completion or that the module complete all requests by buffer or module name regardless of the specific request.

Testing a buffer to determine if a request has completed requires having the “tag” which is returned when making a request and specifying that tracking completion is desired. That tag can then be used to check whether the module which received the request has marked it as completed. Currently, only the utility module depends on the completion of requests for determining which productions should receive rewards when the completion credit assignment mechanism is enabled, and the procedural module does not provide a mechanism for accessing the tags which it has generated. However, other modules or code which make requests could track their own requests to any module with this mechanism.

For many of these commands there are two versions. One which performs an immediate action and one which schedules the action to occur as an event. In general, the scheduled versions of the requests and modifications are preferable to the non-scheduled ones because they will then be recorded in the trace and thus be detectable by other things and also be ordered appropriately. However, sometimes it may be necessary to use the immediate version, and in particular, the reading and testing functions often need to be done immediately because the current result is what is important.

In addition to the commands for using the buffers, there are also some additional commands described below for accessing other buffer information. One allows one to find the name of the buffer’s owning module. This may be useful when implementing tracing tools or other event monitoring utilities which might have access to only the buffer names. Another allows one to find the value of the buffer’s activation spread parameter without knowing the name of the parameter the module has made available for that purpose. That may be useful when developing alternative equations for the spreading activation calculation or for creating tools for monitoring and tracking the activation calculations.

Commands

buffer-read

Syntax:

```
buffer-read buffer -> [chunk-name | nil]  
schedule-buffer-read buffer time-delta {:module module } {:priority priority }  
                                {:output output } {:time-in-ms time-units } -> [event | nil]  
buffer-read-report buffer {:module module } -> [chunk-name | nil]
```

Arguments and Values:

buffer ::= a symbol which should name a buffer
chunk-name ::= a symbol that names the chunk which is in the buffer
time-delta ::= a number indicating when to schedule the event
module ::= a symbol which will be used as the module of the event and in the trace
priority ::= [**:max** | **:min** | *priority-val*]
priority-val ::= a number indicating the priority to use for the event
output ::= [**t** | **high** | **medium** | **low** | **nil**]
time-units ::= a generalized boolean indicating whether *time-delta* is in seconds or milliseconds
 (default is **nil** which means seconds)
event ::= an ACT-R event

Description:

The `buffer-read` commands are used to read the name of the chunk which is currently in the named buffer of the current model of the current meta-process.

For `buffer-read`, the name of the chunk is returned directly. If the buffer is empty the return value is **nil**. If the buffer named is invalid or there is no current model or current meta-process then a warning is printed and **nil** is returned.

`Schedule-buffer-read` is used to record a buffer reference in the trace of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'buffer-read-action  
                        :time-in-ms time-units  
                        :module module  
                        :priority priority  
                        :params (list buffer)  
                        :output output)
```

The default values for the parameters are **:none** for *module*, 0 for *priority*, and **t** for *output* if not provided. The `buffer-read-action` function essentially does nothing and is only there to record a reference for the trace. It returns the event that is scheduled. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled.

Buffer-read-report is used to read a buffer's chunk and have a record of the buffer-read recorded in the trace of the model. It is essentially a combination of buffer-read and schedule-buffer-read. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta 'buffer-read-action
                        :module module
                        :priority :max
                        :params (list buffer)
                        :output t)
```

The name of the chunk currently in the named buffer is returned. If the buffer is empty then **nil** is returned. If the buffer named is invalid or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Examples:

This example assumes that the model starts with no chunk in the goal buffer and has the :trace-detail parameter set to **high**.

```
1> (buffer-read 'goal)
NIL
```

```
2> (goal-focus free)
FREE
```

```
3> (run .1)
    0.000   GOAL                SET-BUFFER-CHUNK GOAL FREE REQUESTED NIL
    0.000   PROCEDURAL          CONFLICT-RESOLUTION
    0.000   -----             Stopped because no events left to process
0.0
4
NIL
```

```
4> (buffer-read 'goal)
FREE-0
```

```
5> (schedule-buffer-read 'goal .2)
#S(ACT-R-EVENT ...)
```

```
6> (mp-show-queue)
Events in the queue:
    0.200   NONE                BUFFER-READ-ACTION GOAL
    0.200   PROCEDURAL          CONFLICT-RESOLUTION
2
```

```
7> (run .2)
    0.200   NONE                BUFFER-READ-ACTION GOAL
    0.200   PROCEDURAL          CONFLICT-RESOLUTION
    0.200   -----             Stopped because time limit reached
0.2
2
NIL
```

```
8> (buffer-read-report 'goal :module 'test-module)
FREE-0
```

```

9> (mp-show-queue)
Events in the queue:
    0.200  TEST-MODULE          BUFFER-READ-ACTION GOAL
    0.200  PROCEDURAL          CONFLICT-RESOLUTION
2

E> (buffer-read 'bad-name)
#|Warning: Buffer-read called with an invalid buffer name BAD-NAME |#
NIL

E> (schedule-buffer-read 'bad-name .1)
#|Warning: schedule-buffer-read called with an invalid buffer name BAD-NAME |#
NIL

E> (schedule-buffer-read 'goal 'bad-time)
#|Warning: schedule-buffer-read called with a non-number time-delta: BAD-TIME |#
NIL

E> (schedule-buffer-read 'goal '.1 :priority 'bad)
#|Warning: schedule-buffer-read called with an invalid priority BAD |#
NIL

E> (buffer-read-report 'bad-name)
#|Warning: buffer-read-report called with an invalid buffer name BAD-NAME |#
NIL

E> (buffer-read 'goal)
#|Warning: buffer-read called with no current model. |#
NIL

E> (buffer-read 'goal)
#|Warning: buffer-read called with no current meta-process. |#
NIL

```

query-buffer

Syntax:

```

query-buffer buffer queries -> [t | nil]
schedule-query-buffer buffer queries time-delta {:module module} {:priority priority}
    {:output output} {:time-in-ms time-units} -> [event | nil]

```

Arguments and Values:

buffer ::= a symbol which should name a buffer
queries ::= [query-list | query-spec]
query-list ::= ({{mod } query value}*)
query-spec ::= an ACT-R chunk-spec which contains valid query information
mod ::= [= | -]
query ::= a symbol which names a query to test
value ::= a Lisp value to test for with the associated query
time-delta ::= a number indicating when to schedule the event
module ::= a symbol which will be used to as the module of the event and in the trace
priority ::= [:**max** | :**min** | priority-val]
priority-val ::= a number indicating the priority to use for the event
output ::= [t | **high** | **medium** | **low** | **nil**]
time-units ::= a generalized boolean indicating whether time-delta is in seconds or milliseconds
 (default is **nil** which means seconds)

event ::= an ACT-R event

Description:

The query-buffer commands are used to query the state of the buffer and/or its module in the current model of the current meta-process. The query provided may be specified either as a list of query items or a chunk-spec containing slot-specs that are valid queries. A query list must contain two or three items per query and may contain any number of queries. Within a single query the optional item is a modifier to use which can be either = or -, with = being a test that the query is true and - being a test that it is false. The default is = if not provided. The next item in the query is the name of the query and that is followed by the value to be tested for that query. A valid chunk-spec for a query would be one that was created with a specification as described for the query list.

For query-buffer each of the queries specified is tested from left to right and if they all return true then query-buffer returns **t**. If any query fails, then **nil** is returned without making any more queries. If the buffer named is invalid, any of the queries are not valid for the buffer or module, or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Schedule-query-buffer is used to record buffer querying in the trace of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'query-buffer-action
                          :time-in-ms time-units
                          :module module
                          :priority priority
                          :params (list buffer queries)
                          :output output
                          :details (format nil "query-buffer-action ~a" buffer))
```

The default values for the parameters are **:none** for module, 0 for priority, and **t** for output if not provided. The query-buffer-action function essentially does nothing and is only there to record a reference for the trace. It returns the event that is scheduled. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled.

Examples:

This example assumes that the model starts with no chunk in the goal and buffer and has the :trace-detail parameter set to **high**.

```
1> (query-buffer 'goal '(state free))
T
```

```

2> (query-buffer 'goal '(= buffer empty error nil))
T

3> (query-buffer 'goal (define-chunk-spec - buffer full error nil))
T

4> (query-buffer 'goal nil)
T

5> (query-buffer 'goal '(state busy))
NIL

6> (schedule-query-buffer 'goal '(buffer empty) .1)
#S(ACT-R-EVENT ...)

7> (mp-show-queue)
Events in the queue:
    0.000  NONE                CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS ...)
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    0.100  NONE                QUERY-BUFFER-ACTION GOAL
3

8> (run .1)
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    0.100  NONE                QUERY-BUFFER-ACTION GOAL
    0.100  PROCEDURAL          CONFLICT-RESOLUTION
    0.100  -----            Stopped because time limit reached
0.1
4
NIL

E> (query-buffer 'bad-name nil)
#|Warning: query-buffer called with an invalid buffer name BAD-NAME |#
NIL

E> (query-buffer 'goal '(bad-query free))
#|Warning: Invalid query name BAD-QUERY in call to define-query-spec. |#
#|Warning: Invalid query-buffer of buffer GOAL with queries-list-or-spec (BAD-QUERY FREE)
|#
NIL

E> (schedule-query-buffer 'bad-name nil .1)
#|Warning: schedule-query-buffer called with an invalid buffer name BAD-NAME |#
NIL

E> (query-buffer 'goal '(< state free))
#|Warning: Query specs only allow = or - modifiers. |#
#|Warning: Invalid query-buffer of buffer GOAL with queries-list-or-spec (< STATE FREE) |#
NIL

E> (schedule-query-buffer 'goal (define-chunk-spec < state free) .1)
#|Warning: schedule-query-buffer called with an invalid query specification
#S(ACT-R-CHUNK-SPEC ...)) |#
NIL

E> (schedule-query-buffer 'goal nil nil)
#|Warning: schedule-query-buffer called with non-number time-delta: NIL |#
NIL

E> (schedule-query-buffer 'goal nil 1.0 :priority 'bad)
#|Warning: schedule-query-buffer called with an invalid priority BAD |#
NIL

E> (query-buffer 'goal nil)
#|Warning: query-buffer called with no current model. |#
NIL

```

```
E> (query-buffer 'goal nil)
#|Warning: query-buffer called with no current meta-process.|#
NIL
```

clear-buffer

Syntax:

```
clear-buffer buffer -> [chunk-name | nil]
schedule-clear-buffer buffer time-delta {:module module} {:priority priority}
                                     {:output output} {:time-in-ms time-units} -> [event | nil]
```

Arguments and Values:

buffer ::= a symbol which should name a buffer
chunk-name ::= a symbol that names the chunk which is in the buffer
time-delta ::= a number indicating when to schedule the event
module ::= a symbol which will be used to as the module of the event and in the trace
priority ::= [**:max** | **:min** | *priority-val*]
priority-val ::= a number indicating the priority to use for the event
output ::= [**t** | **high** | **medium** | **low** | **nil**]
time-units ::= a generalized boolean indicating whether *time-delta* is in seconds or milliseconds
(default is **nil** which means seconds)
event ::= an ACT-R event

Description:

The clear-buffer commands are used to clear any chunk which may be in a buffer which will leave the buffer empty, and it will also clear the buffer's failure flag if it is set. Any module which is [monitoring for buffer clearing](#) will be notified that the chunk has been cleared – in particular the [declarative module](#) will merge the cleared chunk into the model's declarative memory.

For clear-buffer, the buffer is cleared immediately and the name of the chunk which is in the buffer currently is returned. If the buffer is empty the return value is **nil**. If the buffer named is invalid or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Schedule-clear-buffer is used to schedule the clearing of the buffer such that it occurs during the running of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'clear-buffer
                        :time-in-ms time-units
                        :module module
                        :priority priority
                        :params (list buffer)
                        :output output)
```

The default values for the parameters are **:none** for module, 0 for priority, and **low** for output if not provided. It returns the event that is scheduled. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled.

Examples:

This example assumes that the model starts with no chunk in the goal buffer.

```
1> (clear-buffer 'goal)
NIL

2> (goal-focus free)
FREE

3> (run 1)
   0.000   GOAL           SET-BUFFER-CHUNK GOAL FREE REQUESTED NIL
   0.000   PROCEDURAL    CONFLICT-RESOLUTION
   0.000   -----      Stopped because no events left to process
0.0
4
NIL

4> (clear-buffer 'goal)
FREE-0

5> (buffer-read 'goal)
NIL

6> (goal-focus free)
FREE

7> (schedule-clear-buffer 'goal .2 :module 'example)
#S(ACT-R-EVENT ...)

8> (run 1)
   0.200   EXAMPLE       CLEAR-BUFFER GOAL
   0.200   PROCEDURAL    CONFLICT-RESOLUTION
   0.200   -----      Stopped because no events left to process
0.2
2
NIL

9> (buffer-read 'goal)
NIL

E> (clear-buffer 'bad-name)
#|warning: clear-buffer called with an invalid buffer name BAD-NAME |#
NIL

E> (schedule-clear-buffer 'bad-name .1)
#|warning: schedule-clear-buffer called with an invalid buffer name BAD-NAME |#
NIL

E> (schedule-clear-buffer 'goal nil)
#|warning: schedule-clear-buffer called with a non-number time-delta: NIL |#
NIL

E> (clear-buffer 'goal)
#|warning: clear-buffer called with no current model. |#
NIL
```

```
E> (clear-buffer 'goal)
#|Warning: clear-buffer called with no current meta-process.|#
NIL
```

set-buffer-chunk

Syntax:

```
set-buffer-chunk buffer chunk-name {:requested requested} -> [new-chunk-name | nil]
schedule-set-buffer-chunk buffer chunk-name time-delta {:module module} {:priority priority}
{:output output} {:requested requested} {:time-in-ms time-units} -> [event | nil]
```

Arguments and Values:

buffer ::= a symbol which should name a buffer
chunk-name ::= a symbol which should name a chunk
new-chunk-name ::= a symbol which will name a new chunk
requested ::= a generalized boolean
time-delta ::= a number indicating when to schedule the event
module ::= a symbol which will be used to as the module of the event and in the trace
priority ::= [**:max** | **:min** | *priority-val*]
priority-val ::= a number indicating the priority to use for the event
output ::= [**t** | **high** | **medium** | **low** | **nil**]
time-units ::= a generalized boolean indicating whether *time-delta* is in seconds or milliseconds
(default is **nil** which means seconds)
event ::= an ACT-R event

Description:

The `set-buffer-chunk` commands are used to copy a chunk into a buffer. It will always make a copy of the chunk specified, and then place that copy into the buffer. Before it places the new chunk into the buffer, it will first clear the buffer. Thus, any chunk which may be there will be cleared as described for the [clear-buffer command](#). The buffer clearing is not scheduled, thus there will not be an event in the trace to indicate explicitly that the buffer was cleared.

For `set-buffer-chunk`, the buffer is cleared and then set immediately with the copied chunk and the name of the chunk which is placed into the buffer is returned. If either the buffer or chunk named is invalid or there is no current model or current meta-process then a warning is printed and **nil** is returned.

The `requested` keyword parameter specifies whether the buffer will indicate that the chunk was requested or not. If `requested` is specified as **t** (which is the default) or any other true value, then the setting of the chunk is to be interpreted as being the result of a request to a module and the query of that buffer for “buffer requested” will be true. If the `requested` keyword parameter is specified as **nil**,

which should be done when the setting of the chunk is not the result of a module request, then the “buffer requested” query for the buffer will be **nil** and the “buffer unrequested” query will be true. An example of when the requested value should be **nil** is the buffer stuffing which the [visual-location buffer](#) performs when there is a screen change. That does not happen because of an explicit request to the module, and thus if a chunk is placed into the buffer it should be marked accordingly.

Schedule-set-buffer-chunk is used to schedule the setting of the buffer such that it occurs during the running of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'set-buffer-chunk
                          :time-in-ms time-units
                          :module module
                          :priority priority
                          :params (if requested
                                   (list buffer-name chunk-name)
                                   (list buffer-name chunk-name :requested requested))
                          :output output)
```

The default values for the parameters are **none** for module, 0 for priority, and **low** for output if not provided. It returns the event that is scheduled. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled.

Examples:

```
1> (define-chunks (a) (b color b) (c name c))
(A B C)
```

```
2> (dm)
NIL
```

```
3> (set-buffer-chunk 'goal 'a)
A-0
```

```
4> (buffer-chunk goal)
GOAL: A-0 [A]
A-0
```

```
(A-0)
```

```
5> (dm)
NIL
```

```
6> (schedule-set-buffer-chunk 'goal 'b .1)
#S(ACT-R-EVENT ...)
```

```
7> (run 1)
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.100 NONE SET-BUFFER-CHUNK GOAL B
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 ----- Stopped because no events left to process
```

```
0.1
4
NIL
```

```

8> (dm)
A-0

(A-0)

9> (buffer-chunk goal)
GOAL: B-0 [B]
B-0
  COLOR B

(B-0)

10> (query-buffer 'goal '(buffer requested))
T

11> (set-buffer-chunk 'goal 'c :requested nil)
C-0

12> (buffer-chunk goal)
GOAL: C-0 [C]
C-0
  NAME C

(C-0)

13> (dm)
B-0
  COLOR B

A-0

(B-0 A-0)

14> (query-buffer 'goal '(buffer requested))
NIL

15> (query-buffer 'goal '(buffer unrequested))
T

E> (set-buffer-chunk 'bad-name 'a)
#|Warning: set-buffer-chunk called with an invalid buffer name BAD-NAME |#
NIL

E> (set-buffer-chunk 'goal 'bad-name)
#|Warning: set-buffer-chunk called with an invalid chunk name BAD-NAME |#
NIL

E> (schedule-set-buffer-chunk 'goal 'b nil)
#|Warning: schedule-set-buffer-chunk called with time-delta that is not a number: NIL |#
NIL

E> (set-buffer-chunk 'goal 'a)
#|Warning: set-buffer-chunk called with no current model. |#
NIL

E> (set-buffer-chunk 'goal 'b)
#|Warning: set-buffer-chunk called with no current meta-process. |#
NIL

```

overwrite-buffer-chunk

Syntax:

overwrite-buffer-chunk *buffer chunk-name* **{:requested requested}** -> [new-chunk-name | **nil**]
schedule-overwrite-buffer-chunk *buffer chunk-name time-delta* **{:module module}** **{:priority priority}**
{:output output} **{:requested requested}** **{:time-in-ms time-units}** -> [event | **nil**]

Arguments and Values:

buffer ::= a symbol which should name a buffer
chunk-name ::= a symbol which should name a chunk
requested ::= a generalized boolean
new-chunk-name ::= a symbol which will name a new chunk
time-delta ::= a number indicating when to schedule the event
module ::= a symbol which will be used to as the module of the event and in the trace
priority ::= [**:max** | **:min** | priority-val]
priority-val ::= a number indicating the priority to use for the event
output ::= [**t** | **high** | **medium** | **low** | **nil**]
time-units ::= a generalized boolean indicating whether time-delta is in seconds or milliseconds
(default is **nil** which means seconds)
event ::= an ACT-R event

Description:

The `overwrite-buffer-chunk` commands are used to copy a chunk into a buffer like [set-buffer-chunk](#). It will always make a copy of the chunk specified, and then place that copy into the buffer. The difference between `overwrite-buffer-chunk` and `set-buffer-chunk` is that the `overwrite` operation does not clear the buffer first. Thus, if there is a chunk in the buffer when the `overwrite-buffer-chunk` command is issued there will be no clearing of that chunk and no modules will be notified that it has been cleared. One specific consequence of that is that the previous chunk in the buffer, if there was one, will not be collected by the declarative module and added to the model's declarative memory.

Typically, one will use `set-buffer-chunk` when building a module or otherwise working with the buffers. However, there are rare situations where the model or a module may want to destructively erase a chunk which is in a buffer with this command.

The `requested` keyword parameter specifies whether the buffer will indicate that the chunk was requested or not. If `requested` is specified as **t** or any other true value, then the setting of the chunk is to be interpreted as being the result of a request to a module and the query of that buffer for "buffer requested" will be true. If the `requested` keyword parameter is specified as **nil** (which is the default value), which should be done when the setting of the chunk is not the result of a module request, then the "buffer requested" query for the buffer will be **nil** and the "buffer unrequested" query will be true.

For `overwrite-buffer-chunk`, the buffer is set immediately to hold the newly copied chunk and the name of the copy of the chunk which is placed into the buffer is returned. If either the buffer or

chunk named is invalid or there is no current model or current meta-process then a warning is printed and **nil** is returned.

Schedule-overwrite-buffer-chunk is used to schedule the setting of the buffer such that it occurs during the running of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'overwrite-buffer-chunk
                        :time-in-ms time-units
                        :module module
                        :priority priority
                        :params (if requested
                                (list buffer-name chunk-name)
                                (list buffer-name chunk-name :requested requested))
                        :output output)
```

The default values for the parameters are **none** for module, 0 for priority, and **low** for output if not provided. It returns the event that is scheduled. If any of the parameters are not valid or there is no current model then a warning is printed, **nil** is returned and nothing is scheduled.

Examples:

```
1> (define-chunks (a) (b color b) (c name c))
(A B C)

2> (dm)
NIL

3> (overwrite-buffer-chunk 'goal 'a)
A-0

4> (buffer-chunk goal)
GOAL: A-0 [A]
A-0

(A-0)

5> (dm)
NIL

6> (schedule-overwrite-buffer-chunk 'goal 'b .1)
#S(ACT-R-EVENT ...)

7> (run 1)
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.100  NONE                OVERWRITE-BUFFER-CHUNK GOAL B REQUESTED NIL
  0.100  PROCEDURAL          CONFLICT-RESOLUTION
  0.100  -----            Stopped because no events left to process

0.1
4
NIL

8> (dm)
NIL

9> (buffer-chunk goal)
GOAL: B-0 [B]
B-0
  COLOR B
```

(B-0)

```
10> (query-buffer 'goal '(buffer requested))
NIL
```

```
11> (overwrite-buffer-chunk 'goal 'c :requested t)
C-0
```

```
12> (buffer-chunk goal)
GOAL: C-0 [C]
C-0
  NAME  C
```

(C-0)

```
13> (dm)
NIL
```

```
14> (query-buffer 'goal '(buffer requested))
T
```

```
E> (overwrite-buffer-chunk 'bad-name 'a)
#|Warning: overwrite-buffer-chunk called with an invalid buffer name BAD-NAME |#
NIL
```

```
E> (overwrite-buffer-chunk 'goal 'bad-name)
#|Warning: overwrite-buffer-chunk called with an invalid chunk name BAD-NAME |#
NIL
```

```
E> (schedule-overwrite-buffer-chunk 'goal 'a nil)
#|Warning: schedule-overwrite-buffer-chunk called with a non-number time-delta: NIL |#
NIL
```

```
E> (overwrite-buffer-chunk 'goal 'a)
#|Warning: overwrite-buffer-chunk called with no current model. |#
NIL
```

```
E> (overwrite-buffer-chunk 'goal 'a)
#|Warning: overwrite-buffer-chunk called with no current meta-process. |#
NIL
```

mod-buffer-chunk

Syntax:

```
mod-buffer-chunk buffer modifications -> [ chunk-name | nil ]
schedule-mod-buffer-chunk buffer modifications time-delta {:module module} {:priority priority}
  {:output output} {:time-in-ms time-units} -> [event | nil]
```

Arguments and Values:

buffer ::= a symbol which should name a buffer

chunk-name ::= a symbol which will be the name of the chunk in the buffer named

modifications ::= [mod-list | mod-spec]

mod-list ::= ({slot-name slot-value}*)

mod-spec ::= an ACT-R chunk-spec which contains valid modification information

slot-name ::= a symbol that should be the name of a slot of the chunk in the named buffer

slot-value ::= a Lisp value to set for the corresponding slot-name of the chunk in the named buffer

time-delta ::= a number indicating when to schedule the event

module ::= a symbol which will be used to as the module of the event and in the trace
priority ::= [:max | :min | priority-val]
priority-val ::= a number indicating the priority to use for the event
output ::= [t | high | medium | low | nil]
time-units ::= a generalized boolean indicating whether time-delta is in seconds or milliseconds
(default is **nil** which means seconds)
event ::= an ACT-R event

Description:

The mod-buffer-chunk commands are used to make modifications to the chunk which is in a buffer. It works similar to the [mod-chunk command](#), but there are a few differences between mod-chunk and mod-buffer-chunk. The first is that instead of providing the name of a chunk, a buffer is named and the chunk which is in that buffer is the one which is modified. Another is that in addition to providing a list of slot-value pairs to specify the modifications to make mod-buffer-chunk will also accept a [chunk-spec](#) with slot-specs that indicate the modifications to make (such a chunk-spec must only use the modifier = in the slot-specs). Finally, if a slot specified in the modification is not currently a possible slot it will be added to the possible slots using the [extend-possible-slots command](#) before the modifications are performed.

If there is a chunk in the buffer which is named and the modifications provided are valid then those modifications are made to that chunk. To be valid, the modifications may only specify each slot once. If any value specified for a slot in the modification is a symbol and not the name of a chunk in the current model then it is created as a new chunk with no slots and a warning is displayed before the modification is made.

For mod-buffer-chunk, if there is not a chunk in the named buffer, the modifications provided are not valid, or there is no current model or current meta-process then a warning is displayed, no changes are made, and **nil** is returned.

If there is a chunk in the buffer and the modifications are valid, then the chunk is modified and the name of that chunk is returned.

Schedule-mod-buffer-chunk is used to schedule the modifications to the chunk in the buffer such that it occurs during the running of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'mod-buffer-chunk
                        :time-in-ms time-units
                        :module module)
```

```

:priority priority
:params (list buffer modifications)
:details (format nil "~s ~s" 'mod-buffer-chunk buffer-name)
:output output)

```

The default values for the parameters are **none** for module, 0 for priority, and **low** for output if not provided. It returns the event that is scheduled. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled.

When `schedule-mod-buffer-chunk` is called with a slot that is not currently a possible slot name the extension will be performed immediately at the time it is called and not at the scheduled time for the modification.

Examples:

```

1> (define-chunks (a))
(A)

2> (set-buffer-chunk 'goal 'a)
A-0

3> (buffer-chunk goal)
GOAL: A-0 [A]
A-0

(A-0)

4> (mod-buffer-chunk 'goal '(color blue))
A-0

5> (buffer-chunk goal)
GOAL: A-0
A-0
    COLOR  BLUE

(A-0)

6> (schedule-mod-buffer-chunk 'goal (define-chunk-spec size 10) .1)
#S(ACT-R-EVENT ...)

7> (mp-show-queue)
Events in the queue:
    0.000  NONE                CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS ...)
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    0.100  NONE                MOD-BUFFER-CHUNK GOAL
3

8> (run .1)
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    0.100  NONE                MOD-BUFFER-CHUNK GOAL
    0.100  PROCEDURAL          CONFLICT-RESOLUTION
    0.100  -----            Stopped because time limit reached
0.1
4
NIL

9> (buffer-chunk goal)

```

```

GOAL: A-0
A-0
  COLOR BLUE
  SIZE 10

(A-0)

10E> (mod-buffer-chunk 'goal '(size 11 new-slot new-chunk))
#|Warning: Chunks extended with slot NEW-SLOT during a chunk-spec definition.|#
#|Warning: Creating chunk NEW-CHUNK with no slots|#
A-0

11> (buffer-chunk goal)
GOAL: A-0
A-0
  COLOR BLUE
  SIZE 11
  NEW-SLOT NEW-CHUNK

(A-0)

12E> (schedule-mod-buffer-chunk 'goal '(new-new-slot t color nil) .1)
#|Warning: Chunks extended with slot NEW-NEW-SLOT during a chunk-spec definition.|#
#S(ACT-R-EVENT ...)

13> (buffer-chunk goal)
GOAL: A-0
A-0
  COLOR BLUE
  SIZE 11
  NEW-SLOT NEW-CHUNK

(A-0)

14> (run .1)
      0.200  NONE                MOD-BUFFER-CHUNK GOAL
      0.200  PROCEDURAL          CONFLICT-RESOLUTION
      0.200  -----            Stopped because time limit reached
0.1
2
NIL

15> (buffer-chunk goal)
GOAL: A-0
A-0
  SIZE 11
  NEW-SLOT NEW-CHUNK
  NEW-NEW-SLOT T

(A-0)

E> (mod-buffer-chunk 'not-a-buffer nil)
#|Warning: mod-buffer-chunk called with an invalid buffer name NOT-A-BUFFER|#
NIL

E> (mod-buffer-chunk 'goal '(color blue color red))
#|Warning: mod-buffer-chunk called with an invalid modification (COLOR BLUE COLOR RED)|#
NIL

E> (schedule-mod-buffer-chunk 'goal '(color blue) nil)
#|Warning: schedule-mod-buffer-chunk called with non-number time-delta: NIL|#
NIL

E> (mod-buffer-chunk 'goal nil)
#|Warning: mod-buffer-chunk called with no current model.|#
NIL

```

```
E> (mod-buffer-chunk 'goal nil)
#|Warning: mod-buffer-chunk called with no current meta-process.|#
NIL
```

set-buffer-failure

Syntax:

```
set-buffer-failure buffer {:requested requested} {:ignore-if-full if-full} -> [t | nil]
```

Arguments and Values:

buffer ::= a symbol which should name a buffer

requested ::= a generalized boolean

if-full ::= a generalized boolean

Description:

The `set-buffer-failure` command is used to set the failure flag in a buffer. A buffer's failure flag will remain set until the buffer is cleared. If the flag is successfully set then the function returns **t**.

If *buffer* does not name a buffer in the current model, or there is no current model or current meta-process then a warning is printed and **nil** is returned.

The *requested* keyword parameter specifies whether the buffer will indicate that the failure was requested or not. If *requested* is specified as **t** (which is the default) or any other true value, then the setting of the failure flag is to be interpreted as being the result of a request to a module having failed and the query of that buffer for "buffer requested" will be true. If the *requested* keyword parameter is specified as **nil**, then the "buffer requested" query for the buffer will be **nil** and the "buffer unrequested" query will be true.

The failure flag cannot be set when there is currently a chunk in the specified buffer. The *if-full* keyword parameter specifies what to do if there is a chunk in the buffer when `set-buffer-failure` is called. If the parameter is specified as **nil** (the default value) then it will print a warning and return **nil** if there is a chunk in the buffer. If the parameter is true then no warning will be displayed and **nil** will be returned.

Examples:

```
1> (buffer-status goal)
GOAL:
  buffer empty      : T
  buffer full      : NIL
```

```

buffer failure      : NIL
buffer requested   : NIL
buffer unrequested : NIL
state free        : T
state busy        : NIL
state error       : NIL
(GOAL)

2> (set-buffer-failure 'goal)
T

3> (buffer-status goal)
GOAL:
buffer empty      : NIL
buffer full       : NIL
buffer failure    : T
buffer requested  : T
buffer unrequested : NIL
state free       : T
state busy       : NIL
state error      : NIL
(GOAL)

4> (clear-buffer 'goal)
NIL

5> (buffer-status goal)
GOAL:
buffer empty      : T
buffer full       : NIL
buffer failure    : NIL
buffer requested  : NIL
buffer unrequested : NIL
state free       : T
state busy       : NIL
state error      : NIL
(GOAL)

6> (set-buffer-failure 'goal :requested nil)
T

7> (buffer-status goal)
GOAL:
buffer empty      : NIL
buffer full       : NIL
buffer failure    : T
buffer requested  : NIL
buffer unrequested : T
state free       : T
state busy       : NIL
state error      : NIL
(GOAL)

8> (set-buffer-chunk 'goal 'red)
RED-0

9> (buffer-status goal)
GOAL:
buffer empty      : NIL
buffer full       : T
buffer failure    : NIL
buffer requested  : T
buffer unrequested : NIL
state free       : T
state busy       : NIL
state error      : NIL
(GOAL)

```

```

10E> (set-buffer-failure 'goal)
#|Warning: cannot set the failure flag when there is a chunk in the GOAL buffer |#
NIL

11> (set-buffer-failure 'goal :ignore-if-full t)
NIL

12> (buffer-status goal)
GOAL:
  buffer empty      : NIL
  buffer full       : T
  buffer failure    : NIL
  buffer requested  : T
  buffer unrequested : NIL
  state free        : T
  state busy        : NIL
  state error       : NIL
(GOAL)

E> (set-buffer-failure 'not-a-buffer)
#|Warning: set-buffer-failure called with an invalid buffer name NOT-A-BUFFER |#
NIL

E> (set-buffer-failure 'goal)
#|Warning: set-buffer-failure called with no current model. |#
NIL

E> (set-buffer-failure 'goal)
#|Warning: set-buffer-failure called with no current meta-process. |#
NIL

```

module-request

Syntax:

```

module-request buffer chunk-spec {track} -> [ t | nil {tag} ]
schedule-module-request buffer chunk-spec time-delta {:module module} {:priority priority}
  {:output output} {:details details} {:time-in-ms time-units}
  {:track track} -> [event | nil {tag}]

```

Arguments and Values:

buffer ::= a symbol which should name a buffer
chunk-spec ::= an ACT-R chunk-spec
track ::= a generalized boolean indicating whether this request should be tracked for completion
tag ::= a value which can be used check the completion status of the request
time-delta ::= a number indicating when to schedule the event
module ::= a symbol which will be used to as the module of the event and in the trace
priority ::= [**:max** | **:min** | priority-val]
priority-val ::= a number indicating the priority to use for the event
output ::= [**t** | **high** | **medium** | **low** | **nil**]
details ::= a string to display in the trace when the request occurs
time-units ::= a generalized boolean indicating whether time-delta is in seconds or milliseconds
 (default is **nil** which means seconds)
event ::= an ACT-R event

Description:

The `module-request` commands are used to send a request to a module through its buffer. This is the only recommended way to interact with a module to make requests.

If the buffer name is valid and a valid `chunk-spec` is provided, then that `chunk-spec` will be sent to the buffer's module as a request. Note that the command does not implicitly clear the buffer as occurs for requests from the productions of a model. If the buffer should be cleared that must be done explicitly in conjunction with the request.

For `module-request`, if the buffer name and `chunk-spec` are valid and the named buffer's module accepts requests then a value of `t` is returned as the first value. If the buffer name is invalid, an invalid `chunk-spec` is provided, the buffer's module does not accept requests, or there is no current model or current meta-process then a warning is displayed and `nil` is returned as the first value. If the `track` parameter is specified as true and the request is valid then a second value will be returned which is a tag that can be used to test the completion of this request using the [completed-request-p](#) command. The exact representation of a tag is not specified in the API and thus it should not be used or modified in any other way.

The only test performed upon the `chunk-spec` to determine if it is valid (assuming that it is a `chunk-spec`) is to verify that any request parameters provided are valid for the buffer to which it is being sent. Any other processing of the request for validity is up to the module which receives the request.

When using `module-request`, even though the request itself is not scheduled the actions performed by the module in response to that request should be (at least if the module is written according to the recommended guidelines). Thus, it is likely that no changes will occur until the model is run.

It is recommended that all requests be scheduled using `schedule-module-request`.

`Schedule-module-request` is used to schedule the sending of the request to the buffer's module such that it occurs during the running of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'module-request
  :time-in-ms time-units
  :module module
  :priority priority
  :params (list buffer chunk-spec)
  :details (if (stringp details)
              details
              (format nil "~s ~s" 'module-request buffer-name)))
```

```
:output output)
```

The default values for the parameters are **none** for module, 0 for priority, **medium** for output, and **nil** for details. It returns the event that is scheduled and the tag if the track keyword parameter is specified as true. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled. Note that for schedule-module-request the test whether the module accepts requests is not performed until the event occurs. Thus any warning about that will not occur until the model is run.

Examples:

These examples assume that there is a module called dummy with a buffer called dummy which does not accept requests.

```
1> (buffer-chunk goal)
GOAL: NIL
(NIL)

2> (module-request 'goal (define-chunk-spec value free))
T

3> (buffer-chunk goal)
GOAL: NIL
(NIL)

4> (mp-show-queue)
Events in the queue:
    0.000  NONE                CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS ...)
    0.000  GOAL                CREATE-NEW-BUFFER-CHUNK GOAL
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
3

5> (run .001)
    0.000  GOAL                SET-BUFFER-CHUNK GOAL CHUNK0
    0.000  PROCEDURAL          CONFLICT-RESOLUTION
    0.000  -----            Stopped because no events left to process
0.0
5
NIL

6> (buffer-chunk goal)
GOAL: CHUNK0-0
CHUNK0-0
  VALUE  FREE

(CHUNK0-0)

7> (schedule-module-request 'goal (define-chunk-spec value busy) .1)
#S(ACT-R-EVENT ...)

8> (buffer-chunk goal)
GOAL: CHUNK0-0
CHUNK0-0
  VALUE  FREE

(CHUNK0-0)

9> (mp-show-queue)
```

```

Events in the queue:
  0.100  NONE          MODULE-REQUEST GOAL
  0.100  PROCEDURAL   CONFLICT-RESOLUTION
2
10> (run .1)
  0.100  NONE          MODULE-REQUEST GOAL
  0.100  GOAL         SET-BUFFER-CHUNK GOAL CHUNK1
  0.100  PROCEDURAL   CONFLICT-RESOLUTION
  0.100  -----      Stopped because time limit reached
0.1
5
NIL

11> (buffer-chunk goal)
GOAL: CHUNK1-0
CHUNK1-0
  VALUE  BUSY

(CHUNK1-0)

E> (module-request 'dummy (define-chunk-spec))
#|Warning: Module DUMMY does not handle requests.|#
NIL

1> (schedule-module-request 'dummy (define-chunk-spec) 0)
#S(ACT-R-EVENT ...)

2E> (run .1)
  0.100  NONE          MODULE-REQUEST DUMMY
#|Warning: Module DUMMY does not handle requests.|#
  0.100  PROCEDURAL   CONFLICT-RESOLUTION
  0.100  -----      Stopped because no events left to process
0.0
2
NIL

E> (module-request 'not-a-buffer (define-chunk-spec))
#|Warning: module-request called with an invalid buffer name NOT-A-BUFFER|#
NIL

E> (module-request 'goal 'not-a-chunk-spec)
#|Warning: module-request called with an invalid chunk-spec NOT-A-CHUNK-SPEC|#
NIL

E> (schedule-module-request 'goal (define-chunk-spec) nil)
#|Warning: schedule-module-request called with a non-number time-delta: NIL|#
NIL

E> (module-request 'goal *spec*)
#|Warning: module-request called with no current model.|#
NIL

E> (module-request 'goal *spec*)
#|Warning: module-request called with no current meta-process.|#
NIL

```

module-mod-request

Syntax:

```

module-mod-request buffer modifications {track} -> [ t | nil {tag}]
schedule-module-mod-request buffer modifications time-delta {:module module} {:priority priority}
  {:output output} {:time-in-ms time-units}

```

`{:track track} -> [event | nil {tag}]`

Arguments and Values:

buffer ::= a symbol which should name a buffer

modifications ::= [mod-list | mod-spec]

mod-list ::= ({slot-name slot-value }*)

mod-spec ::= an ACT-R chunk-spec which contains valid modification information

slot-name ::= a symbol that should be the name of a slot of the chunk in the named buffer

slot-value ::= a Lisp value to set for the corresponding slot-name of the chunk in the named buffer

track ::= a generalized boolean indicating whether this request should be tracked for completion

tag ::= a value which can be used check the completion status of the request

time-delta ::= a number indicating when to schedule the event

module ::= a symbol which will be used to as the module of the event and in the trace

priority ::= [:max | :min | priority-val]

priority-val ::= a number indicating the priority to use for the event

output ::= [t | high | medium | low | nil]

time-units ::= a generalized boolean indicating whether time-delta is in seconds or milliseconds
(default is **nil** which means seconds)

event ::= an ACT-R event

Description:

The `module-mod-request` commands are used to send a modification request to a module through its buffer. This is the only recommended way to interact with a module to make modification requests. A modification request requires that there be a chunk in the specified buffer at the time these commands are issued.

For `module-mod-request`, if the buffer name is valid, the modifications are valid, there is a chunk in the specified buffer, and the named buffer's module accepts requests then a value of **t** is returned as the first value. If the buffer name is invalid, an invalid modification is provided, the buffer is empty, the buffer's module does not accept modification requests, or there is no current model or current meta-process then a warning is displayed and **nil** is returned as the first value. If the `track` parameter is specified as true and the modification is valid then a second value will be returned which is a tag that can be used to test the completion of this request using the [completed-request-p](#) command. The exact representation of a tag is not specified in the API and thus it should not be used or modified in any other way.

The modifications may be specified as a list of slot-value pairs or a [chunk-spec](#) with slot-specs that indicate the modifications to make (such a chunk-spec must only use the modifier = in the slot-specs and may contain request parameters valid for the specified buffer). If a slot specified in a

modification list is not currently a possible slot it will be added to the possible slots using the [extend-possible-slots command](#) at the time `module-mod-request` function is called. There is no test performed by `module-request` to determine if the modification list provided is acceptable to the module prior to sending it other than the test for valid request parameters. All processing of that nature is done by the module once it has received the modification request.

When using `module-mod-request`, even though the request itself is not scheduled the actions performed by the module in response to the request should be (at least if the module is written according to the recommended guidelines). Thus, it is likely that no changes will occur until the model is run.

It is recommended that all modification requests be scheduled using `schedule-module-mod-request`.

`Schedule-module-mod-request` is used to schedule the sending of the modification request to the buffer's module such that it occurs during the running of the model. It schedules an event to occur as if the following was executed:

```
(schedule-event-relative time-delta 'module-mod-request
                        :time-in-ms time-units
                        :module module
                        :priority priority
                        :params (list buffer modifications)
                        :details (format nil "~s ~s" 'module-mod-request buffer-name)
                        :output output)
```

The default values for the parameters are **none** for module, 0 for priority, and **low** for output if not provided. It returns the event that is scheduled and the tag if the `track` keyword parameter is specified as true. If any of the parameters are not valid or there is no current model or current meta-process then a warning is printed, **nil** is returned and nothing is scheduled. Note that for `schedule-module-mod-request` the test whether the module accepts modification requests is not performed until the event occurs. Thus the warning will not occur until during the model run.

Examples:

These examples assume that there is a module called `dummy` with a buffer called `dummy` which does not accept modification requests.

```
1> (set-buffer-chunk 'goal 'free)
FREE-0

2> (module-mod-request 'goal '(value t))
T

3> (buffer-chunk goal)
```

```

GOAL: FREE-0 [FREE]
FREE-0
  NAME  FREE

(FREE-0)

4> (mp-show-queue)
Events in the queue:
  0.000  NONE          CHECK-FOR-ESC-NIL #S(CENTRAL-PARAMETERS ...)
  0.000  GOAL          MOD-BUFFER-CHUNK GOAL
  0.000  PROCEDURAL   CONFLICT-RESOLUTION
3

5> (run .01)
  0.000  GOAL          MOD-BUFFER-CHUNK GOAL
  0.000  PROCEDURAL   CONFLICT-RESOLUTION
  0.000  -----      Stopped because no events left to process
0.0
3
NIL

6> (buffer-chunk goal)
GOAL: FREE-0
FREE-0
  NAME  FREE
  VALUE T

(FREE-0)

7> (schedule-module-mod-request 'goal (define-chunk-spec name busy) 0)
#S(ACT-R-EVENT :MSTIME 0 :PRIORITY 0 :ACTION MODULE-MOD-REQUEST :MODEL F :MP ...)

8> (mp-show-queue)
Events in the queue:
  0.000  NONE          MODULE-MOD-REQUEST GOAL
  0.000  PROCEDURAL   CONFLICT-RESOLUTION
2

9> (buffer-chunk goal)
GOAL: FREE-0
FREE-0
  NAME  FREE
  VALUE T

(FREE-0)

10> (run .1)
  0.000  NONE          MODULE-MOD-REQUEST GOAL
  0.000  GOAL          MOD-BUFFER-CHUNK GOAL
  0.000  PROCEDURAL   CONFLICT-RESOLUTION
  0.000  -----      Stopped because no events left to process
0.0
3
NIL

11> (buffer-chunk goal)
GOAL: FREE-0
FREE-0
  NAME  BUSY
  VALUE T

(FREE-0)

1E> (module-mod-request 'dummy nil)
#|Warning: module-mod-request called with no chunk in buffer DUMMY|#
NIL

```

```

2> (set-buffer-chunk 'dummy 'busy)
BUSY-1

3E> (module-mod-request 'dummy nil)
#|Warning: Module DUMMY does not support buffer modification requests.|#
NIL

E> (module-mod-request 'not-a-buffer nil)
#|Warning: module-mod-request called with an invalid buffer name NOT-A-BUFFER|#
NIL

E> (module-mod-request 'goal 'bad-modification)
#|Warning: module-mod-request called with an invalid modification BAD-MODIFICATION|#
NIL

E> (module-mod-request 'goal '(new-slot t))
#|Warning: Chunks extended with slot NEW-SLOT during a chunk-spec definition.|#
T

E> (schedule-module-mod-request 'goal nil nil)
#|Warning: schedule-module-mod-request called with a non-number time-delta: NIL|#
NIL

E> (module-mod-request 'goal nil)
#|Warning: module-mod-request called with no current model.|#
NIL

E> (module-mod-request 'goal nil)
#|Warning: module-mod-request called with no current meta-process.|#
NIL

```

buffer-spread

Syntax:

buffer-spread *buffer* -> [*spread-value* | **nil**]

Arguments and Values:

buffer ::= a symbol which should name a buffer

spread-value ::= a number indicating the buffer spreading parameter value for the buffer

Description:

For the [spreading activation calculation](#) in declarative memory each buffer has associated with it a value for the amount of activation it can spread. Each module may specify a particular name for that parameter for each buffer which it owns. This command allows one to get the value of that parameter for any buffer without knowing the specific name of the parameter which the module assigned for it.

If the buffer name given is the name of a buffer in the current model of the current meta-process, then the value of that buffer's activation spread parameter will be returned. If the buffer name is invalid or

there is no current model or current meta-process then a warning will be printed and **nil** will be returned.

This command is not likely to be used by the typical modeler or module developer, but it may be useful when investigating alternative equations for implementing the spreading activation and is used internally by the declarative module.

Examples:

```
1> (sgp :ga 1.5 :imaginal-activation 2.0)
(1.5 2.0)

2> (buffer-spread 'goal)
1.5

3> (buffer-spread 'imaginal)
2.0

4> (buffer-spread 'retrieval)
0

E> (buffer-spread 'bad)
#|Warning: buffer-spread called with an invalid buffer name bad |#
nil

E> (buffer-spread 'goal)
#|Warning: buffer-spread called with no current model. |#
nil

E> (buffer-spread 'goal)
#|Warning: buffer-spread called with no current meta-process. |#
NIL
```

buffers-module-name

Syntax:

buffers-module-name *buffer* -> [name | **nil**]

Arguments and Values:

buffer ::= a symbol which should name a buffer

name ::= a symbol which names the module that owns the buffer specified

Description:

The `buffers-module-name` command can be used to find the name of a module when you know the name of a buffer. If the buffer name given names a valid buffer in the current model of the current meta-process then the symbol which is the name of the module which owns that buffer is returned. If the buffer name is not valid or there is no current model then **nil** is returned.

This command will typically be used if one is building general tools for tracing events or otherwise producing code which will be monitoring the model's event stream and need to determine module names from buffer actions.

Examples:

```
> (buffers-module-name 'goal)
goal

> (buffers-module-name 'visual-location)
:vision

E> (buffers-module-name 'bad)
#|Warning: invalid buffer name bad |#
NIL

E> (buffers-module-name 'goal)
#|Warning: buffers-module-name called with no current model. |#
NIL

E> (buffers-module-name 'goal)
#|Warning: buffers-module-name called with no current meta-process. |#
NIL
```

request-completed-p

Syntax:

```
request-completed-p tag -> [ t | nil ]
```

Arguments and Values:

tag ::= a tag as returned by one of the request creation functions

Description:

The request-completed-p command is used to determine if a tag that corresponds to a request which was made to a module has been marked as completed. If there is a current model and current meta-process then request-completed-p will return **t** unless the request indicated by tag was sent to a buffer which tracks requests and that request has not yet been marked as complete. Thus any value specified as a tag which was not returned by a request creation function will also return **t** indicating that it is complete because all that is actually tracked are the currently incomplete requests – all others are assumed to be complete (including values which aren't actually returned tags). If the tag provided corresponds to a currently incomplete request then **nil** will be returned. If there is no current meta-process or current model then a warning will be printed and **nil** will be returned.

Examples:

These examples assume the default operations of the imaginal module which takes 200ms to complete a request and the goal module not tracking its requests.

```
1> (defvar *tag*)
*TAG*

2> (defvar *result*)
*RESULT*

3> (reset)
DEFAULT

4> (multiple-value-setq (*result* *tag*) (module-request 'imaginal (define-chunk-spec) t))
T

5> (request-completed-p *tag*)
NIL

6> (run .2)
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.200 IMAGINAL SET-BUFFER-CHUNK IMAGINAL CHUNK0
0.200 PROCEDURAL CONFLICT-RESOLUTION
0.200 ----- Stopped because time limit reached

0.2
8
NIL

7> (request-completed-p *tag*)
T

8> (multiple-value-setq (*result* *tag*) (module-request 'goal (define-chunk-spec) t))
T

9> (request-completed-p *tag*)
T

> (request-completed-p :not-a-real-tag)
T

E> (request-completed-p *tag*)
#|Warning: request-completed-p called with no current model.|#
NIL

E> (request-completed-p *tag*)
#|Warning: request-completed-p called with no current meta-process.|#
NIL
```

complete-request

Syntax:

complete-request chunk-spec -> [t | nil]

Arguments and Values:

chunk-spec ::= a chunk-spec which was sent to a module as a request or modification request

Description:

The `complete-request` command is to be used by a module which tracks its requests to indicate that a specific request has been completed. If there is a current model in the current meta-process then the request which was made using the `chunk-spec` passed to `complete-request` will be marked as completed and `t` will be returned. If there is no current meta-process or current model then a warning will be printed and `nil` will be returned. Note that `t` will be returned even if the specified `chunk-spec` had previously been completed or even if it is not a valid `chunk-spec`. Another thing to note is that the `chunk-spec` provided must be the same `chunk-spec` which was passed to the module in the request (it needs to be `eq`) for it to be properly completed.

Examples:

This example depends on this module definition which calls `complete-request` .1 seconds after a request:

```
(define-module-fct :test (list (define-buffer test :trackable t)) nil
  :version "" :documentation "" :query 'goal-style-query
  :request (lambda (mod buffer spec)
             (schedule-event-relative .1 'complete-request
                                     :params (list spec))))

1> (defvar *tag*)
*TAG*

2> (defvar *result*)
*RESULT*

3> (multiple-value-setq (*result* *tag*) (module-request 'test (define-chunk-spec) t))
T

4> (request-completed-p *tag*)
NIL

5> (run .05)
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.050  -----          Stopped because time limit reached
0.05
2
NIL

6> (request-completed-p *tag*)
NIL

7> (run .05)
  0.100  PROCEDURAL          CONFLICT-RESOLUTION
  0.100  -----          Stopped because time limit reached
0.05
2
NIL

8> (request-completed-p *tag*)
T

> (request-completed-p :not-a-spec)
T

E> (request-completed-p *tag*)
#|Warning: request-completed-p called with no current model.|#
```

```
NIL
```

```
E> (request-completed-p *tag*)  
#|warning: request-completed-p called with no current meta-process. |#  
NIL
```

complete-all-buffer-requests

Syntax:

```
complete-all-buffer-requests buffer-name -> [ t | nil ]
```

Arguments and Values:

buffer-name ::= a symbol which names a buffer

Description:

The `complete-all-buffer-requests` command is to be used by a module which tracks its requests to indicate that all requests to a specific buffer are now completed. If there is a current model in the current meta-process then all requests which were made to the indicated buffer will be marked as completed and `t` will be returned. If there is no current meta-process or current model then a warning will be printed and `nil` will be returned. Note that `t` will be returned even if the specified buffer does not currently have any uncompleted requests or even if it is not the name of a valid buffer in the current model of the current meta-process.

This command is intended to be used only by the module which owns the specified buffer to avoid potential problems with a module having its requests marked as completed outside of its control which could lead to errors if a module depends on tracking the completed status of its requests internally.

Examples:

This example depends on this module definition to provide a new trackable buffer which does not complete its requests immediately to allow testing this command. The example is a violation of the recommendation that only the module itself should complete its requests, but with this simple module that will not cause any problems:

```
(define-module-fct :test (list (define-buffer test :trackable t)) nil  
  :version "" :documentation "" :query 'goal-style-query  
  :request (lambda (mod buffer spec)  
             (schedule-event-relative .1 'complete-request  
                                     :params (list spec))))
```

```
1> (defvar *tag*)  
*TAG*
```

```
2> (defvar *result*)
```

```

*RESULT*

3> (multiple-value-setq (*result* *tag*) (module-request 'test (define-chunk-spec) t))
T

4> (request-completed-p *tag*)
NIL

5> (complete-all-buffer-requests 'test)
T

6> (request-completed-p *tag*)
T

> (complete-all-buffer-requests :not-a-buffer)
T

E> (complete-all-buffer-requests 'test)
#|Warning: complete-all-buffer-requests called with no current model.|#
NIL

E> (complete-all-buffer-requests 'test)
#|Warning: complete-all-buffer-requests called with no current meta-process.|#
NIL

```

complete-all-module-requests

Syntax:

```
complete-all-module-requests module-name -> [ t | nil ]
```

Arguments and Values:

module-name ::= a symbol which names a module

Description:

The `complete-all-module-requests` command is to be used by a module which tracks its requests to indicate that all requests to all of its buffers are now completed. If there is a current model in the current meta-process then all requests which were made to the indicated module will be marked as completed and `t` will be returned. If there is no current meta-process or current model then a warning will be printed and `nil` will be returned. Note that `t` will be returned even if the specified module does not currently have any uncompleted requests or even if it is not the name of a valid module in the current model of the current meta-process.

This command is intended to be used only by the module itself to avoid potential problems with a module having its requests marked as completed outside of its control which could lead to errors if a module depends on tracking the completed status of its requests internally.

Examples:

This example depends on this module definition to provide a new trackable buffer which does not complete its requests immediately to allow testing this command. The example is a violation of the recommendation that only the module itself should complete its requests, but with this simple module that will not cause any problems:

```
(define-module-fct :test (list (define-buffer test :trackable t)) nil
  :version "" :documentation "" :query 'goal-style-query
  :request (lambda (mod buffer spec)
             (schedule-event-relative .1 'complete-request
                                     :params (list spec))))
```

```
1> (defvar *tag*)
*TAG*
```

```
2> (defvar *result*)
*RESULT*
```

```
3> (multiple-value-setq (*result* *tag*) (module-request 'test (define-chunk-spec) t))
T
```

```
4> (request-completed-p *tag*)
NIL
```

```
5> (complete-all-module-requests :test)
T
```

```
6> (request-completed-p *tag*)
T
```

```
> (complete-all-module-requests :not-a-module)
T
```

```
E> (complete-all-module-requests :test)
#|Warning: complete-all-module-requests called with no current model.|#
NIL
```

```
E> (complete-all-module-requests :test)
#|Warning: complete-all-module-requests called with no current meta-process.|#
NIL
```

Extending Chunks

Sometimes it may be convenient to associate supporting information with chunks i.e. information which is not appropriate to encode within the chunk using the slots. This is often useful when creating a new module. As noted in the general description of chunks, each chunk has associated with it a set of parameters which can be used to hold such information. This section will describe how to add and use new chunk parameters.

The command to add a new parameter to chunks is called [extend-chunks](#) and will be described in detail below. Here the general concepts and support for chunk parameters will be described.

The first thing to note is that extending chunks is a system wide operation. Once the parameter is added all chunks in all models will have such a parameter available. Thus, extending chunks is a one-time operation. It is recommended that chunk parameters be added when there are no models defined and thus no preexisting chunks. If there are chunks defined when a new parameter is added, then an attempt to access that new parameter value in one of those existing chunks will have unpredictable consequences and could result in errors. The recommended way to add a new parameter is to place the extend-chunks call at the top-level in a file that is loaded automatically by ACT-R at initial load time. That will ensure that the functions for setting and accessing the parameter value are compiled with the rest of the code and that there will be no problems or warnings with attempts to redefine that parameter. There is no mechanism provided for removing a chunk parameter once it has been added and redefinition of an existing chunk parameter is not allowed.

Adding a new chunk parameter causes two functions to be created. The first one is the accessor for the parameter, and the other is an update function so that one can use `setf` with the accessor. Only the accessor is intended for direct use. That accessor function will have the name **chunk-param** where param is the name of the parameter which was added. Thus, if you were to add a parameter named `foo` to chunks the accessor for that parameter would be **chunk-foo**. The accessor takes one parameter which should be the name of a chunk in the current model and it will return the current value of that parameter for that chunk. To change the parameter you can use `setf` in conjunction with the accessor. Thus, assuming one has added a parameter called `foo` this shows how one could get and set the value of that parameter for a chunk named `free`:

```
1> (chunk-foo 'free)
NIL
2> (setf (chunk-foo 'free) 10)
10
```

```
3> (chunk-foo 'free)
10
```

The accessor function will test that there is a current model and that the chunk name is valid and it will print warnings and return **nil** if there is a problem:

```
E> (chunk-foo 'bad-name)
#|Warning: Chunk BAD-NAME does not exist in attempt to access CHUNK-FOO.|#
NIL
```

```
E> (chunk-foo 'free)
#|Warning: get-chunk called with no current model.|#
#|Warning: Chunk FREE does not exist in attempt to access CHUNK-FOO.|#
NIL
```

```
E> (chunk-foo 'free)
#|Warning: get-chunk called with no current meta-process.|#
#|Warning: Chunk FREE does not exist in attempt to access CHUNK-FOO.|#
NIL
```

After adding a parameter, it will be shown when the [pprint-chunks-plus](#) command is called and will display the current value just like all of the other chunk parameters which are currently defined:

```
> (pprint-chunks-plus free)
FREE
  ISA CHUNK

  --chunk parameters--
  FOO 10
  FAN-LIST (FREE)
  ACTIVATION 0
  ...
```

When creating a parameter for the chunks it is also possible to control how that parameter is treated under the following conditions:

- How it is initially set when a chunk is created
- How it is set in the copy when a chunk is copied
- How it is set when two chunks are merged

The details of how to specify those controls when creating a chunk parameter and examples of their use are provided with the details of the `extend-chunks` command. How they apply to the operation of the system will be described here.

There are two possible ways one can specify the initial value for the parameter. If neither is used, then the initial value for the parameter when a chunk is created will be **nil**. The first option is to specify a default value. The parameter will be set to that value for each new chunk created. The

other method for setting the initial value is to provide a function to set the value. If a default value function is provided, then that function will be called to get the initial value for the parameter. The function will be called with one parameter, which will be the name of the chunk in which the default value is being set. The return value of that function call will be the initial setting of that chunk parameter. The parameters for a chunk will be initialized in no particular order, and thus the initialization function should not rely on any other parameters of the chunk having been set. Also, the initialization function will not be called until the first time that the chunk's parameter value is requested. Thus, it may not be called for every chunk and may not be called until well after the initial creation time of the chunk.

When a chunk is copied using the `copy-chunks` command the settings of the parameters for the new copy are determined as follows. By default, the parameter will be set in the same manner as the initial value for a newly created chunk i.e. either `nil`, the specified default value, or the result of calling the default value function. However, it is possible to instead specify a function to use to set the value for the copy which is different from the function used to set the initial value (if such a function was also provided). The copy function will be passed one parameter and its return value will be the setting for the parameter in the copied chunk. There are two options available for specifying a function to set a parameter value in a copied chunk and the difference between them is what value is passed to that function. One option is to set the “copy” function which will be passed the current value of the parameter from the chunk which is being copied. The other option is to set the “copy from chunk” function which will be passed the name of the chunk which is being copied. Only one of the copying mechanisms can be used, and if both are specified then the “copy” function will be the one that gets used. If one wants to have the copied chunk's parameter set the same as the original chunk's parameter specifying the copy function as the Lisp function `identity` is recommended.

When two chunks are merged using the [merge-chunks command](#) (which will happen automatically when the declarative module collects chunks cleared from the buffers) the values of the parameters for the merged chunk after the merging are set by default to the values that existed for the primary chunk (the first one in the call to `merge-chunks`) prior to the merge. That can be overridden by providing a merge function for the parameter. If a merge function is provided then when two chunks are merged that function will be called with the names of the chunks being merged as its only two parameters. They will be passed to the parameter's merge function in the same order as they were given to `merge-chunks`, and the return value of that function will be the setting of that parameter in the merged chunk.

Commands

extend-chunks

Syntax:

```
extend-chunks parameter-name {:default-value value} {:default-function def-fn}  
              {:copy-function copy-fn} {:copy-from-chunk-function copy-from-fn}  
              {:merge-function [merge-fn | :second]} -> [accessor | :duplicate-parameter | nil]
```

Arguments and Values:

parameter-name ::= a symbol which will be the name of the new parameter for the chunk

value ::= any Lisp value

def-fn ::= a function designator for a function which takes one parameter

copy-fn ::= a function designator for a function which takes one parameter

copy-from-fn ::= a function designator for a function which takes one parameter

merge-fn ::= a function designator for a function which takes two parameters

accessor ::= a symbol which names the accessor function for the new parameter

Description:

The `extend-chunks` command is used to add a new parameter to chunks. If chunks do not already have a parameter by that name, then an accessor called `chunk-parameter-name` (where *parameter-name* is the *parameter-name* symbol provided) will be created which takes one parameter which should always be a chunk name. That accessor can be used to get the corresponding parameter of a chunk or used with `setf` to change the value of that parameter for the chunk. This command is a macro which does not evaluate its parameters and there is not a corresponding function.

The initial value for the new parameter when a chunk is created will be **nil** unless either the `default-value` or `default-function` keyword parameter is provided. If a `default-value` is specified then that will be set as the value of the parameter for a newly created chunk. If a `default-function` is specified, then that function will be called with the name of the chunk as its only parameter and the return value of that function call will be used as the initial value. If both a `default-value` and `default-function` are given, then the function will be used.

When a copy of a chunk is being made the parameter value for the new copy of the chunk will be set to the default value as determined above unless either a `copy-function` or a `copy-from-chunk-function` is provided. If one of those functions is provided, then the new chunk will have its value set to the value returned from calling that function. The difference between the copying functions is that the `copy-function` is called with the current parameter value of the chunk which is being copied and the `copy-from-chunk-function` gets called with the name of the chunk which is being copied. If both functions are specified then the `copy-from-chunk-function` will be used.

If a merge-function is provided as a function, then when two chunks are being merged that function will be called with the names of those chunks in the order which they were merged and the value it returns will be the setting for the parameter in the merged chunk. If no merge-function is given, then the merged chunk will take the value of the parameter from the first of the two chunk names specified in the merging. If the merge-function is specified with the keyword `:second`, then the value of the parameter in the merged chunk will be the value from the second of the chunks being merged.

If a new parameter is created, then the name of that accessor is returned.

If a parameter by that name already exists, then a warning will be printed and the keyword **`:duplicate-parameter`** will be returned.

If parameter-name is not valid a warning is printed and **`nil`** is returned.

If there is already a function with the same name as the accessor or the update function created then a warning will be printed to indicate that and those functions will be redefined for the chunk parameter. In some cases, that warning may appear if a file containing an `extend-chunks` call is compiled and then loaded. In those circumstances the warning can typically be safely ignored, and if one wants to eliminate the warning then the `extend-chunks` command can be called between calls to the functions `suppress-extension-warnings` and `unsuppress-extension-warnings` to prevent the output of the warnings:

```
(suppress-extension-warnings)
(extend-chunks ...)
(unsuppress-extension-warnings)
```

The use of the warning suppression function is only recommended for use when an `extend-chunks` call occurs at the top level of a file and one is certain the parameter name's redefinition of the function(s) is not a problem.

Examples:

```
1> (defun count-copies (n) (1+ n))
COUNT-COPIES
```

```
2> (defun merge-use-second (a b) b)
MERGE-USE-SECOND
```

```
3> (defun start-with-name (x) (string x))
START-WITH-NAME
```

```
4> (extend-chunks simple)
CHUNK-SIMPLE
```

```
5> (extend-chunks counter :default-value 0 :copy-function count-copies)
CHUNK-COUNTER
```

```

6> (extend-chunks merge-demo :default-function start-with-name
                                :merge-function merge-use-second)
CHUNK-MERGE-DEMO

7> (define-chunks (a))
(A)

8> (pprint-chunks-plus a)
A

  --chunk parameters--
  MERGE-DEMO "A"
  COUNTER 0
  SIMPLE NIL
  ...

(A)

9> (copy-chunk a)
A-0

10> (pprint-chunks-plus a-0)
A-0

  --chunk parameters--
  MERGE-DEMO "A-0"
  COUNTER 1
  SIMPLE NIL
  ...

(A-0)

11> (merge-chunks a a-0)
A

12> (pprint-chunks-plus a)
A

  --chunk parameters--
  MERGE-DEMO A-0
  COUNTER 0
  SIMPLE NIL
  ...

(A)

13E> (extend-chunks simple)
#|Warning: Parameter SIMPLE already defined for chunks.|#
:DUPLICATE-PARAMETER

1> (defun chunk-dummy ())
CHUNK-DUMMY

2E> (extend-chunks dummy)
#|Warning: Function CHUNK-DUMMY already exists and is being redefined.|#
CHUNK-DUMMY

E> (extend-chunks "not a symbol")
#|Warning: "not a symbol" is not a valid symbol for specifying a chunk parameter.|#
NIL

```

Defining New Modules

The primary means of extending the system, both in terms of extending the cognitive capabilities of models and extending the software system itself, is through adding new modules. This section will describe the mechanism for defining a new module and detail how it interacts with the system.

To create a new module one uses the [define-module command](#). A module can only be defined when there are currently no models in the system and only the default meta-process exists. The typical way to do that is to place the file with the module definition into one of the directories from which all of the .lisp files are loaded automatically at startup. That way the new module will be compiled and loaded with the rest of the ACT-R files. However it is not necessary to do it that way, and one can define modules by loading files or calling the define-module command at any time when there are no models or additional meta-processes defined.

When a module is defined it must be given a name. That name must be unique among the currently defined modules i.e. only one module with a given name may exist in the system. Once a module is defined it cannot be overwritten with a new definition for a module with the same name. However, one can use the [undefine-module command](#) to explicitly remove a module from the system if it is necessary to replace it. Undefining a module can also only happen when there are no models and additional meta-processes defined.

Once a module is defined every model which gets defined will have its own instance of that module. What constitutes an “instance” is determined by the module’s creation function (described in detail below). The recommendation is that a new instance should be created for each model and that there should be no sharing of state between module in separate models, particularly for the cognitive modules. However, there is nothing which prevents such operation and in some circumstances that may be useful, for example, a module might make one connection to an external simulation and then allow all the models to access that same connection through their own instances of the module.

There are several components which may be specified when defining a module. It is not necessary for a module to specify all of them, and other than a name no others are required. All of these components will be described in the following sections.

Documentation

When the module is defined one should provide a string for the version number of the module and a string containing the description of the module. These strings are displayed when the system is initially loaded and when the [mp-print-versions command](#) is called. There are no requirements on what the string must contain, but the recommendation is to provide some sort of number for the version string and a one sentence description of the module for the documentation. If documentation strings are not provided then a warning will be displayed indicating that they should be, but that will not prevent the module from being created.

Buffers

When the model is defined it may have any number of buffers associated with it. This is the only way to add new buffers to the system – they do not exist independently of a module. The buffers for a module will provide the interface for other modules to interact with the new module. The buffers provide the means of making requests and queries to the module, and it may respond to requests or other actions by placing chunks into its buffers. Note that every module has access to all of the buffers of the system and can place chunks into any of them, but the recommendation is that a module should only manipulate its own buffers. Of the default modules, the only one which does not follow that recommendation is the procedural module where the productions are allowed to manipulate, clear, or place chunks in any buffer.

There are several options which can be given when creating a new buffer. The only required component is a name. Each buffer must be given a name and that name must be unique in the system – there is can only be one buffer with any given name. The other components are described in the following sections.

spreading activation weight

The name of the parameter used to set that buffer's weight for the spreading activation calculation and the default value for that parameter may be specified. If they are not specified then the parameter will be named *buffer-activation* where *buffer* is the name of the buffer and the default value for the parameter will be 0.

request parameters

Any request parameters which are to be used with the buffer must be specified when the module is defined. Only those request parameters which are specified when the module is created will be considered valid when making a request to that buffer through the provided module request functions.

However, there is no constraint placed on the generation of a chunk-spec since it does not make reference to any buffer at its creation time. Therefore, it is possible for code which doesn't use normal request mechanisms to pass a chunk-spec with invalid request parameters to the handler functions of a module. It is not recommended that one use the request handler functions of a module directly in that fashion, but if those functions are to be used that way they should perform additional tests in verifying the validity of any request parameters in the chunk-spec provided.

queries

Any queries which the module will accept through the buffer, other than the default query of state which all modules must accept, need to be specified when the module is defined. Only those queries which are specified at definition time will be accepted as valid when using the [query-buffer command](#). However, as described with the specification of a buffer's request parameters, if one calls the module's query function directly there is no pre-testing of the values and it should not assume that all queries are valid.

query printing

By default, `buffer-status` will print out the results of querying the buffer's buffer and state queries (those to which every buffer must respond). If the module has other queries that one would like to show when `buffer-status` is called that can be done by providing a query printing function to do so. That function should use the [command-output command](#) to print any additional status information desired for the buffer, and that output will be displayed after the normal output by `buffer-status`.

multi-buffer

A buffer may indicate whether or not it is to be treated as a "multi-buffer". A multi-buffer is a construct which allows one to specify a set of chunks which can be used with the buffer that will not be copied when placed into the buffer – they will be placed directly into the buffer. Details on [multi-buffers are described in their own section](#) and will not be covered further here.

request tracking

A buffer may indicate whether or not it will track the requests which are made to it. If a buffer indicates that it tracks requests then the module is responsible for indicating that requests are completed at the appropriate times.

Parameters

Each module may specify a set of parameters which it will use. Those parameters can be either new parameters which the module will “own” and make available to the system, or existing parameters owned by other modules which it would like to be informed about when they change. Whenever a parameter is changed through the use of the `sgp` command the module which owns that parameter is given the new value specified in the `sgp` call. The module is responsible for recording that new value, or whatever value it wants to use based on that value, and then return the current value for the parameter. If any other modules are watching that parameter as non-owners, then they are notified of the new value which the owning module has reported.

The parameters are specified using the [define-parameter command](#), and these are the components of a parameter which may be specified by the module.

name

Each defined parameter must specify the name of the parameter. The name must be a Lisp keyword. If it is the name of a parameter that has already been defined by another module then it is only valid if this definition explicitly claims that it is not the “owner” of that parameter. If it specifies that it is the owner of the parameter, then that name must not already be taken by a parameter of another module.

owner

When defining the parameter it must be specified whether it should be created as a new parameter or one which is to be watched and assumed to already exist. If it is being specified as a new parameter, that is, the specification claims that it is being defined by its owner, then the remaining properties may also be specified for the parameter. If the specification claims to not be the owner of the parameter then only the name of the parameter is relevant.

documentation

A string describing the parameter can be specified. That string will be printed by the `sgp` command when displaying the details of the parameter.

default-value

The value the parameter will have by default. If no default value is given then the default value will be **nil**.

valid-test

A function, or the name of a function, may be given to use when testing the parameter values set by the `sgp` command. If a `valid-test` is specified, then that function is passed one value, which is the requested setting in an `sgp` call. If the function returns a non-**nil** value then the setting is passed on to the module to handle. If the `valid-test` function returns **nil** then a warning is displayed and that parameter setting is not passed to the module.

warning

If a parameter value fails the `valid-test` the warning which is printed is constructed based on the warning string provided when the parameter is defined. The warning will always be of this form:

```
#|Warning: Parameter param cannot take value val because it must be warn. |#
```

where *param* is the name of the parameter, *val* is the invalid value given and *warn* is the warning string specified when defining the parameter.

Interface functions

A module interacts with the rest of the system by providing functions to be called at the appropriate times. There are several situations for which a module can provide a function. None of these functions are required unless the module also provides one or more buffers or one or more parameters for the system. If the module has a buffer then it must provide a function to handle queries of the buffer, and if it has any parameters then it must provide a function for setting and getting the parameter values.

Here are the possible situations for which a module may provide a function and the parameters which it will be passed and how the return value is used.

creation

Whenever a new model is defined it will create an instance of each module in the system. To create that instance each module's creation function is called. That function will be passed one parameter which will be the name of the model being defined. The value returned by that function will be saved as the instance of that module in that model. The module's instance should be a data structure capable of holding any parameter values and state information necessary for the module to operate. That instance will be passed to all of the other interface functions which the module provides. If a

module does not provide a creation function then its instance will be the value **nil** for the purpose of calling any of its other interface functions.

reset

The module may specify one, two or three functions to call when a model is reset. Those functions will be called with the instance of the module as their only parameter. The three reset functions are called at different times during the resetting of a model and a module may use any or all of them if needed. The primary reset function is called before the parameters get reset to their default values. The secondary reset gets called after the default parameters have been set and before any user code is evaluated. The tertiary reset gets called after the user code of the model definition has been evaluated. Most modules are likely only going to need one of the reset functions, and typically only the primary reset is necessary. However, if a module needs to modify the value of another module's parameter at reset time it will have to do so in a secondary or tertiary reset function, otherwise any change will be overwritten. Although using the secondary or tertiary reset function is necessary to adjust another module's parameter from the default value, it may not be sufficient because other modules may also have secondary or tertiary reset functions which will also adjust or restore that value. In such cases, watching the parameter, by adding it to the module as a parameter which it does not own, will allow for more control and monitoring of the changes. The return value of a module's reset function is not used.

delete

The module may specify a function to be called when the model in which it has been created is deleted. The delete function will be passed the instance of the module in the model being deleted as its only parameter. The instance of the module will not be referenced by the given system code after it has been deleted and thus may be destructively modified, deallocated, or any other explicit cleanup as needed may be performed upon it. To ensure that deleted module instances are not referenced, one should not access any parameters using `sgp` in a module's delete method. The return value of the module's delete function is not used.

parameters

If the module makes parameters available to the system or is monitoring the parameters of other modules then it must provide a function for setting and getting those parameter values. The module's parameter function will be called with two parameters. The first will be the instance of the module in the current model. The second will be either a keyword which names a parameter, in which case this

is a request for the current value of the parameter which the function should return, or it will be a cons of a keyword parameter name and a value. When a cons is provided as the second parameter it is a request to change the current value of the parameter to the one provided. The module should set the parameter as needed based on that value and then return the current value of the parameter (which may or may not be the same as the value which was provided). A module's parameter function will only be called with parameters which the module has specified when it was defined. For those which it owns it will get both requests for the current value and requests to change the value. If a module does not own the parameter it will only be sent notifications of changes to the parameter and the return value from the function in those cases is ignored.

queries

If the module has one or more buffers, then it must provide a function to handle queries of those buffers. The module's query function will be called whenever the [query-buffer command](#) is used. The query function will be called with four parameters. The first will be the instance of the module in the current model. The second will be the name of the buffer being queried. The third will be one of the valid queries for the module – either state, which all modules must accept, or one of the ones given when the buffer was specified. The fourth parameter will be the value of the query to test. There are no constraints on the values which may be used. Thus, a module should be able to handle any value which is provided, but it does not have to consider them all valid and may display warnings when invalid values are given. Every module is expected to respond to queries of state with values of free, busy, or error, but other than those there are no prespecified queries to which a module must respond. The return value of the query function is considered as a generalized boolean. If the query of the given value is true then the module should return a true value and if the query of the given value is not true the function should return **nil**.

requests

If the module has one or more buffers then it may provide a function to handle requests to those buffers, but it is not required that a module with buffers accept requests. If a request function is specified for the module then it will be called whenever the [module-request command](#) is called for one of the module's buffers. The module's request function will be called with three parameters. The first will be the instance of the module in the current model. The second will be the name of the buffer to which the request was made, and the third will be a chunk-spec which represents the request being made. The chunk-spec provided may include any number of slot tests and request parameters, but only those request parameters which are specified with the buffer's definition will be included.

Other than that there are no tests performed on the chunk-spec before it is passed to the module. Therefore the module's request function is responsible for testing and verification of whether the request being made is something which the module is able to handle and should signal warnings to indicate invalid or malformed requests. The return value of the request function is ignored.

What a module does in response to a request is entirely up to the module writer – there are no mandatory actions which must occur. It is recommended however that the module should schedule events to handle all of the actions which it performs in response to the request. By doing so, the trace will show those actions, other modules will be able to detect that they have occurred (for instance the procedural module may schedule another conflict-resolution event if there is not one pending), the model debugging tools will allow one to see that the events have been scheduled, and the stepping tools can be used for testing and debugging of the module's actions.

buffer modification requests

If the module has one or more buffers then it may provide a function to handle buffer modification requests, but a module is not required to handle such requests. If a function for buffer modification requests is provided for the module then it will be called whenever the [module-mod-request command](#) is called for one of the module's buffers. The module's buffer modification request function will be called with three parameters. The first will be the instance of the module in the current model. The second will be the name of a buffer to which the request was made, and the third will be a chunk-spec of modification information (a chunk-spec that only uses the modifier = in the slot-specs, specifies each slot only once, and may contain request parameters valid for the specified buffer). A buffer modification request will only occur when there is a chunk in the buffer, but the slots specified may or may not exist in that chunk. The return value of the modification request function is ignored.

What a module does in response to a buffer modification request is entirely up to the module writer – there are no mandatory actions which must occur. It is recommended however that the module should schedule events to handle all of the actions which it performs in response to the request. The general purpose of the modification request is to allow the module to perform an action like mod-chunk which has a time cost during which time the module will be busy and unable to handle other requests. However, it does not have to be used that way and may be used to perform other module actions.

notify upon clearing

A module may specify a function to be called when any buffer is cleared. If a module provides a function to be notified when buffers clear that function will be called with three parameters after every buffer clearing occurs. The first parameter will be the instance of the module in the current model. The second parameter will be the name of a buffer, and the third parameter will be the name of the chunk which was cleared from the buffer. The module may use that information however it wants, but the one thing that it should not do is modify the chunk which was cleared from the buffer because other modules may also need the information which is in that chunk. The return value of the notify upon clearing function will be ignored.

notify at the start of a new call to run the system

A module may specify a function to be called whenever one of the [system running commands](#) is called. If a module provides a function to be notified when a run starts then that function will be called with one parameter before the first event of the run is executed. That parameter will be the instance of the module in the current model. This notification function is not recommended for the normal operations of a module because it should not depend on how or when the system is run. The intended use of this notification is to allow a module an opportunity to perform some safety or verification tests before the system runs. The return value of this function is ignored.

Some examples of its current use are the procedural module and the module which handles the ACT-R environment connections. The procedural module uses it to make sure that there is always a conflict-resolution or other appropriate procedural event scheduled or waiting. This avoids a problem that could occur because of an unusual run termination (a Lisp break in the middle of a conflict-resolution action for example) which would leave the procedural module without any events on the queue and thus unable to fire any more productions until it is reset. The environment connection uses it to initialize the stepper tool if it is open when the model starts running.

An important note is that this function will be called regardless of how the running command was called. Some of the running commands may actually result in multiple start/stop pairs because they can involve multiple calls to running commands internally. Thus, start/stop pairs do not map 1:1 with calls in user code, but with the actual starting and stopping of the internal event processing.

notify upon a completion of a call to run the system

A module may also specify a function to be called whenever one of the system running commands is finished running the system. If a module provides a function to be notified when a run ends then that function will be called with one parameter after the last event of the run is executed. That parameter

will be the instance of the module in the current model. Like the run start notification, this is not recommended for the normal operations of a module because it should not depend on how or when the system is run. The intended use of this notification is to allow a module an opportunity to perform some safety or verification tests after the system runs.

One example of this being used is the module which handles the ACT-R Environment connection. It uses this to disable the stepper controls when there is no longer a model running.

As noted with the run start function description, this function will be called regardless of how the running command was called. Some of the running commands may actually result in multiple start/stop pairs because they can involve multiple calls to running commands internally. Thus, start/stop pairs do not map 1:1 with calls in user code, but with the actual starting and stopping of the internal event processing.

warning of an upcoming request

A module may specify a function which will be called to provide the module with a warning of an upcoming request. If a warning function is provided that function will be called with three parameters. The first parameter will be the instance of the module. The second will be the name of one of the module's buffers and the third will be the chunk-spec of the request. The procedural module will call a module's warning function, if it has one, at production selection time to warn about any requests which will be made when the selected production fires. This is the only time that the warning function will be called – requests from other modules or from explicit requests made by other code will not be preceded by warnings. Having a warning function allows a module to know that a production will be making a request which may be dependent on information which was true at the time the production was selected. The module gets called with the warning at the selection time and thus may suspend pending actions or note the relevant information necessary at that time to use when handling the request. The return value of this function is ignored.

The only module which currently uses a warning function is the vision module. It allows the vision module to suspend the processing of a screen update during a production which will be making a move-attention request so that the visual-location chunk which is used in the production does not get invalidated by a change in the display before the request is handled.

search and offset

Two additional functions can be specified when the module has one or more searchable buffers. How those functions are used is described in the [searchable buffers](#) section.

Common Class of Modules – Goal Style

There is a particular type of module which modelers often find useful to create for various situations. That type of module is one which works like the [goal](#) and [imaginal](#) modules where a request is a direct specification of a chunk to create and be placed into the buffer. Those are referred to as “goal style” modules. There are three functions which exist to make creating such modules easy. Those functions can be used as the query, request, and modification request functions for creating such a module, and they are the functions [goal-style-query](#), [goal-style-request](#), and [goal-style-mod-request](#) respectively.

Using those functions, one can create a new module which acts just like the default goal module and buffer using something like this:

```
(define-module name (name) nil :version "1.0" :documentation "Example"  
  :query goal-style-query  
  :request goal-style-request  
  :buffer-mod goal-style-mod-request)
```

Where *name* is the name you want to give to the module and buffer, which must be the same for the assumptions in the goal-style-* functions.

Those commands do not have to only be used in that manner, and it is possible to call them from other functions. For example, one could specify a different request function when creating the module which performs some error checking or possible restrictions on the requests which can be made to the module before then calling goal-style-request to do the rest of the work.

The file which defines the goal-style components is included in the support directory of the distribution. Therefore it is not loaded automatically when the system loads. So, any file which uses those commands should include this line near the top of the file:

```
(require-compiled "GOAL-STYLE-MODULE" "ACT-R-support:goal-style-module")
```

That will ensure that the goal style support code is loaded appropriately. Typically, the main goal module will already have loaded that support, but it is safest to be certain that it has been loaded,

especially when creating modules or other code which may be shared with others who may have made other changes to the modules or default system.

Writing Module Code

When writing code to handle the module's operations there are some assumptions which can be made and practices which should be followed. This section will cover those assumptions and recommended practices.

Because the module functions which one provides will always be passed the current model's instance of the module one can assume that there is a current model in handling those calls. However, if one has additional functions which go along with the module that may be called at other times, or without requiring an instance of the module as a parameter, then the use of the [get-module command](#) is recommended to get the current model's instance of the module to work with. The `get-module` command takes one parameter which should be the name of a module and it returns the instance of that module in the current model. If there is no current model then `get-module` will print a warning and return **nil**.

When a module performs actions in a running model, particularly actions relating to the buffers, it should always schedule those as events instead of directly calling commands which make those changes. When scheduling such events, it should always indicate the module's name when the event is generated. Doing those things makes it much easier to debug a model which uses the module and also makes it easier to integrate new modules from different sources because any unexpected interactions should then be detectable in the model's trace.

If a module needs to reference the values of parameters of other modules then it is recommended that they be specified as non-owned parameters for the module. Those parameters' values should then be recorded in some way in the instance of the module with the module's parameter function even if the module has no parameters of its own. That is recommended for performance reasons because using `sgp` to get parameter values is a relatively slow process and often also requires suppressing the output which also may have a significant cost to performance relative to the work done by the module.

Any chunk-types and chunks which a module is going to define should be defined in the module's primary reset function if possible. If the module depends on the chunks or chunk-types of other modules it should not reference those items during the primary reset function because there is no guarantee that the other module's primary reset function will be called prior to the current module's.

If the module will be [extending chunks](#) or productions with new parameters for its use that should be done at the top level in the module's definition file because extending those items should only occur once. Such calls do not belong in either the creation or reset function for the module.

Module examples

The examples provided below will generally only show the basic syntax of the commands, and will not have “working” module definitions as examples because the necessary support code is outside of the scope of the examples for the reference manual. Also, one often needs a model which uses the module to really have an example that shows the components in operation. In the examples directory of the source code distribution one can find a directory called creating-modules. In that directory are new module definition files which implement demonstration modules along with corresponding models which use the features provided by those modules. The documentation for the modules and how to run the associated models are included in the comments of those files.

Commands

get-module

Syntax:

```
get-module name -> [instance | nil] [ t | nil]
get-module-fct name -> [instance | nil] [ t | nil]
```

Arguments and Values:

name ::= a symbol which should be the name of a module
instance ::= the named module's instance in the current model

Description:

The `get-module` command is used to get the instance of a module in the current model of the current meta-process. It returns two values. If there is a module with that name in the system then the first return value is the instance of that module and the second return value will be `t`. If that name does not name a module in the current model or there is no current model then a warning will be printed and both return values will be `nil`. The purpose of the second return value to indicate that the module instance exists is because a module may use `nil` as its instance.

Examples:

```
> (get-module goal)
#S(GOAL-MODULE ...)
T
```

```

> (get-module-fct 'imaginal)
#S(IMAGINAL-MODULE ...)
T

E> (get-module foo)
#|Warning: FOO is not the name of a module in the current model. |#
NIL
NIL

E> (get-module-fct 'goal)
#|Warning: get-module called with no current model. |#
NIL
NIL

E> (get-module 'goal)
#|Warning: get-module called with no current meta-process. |#
NIL
NIL

```

define-buffer

Syntax:

```

define-buffer name {:param-name param} {:param-default default} {:request-params request-params}
  {:queries queries} {:status-fn status} {:search search} {:multi multi} {:copy copy}
  {:trackable track} -> buffer-list
define-buffer-fct name {:param-name param} {:param-default default} {:request-params request-params}
  {:queries queries} {:status-fn status} {:search search} {:multi multi} {:copy copy}
  {:trackable track} -> buffer-list

```

Arguments and Values:

name ::= a symbol which is the name of the buffer
param ::= a keyword that names the source activation parameter for the buffer
default ::= the default value for the source activation of this buffer
request-params ::= (request-param*)
request-param ::= a keyword indicating a valid request parameter for this buffer
queries ::= (query*)
query ::= a symbol indicating a valid query for this buffer
status ::= a function or name of a function to be called when buffer-status reports on this buffer
multi ::= a generalized boolean indicating whether this is a regular multi-buffer
search ::= a generalized boolean indicating whether this is a searchable multi-buffer
copy ::= a generalized boolean indicating whether this is a multi-buffer which should copy chunks
track ::= a generalized boolean indicating whether this buffer tracks its requests
buffer-list ::= (name (p-name p-default) request-params queries status multi-type track)
p-name ::= *param* if it is provided, otherwise the keyword *:name-activation*
p-default ::= *default* if it is provided, otherwise **nil**
multi-buffer ::= [**:multi** | **:search** | **:multi-copy** | **:search-copy** | **nil**] depending on the values of *multi*
 search and *copy*

Description:

The `define-buffer` command can be used to create a buffer definition list for use in a module definition. The value returned from this function has no real use in the system other than as a member of the list of buffers when defining a module. It is recommended that `define-buffer` be used to create a buffer definition list whenever the buffer uses any of the needed components to make the definition more understandable. The only parameter required when calling `define-buffer` is the name of the parameter which must be a symbol. If a value is not provided for one of the keyword parameters then a **nil** value will be placed into the definition list for that component, with two exceptions – if the `param-name` is not provided but a `param-default` is then the standard default activation name will be used and both the search and multi values must be **nil** (or not provided) for the multi-buffer specifier to be **nil**.

`Define-buffer` does not test any of the values for validity and just creates an appropriately formatted list for use in a module definition.

Examples:

```
> (define-buffer test :param-name :test-val)
TEST (:TEST-VAL NIL) NIL NIL NIL NIL NIL)

> (define-buffer test :multi t :trackable t :param-default 3)
(TEST (:TEST-ACTIVATION 3) NIL NIL NIL :MULTI T)

> (define-buffer-fct 'test :queries (list 'status 'size)
      :status-fn (lambda (x) (command-output " status unknown")))
(TEST NIL NIL (STATUS SIZE) #<Interpreted Function (unnamed) @ #x23ecc8c2> NIL NIL)
```

define-parameter

Syntax:

```
define-parameter param-name {:owner owner} {:default-value default} {:documentation docs}
  {:valid-test test} {:warning warn} -> [parameter | nil]
```

Arguments and Values:

`param-name` ::= a keyword for the name of the parameter being defined

`owner` ::= a generalized boolean indicating whether this definition is specifying the parameter's owner

`default` ::= a value which will be the default value for the parameter

`docs` ::= a string describing the parameter

`test` ::= a function, function name or nil used to test value for this parameter through `sgp`

`warn` ::= a string to print when an invalid parameter value is detected

`parameter` ::= an instance of an ACT-R internal parameter item

Description:

The `define-parameter` command is used to create a parameter for a module. The value returned from this function has no use in the system other than as a member of the list of parameters when defining a module. A parameter defined through this command will not be available to the system until it is part of a module which owns it.

The only parameter required when calling `define-parameter` is the name of the parameter which must be a Lisp keyword.

The `owner` parameter specifies whether the module which uses this parameter definition is to be considered the owner of that parameter. If not provided the default value is `t`.

The `default` parameter specifies the value the parameter should have as its default when the system is reset. If not provided, then the default value for the parameter will be `nil`. The only restriction on the default value is that it cannot be a Lisp keyword because it is not possible to use `sgp` to set a single parameter to a keyword value.

The `docs` parameter should be a string which describes the parameter. That string is printed by `sgp` when the parameter value is displayed.

The `valid` test may be specified as a function or the name of a function. That function is called whenever this parameter's value is changed using the `sgp` command. The test function will be passed the new value specified in the `sgp` call. If the test function returns a true value then the new value is considered valid and it is passed on to the owning module's parameter function to set. If the test function returns `nil` then a warning is printed and the parameter value is left unchanged. If no test is specified in the definition for the parameter then no testing is performed on the values for.

If a test function is provided, then when it reports an invalid value the warning which gets printed will look like this:

```
#|Warning: Parameter param-name cannot take value val because it must be warn.|#
```

where *param-name* is the name of the parameter, *val* is the invalid value given and *warn* is the warn string specified when defining the parameter. If no warn string is provided, then it defaults to an empty string.

If any of the parameters passed to `define-parameter` are invalid then a warning will be displayed and `nil` will be returned.

Examples:

```
> (define-parameter :new :documentation "A new parameter")
#S(ACT-R-PARAMETER ...)

> (define-parameter :count :valid-test 'fixnum :warning "a fixnum")
#S(ACT-R-PARAMETER ...)

> (define-parameter :bll :owner nil)
#S(ACT-R-PARAMETER ...)

E> (define-parameter 'not-a-keyword)
#|Warning: Parameter name must be a keyword.|#
NIL

E> (define-parameter :new :documentation 'not-a-string)
#|Warning: documentation must be a string.|#
NIL

E> (define-parameter :new :valid-test 'not-a-function)
#|Warning: valid-test must be a function, the name of a function, or nil.|#
NIL

E> (define-parameter :new :default-value :keywords-not-allowed)
#|Warning: default-value cannot be a keyword.|#
NIL

E> (define-parameter :new :warning 'not-a-string)
#|Warning: warning must be a string.|#
NIL
```

define-module

Syntax:

```
define-module module-name (buffer-def*) (param*) {:version version} {:documentation doc-string}
  {:creation create} {:reset reset} {:request request} {:query query} {:buffer-mod mod}
  {:params params} {:delete delete} {:notify-on-clear clear} {:warning warn}
  {:run-start run-start} {:run-end run-end} {:search search} {:offset offset}
  -> [module-name | nil]
```

```
define-module-fct module-name (buffer-def*) (param*) {:version version} {:documentation doc-string}
  {:creation create} {:reset reset} {:request request} {:query query} {:buffer-mod mod}
  {:params params} {:delete delete} {:notify-on-clear clear} {:warning warn}
  {:run-start run-start} {:run-end run-end} {:search search} {:offset offset}
  -> [module-name | nil]
```

Arguments and Values:

module-name ::= a symbol which will be the name for the module being defined

buffer-def ::= [*buffer-name* |
 ([*buffer-name* | *buffer-name* *buff-params* | *buffer-name* *buff-params* (*req-param**) |
 buffer-name *buff-params* (*req-param**) (*query-names**) |
 buffer-name *buff-params* (*req-param**) (*query-names**) [*print-status* | **nil**] |
 buffer-name *buff-params* (*req-param**) (*query-names**) [*print-status* | **nil**] *multi-buffer* |
 buffer-name *buff-params* (*req-param**) (*query-names**) [*print-status* | **nil**] *multi-buffer* *track*)]]

buffer-name ::= a symbol which names the buffer being defined

buff-params ::= [*pname* | (*pname* *pdefault*) | (**nil** *pdefault*) | (*pname* **nil**) | **nil**]

pname ::= a keyword which names the parameter to create for the buffer's spreading activation value
 pdefault ::= a number which will be the default for the buffer's spreading activation value parameter
 req-param ::= a keyword which names a request parameter this buffer will accept
 query-names ::= a symbol which names a new query which is valid for this buffer
 print-status ::= a function or name of a function to be called when buffer-status reports on this buffer
 multi-buffer ::= [**:multi** | **:search** | **:multi-copy** | **:search-copy** | **nil**]
 track ::= a generalized boolean indicating whether this buffer tracks its requests
 param ::= a parameter item as returned by the define-parameter command
 version ::= a string providing version information for the module
 doc-string ::= a string to provide a brief description of the module
 create ::= a function or the name of a function to handle the module creation
 reset ::= [primary-reset | ([primary-reset | **nil**] [secondary-reset| **nil**] {[tertiary-reset| **nil**]})]

primary-reset ::= a function or the name of a function to handle the module's primary reset
 secondary-reset ::= a function or the name of a function to handle the module's secondary reset
 tertiary-reset ::= a function or the name of a function to handle the module's tertiary reset
 request ::= a function or the name of a function to handle module requests
 query ::= a function or the name of a function to handle module queries
 mod ::= a function or the name of a function to handle module modification requests
 params ::= a function or the name of a function to handle the module's parameter
 delete ::= a function or the name of a function to handle the deletion of the module
 clear ::= a function or the name of a function to call whenever a buffer is cleared
 warn ::= a function or the name of a function to be called to warn of a future request to the module
 run-start ::= a function or the name of a function to be called each time a running command is called
 run-end ::= a function or the name of a function to be called each time a running command finishes
 search ::= a function or the name of a function to be called when searching a multi-buffer
 offset ::= a function or the name of a function to be called after a multi-buffer search for preferences

Description:

The define-module command is used to add a new module to the system. A new module may only be added if there are currently no models defined and only the default meta-process is defined. The first parameter to define-module must be a symbol that names the new module being defined. If there is not already a module defined with that name and all of the remaining parameters provided to describe the module are valid (as described below) then a new module is added to the system and the name of the module is returned. Otherwise, a warning is printed and **nil** is returned.

The second parameter must be a list of the buffer descriptions for the buffers of the module. It may be an empty list, in which case the module has no buffers. If it is not an empty list, then the elements of that list must be either symbols or lists.

If an element of the buffer definition list is a symbol, then that symbol names a buffer which the module will have and that buffer will be given the default values for its components. If a list is provided then one can specify the other components of the buffer as well. The [define-buffer](#)

[command](#) is recommended for creating a buffer definition list, but is not required. The list may be up to seven elements long and the components of the list are as follows:

- The first element must be a symbol which names the buffer.
- The second element specifies the name of the spreading activation parameter to add to the system for this buffer and/or the default value for the buffer's spreading activation parameter.
 - o If it is a keyword which is not already the name of a parameter in the system then that name will be used for the parameter and it will have a default value of 0.
 - o If it is a list, then the first element of the list is the name of the parameter and the second element is the default value, which should be a number. If the name is specified as **nil** then the default name for the parameter is used.
- The third element is the list of request parameters which the buffer will accept. It must be a list of keywords or **nil**. Only those items specified in this list will be considered as valid request parameters by the [module-request command](#).
- The fourth element is the list of queries which may be made to the module in addition to the standard query for state. It must be a list of symbols or **nil**. Only those items specified in this list will be considered as valid queries by the [query buffer command](#).
- The fifth element should be a function or the name of a function. This function will be called after the [buffer-status command](#) has printed the default query information for the buffer. This is typically added when one provides additional queries so that their status may be displayed for the user along with the normal queries using buffer-status.
- The sixth element, if provided, indicates that this should be considered as a [multi-buffer](#) and specifies which multi-buffer mechanism to use.
- The seventh element, if provided, indicates whether or not this buffer will track its requests

If a module has one or more buffers then it must provide a query function. It is not required to provide a request or modification request function.

The third parameter to define-module must be a list of parameter specifications returned by the define-parameter command. If the list is non-empty then each parameter must have an appropriate ownership. If a parameter in the list is specified as being owned by the module then it must not be owned by any other module, and if a parameter in the list is indicated as not being owned then that parameter must be owned by some other module. A module which specifies parameters must also provide a params function to support them.

The remaining parameters to define-module are keyword parameters and may occur in any order. None of the keyword parameters are required except when the module has buffers and/or parameters as described above.

The version and documentation parameters should be strings and are used to display information about the module when the [mp-print-versions command](#) is called. They are not required, but if either is not provided a warning will be displayed before defining the module.

The rest of the keyword parameters are used to specify the functions to call for interfacing the module to the rest of the system. How they interface to the system is described in the [interface functions section](#). Here the details of the parameters they will be passed and expected return values will be described.

The create function will be passed one parameter. That will be the symbol naming the model in which a new instance of the module is being created. The return value of the module's create function will be the instance used for calling all of the module's other functions from that model.

The module's reset function (or functions) are called with one parameter. That parameter will be the instance of the module in the current model. The return values of the reset functions are ignored by the system.

The request function will be passed three parameters. The first will be the instance of the module in the current model. The second will be a symbol naming one of the module's buffers indicating to which buffer the request was made. The third parameter will be the chunk-spec which describes the request. The return value of the request function is ignored by the system.

The query function will be passed four parameters. The first will be the instance of the module in the current model. The second will be a symbol naming one of the module's buffers indicating which buffer is being queried. The third will be the symbol naming the query being made and the fourth will be the value being queried. The return value of the query function will be considered as a generalized boolean and should indicate whether the value specified for the query was true or false.

The mod function will be passed three parameters. The first will be the instance of the module in the current model. The second will be a symbol naming one of the module's buffers indicating to which buffer the modification request was made. The third will be a chunk-spec of modification information (a chunk-spec that only uses the modifier = in the slot-specs, specifies each slot only

once, and may contain request parameters valid for the specified buffer). The return value of the mod function is ignored by the system.

The params function will be passed two parameters. The first will be the instance of the module in the current model. The second will be either a keyword naming one of the module's owned parameters or a cons of a keyword naming a parameter specified for the module and a new value for that parameter. If the second parameter is a keyword then that indicates that the current value for that parameter is being asked for and the return value of the function should be that parameter's current value in the current model. If the second parameter is a cons then that means sgp has been called to change that parameter's value. If this module owns the parameter being changed then the return value of the params function will be considered the current value of that parameter. If the module does not own the parameter being changed then the return value of the params function is ignored.

The delete function will be passed one parameter. That parameter will be the instance of the module in the model being deleted. The return value of the delete function is ignored by the system.

The clear function will be passed three parameters. The first will be the instance of the module in the current model. The second will be a symbol naming the buffer which is being cleared. The third will be a symbol naming the chunk which is being cleared from that buffer. The return value of the clear function is ignored by the system.

The warn function will be passed three parameters. The first will be the instance of the module in the current model. The second will be a symbol naming one of the module's buffers indicating to which buffer the request will be made. The third will be a chunk-spec indicating a request that will be made. The return value of the warn function is ignored by the system.

The run-start function will be passed one parameter. That parameter will be the instance of the module in the current model. The return value of the run-start function is ignored by the system.

The run-end function will be passed one parameter. That parameter will be the instance of the module in the current model. The return value of the run-end function is ignored by the system.

The search function will be passed two parameters. The first will be the instance of the module in the current model. The second will be the name of a multi-buffer of the module. The return value of the search function should be a list of chunks in that buffer's buffer set in the order they are to be searched or **nil** indicating no order preference.

The offset function will be passed three parameters. The first will be the instance of the module in the current model. The second will be the name of a multi-buffer of the module. The third will be a list of chunk names from the buffer set of that multi-buffer. The return value of the offset function should be a list of numbers representing the offset value for the matching of the chunks in the list provided or **nil** if no offsets are to be used.

Examples:

Examples of modules along with demo models which use them can be found in the examples/creating-modules directory of the distribution. The examples here are only to show the syntax of the command and some of the warnings and errors which may occur.

```
> (define-module foo nil nil)
#|Warning: Modules should always provide a version and documentation string.|#
FOO

1> (define-module bar nil nil :version "" :documentation "")
BAR

2E> (define-module bar nil nil :version "" :documentation "")
#|Warning: Module BAR already exists and cannot be redefined. Delete it with undefine-
module first if you want to redefine it.|#
NIL

> (define-module-fct 'foo '(foo-1 (foo-2 (:foo-spread 1.5)))
      (list (define-parameter :foo-value)
            :version "0.3"
            :documentation "example"
            :params 'foo-param-function
            :query 'foo-query-function)
FOO

E> (define-module-fct nil nil nil)
#|Warning: Modules should always provide a version and documentation string.|#
#|Warning: Nil is not a valid module-name. No module defined.|#
NIL

E> (define-module-fct 'bad '(buffer-1) nil)
#|Warning: Modules should always provide a version and documentation string.|#
#|Warning: A module with a buffer must support queries.
Module BAD not defined.|#
NIL

E> (define-module-fct 'bad nil (list 'bad-parameter))
#|Warning: Modules should always provide a version and documentation string.|#
#|Warning: Invalid params-list (BAD-PARAMETER).
Module BAD not defined.|#
NIL

E> (define-module-fct 'bad '(buffer-1 (buffer-2 bad-param-name)) nil :query 'query-fn)
#|Warning: Modules should always provide a version and documentation string.|#
#|Warning: Invalid buffer specification: (BUFFER-2 BAD-PARAM-NAME)|#
#|Warning: Error in module buffer definitions.
Module BAD not defined.|#
NIL
```

```
E> (define-module 'bad nil nil)
#|Warning: Modules should always provide a version and documentation string.|#
#|Warning: Cannot create a new module when there are models defined.|#
NIL
```

```
E> (define-module 'bad nil nil)
#|Warning: Modules should always provide a version and documentation string.|#
#|Warning: Cannot create a new module when there is a meta-process other than the default
defined.|#
NIL
```

undefine-module

Syntax:

```
undefine-module module-name -> [t | nil]
undefine-module-fct module-name -> [t | nil]
```

Arguments and Values:

module-name ::= a symbol which should name a module

Description:

The `undefine-module` command is used to remove a module from the system. It takes one required parameter which is the name of the module to remove. If that names a module in the system and there are no models defined and there are no meta-processes defined other than the default then that module, all of its buffers, and all of its parameters are removed from the system and the value **t** will be returned.

If the name does not name a module in the system or there is a model defined or a meta-process other than default defined then a warning is printed and a value of **nil** is returned.

There are typically two reasons for using this command. The first is when first writing and debugging a module. Often one finds that they need to change something with it and this command is the only way to remove a broken or incomplete module without quitting the system and reloading everything. The other is when one has built a module which is a replacement for an existing module. In that case, the file with a new version of the module undefines the original version of the module before defining the replacement.

Examples:

```
1> (define-module bar nil nil :version "" :documentation "")
BAR
```

```
2> (define-module foo nil nil :version "" :documentation "")
FOO
```

```

3> (undefine-module bar)
T

4> (undefine-module-fct 'foo)
T

5E> (undefine-module-fct 'foo)
#|Warning: FOO is not the name of a currently defined module. |#
NIL

E> (undefine-module goal)
#|Warning: Cannot delete a module when there are models defined. |#
NIL

E> (undefine-module goal)
#|Warning: Cannot delete a module when there is a meta-process other than the default
defined. |#
NIL

```

goal-style-query

Syntax:

goal-style-query *instance buffer slot value* -> [t | nil]

Arguments and Values:

instance ::= this parameter is ignored
buffer ::= a symbol which names the buffer being queried
slot ::= this parameter is ignored
value ::= a symbol for the value of the query

Description:

The goal-style-query command is designed to be used as the query function for a module's definition. It assumes that only the default queries for state will be made. It responds to those queries as follows: a state busy query will always return **nil**, a state free query will always return **t**, and a state error query will always return **nil**. For any other values it will print a warning indicating that the query was invalid and return **nil**.

Examples:

```

> (goal-style-query nil 'goal 'state 'free)
T

> (goal-style-query nil 'goal 'state 'busy)
NIL

> (goal-style-query nil 'goal 'state 'error)
NIL

E> (goal-style-query nil 'goal 'state 'bad)
#|Warning: Unknown state query BAD to GOAL buffer |#
NIL

```

```
E> (goal-style-query nil 'goal 'bad-query t)
#|Warning: Unknown query BAD-QUERY T to the GOAL buffer|#
NIL
```

goal-style-request

Syntax:

goal-style-request *instance buffer chunk-spec {delay {priority}}* -> [event | nil]

Arguments and Values:

instance ::= this parameter is ignored

buffer ::= a symbol which names the buffer to which the request was made

chunk-spec ::= a chunk-spec of the request

delay ::= a number indicating how many seconds to wait before creating the chunk

priority ::= [:max | :min | priority-val]

priority-val ::= a number indicating the priority to use for the event

event ::= the scheduled event which will create the new chunk in the buffer

Description:

The `goal-style-request` command is designed to be used as the request function for a module's definition, but it may also be called by a other functions to handle the creation of a chunk in the buffer and provides optional delay and priority parameters for such use. If the chunk-spec provided is valid for the [chunk-spec-to-chunk-def command](#), then this command schedules an event to create a chunk using that chunk definition and place that chunk into the specified buffer after the provided delay time has passed with the indicated priority. If no delay time is provided, then it default to 0 seconds. If the priority is not specified then it defaults to -100. The event scheduled is returned. If the chunk-spec is invalid then a warning is printed and **nil** is returned.

The scheduled event will result in multiple other events being generated. The first event will have an action of [create-new-buffer-chunk](#). That event's action will create a chunk using the result of the chunk-spec-to-chunk-def call. It will also schedule a set-buffer-chunk event using the [schedule-set-buffer-chunk command](#) with that chunk, a delay of 0, and a priority of -1000 and schedule an event with an action named clean-up-goal-chunk with a delay of 0 and a priority of -1001 which will not be shown in the trace. The clean-up-goal-chunk action will [delete](#) the chunk that was generated by the create-new-buffer-chunk command and [release its name](#) since that chunk is no longer needed because its copy is now in the buffer.

Examples:

```
1> (goal-style-request nil 'goal (define-chunk-spec isa chunk))
#S(ACT-R-EVENT ...)
```

```

2> (goal-style-request nil 'goal (define-chunk-spec isa chunk) .2)
#S(ACT-R-EVENT ...)

3> (run .25)
0.000 GOAL CREATE-NEW-BUFFER-CHUNK GOAL
0.000 GOAL SET-BUFFER-CHUNK GOAL CHUNK0
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.200 GOAL CREATE-NEW-BUFFER-CHUNK GOAL
0.200 GOAL SET-BUFFER-CHUNK GOAL CHUNK1
0.200 PROCEDURAL CONFLICT-RESOLUTION
0.200 ----- Stopped because no events left to process

0.2
9
NIL

E> (goal-style-request nil 'goal 'bad-spec)
#|warning: chunk-spec-to-chunk-def called with something other than a chunk-spec.|#
#|warning: Invalid request made of the GOAL buffer.|#
NIL

```

create-new-buffer-chunk

Syntax:

create-new-buffer-chunk *buffer chunk-description* {:**priority** *priority*} -> nil

Arguments and Values:

buffer ::= a symbol which names a buffer

chunk-description ::= a valid chunk definition

priority ::= [:**max** | :**min** | *priority-val*]

priority-val ::= a number indicating the priority to use for setting the chunk in the buffer

Description:

This command defines a new chunk using the *chunk-description* provided (which should be a list that is valid for the chunk list of [define-chunks-fct](#)), schedules an event to copy that chunk into the buffer specified at the current time with the priority provided (or -1000 if not provided), and schedules an event to [delete](#) the chunk that was generated and [release its name](#) since that chunk is no longer needed because its copy is now in the buffer. The event which places the chunk into the buffer is a normal event, and the event which deletes the chunk is a maintenance event. This is the function which does the work for a [goal-style-request](#), but it may be useful in other situations as well. It does not do any validity testing on the parameters provided thus care should be taken when using it to avoid errors.

Examples:

```

1> CG-USER(134): (buffer-read 'goal)
NIL

```

```

2> (create-new-buffer-chunk 'goal '(color blue state free))
NIL

3> (mp-show-queue)
Events in the queue:
    0.000    GOAL                SET-BUFFER-CHUNK GOAL CHUNK0
    0.000    NONE                Clean-up unneeded chunk
    0.000    PROCEDURAL          CONFLICT-RESOLUTION
3

4> (run 1)
    0.000    GOAL                SET-BUFFER-CHUNK GOAL CHUNK0
    0.000    PROCEDURAL          CONFLICT-RESOLUTION
    0.000    -----            Stopped because no events left to process
0.0
3
NIL

5> (buffer-chunk goal)
GOAL: CHUNK0-0
CHUNK0-0
  STATE  FREE
  COLOR  BLUE

(CHUNK0-0)

```

goal-style-mod-request

Syntax:

goal-style-mod-request *instance buffer mods {delay {priority}}* -> [event | nil]

Arguments and Values:

instance ::= this parameter is ignored

buffer ::= a symbol which names the buffer to which the request was made

mods ::= a modification chunk-spec

delay ::= a number indicating how many seconds to wait before modifying the chunk

priority ::= [**:max** | **:min** | priority-val]

priority-val ::= a number indicating the priority to use for the event

event ::= the scheduled event which will modify the chunk in the buffer

Description:

The `goal-style-mod-request` command is designed to be used as the buffer modification request function for a module's definition, but it may also be called by other functions to handle the modification of a chunk in the buffer and provides optional delay and priority parameters for such use. This command just calls [schedule-mod-buffer-chunk](#) with the buffer, chunk-spec, delay, and priority provided. If no delay time is provided, then it default to 0 seconds. If no priority is provided then the default value is 20. The value returned from that call to `schedule-mod-buffer-chunk` is returned.

Examples:

```

1> (set-buffer-chunk 'goal (car (define-chunks (value t))))
CHUNK0-0

2> (goal-style-mod-request nil 'goal (define-chunk-spec value nil))
#S(ACT-R-EVENT ...)

3> (goal-style-mod-request nil 'goal (define-chunk-spec value 5) .5)
#S(ACT-R-EVENT ...)

4> (run .5)
  0.000  GOAL                MOD-BUFFER-CHUNK GOAL
  0.000  PROCEDURAL          CONFLICT-RESOLUTION
  0.500  GOAL                MOD-BUFFER-CHUNK GOAL
  0.500  PROCEDURAL          CONFLICT-RESOLUTION
  0.500  -----            Stopped because time limit reached

0.5
5
NIL

```

Multi-buffers

A multi-buffer provides some new possibilities for how a module can use its buffer as well as possibly changing how the [procedural module](#) will interact with that buffer. These capabilities are still considered an experimental modification to the system and thus the reason for covering them in a separate section instead of in the other sections. If you use multi-buffers in your own modules please let me know of any problems you encounter or suggestions for improvements which you have.

Multi-buffers were introduced as a general mechanism to allow for the implementation of modules which do things which were not easy to do with the system previously. One such module was the threaded cognition module by Salvucci and Taatgen. Prior to the introduction of the multi-buffers, the implementation of the threaded cognition module required making changes to many internal functions of the system. That made that module difficult to maintain since it had to be updated often to stay in line with updates and changes to the main system. Those changes also had the potential to interfere with modules written by other people which could lead to problems integrating different new mechanisms. The current design of multi-buffers was sufficient to allow threaded cognition to be implemented as a module without needing to change any of the main system code, and the expectation is that the multi-buffer concept may be useful in other areas as well.

The difference between a multi-buffer and a “normal” buffer is that a module with a multi-buffer may specify a set of chunks which can be placed into that buffer without being copied first. Despite the name, a multi-buffer is still restricted to only holding one chunk at any time. It is that additional set of chunks which accompany the buffer to which the multi prefix refers, but when used as a [searchable buffer](#) that chunk may be changed among those in the set automatically by the conflict resolution process thus being treated as if it holds any one of them for production matching purposes. When a chunk which is not in the “buffer set” of a multi-buffer is placed into the multi-buffer, it will work like a normal buffer and copy that chunk first.

Some of the buffer related commands described above work slightly differently when the buffer provided is a multi-buffer. [Set-buffer-chunk](#) and [overwrite-buffer-chunk](#) do not copy the chunk being placed into a multi-buffer if the chunk is a member of the buffer set for that buffer. When [clear-buffer](#) is passed a multi-buffer then the chunk being cleared from the buffer is also removed from the buffer set of that buffer. Finally, the [buffer-chunk command](#) will show the buffer set of a multi-buffer when that buffer’s information is displayed. The buffer set will be shown in curly braces after the name of the chunk currently in the buffer if the buffer set is non-empty.

Not every chunk may be added to the buffer set of a multi-buffer. Essentially, only those chunks which are created by the module which created the multi-buffer are valid members of the multi-set. Chunks which are, or could potentially be, referenced by other modules are marked as being invalid as members of a buffer set. In particular, chunks which have been cleared from a buffer and chunks which have been explicitly added into declarative memory cannot be put into a buffer set. Additionally, one may mark a chunk as explicitly unavailable for the buffer set of any multi-buffer by setting the `buffer-set-invalid` parameter of the chunk to `t`:

```
(setf (chunk-buffer-set-invalid chunk) t)
```

That may be useful when creating a module which defines chunks that are made available for other uses, but which should not be directly placed into buffers. If a chunk is already in a buffer set when it is marked as invalid then it will remain in that buffer set, but will be copied into the buffer from that point forward.

When working with a multi-buffer a module should be careful to not allow chunks which it wants to remain in the buffer set to be cleared from the buffer. The module should probably use `overwrite-buffer-chunk` instead of `set-buffer-chunk` internally to avoid clearing the current chunk first. Similarly, the module may want to turn off [strict-harvesting](#) for the multi-buffer to avoid the automatic clearing that the procedural module may perform. Note however that a request to the multi-buffer from a production will still clear the buffer. Thus, care may be required in how a model's productions interact with the multi-buffer if the buffer set is to be maintained.

To create a multi-buffer the module needs to provide a sixth item in the buffer definition list for the buffer in the module definition. A value of `:multi` will make the buffer a multi-buffer. There are other keyword values which create a searchable buffer described in the section below. If an invalid sixth item is given then a warning will be printed and the module will not be created.

Commands

There are four additional commands that are used to work with the buffer set of a multi-buffer and one other general buffer command which is useful for multi-buffers. For all of the examples shown in the new commands this module with two multi-buffers, `foo` and `bar`, is assumed to exist:

```
(define-module :foo ((foo nil nil nil nil :multi)(bar nil nil nil nil :multi)) nil
  :query goal-style-query :version "" :documentation "")
```

As is this model which creates some initial chunks to use in the examples:

```
(define-model foo (sgp :v t)
```

```
(define-chunks (a) (b) (c))
(add-dm (d)))
```

store-m-buffer-chunk

Syntax:

```
store-m-buffer-chunk buffer chunk -> [ chunk | nil ]
```

Arguments and Values:

buffer ::= a symbol which is the name of a multi-buffer

chunk ::= a symbol which is the name of a chunk to add to the buffer set of *buffer*

Description:

The `store-m-buffer-chunk` command is used to add a chunk to the buffer set of a multi-buffer. If the *buffer* name given is the name of a multi-buffer and *chunk* names a chunk defined in the current model of the current meta-process which is not marked as invalid for a buffer set then that chunk will be added to the buffer set of that multi-buffer in the current model and the chunk name will be returned. If the chunk is already a member of the buffer set then no change to the buffer set is made and the chunk name is still returned.

If there is no current model, no current meta-process, *buffer* does not name a multi-buffer, *chunk* is not a valid chunk name, or *chunk* names a chunk which is marked as invalid for a buffer set then there will be a warning printed and the value **nil** will be returned.

Examples:

These examples assume the module and model [shown above](#) have been defined.

```
1> (reset)
DEFAULT

2> (buffer-chunk foo bar)
FOO: NIL
BAR: NIL
(NIL NIL)

3> (store-m-buffer-chunk 'foo 'a)
A

4> (store-m-buffer-chunk 'foo 'b)
B

5> (store-m-buffer-chunk 'bar 'b)
B

6> (buffer-chunk foo bar)
```

```
F00: NIL {A B}
BAR: NIL {B}
(NIL NIL)
```

```
7> (set-buffer-chunk 'foo 'a)
A
```

```
8> (buffer-chunk foo)
F00: A {A B}
A
```

```
(A)
```

```
9> (set-buffer-chunk 'foo 'b)
B
```

```
10> (buffer-chunk foo)
F00: B {B}
B
```

```
(B)
```

```
11> (setf (chunk-buffer-set-invalid 'b) t)
T
```

```
12> (overwrite-buffer-chunk 'foo 'b)
B-0
```

```
13> (buffer-chunk foo)
F00: B-0 [B] {B}
B-0
```

```
(B-0)
```

```
E> (store-m-buffer-chunk 'goal 'a)
#|Warning: store-m-buffer-chunk cannot store a chunk in buffer GOAL because it is not a
multi-buffer |#
NIL
```

```
E>: (store-m-buffer-chunk 'foo 'd)
#|Warning: store-m-buffer-chunk cannot store D in a buffer set because it has been marked
as invalid for a buffer set most likely because it has been previously cleared from a
buffer |#
NIL
```

get-m-buffer-chunks

Syntax:

```
get-m-buffer-chunks buffer -> (chunk*)
```

Arguments and Values:

buffer ::= a symbol which is the name of a multi-buffer

chunk ::= a symbol which is the name of a chunk in the buffer set of *buffer*

Description:

The `get-m-buffer-chunks` command is used to get the buffer set of a multi-buffer. If the buffer name given is the name of a multi-buffer then a list of the chunks which are in the buffer set of that multi-buffer in the current model of the current meta-process will be returned. There is no specification for the ordering of the chunks in the list which is returned. Thus, one should not assume anything about the buffer set based on that ordering.

If there is no current model, no current meta-process, or buffer does not name a multi-buffer a warning is printed and **nil** will be returned.

Examples:

These examples assume the module and model [shown above](#) have been defined.

```
1> (reset)
DEFAULT
2> (store-m-buffer-chunk 'foo 'a)
A
3> (store-m-buffer-chunk 'foo 'c)
C
4> (store-m-buffer-chunk 'bar 'a)
A
5> (store-m-buffer-chunk 'bar 'b)
B
6> (get-m-buffer-chunks 'foo)
(A C)
7> (get-m-buffer-chunks 'bar)
(A B)
E> (get-m-buffer-chunks 'goal)
#|warning: get-m-buffer-chunks cannot return a buffer set for buffer GOAL because it is
not a multi-buffer |#
NIL
```

remove-m-buffer-chunk

Syntax:

```
remove-m-buffer-chunk buffer chunk -> [ chunk | nil ]
```

Arguments and Values:

`buffer ::=` a symbol which is the name of a multi-buffer

`chunk ::=` a symbol which is the name of a chunk to remove from the buffer set of `buffer`

Description:

The `remove-m-buffer-chunk` command is used to remove a chunk from the buffer set of a multi-buffer. If the buffer name given is the name of a multi-buffer and the chunk names a chunk defined in the current model of the current meta-process which is currently in the buffer set of that multi-buffer then that chunk will be removed from that buffer set and the chunk name will be returned. If the chunk is not a member of the buffer set then no change to the buffer set is made and the chunk name is still returned.

If there is no current model, no current meta-process, buffer does not name a multi-buffer, or chunk is not a valid chunk name then there will be a warning printed and **nil** will be returned.

Examples:

These examples assume the module and model [shown above](#) have been defined.

```
1> (reset)
DEFAULT

2> (store-m-buffer-chunk 'foo 'a)
A

3> (store-m-buffer-chunk 'foo 'c)
C

4> (store-m-buffer-chunk 'bar 'a)
A

5> (store-m-buffer-chunk 'bar 'b)
B

6> (get-m-buffer-chunks 'foo)
(A C)

7> (get-m-buffer-chunks 'bar)
(A B)

8> (remove-m-buffer-chunk 'foo 'a)
A

9> (remove-m-buffer-chunk 'bar 'b)
B

10> (get-m-buffer-chunks 'foo)
(C)

11> (get-m-buffer-chunks 'bar)
(A)

12> (remove-m-buffer-chunk 'bar 'd)
D

13> (get-m-buffer-chunks 'bar)
(A)

E> (remove-m-buffer-chunk 'goal 'a)
#|Warning: remove-m-buffer-chunk cannot remove a chunk from buffer GOAL because it is not
a multi-buffer |#
NIL
```

```
E> (remove-m-buffer-chunk 'bar :not-a-chunk)
#|Warning: remove-m-buffer-chunk cannot remove :NOT-A-CHUNK from the buffer set because it
does not name a chunk |#
NIL
```

remove-all-m-buffer-chunks

Syntax:

```
remove-all-m-buffer-chunks buffer -> [ t | nil ]
```

Arguments and Values:

buffer ::= a symbol which is the name of a multi-buffer

Description:

The `remove-all-m-buffer-chunks` command is used to remove all of the chunks from the buffer set of a multi-buffer. If the buffer name given is the name of a multi-buffer then the buffer set of that multi-buffer in the current model of the current meta-process will be cleared and the value `t` will be returned.

If there is no current model, no current meta-process, or *buffer* does not name a multi-buffer then there will be a warning printed and the value `nil` will be returned.

Examples:

These examples assume the module and model [shown above](#) have been defined.

```
1> (reset)
DEFAULT

2> (store-m-buffer-chunk 'foo 'a)
A

3> (store-m-buffer-chunk 'foo 'b)
B

4> (get-m-buffer-chunks 'foo)
(A B)

5> (remove-all-m-buffer-chunks 'foo)
T

6> (get-m-buffer-chunks 'foo)
NIL

E> (remove-all-m-buffer-chunks 'goal)
#|Warning: remove-all-m-buffer-chunks cannot remove a chunk from buffer GOAL because it is
not a multi-buffer |#
NIL
```

erase-buffer

Syntax:

erase-buffer *buffer* -> [chunk | **nil**]

Arguments and Values:

buffer ::= a symbol which is the name of a buffer

chunk ::= a symbol which is the name of a chunk

Description:

The `erase-buffer` command is used to remove a chunk from a buffer without notifying other modules. This can be used with a normal buffer or a multi-buffer. It is similar to [clear-buffer](#) except that it skips the notification step. That will prevent the chunk from being collected by declarative memory and thus being marked as invalid for a buffer set. Thus, `erase-buffer` is to `clear-buffer` as [overwrite-buffer-chunk](#) is to [set-buffer-chunk](#). If the buffer name given is the name of a buffer and there is a chunk in that buffer that chunk will be removed from the buffer and the chunk name will be returned. If the buffer is already empty then **nil** will be returned.

If there is no current model, no current meta-process, or *buffer* does not name a buffer then there will be a warning printed and the value **nil** will be returned.

Examples:

These examples assume the module and model [shown above](#) have been defined.

```
1> (reset)
DEFAULT

2> (store-m-buffer-chunk 'foo 'a)
A

3> (store-m-buffer-chunk 'foo 'c)
C

4> (set-buffer-chunk 'foo 'a)
A

5> (set-buffer-chunk 'goal 'a)
A-0

6> (buffer-chunk foo goal)
FOO: A {A C}
A

GOAL: A-0 [A]
A-0
  ISA CHUNK

(A A-0)
```

```
7> (erase-buffer 'foo)
A
```

```
8> (buffer-chunk foo)
FOO: NIL {A C}
(NIL)
```

```
9> (erase-buffer 'goal)
A-0
```

```
10> (set-buffer-chunk 'foo 'a)
A
```

```
11> (buffer-chunk foo)
FOO: A {A C}
A
```

```
(A)
```

```
12> (clear-buffer 'foo)
A
```

```
13> (buffer-chunk foo)
FOO: NIL {C}
(NIL)
```

```
E> (erase-buffer :not-a-buffer)
#|Warning: erase-buffer called with an invalid buffer name :NOT-A-BUFFER|#
NIL
```

Searchable buffers

A searchable buffer is a multi-buffer which the procedural module will treat differently than other buffers. Except for the different treatment from the procedural module, a searchable buffer continues to operate like a [multi-buffer](#) as described in the previous. When the procedural module matches a production's condition against a searchable buffer it does not restrict the matching to only the chunk which is in the buffer. Instead, it will search the buffer set of that buffer to find a chunk which matches. How that search takes place and how the module may influence the search will be detailed in this section.

An important consideration in the development of the searchable buffers was to maintain plausibility. That means avoiding the potential for multiple instantiation issues as well as not allowing the capability to solve NP-hard problems within a single production. The concept that seems to best fit with those constraints is to limit the search such that it is performed in parallel for all searchable buffers within a production and to terminate the search at the first chunk found as a match.

The testing of the conditions for a production thus follows these steps:

- Normal buffer's conditions are tested and variable bindings occur
 - o If it is a dynamic production this includes bindings from the variableized slots
- Other conditions which use only those variables (or none) are tested (!eval!, !bind!, queries)
- Each search buffer has its buffer set searched to find a chunk which matches all of the conditions that are currently defined i.e. involve constants or already bound variables
- Bind all of the variables necessary from the search buffer chunks found
- Test all remaining condition

There are a couple of additional details to add to that. One important issue is that if there is more than one search buffer tested within a production this is not guaranteed to find a set of chunks which satisfy all of the conditions in that production because tests among the search buffer chunks is performed after the search has completed – it will not go back and search again to find other chunks if those conditions fail to match.

Another thing to note relates to dynamic productions. Because the variableized slot bindings occur before the search, a variable bound in a search buffer condition cannot be used as a variableized slot. However, a search buffer could test variableized slots that were bound in the normal buffer conditions.

This is the result of generalizing the constraint of dynamic testing which only allows one level of indirection to include the search as a level of indirection.

Once the conflict set has been determined the production with the highest utility in the set will be selected and fired. If that production involved one or more search buffers, then the chunk(s) found during the search will be placed into the appropriate buffer(s) at the time it is selected.

During the condition matching processes with a search buffer any references to the buffer name variable of the search buffer will reflect a binding to the name of a chunk in the buffer set. However, once a production is selected that variable will be bound to the actual chunk in the buffer. That only differs if it is a search-copy buffer (as described below) because a new chunk will be created as a copy for the buffer instead of placing the buffer set chunk there directly. The reason for that is because actions of the production may need to refer to the actual chunk in the buffer. That changing of the variable after the matching may seem like a problem, but it makes the copy and non-copy buffers operate the same for matching purposes if the buffer variable is used in the conditions (probably not something which would typically be done). The alternative would be to actually create a copy of each chunk in the buffer set for the matching of a search-copy buffer, but since that name would be new it wouldn't be very useful to match it in the conditions of a production.

There are two ways for the module which owns the searchable buffer to affect the search and selection process. First, the module may specify the order in which to search the buffer set and possibly exclude some members. The module does that by specifying a function with the `:search` keyword in the module definition. The search function will be called once during each conflict-resolution event. It will be passed the module instance and the symbol naming a searchable buffer of the module. It should return a list of chunks from the buffer set in the order that they are to be searched. Any chunks in the list returned which are not in the buffer set will be ignored in the search. If no search function is provided for the module then the whole buffer set will be searched in an arbitrary order.

The other thing the module can do is specify a preference for each of the chunks which have matched to offset the utilities of the productions which matched them. To do that the module must specify an offset function with the `:offset` keyword in the module definition. The offset function will be called once during each conflict-resolution event in which the conflict set is non-empty. The function will be passed three parameters: the module instance, a symbol naming a search buffer of the module, and a list of the chunks from the buffer set of that buffer which were found as matches in some production. The return value from that function should be a list of numbers of the same length as the

list of chunks it was passed where each number will be the offset for the corresponding chunk's matching in a production. Those offset values will be added to the utility of a production which matched that chunk before determining which production to fire. The recommendation for offsets is to return 0 for a preferred chunk and negative values for non-preferred chunks.

To create a searchable buffer for a module one must specify a sixth parameter in the buffer definition similar to how a multi-buffer is specified. For search buffers there are actually two options available. The first is to specify the keyword `:search`. That will create a multi-buffer which has the search capability described above. The other option is to specify the value `:search-copy`. That will create a special variation of a multi-buffer. A buffer created with `:search-copy` will actually have the chunks from the buffer set copied into the buffer like a normal buffer does. Otherwise, the search-copy buffer operates just like a standard searchable buffer. The advantage of using a search-copy buffer is that the chunks in the buffer set are then protected from being unintentionally cleared from the buffer (and thus being removed from the buffer set) through strict harvesting or module requests and they will not be modified since only a copy is made available through the module. This "copied" multi-buffer operation is also available for a normal multi-buffer by specifying a value of `:multi-copy` when creating it, but such a buffer would act the same as a normal buffer and that option is not recommended.

Multiple Models

In the model and meta-process sections it indicates that it is possible to run more than one model at a time, but it does not cover the details on how to do so. This section will describe how to create and use multiple models and show the commands that are available for working with them.

The first thing to discuss is the notion of the current model and the current meta-process. Those terms were introduced in the model and meta-process sections above and most of the commands described in this manual indicate that they only work with respect to those current items. When there is only one meta-process and one model defined then those will always be the current ones. However if more than one of either is defined which one is the current one will typically need to be specified by the user. If there are multiple meta-processes then the user must always specify the current one – there will not be a current meta-process otherwise. If there is more than one model defined within a meta-process (each meta-process has its own set of models) then which one is the current model will be set either by the user or by the system. When a model is initially defined and whenever the model is reset the code provided in the model’s definition will be executed with that model set as the current one by the system. If the meta-process is not running, then, as with meta-processes, there will not be a current model unless specified by the user. When the meta-process is running the current model will be set based on the events being executed. When each event is executed the meta-process will set the current model to be the model which generated that event before it performs the event’s action. After an event’s action is complete the meta-process will return to not having a current model. It is possible for the code in the event’s action to also modify which model is current if necessary, for example to send a notice to some other model.

Because the meta-process maintains the current model automatically based on the events, user defined modules (or other code) which generate events to perform their actions should not need to have any changes made to them to be able to work with multiple models. The only thing that would be important for a module is that if it has any internal state it would need to generate a unique instance for each model when it is created. Thus, as long as the creation function for the module returns a new “instance” (structure, class, vector, etc.) for each model and doesn’t rely on globally defined objects it should work with multiple models without any modifications. All of the standard modules provided with the system will work as described in this manual whether there is only one model defined or if there are multiple models defined.

The next thing to discuss is the two ways in which multiple models can be run. The choice is whether they are to be run synchronously (at the same time with one clock) or asynchronously (individually with their own clocks). The distinction is important because it is determined at the time the models are defined and cannot be changed. Note however that those are not mutually exclusive and it is possible to use both mechanisms with different models all simultaneously defined where there are distinct sets of models which run synchronously within a set, but asynchronously between the sets.

What determines how the models will run is whether or not they are defined within the same meta-process. If there is only one meta-process defined then all models will be defined within that meta-process and will run synchronously – any command which runs the system will run all of those models. However, if there are multiple meta-processes defined then each meta-process will run the models defined within it on its own clock which will be independent of the clocks in other meta-processes. Those meta-processes can only be run one at a time, since there is only ever one current meta-process which can be run.

[It is worth noting here that the ACT-R system is only designed to be run single threaded. When a meta-process is running multiple models they are all being run in one thread and all of their events are being executed off of a single event queue. Similarly, when there are multiple meta-processes defined only one should ever be run at a time. Even though in a threaded Lisp system it would be possible to spawn separate threads with different current meta-processes to run, the ACT-R code has no protection to prevent unintended interactions in such a situation and doing so will likely lead to improper operations and errors.]

When working with multiple models one thing to be aware of is that the system does not come with any built in ways for those models to interact. Other than sharing a clock in the case of synchronous models there is no interaction among the models. Each one has its own set of modules, chunk-types, chunks, and parameters and unless there is some interaction implemented by the user the models will run the same when they are run together as each would run on its own because they are not aware of the other models' existence. In particular, they do not “see” the other models' events in the queue. Thus, events waiting to be scheduled will not be removed from the waiting queue until an appropriate trigger from the same model occurs.

Synchronous models

Synchronous models are usually easier to work with since they are all defined within the same meta-process, and if there is only one such meta-process that simplifies things even more. For this section the assumption will be that there is only one meta-process. These same principles would also apply to a situation with multiple meta-processes if one wants to create asynchronous sets of synchronous models. The only difference would be the need to specify the appropriate current meta-process when defining the models, as described in the asynchronous section.

To create synchronous models all one needs to do is define them. There is no special syntax needed, and the standard [define-model command](#) will create each model the same as it would if that were the only model being defined. Each model defined will exist within the current meta-process and they will all run when that meta-process is run.

An example of defining multiple synchronous models can be found in the examples directory of the distribution in the file called “unit-1-together-1-mp.lisp”. That file contains the definitions for all three of the example models from unit 1 of the tutorial (count, addition, and semantic). Each of the model definitions is just copied from the corresponding tutorial model file. The only difference between loading that file with all three defined and loading the individual model files is that the individual model files each call [clear-all](#) when loaded which results in removing all models and all meta-processes except for the default one before defining the model. The combined file only calls clear-all once to remove other models and meta-processes and then defines each of the models one after the other. That model file will be used to create some of the examples when describing the additional commands available for working with multiple models.

After loading that file one can run it just as one would have run any of the individual models by calling the run command. Here is the beginning of the trace of a run of those models:

```
> (run 10)
0.000 COUNT      GOAL      SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
0.000 SEMANTIC   GOAL      SET-BUFFER-CHUNK GOAL G1 REQUESTED NIL
0.000 ADDITION   GOAL      SET-BUFFER-CHUNK GOAL SECOND-GOAL REQUESTED NIL
0.000 COUNT      PROCEDURAL CONFLICT-RESOLUTION
0.000 COUNT      PROCEDURAL PRODUCTION-SELECTED START
0.000 COUNT      PROCEDURAL BUFFER-READ-ACTION GOAL
0.000 SEMANTIC   PROCEDURAL CONFLICT-RESOLUTION
0.000 ADDITION   PROCEDURAL CONFLICT-RESOLUTION
0.050 COUNT      PROCEDURAL PRODUCTION-FIRED START
0.050 COUNT      PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 COUNT      PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 COUNT      PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 SEMANTIC   PROCEDURAL PRODUCTION-FIRED INITIAL-RETRIEVE
0.050 SEMANTIC   PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 ADDITION   PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 ADDITION   PROCEDURAL CLEAR-BUFFER RETRIEVAL
...
```

When there are multiple models running there will be an additional column shown in the trace indicating which model is performing the action. Here we see actions from each of the count, semantic, and addition models occurring.

Commands

current-model

Syntax:

current-model -> [name | **nil**]

Arguments and Values:

name ::= a symbol which will be the name of the current model in the current meta-process

Description:

The `current-model` command can be used to get the name of the current model in the current meta-process. If there is a current model then that model's name will be returned. If there is no current model in the current meta-process or there is no current meta-process then **nil** will be returned.

This command is likely to be used when one has code which can be called for different models and it's important to know which model is the current one. An example of that would be the event handler methods for a device which may be called for different models using the same type of device.

Examples:

```
1> (clear-all)
NIL

2> (current-model)
NIL

3> (define-model m1)
M1

4> (current-model)
M1

5> (define-model m2)
M2

6> (current-model)
NIL

7> (define-model m3 (format t "Current-model is: ~S~%" (current-model)))
Current-model is: M3
M3

8> (current-model)
```

NIL

mp-models

Syntax:

mp-models -> (name*)

Arguments and Values:

name ::= a symbol which will be the name of a model in the current meta-process

Description:

The `mp-models` command can be used to get the names of all of the models defined in the current meta-process. If there is a current meta-process then a list with the names of all the currently defined models will be returned. If there is no current meta-process then a warning will be printed and **nil** will be returned.

This command is likely to be used when one has code which needs to do something for all the current models (like provide some external percept) or perhaps to send a message from one model to all the others in some action (like a vocal speech act).

Examples:

```
1> (clear-all)
NIL
```

```
2> (mp-models)
NIL
```

```
3> (define-model m1)
M1
```

```
4> (mp-models)
(M1)
```

```
5> (define-model m2)
M2
```

```
6> (mp-models)
(M2 M1)
```

```
E> (mp-models)
#|Warning: mp-models called with no current meta-process.|#
NIL
```

delete-model

Syntax:

```
delete-model {name} -> [ t | nil ]  
delete-model-fct name -> [ t | nil ]
```

Arguments and Values:

name ::= a symbol which should name a model in the current meta-process

Description:

The `delete-model` command can be used to completely remove a model from the current meta-process. When a name is provided and that name names a model in the current meta-process that model will be completely removed from the meta-process and `t` will be returned. If *name* does not name a model in the current meta-process or there is no current meta-process then a warning will be printed and `nil` will be returned.

If no name is provided to `delete-model` then the current model in the current meta-process will be removed from that meta-process. If there is a current model and a current meta-process then that model is removed and `t` will be returned. If no name is provided and there is no current model or no current meta-process then a warning will be printed and `nil` will be returned.

Examples:

```
1> (clear-all)  
NIL  
  
2> (define-model m1)  
M1  
  
3> (delete-model m1)  
T  
  
4> (define-model m2)  
M2  
  
5> (delete-model)  
T  
  
6> (define-model m3)  
M3  
  
7> (define-model m4)  
M4  
  
8> (delete-model-fct 'm3)  
T  
  
9> (mp-models)  
(M4)  
  
E> (delete-model foo)  
#|Warning: No model named F00 in current meta-process.|#  
NIL
```

```
E> (delete-model)
#|Warning: No current model to delete. |#
NIL

E> (delete-model)
#|Warning: delete-model called with no current meta-process.
No model deleted. |#
NIL
```

with-model

Syntax:

```
with-model name form* -> [ result | nil ]
with-model-eval name form* -> [ result | nil ]
with-model-fct name (form*) -> [ result | nil ]
```

Arguments and Values:

name ::= a symbol which should name a model in the current meta-process
form ::= a valid Lisp expression to evaluate
result ::= the value returned from the last form evaluated

Description:

The with-model commands are used to set a model to be the current model and then execute some commands. If the name provided is the name of a model in the current meta-process then that model will be set as the current model before evaluating the forms provided. After those forms have been evaluated the current model will be returned to whichever model was the current one before with-model was called. The return value from the last form evaluated will be returned by with-model. If the name provided does not name a model in the current meta-process then a warning will be printed and **nil** will be returned without evaluating any of the forms.

In addition to the usual macro and functional versions of the command there is an additional macro version for with-model – with-model-eval. That command is a hybrid of the functional and macro versions and probably the one which is most likely to be used in code. The difference between with-model and with-model-eval is that with-model-eval will evaluate the expression provided for the name position to determine the name of the model to use.

Examples:

These examples assume that the “unit-1-together-1-mp.cl” example file has been loaded.

```
> (dolist (model (mp-models))
  (format t "Model: ~s~%" model)
  (with-model-eval model
```

```

(goal-focus)))
Model: COUNT
Will be a copy of FIRST-GOAL when the model runs
FIRST-GOAL
  START 2
  END 4

```

```

Model: SEMANTIC
Will be a copy of G1 when the model runs
G1
  OBJECT CANARY
  CATEGORY BIRD

```

```

Model: ADDITION
Will be a copy of SECOND-GOAL when the model runs
SECOND-GOAL
  ARG1 5
  ARG2 2

```

NIL

```

1> (run 10)
  0.000 COUNT      GOAL      SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000 SEMANTIC  GOAL      SET-BUFFER-CHUNK GOAL G1 REQUESTED NIL
  0.000 ADDITION  GOAL      SET-BUFFER-CHUNK GOAL SECOND-GOAL REQUESTED NIL
  ...
  0.500 -         - - - - - Stopped because no events left to process
0.5
147
NIL

```

```

2> (with-model semantic
    (goal-focus g2))
G2

```

```

3> (run 10)
  0.500 SEMANTIC  GOAL      SET-BUFFER-CHUNK GOAL G2 REQUESTED NIL
  0.500 SEMANTIC  PROCEDURAL CONFLICT-RESOLUTION
  0.550 SEMANTIC  PROCEDURAL PRODUCTION-FIRED INITIAL-RETRIEVE
  0.550 SEMANTIC  PROCEDURAL CLEAR-BUFFER RETRIEVAL
  0.550 SEMANTIC  DECLARATIVE START-RETRIEVAL
  0.550 SEMANTIC  PROCEDURAL CONFLICT-RESOLUTION
  0.600 SEMANTIC  DECLARATIVE RETRIEVED-CHUNK P14
  0.600 SEMANTIC  DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL P14
  0.600 SEMANTIC  PROCEDURAL CONFLICT-RESOLUTION
  0.650 SEMANTIC  PROCEDURAL PRODUCTION-FIRED CHAIN-CATEGORY
  0.650 SEMANTIC  PROCEDURAL CLEAR-BUFFER RETRIEVAL
  0.650 SEMANTIC  DECLARATIVE START-RETRIEVAL
  0.650 SEMANTIC  PROCEDURAL CONFLICT-RESOLUTION
  0.700 SEMANTIC  DECLARATIVE RETRIEVED-CHUNK P20
  0.700 SEMANTIC  DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL P20
  0.700 SEMANTIC  PROCEDURAL CONFLICT-RESOLUTION
  0.750 SEMANTIC  PROCEDURAL PRODUCTION-FIRED DIRECT-VERIFY
  0.750 SEMANTIC  PROCEDURAL CLEAR-BUFFER RETRIEVAL
  0.750 SEMANTIC  PROCEDURAL CONFLICT-RESOLUTION
  0.750 -         - - - - - Stopped because no events left to process
0.25
36
NIL

```

```

4> (with-model-fct 'count
    '((goal-focus)))
Goal buffer is empty
NIL

```

```

E> (with-model foo (goal-focus))
#|Warning: F00 does not name a model in the current meta-process|#

```

NIL

Asynchronous models

Asynchronous models are potentially a little trickier to work with than synchronous models because they require having multiple meta-processes. When there is more than one meta-process there will not be a current meta-process. Thus, all interactions with the system will require specifying which meta-process to use as the current one.

To create asynchronous models one needs to define the additional meta-processes and then define the models within the appropriate meta-processes. To do that one will need to use the [define-meta-process](#) and [with-meta-process](#) commands described below along with the [define-model command](#) as would normally be used to create a model.

There are some other things to consider when creating additional meta-processes. First, each model only exists in one meta-process – the meta-process in which it is defined. The names for models are only required to be unique within a meta-process (each meta-process has its own set of model names). Thus, it is possible to have models with the same name defined in different meta-processes, but that does not create any connection between those models. Similarly, the current model is relative to the current meta-process because each meta-process has its own set of models and thus its own current model. If there is only one model defined in each meta-process then it will not be necessary to also specify which model to use because specifying the meta-process will be sufficient. However, if you have multiple meta-processes defined and more than one model within a meta-process you will have to specify both which meta-process and which model to make current.

An example of defining multiple asynchronous models can be found in the examples directory of the distribution in the file called “unit-1-together-3-mp.lisp”. That file contains the definitions for all three of the example models from unit 1 of the tutorial (count, addition, and semantic) each being defined within a different meta-process.

After loading that file one cannot just run it using the run command as would be done with the models individually or with the synchronously defined version of the three models described above. Trying to do so would result in this warning with nothing having run:

```
> (run 10)
#|Warning: run called with no current meta-process.|#
NIL
```

Instead, one must specify a current meta-process and then run it. Here is an example running the count model (which is defined in the meta-process named default):

```
> (with-meta-process default
  (run 10))
  0.000 DEFAULT   GOAL           SET-BUFFER-CHUNK GOAL FIRST-GOAL REQUESTED NIL
  0.000 DEFAULT   PROCEDURAL      CONFLICT-RESOLUTION
  0.000 DEFAULT   PROCEDURAL      PRODUCTION-SELECTED START
  ...
  0.300 DEFAULT   -----          Stopped because no events left to process
0.3
50
NIL
```

To run one of the other models one would then need to specify that meta-process and run it as well. Here is the semantic model being run using the meta-process named semantic in which it was defined:

```
> (with-meta-process semantic
  (run 10))
  0.000 SEMANTIC  GOAL           SET-BUFFER-CHUNK GOAL G1 REQUESTED NIL
  0.000 SEMANTIC  PROCEDURAL      CONFLICT-RESOLUTION
  0.050 SEMANTIC  PROCEDURAL      PRODUCTION-FIRED INITIAL-RETRIEVE
  0.050 SEMANTIC  PROCEDURAL      CLEAR-BUFFER RETRIEVAL
  0.050 SEMANTIC  DECLARATIVE     START-RETRIEVAL
  0.050 SEMANTIC  PROCEDURAL      CONFLICT-RESOLUTION
  0.100 SEMANTIC  DECLARATIVE     RETRIEVED-CHUNK P14
  0.100 SEMANTIC  DECLARATIVE     SET-BUFFER-CHUNK RETRIEVAL P14
  0.100 SEMANTIC  PROCEDURAL      CONFLICT-RESOLUTION
  0.150 SEMANTIC  PROCEDURAL      PRODUCTION-FIRED DIRECT-VERIFY
  0.150 SEMANTIC  PROCEDURAL      CLEAR-BUFFER RETRIEVAL
  0.150 SEMANTIC  PROCEDURAL      CONFLICT-RESOLUTION
  0.150 SEMANTIC  -----          Stopped because no events left to process
0.15
24
NIL
```

When there are multiple meta-processes defined the trace will include an additional column to indicate which meta-process is running. Notice that the semantic meta-process started at time 0 even though the default meta-process had already been run since each meta-process has its own clock.

Commands

define-meta-process

Syntax:

```
define-meta-process name -> [ name | nil ]
define-meta-process-fct name -> [ name | nil ]
```

Arguments and Values:

name ::= a symbol which will be the name for a meta-process

Description:

The `define-meta-process` command is used to create a new meta-process. If the name given is a symbol which doesn't name an existing meta-process then a new meta-process with that name is created and that name is returned. If there is already a meta-process with that name or if the name is not a symbol then no meta-process is created, a warning will be printed and **nil** will be returned.

Examples:

```
> (define-meta-process new)
NEW

> (define-meta-process-fct 'another)
ANOTHER

E> (define-meta-process "what")
#|Warning: "what" is not a symbol and thus not valid as a meta-process name. |#
NIL

E> (define-meta-process-fct 'default)
#|Warning: There is already a meta-process named DEFAULT. |#
NIL
```

meta-process-names

Syntax:

```
meta-process-names -> (name*)
```

Arguments and Values:

name ::= a symbol which will be the name of a meta-process

Description:

The `meta-process-names` command can be used to get the names of all of the currently defined meta-processes. A list with the names of all the defined meta-process will be returned in no particular order. There will always be at least one member in the list because the default meta-process will always exist.

Examples:

```
1> (clear-all)
NIL

2> (meta-process-names)
(DEFAULT)

3> (define-meta-process new)
NEW
```

```
4> (meta-process-names)
(DEFAULT NEW)
```

delete-meta-process

Syntax:

```
delete-meta-process name -> [ t | nil ]
delete-meta-process-fct name -> [ t | nil ]
```

Arguments and Values:

name ::= a symbol which should name a meta-process

Description:

The `delete-meta-process` command is used to delete a meta-process. When the name provided names a meta-process that meta-process and all models defined within it will be removed from the system and **t** will be returned. If *name* does not name a meta-process then a warning will be printed and **nil** will be returned.

Note that the default meta-process cannot be deleted. If an attempt to delete it is made a warning will be printed and **nil** will be returned.

Examples:

```
1> (clear-all)
NIL

2> (meta-process-names)
(DEFAULT)

3> (define-meta-process new)
NEW

4> (define-meta-process x)
X

5> (meta-process-names)
(DEFAULT X NEW)

6> (delete-meta-process x)
T

7> (meta-process-names)
(DEFAULT NEW)

8> (delete-meta-process-fct 'new)
T

9> (meta-process-names)
(DEFAULT)
```

```
10E> (delete-meta-process new)
#|Warning: NEW does not name a meta-process.|#
NIL

E> (delete-meta-process-fct 'default)
#|Warning: Cannot delete the default meta-process.|#
NIL
```

current-meta-process

Syntax:

current-meta-process -> [name | nil]

Arguments and Values:

name ::= a symbol which will be the name of the current meta-process

Description:

The `current-meta-process` command can be used to get the name of the current meta-process. If there is a current meta-process then that meta-process's name will be returned. If there is no current meta-process then **nil** will be returned.

Examples:

```
1> (clear-all)
NIL

2> (current-meta-process)
DEFAULT

3> (define-meta-process new)
NEW

4> (current-meta-process)
NIL

5> (delete-meta-process new)
T

6> (current-meta-process)
DEFAULT
```

with-meta-process

Syntax:

with-meta-process *name form** -> [result | nil]
with-meta-process-eval *name form** -> [result | nil]
with-meta-process-fct *name (form*)* -> [result | nil]

Arguments and Values:

name ::= a symbol which should name a meta-process
 form ::= a valid Lisp expression to evaluate
 result ::= the value returned from the last form evaluated

Description:

The with-meta-process commands are used to set the current meta-process and then evaluate some code. If the name provided is the name of a meta-process then that meta-process will be set as the current one before evaluating the forms provided. After those forms have been evaluated the current meta-process will be returned to whichever meta-process was the current one before with-meta-process was called. The return value from the last form evaluated will be returned by with-meta-process. If the name provided does not name a meta-process then a warning will be printed and **nil** will be returned without evaluating any of the forms.

In addition to the usual macro and functional versions of the command there is an additional macro version for with-meta-process – with-meta-process-eval. That command is a hybrid of the functional and macro versions. The difference between with-meta-process and with-meta-process-eval is that with-meta-process-eval will evaluate the expression provided for the name position to determine the name of the model to use.

Examples:

These examples assume that the “unit-1-together-3-mp.lisp” example file has been loaded.

```
> (dolist (mp (meta-process-names))
  (with-meta-process-eval mp
    (format t "Meta-process ~s has current model ~s~%" mp (current-model))))

Meta-process DEFAULT has current model COUNT
Meta-process SEMANTIC has current model SEMANTIC
Meta-process ADDITION has current model ADDITION
NIL

> (with-meta-process semantic
  (run 10))
0.000 SEMANTIC GOAL SET-BUFFER-CHUNK GOAL G1 REQUESTED NIL
0.000 SEMANTIC PROCEDURAL CONFLICT-RESOLUTION
0.050 SEMANTIC PROCEDURAL PRODUCTION-FIRED INITIAL-RETRIEVE
0.050 SEMANTIC PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 SEMANTIC DECLARATIVE START-RETRIEVAL
0.050 SEMANTIC PROCEDURAL CONFLICT-RESOLUTION
0.100 SEMANTIC DECLARATIVE RETRIEVED-CHUNK P14
0.100 SEMANTIC DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL P14
0.100 SEMANTIC PROCEDURAL CONFLICT-RESOLUTION
0.150 SEMANTIC PROCEDURAL PRODUCTION-FIRED DIRECT-VERIFY
0.150 SEMANTIC PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.150 SEMANTIC PROCEDURAL CONFLICT-RESOLUTION
0.150 SEMANTIC ----- Stopped because no events left to process

0.15
24
```

NIL

```
> (with-meta-process-fct 'addition
  '((goal-focus)))
Will be a copy of SECOND-GOAL when the model runs
SECOND-GOAL
  ARG1  5
  ARG2  2
```

SECOND-GOAL

```
E> (with-meta-process x
    (goal-focus))
```

```
#|Warning: No actions taken in with-meta-process because X does not name a meta-process |#
```

Combining synchronous and asynchronous models

It is possible to use both synchronous and asynchronous models together by defining multiple models synchronously within different meta-processes. There are no additional commands necessary for doing so, but there are a couple of things worth noting about such situations. First, as was indicated in the asynchronous section it will require specifying both a current meta-process and a current model when interacting with the models. Also, when running synchronous models in asynchronous meta-processes there will be an additional column shown in the trace because it will show both the current meta-process and the current-model.

There is an example file called “unit-1-together-2-mp.lisp” in the examples directory of the distribution. It define all three example models from unit 1 of the tutorial together split between two different meta-processes. The count model is defined in the default meta-process and the semantic and addition models are defined in a meta-process named others. Here is some of the trace from running the others meta-process:

```
> (with-meta-process others
  (run 10))
0.000 OTHERS SEMANTIC GOAL SET-BUFFER-CHUNK GOAL G1 REQUESTED NIL
0.000 OTHERS ADDITION GOAL SET-BUFFER-CHUNK GOAL SECOND-GOAL REQUESTED NIL
0.000 OTHERS SEMANTIC PROCEDURAL CONFLICT-RESOLUTION
0.000 OTHERS ADDITION PROCEDURAL CONFLICT-RESOLUTION
0.050 OTHERS SEMANTIC PROCEDURAL PRODUCTION-FIRED INITIAL-RETRIEVE
0.050 OTHERS SEMANTIC PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 OTHERS ADDITION PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 OTHERS ADDITION PROCEDURAL CLEAR-BUFFER RETRIEVAL
...
0.500 OTHERS - - - - - Stopped because no events left to process
0.5
97
NIL
```

There we can see the additional columns indicating the meta-process being run and then the model for which the event occurred.

Other multiple model examples

There are two other example models which define multiple synchronous models available in the examples directory. Each will be described briefly here and more details can be found in the comments in those files.

The “multi-model-talking.lisp” file shows a custom device being defined which allows multiple models to hear each other as they speak. It also creates three models which can be run to show some very simple interaction among them using that device.

The “game-of-life.lisp” file uses multiple models along with a custom device and the AGI to display a set of cells following the rules of Conway’s Game of Life. Each cell is controlled by a separate model. The custom device allows them to see how many neighbors they have and provides a way for them to announce any change to the cell vocally each cycle.

Configuring Real Time Operation

When a meta-process is run with the real-time flag set to a non-nil value the system is run in step with an external clock. By default, the clock used is the Lisp `get-internal-real-time` function, but the user can change that default behavior to specify an alternate clock as well as customize some other features of how the meta-process executes in real time mode. The command [mp-real-time-management](#) is the mechanism for changing the operation. Before describing that command, a more complete description of how the meta-process advances the clock in real time mode will be provided.

When a meta-process is not run in real time mode, as described in the [running section](#), the scheduled events drive the clock directly and there is no delay between event executions regardless of the difference in time between them. When the meta-process runs in real time mode the “ACT-R time” is still determined by the events on the queue, but their execution is delayed, if necessary, so as to not run ahead of the real time clock provided. Here is some Lisp pseudo-code that describes the operation of the meta-process running in real time mode:

```
(let ((actr-start (mp-time))
      (real-start (funcall real-time-clock)))

  (loop

    (when (run-end-conditions-satisfied) (return))

    (when (and (numberp max-delta) (> (- (evt-time (next-event)) (mp-time)) max-delta))
      (dolist (model (mp-models))
        (with-model-eval model
          (schedule-event-relative max-delta 'dummy-event-function
                                   :priority :max
                                   :details "A dummy event to prevent model skip ahead"
                                   :output nil
                                   :maintenance max-delta-maintenance))))

      (do* ((current-real-time (funcall real-time-clock))
            (funcall real-time-clock))
            (delta-model (- (evt-time (next-event)) actr-start)
                          (- (evt-time (next-event)) actr-start))
            (delta-real (* scale (/ (- current-real-time real-start) real-units-per-second))
                       (* scale (/ (- current-real-time real-start) real-units-per-second)))
            ((>= delta-real delta-model))
            (funcall real-time-slack
                    (/ (- delta-model delta-real) scale)
                    (evt-time (next-event))))

      (setf (mp-time) (evt-time (next-event)))

      (execute (next-event))))
```

The functions in blue are not real ACT-R commands, but should be assumed to work as follows:

`run-end-conditions-satisfied` returns true when conditions specified for the current run have been met.

`next-event` returns the next event from the event queue.

`execute` performs all the actions necessary to deal with an event (calling the event hooks, printing the trace, calling the action function, etc.).

The variables in red are items that can be set by the user to control how real time runs using the mp-real-time-management function and here are some more details about them:

`real-time-clock` must be a function that takes no parameters. It should return a number which represents a time. The time reference is not used as an absolute time, but an interval timer to pace the model (it is the difference between a starting value and the current value that is used to determine when to execute the next model action). It should return non-decreasing values for time in any units desired. The default function for this value is the Lisp function **get-internal-real-time**.

`real-units-per-second` should be a number which is used to scale the return value from the `real-time-clock` function into seconds. The default value is the value of the Lisp variable **internal-time-units-per-second**.

`scale` should be a number which is used to scale the time relative to real time passing. A scale greater than 1 means to run the meta-process faster than real-time and a scale less than one will run the meta-process slower than real time. The default value is 1.0. This value is overridden by a scale provided in a call to run the meta-process using a number for the `:real-time` keyword parameter.

`max-delta` is used to set a maximum time interval between events in the meta-process. If it is set to a number, then that is assumed to be the maximum time in seconds allowed between successive events. If there is not an event scheduled within that amount of time from the current time, then a dummy event is scheduled for every model `max-delta` seconds from the current time. Note that if there are multiple models running in the meta-process then this does not place a bound on the amount of time that a single model may go without an event, just how long the meta-process will go without processing one. The default value for this is **nil** which means there is no bound placed on the interval between events.

`max-delta-maintenance` specifies whether the events scheduled to enforce the `max-delta` timing will be marked as maintenance events. The default value is **nil** which means they will not be marked as maintenance events.

real-time-slack this must be a function which accepts two parameters. It is called whenever the next model event is “ahead” of the current interval length as reported by the **real-time-clock** i.e. when the model must wait before executing the event. The function is passed two values. The first is the duration which the model must wait in seconds (scaled to real seconds by the scale), and the second is the ACT-R time of that next event in seconds. The function should perform some appropriate action to allow the time to pass cleanly which may involve calling some Lisp implementation specific functions to handle system events if necessary. The default value for this function is one which calls the Lisp function **sleep** with the duration which the model must wait if that time is greater than or equal to a value set by the `:real-time-sleep-threshold` system parameter (defaults to .15 seconds).

Dynamic events

Both the `real-time-clock` and `real-time-slack` functions may safely schedule new ACT-R events. This is the recommended way to place events from an external simulation into the event queue. This is especially true if the external communication is being handled in a separate Lisp process because the ACT-R scheduling commands are not “thread safe” and thus should only be called by code running in the main ACT-R process. [How such a process communicates with these functions to schedule events should be properly protected, and such mechanisms are up to the user to implement since the requirements may vary among Lisp implementations. See the “device-interaction-issues” document in the docs directory for more information on working with code which may run outside of the thread running ACT-R.] Those external events may occur before the time when the “current” next event is scheduled in some circumstances e.g. the current-real-time is .5 seconds, the next event is scheduled for 2.5 seconds, and during the `real-time-slack` function something happens which should occur at time 1.5 seconds. Because the real time operation always checks for the next event, when a closer event gets scheduled it will be detected as the next event the next iteration through the loop and be used to determine the appropriate waiting time. However, it is not possible to schedule an event for a time which has already passed. The ACT-R clock will only move forward and any event scheduled with a time that has already passed will be scheduled with the current time instead.

The most likely situation where an event would be entered prior to the current next one would happen when the model is “waiting” for a long request to finish before the next production will match i.e. conflict-resolution fails and thus gets scheduled for after that next event. When not in real time mode that works fine because nothing else could happen during that period, but when connecting to an external simulation some external event could happen during that interval and the model should be able to detect that. If such events could occur in a simulation and the modeler wants the model to be sensitive to those events then there are two things that need to be done. First, when configuring the

real time control with `mp-real-time-management` the `:allow-dynamics` parameter should be specified as true. This lets the meta-process know that any events which are scheduled prior to the current next event are important to the model. With that setting enabled, events which were scheduled based on the contents of the event queue (those scheduled with `schedule-event-after-*` commands) may be reevaluated and rescheduled based on newly scheduled events. Only events which are specified as being dynamic when created will be reconsidered when new events are scheduled. Of the events generated by the standard modules of the system, only the procedural module currently schedules dynamic events. The conflict-resolution events that it generates are specified as dynamic. Thus, conflict-resolution events will be rescheduled to occur after newer events which are added to the queue. That may be all that is needed in many situations, but if the modeler creates other events for the simulation or model which are to be sensitive to newly scheduled events then those events need to have the dynamic flag set to true when they are scheduled.

Commands

`mp-real-time-management`

Syntax:

```
mp-real-time-management {:time-function timefct {:units-per-second ups {:slack-function slackfct  
  {:max-time-delta delta {:delta-maintenance maintenance  
  {:allow-dynamics dynamics {:scale scale -> [ t | nil ]
```

Arguments and Values:

`timefct` ::= a function or a symbol which names a function to use as the real time clock (default is **get-internal-real-time**)

`ups` ::= a positive number specifying the number of units returned by `timefct` in a second (default is **internal-time-units-per-second**)

`slackfct` ::= a function or a symbol which names a function to call when the model needs to wait (default is **real-time-slack**)

`delta` ::= **nil** or a positive number specifying the maximum time in seconds allowed between events (default is **nil**)

`maintenance` ::= a generalized boolean indicating whether events scheduled to enforce the max time delta are marked as maintenance events (default is **nil**)

`dynamics` ::= a generalized boolean indicating whether to reschedule dynamic events (default is **nil**)

`scale` ::= a positive number specifying a scale for the real time clock (default is 1.0)

Description:

The `mp-real-time-management` command is used to modify how the current meta-process will operate when run with the real-time flag set to true. It sets the real time controls for the current meta-

process to the provided values with the default settings for those parameters not provided and returns **t**. How those controls affect the operation of the system is described in the [Configuring Real Time Operation section](#) above.

If there is no current meta-process, the model is currently running (the values cannot be changed once the model starts running), or one or more of the parameters are invalid then no changes will be made, there will be a warning printed, and the value **nil** will be returned.

Examples:

These first two examples show how the real time running could be made faster or slower than real time (respectively) without adjusting any other values.

```
> (mp-real-time-management :scale 2)
T
```

```
> (mp-real-time-management :scale .25)
T
```

The next examples show max-time-delta causing intermediate events to be scheduled and by enabling or disabling the allow-dynamics setting how that affects when conflict-resolution occurs. To do that this model is assumed to be defined prior to these actions:

```
(define-model test-model
  (sgp :esc t :lf 1.0 :trace-detail medium)
```

```
  (p start
    ?goal>
    buffer empty
    ==>
    +goal>
    +retrieval>))
```

```
1> (reset)
DEFAULT
```

```
2> (run 10 :real-time t)
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL CLEAR-BUFFER GOAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 GOAL SET-BUFFER-CHUNK GOAL CHUNK0
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 DECLARATIVE RETRIEVAL-FAILURE
1.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 ----- Stopped because no events left to process
```

```
1.05
17
NIL
```

```
3> (reset)
DEFAULT
```

```

4> (mp-real-time-management :allow-dynamics nil :max-time-delta 0.5)
T

5> (run 10 :real-time t)
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL CLEAR-BUFFER GOAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 GOAL SET-BUFFER-CHUNK GOAL CHUNK0
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 DECLARATIVE RETRIEVAL-FAILURE
1.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 ----- Stopped because no events left to process

1.05
18
NIL

```

Notice that an extra event occurred in the second run relative to the first (as indicated by the second return value from run) although there was nothing different displayed in the trace. A non-printing event occurred at time .550 because the gap between events at .05 and 1.0 was greater than the maximum time of .5 specified by :max-time-delta.

```

5> (reset)
DEFAULT

6> (mp-real-time-management :allow-dynamics t :max-time-delta 0.5)
T

6> (run 10 :real-time t)
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL CLEAR-BUFFER GOAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 GOAL SET-BUFFER-CHUNK GOAL CHUNK0
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.550 PROCEDURAL CONFLICT-RESOLUTION
1.050 DECLARATIVE RETRIEVAL-FAILURE
1.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 ----- Stopped because no events left to process

1.05
19
NIL

```

This time because :allow-dynamics was set to **t** a conflict-resolution event occurred at time .550 because it was allowed to be moved back when the event generated because of the time delta was scheduled.

```

7> (reset)
DEFAULT

8> (mp-real-time-management :allow-dynamics t :max-time-delta 0.5 :delta-maintenance t)
T

9> (run 10 :real-time t)

```

```

0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED START
0.050 PROCEDURAL CLEAR-BUFFER GOAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 GOAL SET-BUFFER-CHUNK GOAL CHUNK0
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 DECLARATIVE RETRIEVAL-FAILURE
1.050 PROCEDURAL CONFLICT-RESOLUTION
1.050 ----- Stopped because no events left to process
1.05
18
NIL

```

This time even though `:allow-dynamics` was set to `t` the conflict-resolution event did not occur at time .550 because the delta time event is marked as a maintenance event and thus the procedural module will ignore it for scheduling a conflict-resolution event.

```

E> (mp-real-time-management :max-time-delta t)
#|Warning: Max-time-delta T is not a positive number or nil|#
NIL

E> (mp-real-time-management :time-function nil)
#|Warning: Time-function NIL not a valid function for mp-real-time-management|#
NIL

E> (mp-real-time-management :units-per-second t)
#|Warning: Units-per-second T must be a positive number|#
NIL

E> (mp-real-time-management :slack-function 'not-function)
#|Warning: Slack-function NOT-FUNCTION not a valid function for mp-real-time-management|#
NIL

1> (reset)
DEFAULT

2> (schedule-event-relative 0 'mp-real-time-management :priority :max)
#S(ACT-R-EVENT ...)

3E> (run-n-events 2)
#|Warning: Mp-real-time-management cannot adjust real-time operation while the model is
running.|#
0.000 ----- Stopped because event limit reached
0.0
2
NIL

E> (mp-real-time-management :allow-dynamics nil :max-time-delta 0.5)
#|Warning: mp-real-time-management called with no current meta-process.|#
NIL

```

Module Activity and Brain Predictions

The activity of the modules in ACT-R (as reported through their buffers) along with a mapping of those modules/buffers onto the brain can be used to make predictions from a model for data from various imaging methods like fMRI, EEG, PET, or MEG. The ACT-R software includes commands for getting a summary of the module activity as well as commands for producing a prediction of a blood-oxygen-level dependent (BOLD) response for comparison to fMRI data.

This section will describe how one can have the system record the module activity and ways to access that data, and the next one will describe the tools which use that data to produce a BOLD response. For information on mappings of modules to brain regions and additional details on predicting the BOLD response there are many papers available from the [ACT-R website](#) under the fMRI category, and a good place to start is the paper “[A central circuit of the mind](#)”.

Configure a model to record module activity

To get the module activity or produce a BOLD response the model must be configured to record that activity as it runs, and the [buffer trace module](#) provides that capability. The model must set the parameters of that module to record the desired information, and there are two parameters which are important: [:save-buffer-trace](#) and [:traced-buffers](#).

The [:save-buffer-trace](#) parameter must be set to **t** so that the activity is recorded. With that set, by default, all buffers’ activities are recorded. If not all the buffers are needed, or one wants to display the output of the data in a particular order, then the [:traced-buffers](#) parameter should be set to the list of buffers for which the activity data is desired, and the order of that list determines the ordering for any output data. Recording the module activity can be a costly operation over the course of a long model run, so it is recommended that one only record the information that’s needed for efficiency purposes.

Commands for getting module activity

There are three commands available for getting the module activity which provide different ways to collect the data. All of these commands report the activity at the buffer level i.e. if a module has more than one buffer each buffer is tracked separately. If one would like to process the module activity differently it is also possible to use the [get-current-buffer-trace](#) command and process the raw buffer-trace information directly.

What counts as buffer activity?

Buffer activity is determined from the buffer-trace data reported by `get-current-buffer-trace`. Thus, the first requirement is that only actions which occur as scheduled events will be recorded. Any function calls which directly affect a buffer (for example `set-buffer-chunk`) will not be counted as buffer activity.

Buffer activity starts when one of three things occurs: a request (or modification request) is made to the buffer, a chunk is placed into the buffer, or a “state busy” query of the buffer is true. Once activity has started the buffer will be considered active until one of three conditions occurs: the “state busy” query returns **nil**, the “state free” query returns true, or a new request (or modification request) is made to the buffer (which ends the current activity and also starts a new one).

Some buffers (in particular the goal buffer) may have activities which are instantaneous i.e. the start and stop times are the same. That activity is still recorded, and how that is treated by the different commands will be discussed with each one.

module-demand-times

Syntax:

```
module-demand-times { :start start } { :end end } -> (buffer-time-list*)
```

Arguments and Values:

`start` ::= a number indicating the starting time for which activity is to be reported in seconds

`end` ::= a number indicating the end time for which activity is to be reported in seconds

`buffer-time-list` ::= a list with a buffer name as the first element and the rest being conses of activity start and end times

Description:

The `module-demand-times` command has two keyword parameters, `:start` and `:end`, which can be used to restrict the reported activity to the specified range. If no start time is provided then it starts at time 0, and if no end time is provided then it ends at the current model time. It returns a list of lists of buffer activity times for all of the currently traced buffers in the current model of the current meta-process. If there is no current model or current meta-process then the command will print a warning and return **nil**. If a start or end value which isn't a number is provided a warning will be displayed and **nil** will be returned.

The list for a buffer's activity will have the symbol which is the buffer name as the first element and the remaining elements will be conses with the car being the time at which that activity started and the cdr being the time at which it ended. The activities will be in order by start time (lowest first). The activity times will be measured in seconds, but returned as a rational number to prevent any loss of accuracy due to the precision of floating point numbers.

Buffers which have instantaneous activities will report those with conses that have the same start and end times.

If a start and/or end time is provided then only activities which overlap with that range will be reported. That may include activities which begin before the provided start time and persist past it as well as activities which begin before the provided end time and continue past it.

Examples:

These examples uses the zbrodoff model from unit 4 of the ACT-R tutorial with the addition of this parameter setting:

```
(sgp :save-buffer-trace t :traced-buffers (goal visual retrieval vocal imaginal manual))
```

and the run was generated with this call:

```
(zbrodoff-trial "a" "2" "c" "k")
```

```
> (module-demand-times)
((GOAL (0 . 0) (447/200 . 447/200)) (VISUAL (1/20 . 27/200) (47/200 . 8/25)
(107/200 . 31/50) (147/200 . 157/200)) (RETRIEVAL (197/200 . 197/200) (247/200 . 247/200)
(297/200 . 297/200) (347/200 . 347/200) (397/200 . 397/200)) (VOCAL (37/200 . 87/200)
(97/200 . 137/200) (147/200 . 187/200) (197/200 . 237/200) (247/200 . 287/200)
(297/200 . 337/200) (347/200 . 387/200) (397/200 . 437/200))(IMAGINAL (37/200 . 77/200))
(MANUAL (447/200 . 507/200)))

> (module-demand-times :start 1.0)
((GOAL (447/200 . 447/200)) (VISUAL) (RETRIEVAL (247/200 . 247/200) (297/200 . 297/200)
(347/200 . 347/200) (397/200 . 397/200))(VOCAL (197/200 . 237/200) (247/200 . 287/200)
(297/200 . 337/200) (347/200 . 387/200) (397/200 . 437/200)) (IMAGINAL) (MANUAL
(447/200 . 507/200)))

> (module-demand-times :end 1.0)
((GOAL (0 . 0)) (VISUAL (1/20 . 27/200) (47/200 . 8/25) (107/200 . 31/50)
(147/200 . 157/200)) (RETRIEVAL (197/200 . 197/200)) (VOCAL (37/200 . 87/200)
(97/200 . 137/200) (147/200 . 187/200) (197/200 . 237/200)) (IMAGINAL (37/200 . 77/200))
(MANUAL))

> (module-demand-times :start 1.0 :end 2.0)
((GOAL) (VISUAL) (RETRIEVAL (247/200 . 247/200) (297/200 . 297/200) (347/200 . 347/200)
(397/200 . 397/200))(VOCAL (197/200 . 237/200) (247/200 . 287/200) (297/200 . 337/200)
(347/200 . 387/200) (397/200 . 437/200)) (IMAGINAL) (MANUAL))
```

```

> (module-demand-times :start 'a :end 2)
#|Warning: Start and end times for module-demand-times must be numbers, but given A,2. |#
NIL

E> (module-demand-times)
#|Warning: Module-demand-times requires a current model. |#
NIL

E> (module-demand-times)
#|Warning: Module-demand-times requires a current meta-process. |#
NIL

```

module-demand-functions

Syntax:

module-demand-functions `{:start start} {:end end} {:step step} -> (buffer-activity-list*)`

Arguments and Values:

start ::= a number indicating the starting time for which activity is to be reported in seconds
end ::= a number indicating the end time for which activity is to be reported in seconds
step ::= a number indicating the granularity of the activity record to report in seconds
buffer-activity-list ::= a list with a buffer name as the first element and the rest being 1's or 0's to indicate activity

Description:

The `module-demand-functions` command has three keyword parameters. `:start` and `:end` can be used to restrict the reported activity to the specified range. `:step` indicates the size of the windows over which activity is reported. If no start time is provided then it starts at time 0, and if no end time is provided then it ends at the current model time. If no step size is given then it will use the value of the [:bold-inc parameter](#) of the current model. It breaks the time between start and end into segments `step` seconds long and records whether any activity for each buffer occurs within that segment and returns a list of lists of the buffer activity for all of the currently traced buffers in the current model of the current meta-process. If there is no current model or current meta-process then the command will print a warning and return **nil**.

The list for a buffer's activity will have the symbol which is the buffer name as the first element and the rest of the list will consist of n elements where:

$$n = \left\lceil \frac{end - start}{step} \right\rceil$$

Each of those elements, indexed by i , represents the activity recorded for the time $[S_i, S_{i+1})$ where:

$s_0 = \text{start}$
 $s_i = s_{i-1} + \text{step}$

If a buffer has any activity within a time segment the corresponding element of the list will be 1 and if it has no activity the value will be 0. Note that if the end time of a buffer activity (as reported by module-demand-times) corresponds with the end time of a segment that will not count as activity occurring at the start of the next segment i.e. if the step size is 2 and a buffer has reported activity of (0.0 . 2.0) that will only count as activity in the first step and not the second.

An activity that takes no time (the start and end times are the same) is still considered activity and will result in a 1 for the step in which it occurs.

If the `:v` and `:cmdt` parameters are both enabled then a table of the values will be written using the `command-output` command.

Examples:

These examples uses the zbrodoff model from unit 4 of the ACT-R tutorial with the addition of this parameter setting:

```
(sgp :save-buffer-trace t :traced-buffers (goal visual retrieval vocal imaginal manual))
```

and the run was generated with this call:

```
(zbrodoff-trial "a" "2" "c" "k")
```

which ended at time 2.535.

```
> (module-demand-functions)
      TIME      GOAL    VISUAL  RETRIEVAL    VOCAL    IMAGINAL    MANUAL
      0.000      1.000      1.000      1.000      1.000      1.000      0.000
      2.000      1.000      0.000      0.000      1.000      0.000      1.000

((GOAL 1 1) (VISUAL 1 0) (RETRIEVAL 1 0) (VOCAL 1 1) (IMAGINAL 1 0) (MANUAL 0 1))

> (module-demand-functions :step .5)
      TIME      GOAL    VISUAL  RETRIEVAL    VOCAL    IMAGINAL    MANUAL
      0.000      1.000      1.000      0.000      1.000      1.000      0.000
      0.500      0.000      1.000      1.000      1.000      0.000      0.000
      1.000      0.000      0.000      1.000      1.000      0.000      0.000
      1.500      0.000      0.000      1.000      1.000      0.000      0.000
      2.000      1.000      0.000      0.000      1.000      0.000      1.000
      2.500      0.000      0.000      0.000      0.000      0.000      1.000

((GOAL 1 0 0 0 1 0) (VISUAL 1 1 0 0 0 0) (RETRIEVAL 0 1 1 1 0 0) (VOCAL 1 1 1 1 1 0)
 (IMAGINAL 1 0 0 0 0 0) (MANUAL 0 0 0 0 1 1))
```

```

> (module-demand-functions :start 1 :end 2 :step .2)
      TIME      GOAL    VISUAL  RETRIEVAL    VOCAL    IMAGINAL    MANUAL
      1.000     0.000    0.000    0.000     1.000     0.000     0.000
      1.200     0.000    0.000    1.000     1.000     0.000     0.000
      1.400     0.000    0.000    1.000     1.000     0.000     0.000
      1.600     0.000    0.000    1.000     1.000     0.000     0.000
      1.800     0.000    0.000    1.000     1.000     0.000     0.000

((GOAL 0 0 0 0 0) (VISUAL 0 0 0 0 0) (RETRIEVAL 0 1 1 1 1) (VOCAL 1 1 1 1 1) (IMAGINAL 0 0
0 0 0)(MANUAL 0 0 0 0 0))

E> (module-demand-functions)
#|Warning: Module-demand-functions requires a current model. |#
NIL

E> (module-demand-functions)
#|Warning: Module-demand-functions requires a current meta-process. |#
NIL

E> (module-demand-functions :step 'bad)
#|Warning: Start, end, and step for module-demand-functions must be numbers, but given
0,2.535,BAD. |#
NIL

```

module-demand-proportion

Syntax:

module-demand-proportion *{:start start} {:end end} {:step step}* -> (buffer-activity-list*)

Arguments and Values:

start ::= a number indicating the starting time for which activity is to be reported in seconds
end ::= a number indicating the end time for which activity is to be reported in seconds
step ::= a number indicating the granularity of the activity record to report in seconds
buffer-activity-list ::= a list with a buffer name as the first element and the rest being proportions of activity in each step

Description:

The module-demand-proportion command works like [module-demand-functions](#) except that instead of a 1 or 0 to indicate activity for each step it returns the proportion of time during the step which the module was active. If a module activity takes 0 time (the start and end times are the same) it is considered to take 1ms for computing the proportion.

The calculations are done using rational numbers for the times and if the start, end, and step times are provided as integers or rationals (true for the default values) then the results will also be rational numbers, but if one of the provided values is a floating point number the results may end up as floating points because of float contagion.

Examples:

These examples uses the zbrodoff model from unit 4 of the ACT-R tutorial with the addition of this parameter setting:

```
(sgp :save-buffer-trace t :traced-buffers (goal visual retrieval vocal imaginal manual))
```

and the run was generated with this call:

```
(zbrodoff-trial "a" "2" "c" "k")
```

which ended at time 2.535.

```
> (module-demand-proportion)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
0.000	0.001	0.153	0.002	0.733	0.100	0.000
2.000	0.001	0.000	0.000	0.093	0.000	0.150

```
((GOAL 1/2000 1/2000) (VISUAL 61/400 0) (RETRIEVAL 1/400 0) (VOCAL 293/400 37/400) (IMAGINAL 1/10 0) (MANUAL 0 3/20))
```

```
> (module-demand-proportion :step 1)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
0.000	0.001	0.305	0.001	0.665	0.200	0.000
1.000	0.000	0.000	0.004	0.800	0.000	0.000
2.000	0.001	0.000	0.000	0.185	0.000	0.300

```
((GOAL 1/1000 0 1/1000) (VISUAL 61/200 0 0) (RETRIEVAL 1/1000 1/250 0) (VOCAL 133/200 4/5 37/200) (IMAGINAL 1/5 0 0) (MANUAL 0 0 3/10))
```

```
> (module-demand-proportion :start 1 :end 2 :step .2)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
1.000	0.000	0.000	0.000	0.925	0.000	0.000
1.200	0.000	0.000	0.005	0.825	0.000	0.000
1.400	0.000	0.000	0.005	0.750	0.000	0.000
1.600	0.000	0.000	0.005	0.750	0.000	0.000
1.800	0.000	0.000	0.005	0.750	0.000	0.000

```
((GOAL 0.0 0.0 0.0 0.0 0.0) (VISUAL 0.0 0.0 0.0 0.0 0.0) (RETRIEVAL 0.0 0.005 0.005 0.005 0.005) (VOCAL 0.925 0.825 0.75 0.75 0.75) (IMAGINAL 0.0 0.0 0.0 0.0 0.0) (MANUAL 0.0 0.0 0.0 0.0 0.0))
```

```
E> (module-demand-proportion)
```

```
##Warning: Module-demand-proportion requires a current model. |#
```

```
NIL
```

```
E> (module-demand-proportion)
```

```
##Warning: Module-demand-proportion requires a current meta-process. |#
```

```
NIL
```

```
E> (module-demand-proportion :start t)
```

```
##Warning: Start, end, and step for module-demand-proportion must be numbers, but given T, 507/200,2. |#
```

```
NIL
```

BOLD module

The BOLD module provides the modeler with a command for predicting blood-oxygen-level dependent (BOLD) response values from the activity of the buffers for comparison to fMRI data. This section will provide a general description of how that calculation is performed and then describe the parameters for configuring it and the command to generate the response. This description assumes that one has read the previous section on [module activity and brain predictions](#) for an understanding of how the buffer activities are determined and that one has some understanding of what the BOLD response in fMRI data represents.

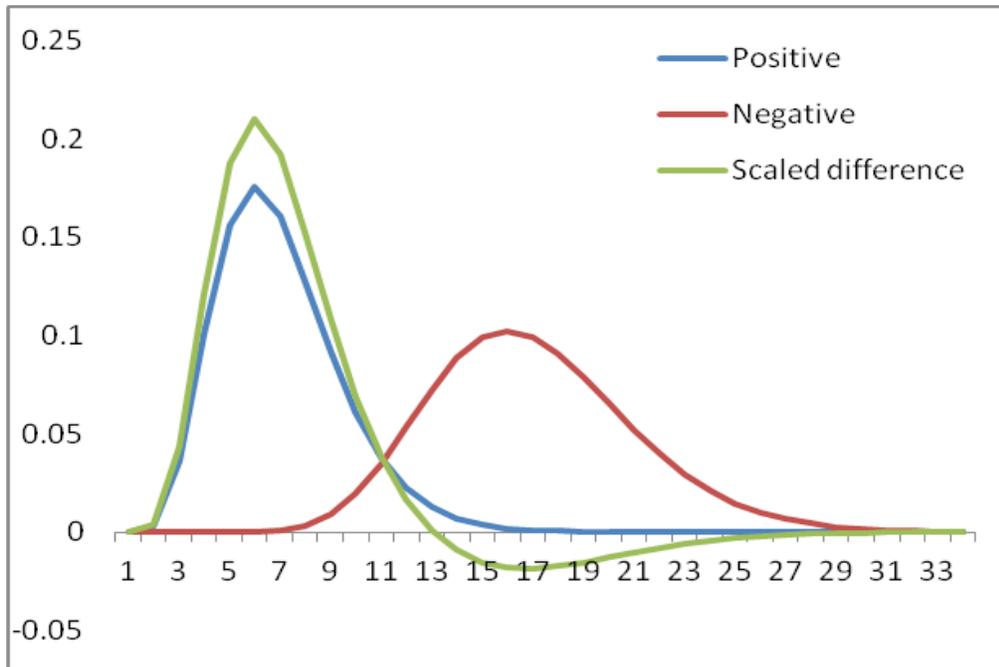
The basic concept used to create the BOLD response prediction for a model is that while a module is active it corresponds to an increased metabolic demand which produces a hemodynamic response that generates a BOLD response. Therefore, to compute the BOLD response for a model run we need two things. The first is the model's buffer activity which we will get from the [module-demand-times](#) command. We will use those times to create a demand function for the buffer activity where the value of the function is 1 when the buffer is active and 0 when it is not. We will call this function $D(t)$. The other is a representation of the hemodynamic response. For that we will use a difference of two Gamma probability density functions which is used by many other researchers and also used in the SPM software package (Friston, Ashburner, Kiebel, Nichols, & Penny, 2011).

The Gamma probability density function is described in terms of two parameters, an exponent (or shape), \mathbf{a} and a scale, \mathbf{b} :

$$\text{gammapdf}(x; a, b) = \frac{x^{a-1} e^{-x/b}}{b^a (a-1)!}$$

The default settings for the parameters in the BOLD module use the same exponent and scale parameters for the positive and negative curves as SPM ($a=6, b=1$ for positive and $a=16, b=1$ for the negative), but they are combined using a slightly different scaling in the ACT-R representation. How to set the parameters for specifying the positive and negative curves and the parameters to control the scale factors combining them are described below along with how one would change that to match the scaling used in SPM.

Here is what the default functions used by the BOLD module look like:



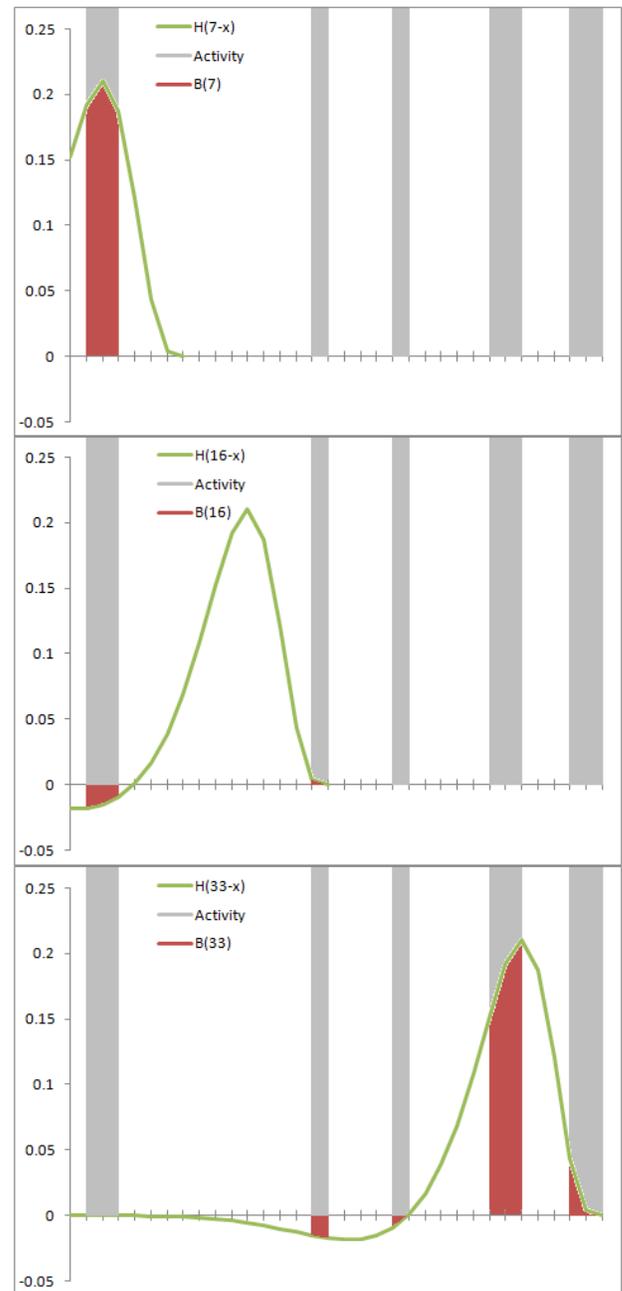
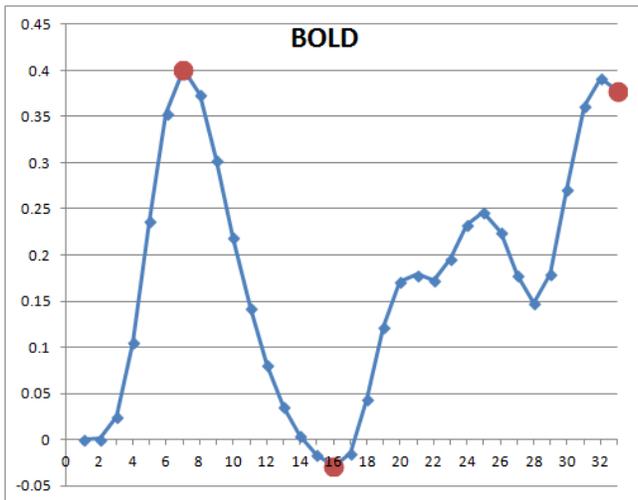
The green curve is the hemodynamic response function we will use and call $H(t)$.

To produce the BOLD response at a given time, $B(t)$, we need to compute the accumulated hemodynamic response over all of the activity of the module up to that time. That can be done using the convolution of the demand function with the hemodynamic response function:

$$B(t) = \int_0^t D(x)H(t-x)dx$$

Because our demand function only has values of 0 or 1 the $B(t)$ value is the area under the reversed and shifted hemodynamic response function in the periods of module activity.

Here's an example showing the BOLD response for a module which has activity during the times 1-3, 15-16, 20-21, 26-28, and 31-33 and how the values were computed for three of those points:



Parameters

:bold-param-mode

This parameter determines how the exponent parameters for the Gamma distributions are set as well as how the scaling is determined for combining the positive and negative curves to generate the hemodynamic response function. It has two possible values: act-r and spm. The default value is act-r. The differences between those modes will be described with the affected parameters.

:bold-exp & :neg-bold-exp

These two parameters are used to set the exponent parameters (the **a** parameters) for the positive and negative Gamma distributions which generate the hemodynamic response function, and the values must be integers. If the `:bold-param-mode` value is `spm` then the `:bold-exp` and `:neg-bold-exp` values should be set to the **a** values desired, which would be 6 and 16 respectively for the default function. If the `:bold-param-mode` value is `act-r` then these parameters must be set to one less than the **a** parameters desired i.e. they would need to be set to 5 and 15 for the default function. Since `act-r` mode is the default these parameters default values are 5 and 15. However, if one changes `:bold-param-mode` to `spm` when both `:bold-exp` and `:neg-bold-exp` are set to their default values they will be automatically changed to the corresponding `spm` values of 6 and 16.

:bold-scale & :neg-bold-scale

These parameters are used to set the scale parameters for the positive and negative Gamma distributions which generate the hemodynamic response function. Their setting does not depend on the `:bold-param-mode` value and both default to a value of 1.

:bold-positive & :bold-negative

These two parameters are the weights used when combining the positive and negative Gamma distributions to generate the hemodynamic response function. The default values are 6 and 1 respectively.

When `:bold-param-mode` is set to `act-r` the weights are used as follows to combine the positive and negative Gamma distributions:

$$H(t) = \left[\frac{bp}{bp - bn} \text{positive}(t) \right] - \left[\frac{bn}{bp - bn} \text{negative}(t) \right]$$

where:

`bp` = `:bold-positive` value

`bn` = `:bold-negative` value

`positive(t)` = the positive Gamma distribution

`negative(t)` = the negative Gamma distribution

With this scaling method a module with a constant activity will have a BOLD value of 1, but the peak BOLD value for a module may exceed a value of 1.

When `:bold-param-mode` is set to `spm` the weights are used as follows to combine the positive and negative Gamma distributions:

$$H(t) = \left[\frac{bp}{\max(bp, bn)} \text{positive}(t) \right] - \left[\frac{bn}{\max(bp, bn)} \text{negative}(t) \right]$$

where:

`bp` = `:bold-positive` value

`bn` = `:bold-negative` value

`positive(t)` = the positive Gamma distribution

`negative(t)` = the negative Gamma distribution

This is the scaling used in SPM to generate the default hemodynamic response function.

These parameters can be set to 0 to disable the positive or negative component if desired (note however that if the positive component is disabled in `act-r` mode the result will be a positive function because the `bp - bn` value will also be negative).

`:bold-settle`

This parameter is used to help with the efficiency of computing the BOLD response values. It can be set to a time in seconds beyond which $H(t)$ can be considered as having a value of 0. Only activity which occurs within `:bold-settle` seconds of the current time is used in computing the BOLD response (essentially the integral is computed from $t - \text{:bold-settle}$ to t instead of 0 to t). The default value is 40 (with the other default values $H(40) = -.0000052$).

`:bold-inc`

This parameter is used to set the interval, in seconds, at which the BOLD predictions are computed. The default value is 2 seconds.

`:point-predict`

This parameter is used to specify a list of buffer names for which the corresponding module activities occur instantaneously, but a BOLD prediction is still desired (since the standard calculation will result in values of 0 for instantaneous activity). For the buffers on this list the demand associated with each activity will be treated as an impulse function with an area of 1 occurring at the time of the activity. Thus, the $B(t)$ value will be the sum of the $H(x-t)$ values at the activity times. The default value is a list containing the buffer name `goal`.

Commands

predict-bold-response

Syntax:

predict-bold-response {start {end}} -> (bold-list*)

Arguments and Values:

start ::= a time in seconds

end ::= a time in seconds

bold-list ::= a list of a buffer name and the BOLD data prediction for that buffer

Description:

The predict-bold-response function is used to get the predictions for the BOLD response for the currently traced buffers in the current model. It takes two optional parameters. Those parameters allow one to specify a start time and an end time over which the BOLD prediction is to be generated. The default start time is 0 if not provided and the default end time is the current model time if not provided. It returns a list of lists where each sublist is the prediction for one buffer. The first element for a buffer's prediction will be the name of the buffer and the rest of the list will be the BOLD response values for that buffer. The BOLD response values are computed as described above at the times specified as follows.

The BOLD predictions are created as if the model were being scanned by an fMRI machine which is creating one complete scan every :bold-inc seconds, starting the first scan at time 0, and reporting the results at the middle of each scan. Thus, with the default value of 2 for :bold-inc the first BOLD prediction for a model will occur at time 1, and each subsequent prediction will occur 2 seconds later. The start time should correspond to the start time of a scan (a multiple of :bold-inc). If it is not, then a warning will be printed and it will be rounded down to the closest scan start time.

If the [:v](#) and [:cmdt](#) parameters are both enabled then a table of the values will be written using the [command-output](#) command.

If either start or end is not a number, the interval between start and end is less than :bold-inc seconds, or there is no current model or meta-process then a warning will be printed and **nil** will be returned.

Examples:

These examples uses the zbrodoff model from unit 4 of the ACT-R tutorial with the addition of this parameter setting:

```
(sgp :save-buffer-trace t :traced-buffers (goal visual retrieval vocal imaginal manual))
```

and the run was generated with this call:

```
(zbrodoff-trial "a" "2" "c" "k")
```

which ended at time 2.535 plus an additional run of 10 seconds using:

```
(run-full-time 10)
```

so that there is more than one time increment over which to compute the BOLD response by default.

```
> (predict-bold-response)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
1.000	0.004	0.000	0.000	0.000	0.000	0.000
3.000	0.122	0.027	0.000	0.067	0.020	0.000
5.000	0.312	0.063	0.000	0.282	0.042	0.027
7.000	0.362	0.052	0.000	0.319	0.033	0.062
9.000	0.232	0.025	0.000	0.193	0.016	0.051
11.000	0.094	0.007	0.000	0.074	0.004	0.025

```
((GOAL 0.0036787947 0.122201666 0.3123068 0.36194223 0.2319162 0.09367731)  
(VISUAL 3.0279756e-4 0.027291834 0.06276049 0.0515962 0.02546558 0.0073800245)  
(RETRIEVAL 0.0 9.128459e-6 1.4167614e-4 2.0907508e-4 1.407141e-4 6.0042497e-5)  
(VOCAL 2.386771e-4 0.066692315 0.28215575 0.31926388 0.19250253 0.07392211)  
(IMAGINAL 1.9477647e-4 0.019534493 0.04173144 0.033012565 0.015866699 0.004363207)  
(MANUAL 0.0 1.730284e-4 0.026856279 0.06210476 0.050781336 0.024944454))
```

```
> (predict-bold-response 4)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
5.000	0.312	0.063	0.000	0.282	0.042	0.027
7.000	0.362	0.052	0.000	0.319	0.033	0.062
9.000	0.232	0.025	0.000	0.193	0.016	0.051
11.000	0.094	0.007	0.000	0.074	0.004	0.025

```
((GOAL 0.3123068 0.36194223 0.2319162 0.09367731) (VISUAL 0.06276049 0.0515962 0.02546558  
0.0073800245)  
(RETRIEVAL 1.4167614e-4 2.0907508e-4 1.407141e-4 6.0042497e-5)  
(VOCAL 0.28215575 0.31926388 0.19250253 0.07392211)  
(IMAGINAL 0.04173144 0.033012565 0.015866699 0.004363207)  
(MANUAL 0.026856279 0.06210476 0.050781336 0.024944454))
```

```
> (predict-bold-response 0 6.5)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
1.000	0.004	0.000	0.000	0.000	0.000	0.000
3.000	0.122	0.027	0.000	0.067	0.020	0.000
5.000	0.312	0.063	0.000	0.282	0.042	0.027

```
((GOAL 0.0036787947 0.122201666 0.3123068) (VISUAL 3.0279756e-4 0.027291834 0.06276049)  
(RETRIEVAL 0.0 9.128459e-6 1.4167614e-4) (VOCAL 2.386771e-4 0.066692315 0.28215575)  
(IMAGINAL 1.9477647e-4 0.019534493 0.04173144) (MANUAL 0.0 1.730284e-4 0.026856279))
```

```
> (predict-bold-response 12 16)
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
13.000	0.011	-0.002	0.000	0.007	-0.001	0.007
15.000	-0.025	-0.005	-0.000	-0.023	-0.004	-0.002

```
((GOAL 0.011476353 -0.025498454) (VISUAL -0.0017629415 -0.0052753445)
(RETRIEVAL 1.15529265e-5 -1.1248277e-5) (VOCAL 0.006810909 -0.02294569)
(IMAGINAL -0.001374807 -0.0035236396) (MANUAL 0.0071796663 -0.0017784594))
```

```
E> (predict-bold-response 3.5)
```

```
#!/warning: Start time should be a multiple of :bold-inc (2). Using start time of 2. |#
```

TIME	GOAL	VISUAL	RETRIEVAL	VOCAL	IMAGINAL	MANUAL
3.000	0.122	0.027	0.000	0.067	0.020	0.000
5.000	0.312	0.063	0.000	0.282	0.042	0.027
7.000	0.362	0.052	0.000	0.319	0.033	0.062
9.000	0.232	0.025	0.000	0.193	0.016	0.051
11.000	0.094	0.007	0.000	0.074	0.004	0.025

```
((GOAL 0.122201666 0.3123068 0.36194223 0.2319162 0.09367731)
(VISUAL 0.027291834 0.06276049 0.0515962 0.02546558 0.0073800245)
(RETRIEVAL 9.128459e-6 1.4167614e-4 2.0907508e-4 1.407141e-4 6.0042497e-5)
(VOCAL 0.066692315 0.28215575 0.31926388 0.19250253 0.07392211)
(IMAGINAL 0.019534493 0.04173144 0.033012565 0.015866699 0.004363207)
(MANUAL 1.730284e-4 0.026856279 0.06210476 0.050781336 0.024944454))
```

```
E> (predict-bold-response 12 13)
```

```
#!/warning: Sample time too short for BOLD predictions - must be at least :bold-inc seconds
(currently 2) |#
```

```
NIL
```

```
E> (predict-bold-response :bad-value)
```

```
#!/warning: Predict-bold-response requires start and end times to be numbers, but given
:BAD-VALUE,12.535. |#
```

```
NIL
```

```
E> (predict-bold-response)
```

```
#!/warning: Predict-bold-response requires a current model. |#
```

```
NIL
```

```
E> (predict-bold-response)
```

```
#!/warning: Predict-bold-response requires a current meta-process. |#
```

```
NIL
```

Adding new production compilation types

Not yet documented.

References

- Anderson, J. R. (2007) *How Can the Human Mind Occur in the Physical Universe?* New York: Oxford University Press.
- Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y . (2004). An integrated theory of the mind. *Psychological Review* 111, (4). 1036-1060.
- Anderson, J.R., Fincham, J. M., Qin, Y., & Stocco, A. (2008). A central circuit of the mind. *Trends in Cognitive Science*. 12(4), 136-143.
- Anderson, J. R. & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Erlbaum.
- Briggs, R., & Hoyer, D. J. (1975). A new distinctive feature theory for upper case letters. *Journal of General Psychology*, 93, 87-93.
- Friston, K. J., Ashburner, J. T., Kiebel, S. J., Nichols, T. E., & Penny, W. D. (Eds.). (2011). *Statistical parametric mapping: The analysis of functional brain images: The analysis of functional brain images*. Academic Press
- Gibson, E. J. (1969). *Principles of perceptual learning and development*. New York: Appleton-Century-Crofts.
- Kieras, D. E., & Meyer, D. E. (1996). *The EPIC architecture: Principles of operation*. Retrieved June 3, 2008, from University of Michigan, Department of Electrical Engineering and Computer Science Web site: <ftp://www.eecs.umich.edu/people/kieras/EPIC/EPICPrinOp.pdf>
- Matsumoto, M. & Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1*. 3-30.
- McClelland, J. L., & Rumelhart, D. E. (1981). An interactive model of context effects in letter perception: I. An account of basic findings. *Psychological Review*, 88, 375-407.
- Pylyshyn, Z. (1999). Is vision continuous with cognition? The case for cognitive impenetrability of visual perception. *Behavioral and Brain Sciences*, 22, 341–423.

Salvucci, D. D. (2001). An integrated model of eye movements and visual encoding. *Cognitive Systems Research*, 1, 201-220.

Taatgen, N. A., Rijn, H. v., & Anderson, J. R. (2007). An Integrated Theory of Prospective Time Interval Estimation: The Role of Cognition, Attention and Learning. *Psychological Review*, 114(3), 577-598.

Tresiman, A.M., & Gelade, G. (1980). A feature-integration theory of attention. *Cognitive Psychology*, 1, 97-136.