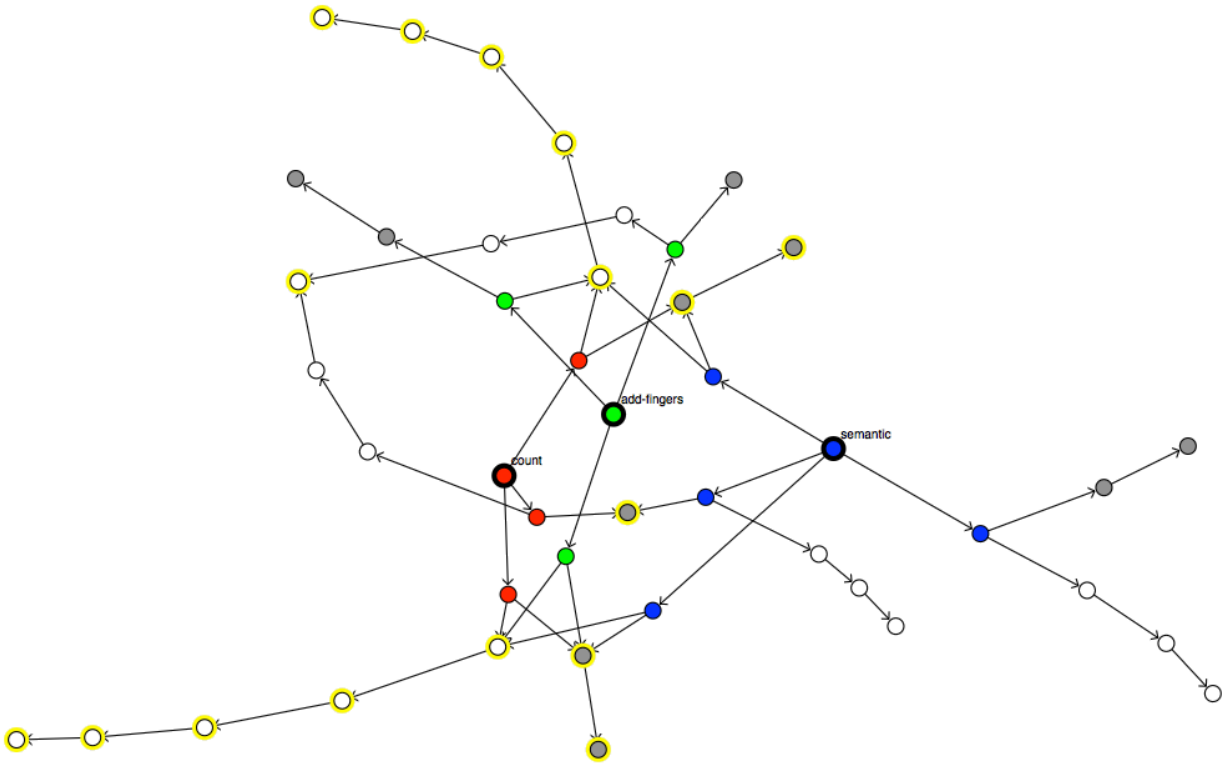
PRIMS TUTORIAL



Niels Taatgen
July 2015

Funded by ERC StG
283597 MULTITASK

UNIT 1

Introduction to PRIMs

This document discusses a new implementation of the ACTRansfer/PRIMs cognitive architecture, which is implemented in Swift.

The structure of a PRIMs model

A PRIMs model consists of several components. It contains the name of the task with some parameters, the specification of a set of operators to perform the task, possibly additional facts necessary to carry out the task, and a specification of the task itself.

The example we will use consists of two models from the standard unit 1 of the ACT-R tutorial, count and semantic, but now translated into PRIMs. The operators in PRIMs model correspond one-to-one with productions in the unit 1 models.

Let us look at the file `count.prims`. It starts by specifying what task we are going to carry out:

```
define task count {
  initial-goals: (count) // In this example the task and the goal are the same
  start-screen: start
  imaginal-autoclear: nil
  default-activation: 1.0
  ol: t
  rt: -2.0
  lf: 0.2
  default-operator-self-assoc: 0.0
}
```

In PRIMs, a task can be implemented by several goals, but the count task is only implemented by one goal, also named count. The `initial-goals: (count)` line specifies this. The input of the model, which we will look at in more detail later, is given in terms of screens, and the screen that we will use here is called `start` (`initial-screen: start`). We will see a specification of "start" further down the file.

The remaining lines in the task definition are model parameters, some of them are straight from ACT-R (`ol:` optimized learning, `rt:` retrieval threshold, `lf:` latency factor).

The next part of the model defines operators to carry out the task. Operators are organized within goals. We have only one goal, the count goal, so we define it as:

```
define goal count {
… Operator definitions …
}
```

The result of this organization is that all the operators within a goal definition will be associated with that goal, that is, whenever a goal is in one of the goal buffer slots, the associated operators receive spreading activation and are therefore more likely to be retrieved.

We then see definitions of the three operators that are needed to count. The first is:

```
operator start-count {
"Start counting"
 V1 <> nil  // There has to be a visual input with the starting number
 WM1 = nil  // Imaginal should be empty
==>
 V1 -> WM1 // Copy the start number to working memory
 count-fact -> RT1 // Start retrieving the next number
 V1 -> RT2
 say -> AC1 // Say the current number
 V1 -> AC2
 }
```

As the name implies, this operator initiates counting. Each operator consists of one or more condition PRIMs, and one or more action PRIMs, each on a separate line. PRIMs always refer to two particular slots in two particular buffers (or one slot in one buffer and nil). In between is either a comparison (= or <>), or a copy operation (–>). An arrow ==> separates the conditions and actions, and anything after // is ignored.

The letters indicate the buffer, and the number the slot number within that buffer. In this example, we will use the following buffers:

- V: Visual, or input buffer. This buffer contains the visual input. In this particular model, V1 has the starting number for counting, and V2 the ending number
- WM: Working memory or imaginal buffer. This buffer is used to store intermediate results. Here we only use one slot to represent the current count.
- RT: Retrieval. Used to retrieve items from declarative memory. In this model it is used to retrieve count-facts from memory.
- AC: Action. Used specify actions the model takes. This model will use it to "say" numbers.
- G: Goal. Goal slots are used to activate operators that are relevant for the current goal. Because of our initial-goals declaration, `count` will be put into `G1`.

- C: Constant. This is not actually a buffer, and it also doesn't show in the operator definition. However, each time a PRIM in an operator has a constant in it, that is, a specification that is not a buffer/slotnumber combination (and not `nil`), that constant is put in one of the slots of the operator chunk. In our example, `count-fact` and `say` are both constants that will be put into `C1` and `C2`, respectively.

If an operator is selected, its conditions are checked first. In this case, a check is made whether `V1` does not contain `nil`, or, in other words, we need a value in `V1`. The second check is to see whether `WM1` is still `nil`, indicating we haven't started counting yet.

If these conditions are satisfied, the model will start carrying out its actions. Typically, actions involve updating WM slots and specifying actions that are carried out by the modules. In this example, we first store the input into working memory in order to maintain a counter: `V1 -> WM1`. We then specify a retrieval request, which consists of two PRIMs: `count-fact -> RT1` specifies `slot1` of the retrieval, and `V1 -> RT2` `slot2` of the retrieval. In other words, we want a count-fact about the starting number. Furthermore, we want to say that number, which is specified by `say -> AC1` and `V1 -> AC2`.

The PRIMs parser translates an operator specification into a real ACT-R chunk, which will look like:

```
start-count
  isa  operator
  slot1  count-fact
  slot2  say
  condition  V1<>nil;WM1=nil
  action  V1->WM1;C1->RT1;V1->RT2;C2->AC1;V1->AC2
```

The chunks in the condition and action slots represent the first in a list of PRIMs. `V1->WM1;C1->RT1;V1->RT2;C2->AC1;V1->AC2` is a PRIM that carries out `V1->WM1` and points to `C1->RT1;V1->RT2;C2->AC1;V1->AC2`, which in turn carries out `C1->RT1`, and points to `V1->RT2;C2->AC1;V1->AC2`, etcetera.

The second and third operator are as follows:

```
  operator iterate {
    "Iterate as long as count isn't done"
    RT2 = WM1
    V2 <> WM1
   ==>
    RT3 -> WM1
    count-fact -> RT1
    RT3 -> RT2
    say -> AC1
```

```
    RT3 ->AC2
  }

  operator final {
    "Stop when reaching final number"
    V2 = WM1
   ==>
    say -> AC1
    stop -> AC2
    stop -> G1
  }
```

They do the rest of the counting: the iterate operator iterates as long as the final number has not been reached (V2 <> WM1), and the third operator terminates the count. By putting `stop` in `G1` we signal the simulation that we have reached the end.

The next definition in the file defines facts that will be put into declarative memory. A fact definition consists of lists of values enclosed in parentheses. Each of these lists is translated into a chunk

```
define facts {
(cf1 count-fact  one  two)
(cf2 count-fact  two  three)
(cf3  count-fact  three  four)
(cf4  count-fact  four  five)
(cf5  count-fact  five  six)
}
```

The first item in the list will become the name of the new chunk, while the remaining items will be put into slots. For example, `(cf1 count-fact one two)` will be translated into the following chunk:

```
cf1
  isa  fact
  slot1  count-fact
  slot2  one
  slot3  two
```

The remainder of the file contains the task specification. This is not really part of the model itself, but specifies the environment that it has to interact with. In the case of counting, this is pretty simple: we need a starting and an ending number. The following definition gives a framework for that input:

```
define screen start {
(?0 ?1)
}
```

It specifies a screen with two items on it, `?0` and `?1`. We have already specified that "start" will be the start-screen (in the task definition). `?0` and `?1` are placeholders for the real items:

```
define inputs {
(two four)
(one three)
(three five)
}
```

Whenever you run the model, the simulation will pick one of the three inputs. The final declaration in the model is the ending action if the model performed correctly:

```
define goal-action {
(say stop)
}
```

Although in this example there isn't really a check whether the model has counted correctly, we will use this later when we are providing the model feedback about its performance.

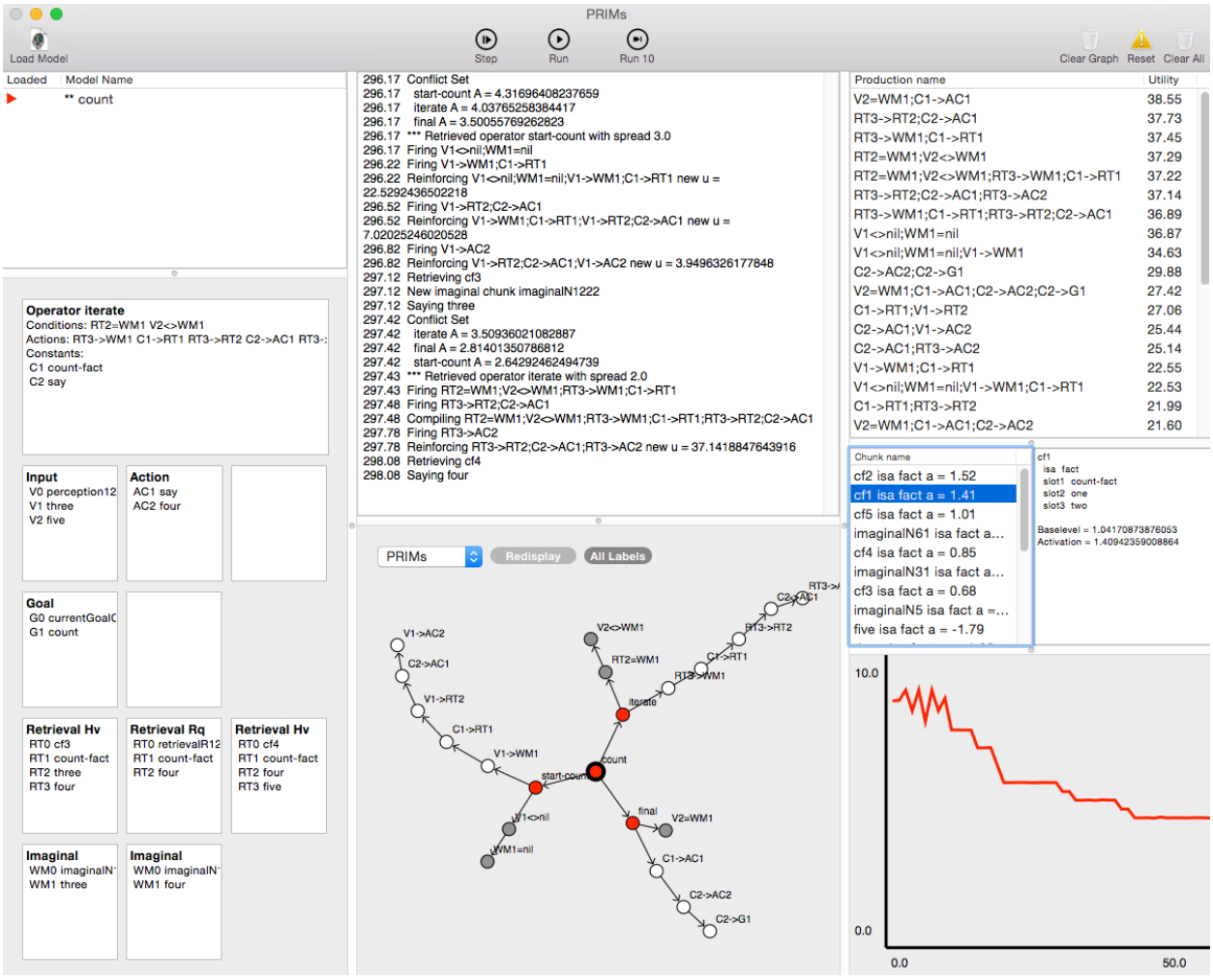
Running a model

To run the count model, start the PRIMs application, and use the "Load Model" button or "Open..." menu option to load the file. Select `count.prims`, after which the top-middle panel will show some information about what is loaded. If there is an error in the model, you will find it in that panel (unless the program crashed first.... Not everything is fool-proof yet).

Once the model has loaded, you can run it with the three buttons in the menu bar (Step, Run and Run 10). Various other parts of the window can help you make sense of what is going on in the model. Let us have a look after the model has run a couple of times (Figure 1).

This model window has the following panels:

- The top-left panel shows a list of tasks that the system currently knows, and are proceeded by a colored triangle if the associated models is actually loaded. The color of the triangle also identifies the task in some of the other panels. The task with ** is the currently active task, clicking a different task activates it.
- The bottom-left panel shows the contents of the different buffers when the model is running. Shown are the currently selected operator, the contents of several buffers before the operator is executed (left column), how buffers are changed by the operator (middle column), and how buffers are changed by module actions (right column).



- The top-center panel typically shows the model trace. We will discuss this in a little more detail in a moment.
- The bottom-center panel shows a graph of the PRIMs. If multiple tasks are loaded, operators will show different colors, and PRIMs that are used by multiple tasks will receive a yellow halo. The popup menu can be used to make graphs of different aspects of the models.
- The top-right panel shows learned production rules with their utilities. Production rules are always combinations of PRIMs.
- The middle-right panel shows all the chunks in declarative memory with their activation. If you click on one, it will show in more detail on the right.
- Finally, the bottom-right panel shows a graph of model results. In the example, the model has run about 50 times, and has improved its speed considerably. Multiple graphs can show after running multiple models.

- The buttons on the top-right of the window do the following: Clear Graph clears the graph on the bottom-right. Reset removes all models from memory and reloads the one that is selected on the top-left (Note: it doesn't actually reload it from the file, just from a stored representation). Clear All, finally, removes everything.

Once you have loaded the model, you can step through it with the Step button, run it all the way with the Run button, or run it 10 times with the Run 10 button (no trace will show). The trace will detail what is going on. Let the Count run model once, after which the trace will show something like:

```
0.00  *** Retrieved operator start-count with spread 3.0
0.00  Firing V1<>nil
0.05  Firing WM1=nil
0.35  Firing V1->WM1
0.65  Firing C1->RT1
0.95  Firing V1->RT2
1.25  Firing C2->AC1
1.55  Firing V1->AC2
1.85  Retrieving cf1 (latency = 0.0605669022304275)
1.85  New imaginal chunk imaginalN5  (latency = 0.2)
1.85  Saying one (latency = 0.3)
      2.16  *** Retrieved operator start-count with spread 1.5
      2.16  Firing V1<>nil
      2.21  Firing WM1=nil
      2.51  Operator start-count9 failed
2.52  *** Retrieved operator iterate with spread 2.0
2.52  Firing RT2=WM1
2.57  Firing V2<>WM1
2.87  Firing RT3->WM1
3.17  Firing C1->RT1
3.47  Firing RT3->RT2
3.77  Firing C2->AC1
4.07  Firing RT3->AC2
4.37  Retrieving cf2 (latency = 0.0592830290521636)
4.37  Saying two (latency = 0.3)
      4.67  *** Retrieved operator start-count with spread 2.0
      4.67  Firing V1<>nil
      4.72  Firing WM1=nil
      5.02  Operator start-count17 failed
5.03  *** Retrieved operator iterate with spread 2.0
5.03  Firing RT2=WM1
5.08  Firing V2<>WM1
5.38  Firing RT3->WM1
5.68  Firing C1->RT1
5.98  Firing RT3->RT2
6.28  Firing C2->AC1
6.58  Firing RT3->AC2
6.88  Retrieving cf3 (latency = 0.0570473510241556)
6.88  Saying three (latency = 0.3)
7.19  *** Retrieved operator final with spread 2.0
7.19  Firing V2=WM1
7.24  Firing C1->AC1
7.54  Firing C2->AC2
7.84  Firing C2->G1
8.14  Saying stop (latency = 0.3)
```

The number in the left column represents time. We can see that at time 0, the model retrieves the start-count operator. It will then start carrying out that operator by firing a production rule for each individual PRIM. After the last PRIM has been carried out, modules do their actions in parallel. In this case, three modules become active: the retrieval module retrieves `cf2`, the action module says two, and the imaginal (working memory) module makes a new chunk to store the count. They each have their own latency, but the longest counts (in this case 0.3 seconds for saying one). Note that anything in the trace that is indented belongs to an operator that failed, so it has no impact on executing, except for taking up time.

If you go through the model one step at a time, the trace will show some more detail:

```
0.00  Conflict Set
0.00    start-count A = 5.98194492136532
0.00    final A = 5.42805591304186
0.00    iterate A = 4.8559583289925
0.00  *** Retrieved operator start-count with spread 3.0
0.00  Firing V1<>nil
0.05  Firing WM1=nil
0.05  Compiling V1<>nil;WM1=nil
0.35  Firing V1->WM1
0.35  Compiling WM1=nil;V1->WM1
0.65  Firing C1->RT1
0.65  Compiling V1->WM1;C1->RT1
0.95  Firing V1->RT2
0.95  Compiling C1->RT1;V1->RT2
1.25  Firing C2->AC1
1.25  Compiling V1->RT2;C2->AC1
1.55  Firing V1->AC2
1.55  Compiling C2->AC1;V1->AC2
1.85  Retrieving cf3 (latency = 0.0411039719523428)
1.85  New imaginal chunk imaginalN5  (latency = 0.2)
1.85  Saying three (latency = 0.3)
```

For each retrieved operator it will show all the competing operators with their activations. The trace also shows production compilation in action. After V1<>nil and WM1=nil have been checked, a rule is learned that makes both comparisons in one step. This rule will need to be reinforced a couple of times before it can compete (this also shows up in the trace if you step it). The compilation process follows the standard rules of ACT-R: you can set the learning speed with the `alpha` parameter.

Modeling Transfer

One of the goals of PRIMs is to model transfer. We therefore need to specify at least one additional model. The example is the Semantic model from unit 1. The goal of the semantic

model is to judge relationships between animals and animal categories, and answer questions like “Is a canary an animal”? The facts the model uses are:

```
define facts {
    (sem1  property  lion  mammal)
    (sem2  property  mammal  animal)
    (sem3  property  animal living-thing)
    (sem4  property  plant living-thing)
    (sem5  property  tulip plant)
    (sem6  property  bird animal)
    (sem7  property  tweety bird)
    (sem8  property  robin bird)
}
```

Answering the question involves two steps: first to retrieve that a canary is a bird, and then that a bird is an animal. Even though this model is semantically different from count, it shares the same type of iteration. The operators in the model are therefore similar:

```
define goal semantic {
    operator start-semantic {
    "Start semantic reasoning"
        V1 <> nil
        WM1 = nil
        ==>
        V1 -> WM1
        property -> RT1
        V1 -> RT2
        subvocalize -> AC1
        V1 -> AC2
    }

    operator move-up-tree {
    "Move up the tree"
        RT2 = WM1
        V2 <> WM1
        ==>
        RT3 -> WM1
        property -> RT1
        RT3 -> RT2
        subvocalize -> AC1
        RT3 -> AC2
    }

    operator say-yes {
    "Say yes when found"
        V2 = WM1
        ==>
        say -> AC1
        yes -> AC2
        stop -> G1
    }

    operator say-no {
    "Say no on retrieval failure"
```

```
        V2 <> WM1
        RT1 = error
        ==>
        say -> AC1
        no -> AC2
        stop -> G1
    }
}
```

The first two operators in this model are the same as in the counting model, with the exception that the slot labels are different. But once the operators are translated into declarative memory, the conditions and actions are identical. The last two operators are different. If a match between the target category and the retrieved category is found, the model should answer "yes", which is slightly different from finalizing the count. Also, if the proposition does not hold, the model will hit a retrieval error at some point. On a retrieval error, the first slot of the retrieval buffer will be set to error (which is, in the example, matched by (RT1 = error)).

Here is an example of a run of the model:

```
0.00  *** Retrieved operator start-semantic with spread 3.0
0.00  Firing V1<>nil
0.05  Firing WM1=nil
0.35  Firing V1->WM1
0.65  Firing C1->RT1
0.95  Firing V1->RT2
1.25  Firing C2->AC1
1.55  Firing V1->AC2
1.85  Retrieving sem1 (latency = 0.0458280985167914)
1.85  New imaginal chunk imaginalN6  (latency = 0.2)
1.85  Subvocalizing lion (latency = 0.3)
2.15  *** Retrieved operator move-up-tree with spread 2.0
2.15  Firing RT2=WM1
2.20  Firing V2<>WM1
2.50  Firing RT3->WM1
2.80  Firing C1->RT1
3.10  Firing RT3->RT2
3.40  Firing C2->AC1
3.70  Firing RT3->AC2
4.00  Retrieving sem2 (latency = 0.0750171372159093)
4.00  Subvocalizing mammal (latency = 0.3)
      4.31  *** Retrieved operator start-semantic with spread 2.0
      4.31  Firing V1<>nil
      4.36  Firing WM1=nil
      4.66  Operator start-semantic16 failed
      4.67  *** Retrieved operator say-no with spread 2.0
      4.67  Firing V2<>WM1
      4.72  Firing RT1=C1
      5.02  Operator say-no19 failed
5.03  *** Retrieved operator move-up-tree with spread 2.0
5.03  Firing RT2=WM1
5.08  Firing V2<>WM1
5.38  Firing RT3->WM1
```

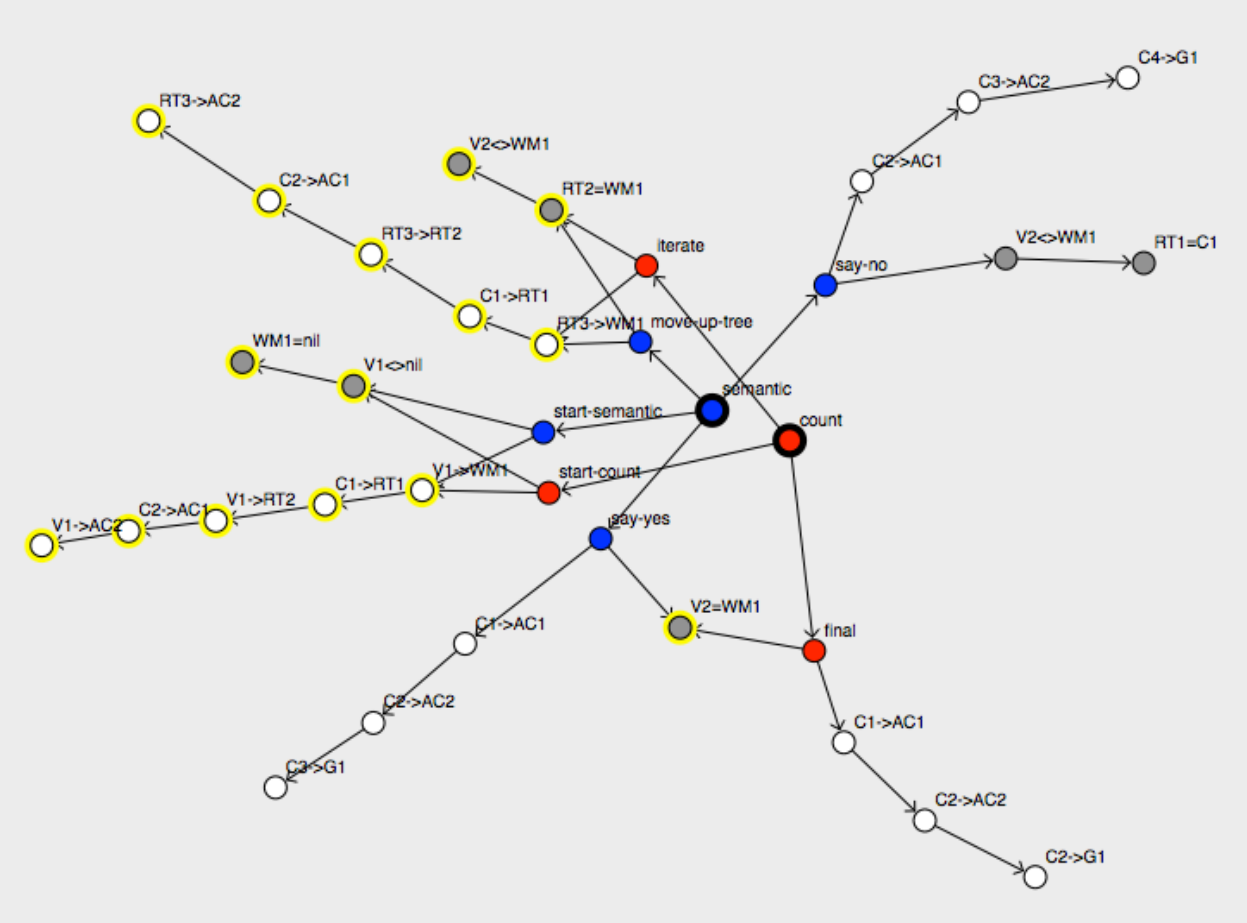
```
5.68  Firing C1->RT1
5.98  Firing RT3->RT2
6.28  Firing C2->AC1
6.58  Firing RT3->AC2
6.88  Retrieving sem3 (latency = 0.0460701809949314)
6.88  Subvocalizing animal (latency = 0.3)
      7.19  *** Retrieved operator start-semantic with spread 2.0
      7.19  Firing V1<>nil
      7.24  Firing WM1=nil
      7.54  Operator start-semantic28 failed
      7.55  *** Retrieved operator say-no with spread 2.0
      7.55  Firing V2<>WM1
      7.60  Firing RT1=C1
      7.90  Operator say-no30 failed
7.90  *** Retrieved operator move-up-tree with spread 2.0
7.90  Firing RT2=WM1
7.95  Firing V2<>WM1
8.25  Firing RT3->WM1
8.55  Firing C1->RT1
8.85  Firing RT3->RT2
9.15  Firing C2->AC1
9.45  Firing RT3->AC2
9.75  Retrieval failure
9.75  Subvocalizing living-thing (latency = 0.3)
11.24  *** Retrieved operator say-yes with spread 2.0
11.24  Firing V2=WM1
11.29  Firing C1->AC1
11.59  Firing C2->AC2
11.89  Firing C3->G1
12.19  Saying yes (latency = 0.3)
```

However, if you first run the count model for a while, productions have been compiled that can be reused:

```
1108.01  *** Retrieved operator start-semantic with spread 3.0
1108.01  Firing V1<>nil;WM1=nil;V1->WM1;C1->RT1
1108.06  Firing V1->RT2;C2->AC1;V1->AC2
1108.36  Retrieving sem2 (latency = 0.0557285933220927)
1108.36  New imaginal chunk imaginalN12151  (latency = 0.2)
1108.36  Subvocalizing mammal (latency = 0.3)
      1108.67  *** Retrieved operator say-no with spread 2.0
      1108.67  Firing V2<>WM1
      1108.72  Firing RT1=C1
      1109.02  Operator say-no12156 failed
1109.03  *** Retrieved operator move-up-tree with spread 2.0
1109.03  Firing RT2=WM1;V2<>WM1;RT3->WM1;C1->RT1;RT3->RT2;C2->AC1;RT3->AC2
1109.08  Retrieving sem3 (latency = 0.0922408709785375)
1109.08  Subvocalizing animal (latency = 0.3)
      1109.39  *** Retrieved operator say-no with spread 2.0
      1109.39  Firing V2<>WM1
      1109.44  Firing RT1=C1
      1109.74  Operator say-no12166 failed
1109.75  *** Retrieved operator move-up-tree with spread 2.0
1109.75  Firing RT2=WM1;V2<>WM1;RT3->WM1;C1->RT1;RT3->RT2;C2->AC1;RT3->AC2
1109.80  Retrieval failure
1109.80  Subvocalizing living-thing (latency = 0.3)
1111.29  *** Retrieved operator say-yes with spread 2.0
```

```
1111.29  Firing V2=WM1
1111.34  Firing C1->AC1
1111.64  Firing C2->AC2
1111.94  Firing C3->G1
1112.24  Saying yes (latency = 0.3)
```

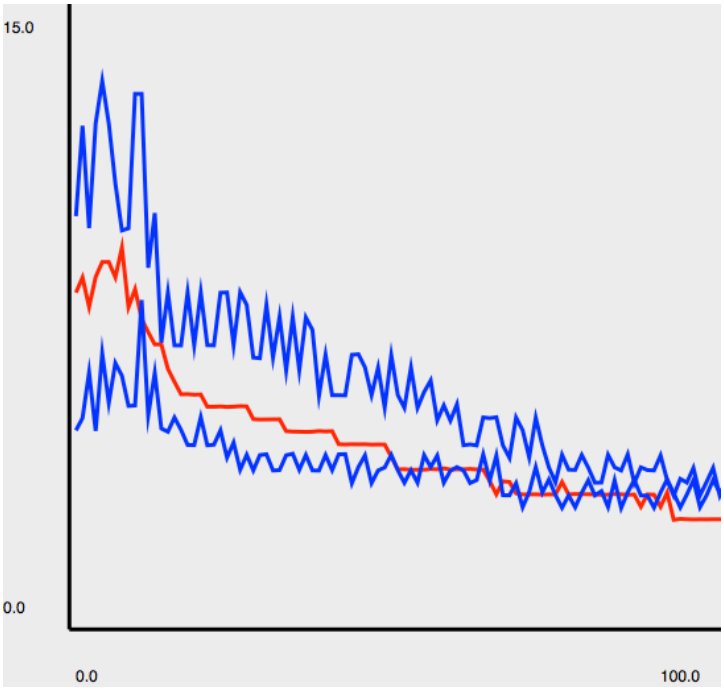
How do we assess transfer between these two models? A first option is to look at the overlap of chunks in declarative memory between the two models. The figure below gives an impression of this overlap (all the yellow halos), which is considerable.



To get a real sense of the amount of transfer, we have to run the model. We first run the Count model a number of times, and then the Semantic model. Then we run the Semantic model without prior training.

To do this in a simple way, do the following. Load in both the Count and Semantic model. Click on the count model in the top-left panel to activate it. Click the "Run 10" button ten times to run the model for 100 times. You will see the learning curve in the right bottom corner. Now activate semantic, and run it 100 times. The new curve will show semantic after

learning count. Now push the Reset button, and run the semantic model again for 100 times. Your graph should look something like:



The top blue curve is semantic without transfer from count, and the bottom curve is the one with transfer.

We can also collect this type of data by automatizing this. Choose the "Run Batch..." option from the Run menu, and select the "testbatch.bprims" as input in the file dialog. Then choose a filename of your choice as output. The system will now run what we just did by hand ten times. It will also generate a datafile that we can analyze with R, Excel or any other packages that takes data tables.

Perception and Action

PRIMs uses a simplified version ACT-R's perception and motor modules. This means that some of the precision is lost, but on the other hand that it is easier to program the experiment part of the model.

For reasonably straight-forward experiments, PRIMs supplies some methods to specify the interaction between the model and the outside world. In the count and semantic models, we have specified a single screen with two pieces of information on it. However, we can specify several screens and conditions under which the model will change screens. Moreover, a screen can have a hierarchy of items on it that the model can traverse. We will look at the latter aspect in Unit 2.

For the current assignment, we are going to build two models of addition by counting. The first one will be done through mental operations, but the second model assumes that one of the two counters that has to be maintained will use the fingers. The assumption of the model is that each time we say a number, we also stick up an additional finger, and the total

number of fingers is available to perception. To model this we need the following specification:

```
define screen start {
(?0 ?1)
}

define screen one-finger {
(?0 ?1 one)
}


define screen two-finger {
(?0 ?1 two)
}


define screen three-finger {
(?0 ?1 three)
}

define screen four-finger {
(?0 ?1 four)
}

transition(start,one-finger) = action(say)
transition(one-finger,two-finger) = action(say)
transition(two-finger,three-finger) = action(say)
transition(three-finger,four-finger) = action(say)
```

In this example, the start screen just has the two numbers that have to be added. The four others screens also have that information, augmented by a number of fingers. The transition specifications determine how one screen is replaced by another screen: by a `say` action. So the first time the model says something, a first finger is put up, etcetera.

Assignment

The assignment is to add a model of addition by counting to the existing two models, and to assess transfer between counting and addition.

There are two possible solutions to explore. The first is a direct translation of the unit 1 model into PRIMs. The two numbers to be added are perceptual input, and two `WM` slots are needed to hold the current count and the current sum.

Your first operator has to initialize the process by putting the first addend (which is in `V1`) into `WM1`, and zero in `WM2`. It then has to make a retrieval request for the a count-fact about the number in `V1`.

The second operator harvests a retrieval count-fact that matches `WM1`, updates the number in `WM1`, and issues a count-fact retrieval request for `WM2`.

A third operator harvest a count-fact retrieval that matches `WM2`, checks whether `WM2` is not equal to the second addend (in `V2`), and issues a count-fact retrieval for a count-fact about `WM1`.

A fourth operator checks whether the count in `WM2` equals `V2`, and gives the contents of `WM1` as an answer.

One important parameter to set is imaginal-autoclear, which should be set to nil (just as in the original count model). We'll explain why in unit 2.

Implement this model, and check whether it works. You should also specify what the start screen looks like, and supply it with a few inputs and count-facts (you can take these from the count model, don't forget to add a count-fact about zero).

Once you have made sure the model works, you can check how fast it learns. You can then also see how much transfer there is from the count model, using the same method as with semantic. If you want to use the method with the batch file, you should edit the `.bprims` file so that it uses your new model instead of semantic (hopefully the structure of the batch file speaks for itself).

You will probably conclude that transfer between counting and this addition model is limited. You can therefore try to build an alternative model using fingers. This model only requires three operators, and should show more transfer than the "standard" model.

PRIMs parameters

imaginal-delay:

Time it takes to put a new chunk in the imaginal buffer (default 0.2)

egs:

Utility noise (default 0.05)

alpha:

Learning parameter for production compilation (default 0.1)

nu:

Utility for newly compiled productions (default 0.0)

primU:

Utility of productions that handle a single PRIM (default 2.0)

dat:

Default time to fire the first production to handle an operator (default 0.05)

production-prim-latency:

Default time to fire subsequent productions to handle an operator. The idea is that this takes longer because it also involves retrieving a chunk from memory. The current implementation doesn't actually do this, but the time this would take should be accounted for. The consequence is that a fully proceduralized operator only takes de default action time (dat:) to carry out, but if multiple productions have to fire it takes substantially longer. (default 0.3)

bll:

Base-level decay (default 0.5)

ol:

Optimized learning (default t). Set to nil for the standard equation.

mas:

Maximum associative strength (default 3.0)

rt:

Retrieval Threshold (default -2.0)

lf:

Latency Factor (default 0.2)

mp:

Mismatch Penalty (default 5.0). Note that the current version of PRIMs doesn't yet have an option to switch on partial matching.

ans:

Activation noise (default 0.2)

ga:

Spreading activation from the goal (default 1.0)

input-activation:

Spreading activation from the input (default 0.0)

default-activation:

In most models, the chunks you specify as part of the model can be assumed to exist for a while, so they probably have reasonable stable activation values. When you give a value to default-activation, that value becomes the lower-bound of baselevel activation for all chunks you specify in the model (including operators). There is no default for this parameter, because when you omit it there is no lower-bound, and any chunk you specify will have a single reference.

default-operator-assoc:

Sji between an operator and the goal it is defined in. This ensures that operators relevant to one of the goals in the goal buffer are more likely to be retrieved instead of other operators. (default 4.0)

goal-chunk-spreads:

Normally, the amount of spreading from the goal is equal to the ga parameter divided by the number of chunks is the goal. If you set this parameter to t, the amount of spreading will be equal to the activation of the goal, allowing you to give goals more or less priority. (default nil)

default-inter-operator-assoc:

Sji between an operator and other operators for the same goal. Make it more likely that an operator that is associated to the same goal as the previous operator is selected. (default 1.0)

default-operator-self-assoc:

Sji between an operator and itself. Should be negative to make it less likely that an operator will fire repeatedly. (default -1.0)

perception-action-latency:
: Default time for any action (default 0.2)

say-latency:
: Latency of a say action (default 0.3)

subvocalize-latency:
: Latency of a subvocalize action (default 0.3)

read-latency:
: Time of a read action (default 0.2)

imaginal-autoclear:
: When set to t (true), a new chunk will be made in the imaginal buffer whenever an operator puts something in a slot of the imaginal buffer in which there is already a value. The old chunk is moved to declarative memory. When set to nil, slot values are simply overwritten. (default: t)

goal-operator-learning:
: Experimental mechanism for a task to find its own operators. When set to t (true), the mechanism will be active. What the mechanism does is try to learn associations between the goal in G1 and operators it retrieves using reinforcement learning. Whenever the model successfully completes a task, all the operators responsible are updated. The next three parameters also need to be specified. (default: nil)

beta:
: Learning speed for goal-operator learning (similar to alpha). (default: 0.1)

reward:
: Reward used in goal-operator learning. The reward also maximizes the time that the model will try to reach the goal (default is 0.0, which switches it off, so if you want to use it you have to give it a sensible value, i.e., something slightly longer than the time necessary to reach the goal).

explore-exploit:
: Goal-operator learning adds extra noise to goal-operator combinations it hasn't tried very often yet. This parameter scales how fast noise on the goal-operator association is reduced with more experience. A higher value corresponds to a longer period of exploration (default 0.0). Still very experimental, so off by default.

UNIT 2

Multiple goals, building memory, more complex input

Building Memory

Up to now we have been using the imaginal or WM buffer as scratchpad with arbitrary slots to store stuff in. However, it is more interesting to see the imaginal buffer as the place where new memory structures are created for storage in long-term declarative memory. PRIMs is therefore a bit more frugal in adding things to DM: only chunks removed from the imaginal buffer end up in DM, instead of chunks from any buffer (most of those chunks are not particularly useful for future purposes anyway).

In order to facilitate but also constrain the building of memory structures, PRIMs automatically creates a new chunk in the imaginal buffer whenever you try to overwrite a slot in that buffer. If you don't want this to happen, set `imaginal-autoclear` to nil. In order to link chunks together you will probably of put a direct reference to another chunk in a slot. To make this possible in PRIMs, each chunk has a virtual "`slot0`", which refers to the name of the chunk itself. We will need this to gradually build up large chunk structures, as we will see in the example of list learning. The list learning model also illustrates a second aspect of PRIMs: multiple goals.

Multiple parallel goals

We encounter new tasks every day, many of which we can carry out without instruction or training. The probable reason is that these new tasks are combinations of things that we already know how to do. Up to now, tasks have corresponded to a single goal. However, a task may consist of several goals, possibly parallel or with some imposed control structure.

In the earlier examples, the goal was stored in slot `G1`. However, we can also store goals in subsequent slots `G2` and up. Each of these goals can be associated with their own set of operators. Each of the goals in slots in the goal buffer spread activation to associated operators (the Sji for this association is `default-operator-assoc`). Operators for a particular goal are also associated with each other (with an Sji of `default-inter-operator-assoc`). This makes sure we normally stay within a goal as long as there are applicable operators. Finally, in most models we don't want the same operator to fire repeatedly, so operators are negatively associated with themselves (by `default-operator-self-assoc`). Operators can put new goals in G slots (e.g., `count-goal->G2`), and can also remove them (by putting nil into them, e.g. `nil->G2`).

To further help reusability of operators, we can define task-specific constants, so that they are no longer part of the operator.

In the list-learning example, we will have four goals: read-in-memory (reads letters and stores them in memory), report-vocal (recalls list from memory and says them), rehearsal (which rehearses the items in the list), and be-lazy (which just waits for something to happen). We further define "letter", "list" and "say" as task-specific constants. In that way, we can change the type of item in the list, or the action that has to be performed on recall without needing new operators. We declare this as follows:

```
define task list-recall {
  initial-goals: (read-in-memory rehearse report-vocal)
  goals: (be-lazy)
  task-constants: (letter list say)
```

Anything in the list after `initial-goals` will be placed in `G1`, `G2`, etc. To make sure they are properly declared, additional goals should be put in a list after `goals`.

Task-specific constants are declared in task-constants. At the start of a run, they are put in slots `GC1`, `GC2`, etc. Let us now see how we can build a list of chunks in declarative memory. The visual input has "letter" in `V1`, and the letter itself in `V2`.

```
define goal read-in-memory {
  operator read-first-letter  {
  "Read the first letter and put it in WM. Store goal id in WM"
   WM1=nil
   V1=letter
   ==>
   V2->WM3
   list->WM1
   G0->WM2
   nil->V1
  }

  operator read-next-letter {
  "Read the next letter, store it in WM"
    WM1<>nil
    V1=letter
    ==>
    V2->WM3
    list->WM1
    WM0->WM2
    nil->V1
  }
}
```

The first of these operators is activated by the first letter in the input. Because letter is actually a task constant, `V1=letter` is translated into `V1=GC1`. Once a letter has been recognized, a chunk is built in the imaginal buffer, with "list" in `slot1`, a reference to the

current goal in slot2 (G0->WM2), and the letter in slot3. The V1 slot is then set to nil to ensure we do not put that letter in the list a second time.

Once we have a first letter, subsequent letters are stored in new chunks in the imaginal, bumping the earlier chunk to DM. The read-next-letter operator has almost the same action as reading the first letter, except that a reference to the previous chunk is put into slot2. WM0->WM2 may look confusing, but what it does is putting a reference of the old list item in slot2 of the new list item.

After reading the list, DM now has a chunk for every item in the list. We now need to recall the list:

```
define goal report-vocal {
  operator start-report {
  "Retrieve the first item and bump anything remaining in WM to DM"
    V1=report
    RT1=nil
    ==>
    G0->RT2
    report->WM1
  }

  operator report-item {
  "Say item and retrieve next"
    V1=report
    ==>
    RT3->AC2
    say->AC1
    RT0->RT2
  }

  operator done-report {
  "On retrieval error end report"
    V1=report
    RT1=error
==>
    stop->G1
  }
}
```

If the input tells us to report the letters, we retrieve the first item by using the goal as a retrieval cue for slot2. Putting something in WM1 ensure the last chunk is cleared out of the imaginal. If an item is retrieved successfully, the report-item operator can then say it, and retrieve the next item in the list. This is accomplished by RT0->RT2: the next item on the list points back to the one we just retrieved, so we need to use the current chunk itself (which is in RT0) as a cue for retrieving the next item (RT2).

One last item to take care of in this model is what happens between the presentation of the letters. Even though we have a rehearsal goal in the specification, the current model does not have any operators for that goal. These can in theory be borrowed from another model.

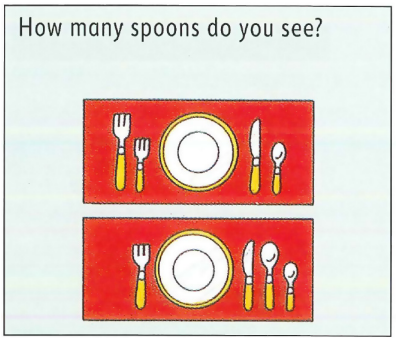
Or we can still add them. Presently, the model will have to fall back on the `rehearse-no-rehearse` operator. This operator will carry out a `wait` action whenever `V1` is equal to nil. The `wait` action is a special action that will do nothing until the next timed screen switch happens (unless there is no next timed switch, in which it will do nothing). The last of the model shows how this timing works:

```
define screen screen1 {
(letter x)
}

define screen screen2 {
(letter k)
}

define screen screen3 {
(letter p)
}

define screen screen4 {
(letter f)
}

define screen report {
(report)
}

transition(screen1,screen2) = absolute-time(2.0)
transition(screen2,screen3) = absolute-time(4.0)
transition(screen3,screen4) = absolute-time(6.0)
transition(screen4,report) = absolute-time(8.0)
```

There are four screens with a letter, followed by a screen with "report". The transitions are specified with respect to the start of the trial. Alternatively, transitions can be specified relative to the moment the screen appears, using `relative-time(`*time*`)`. So we could also have specified `relative-time(2.0)` for each of the transitions.

More complex screens

Up to this point we have only looked at screens that are represented in a single chunk. To allow a slight more complex, but also not overly complex representation, PRIMs allows a more hierarchical representation of structure. Consider the following example:



This picture consists of two placemats, each of which has a number of items, each of which has properties. We can represent this picture as follows:

```
define screen start {
  (placemat one
   (item fork)
   (item fork)
   (item plate)
   (item knife)
   (item spoon))
  (placemat two
   (item fork)
   (item plate)
   (item knife)
   (item spoon)
   (item spoon))
}
```

If this screen is selected, the first top-level item is automatically placed in the input buffer, in this case `(placemat one)`. The rest of the screen can now be traversed using four special actions:

- `focusnext` Move attention to the next item on the current level. A `focusnext` right at the beginning would move attention to placemat two. If there is no next item, the item type of the current level (e.g. `placemat`) is placed in `V1`, and `error` in `V2`.
- `focusdown` Move attention to the first sub-item below the current level. A `focusdown` on `placemat one` would produce `item fork.`
- `focusup` Move attention back up a level.
- `focusfirst` Move attention back to the first item on the current level.

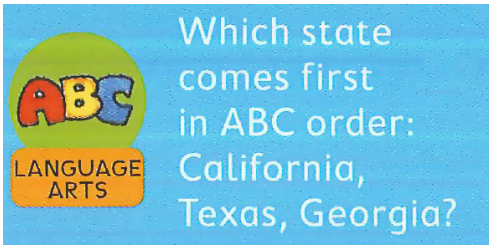
This particular example is worked out in the model count-spoons.prims. Load in the model, and see how it performs. You might conclude you need quite a few operators for such a simple task. However, once we know how to count spoons, we can reuse some of the knowledge for a slightly different task. The more-fish.prims model is a model that looks at which of two acquaria has more red fish. The top-level goal for the task is different, but the subgoal is the same. Therefore there is quite some overlap between the models (both specify the countgoal goal, but they are identical and are merged upon loading).

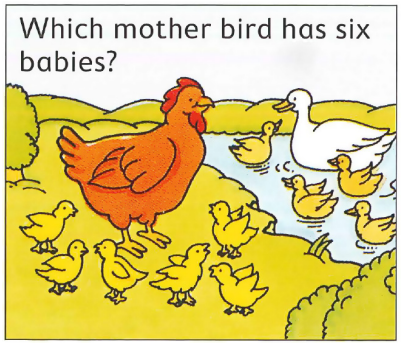
Assignment

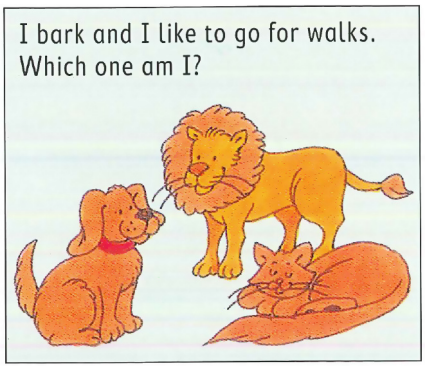
The assignment is to write a model for one or more new cards, and combine these with the two example models, and models made by others. It is then possible to see what the overlap is between the different models. Here are things to try:

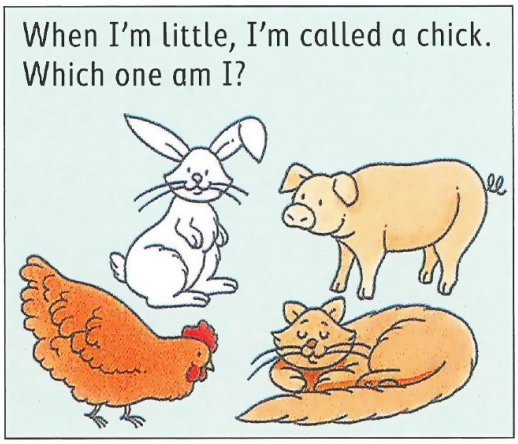
- Look for the most similar task to the task you are interested in, and see how much transfer there is between the two.
- Run all the models except the one you are interested in a couple of times, and then assess transfer to that model.

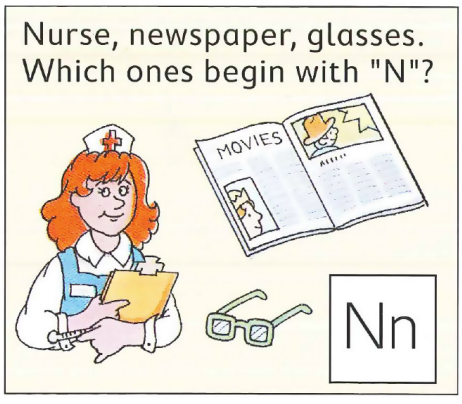
Some cards:

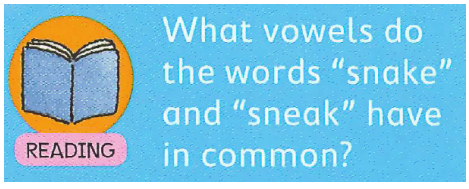












More in separate files cards1.pdf and cards2.pdf.

UNIT 3

Far Transfer and Learning Operators

The challenge of long-term learning is to properly organize skills, to define mechanisms to evaluate knowledge, and to have strategies to discover the best operators for a new task. Although this puzzle is far from solved, we offer some starting points in this Unit.

Evaluating operators

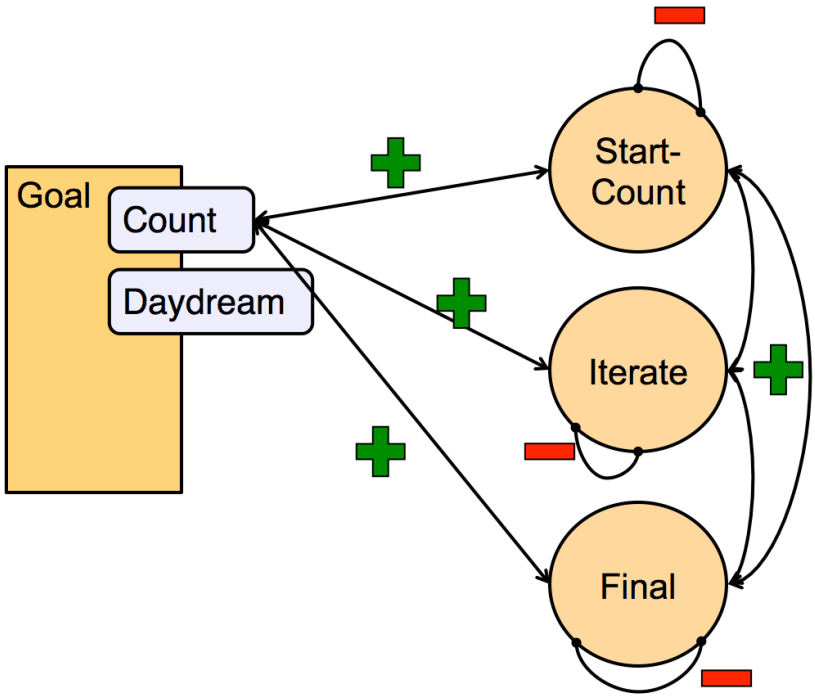
An operator is a chunk in declarative memory, so just like any other chunk it has a baselevel activation, and strengths of association with other chunks. Just like any other chunk, these activations correspond to the odds that we need the operator in the current context.

For operators, the baselevel activation represents how useful the operator is regardless of the task. In the current implementation, it receives an extra reference if it leads to a successful completion of a goal. To set this up properly, we need to specify a number of things in a task. Here is the parameter declaration of count-learn.prims, a model we will use as a demonstration:

```
define task count-learn {
    initial-goals: (learncount)
    task-constants: (count-fact say say stop)
    start-screen: start
    imaginal-autoclear: nil
    default-activation: 1.0
    ol: t
    rt: -2.0
    lf: 0.2
    default-operator-self-assoc: 0.0
    default-operator-assoc: 4.0
    goal-operator-learning: t
    reward: 10.0
    beta: 0.1
}
```

The goal-operator-learning parameter switches on the learning mechanisms for operators. Each time an operator is successful in reaching a goal, it will receive an extra reference. Success is defined in terms of whether the goal-action is carried out. The reward parameter sets the maximum time to reach the goal (10 seconds in the example).

Associative strength will be used to represent the usefulness of an operator for a particular goal. The figure below illustrates the overall picture of associations:



The key associations that need to be learned are the associations between goals (like count and daydream), and operators. The assumption is that these associations are initially set to 0, but increase if an operator is successful in accomplishing a goal.

Remember that there is no hard connection between operators and goals. So in any situation any operator may be retrieved for any set of goals, but typically only operators will be chosen with a positive association to that goal. However, if a goal is in a situation in which there are no associated operators, any matching operator can be tried. If that leads to success, the association between the operator and the goal will be strengthened using reinforcement learning. The equation for the update is:

delta Sji = beta(payoff - Sji)

in which

payoff = MaximumSji * (reward - timeToReward)/reward

If the operator does not lead to a reward, it is penalized:

payoff = MaximumSji * (0 - timeToFailure)/reward

In these equations, reward, beta and MaximumSji are parameters that are set by the model (see example above). MaximumSji is the default-operator-assoc parameter.

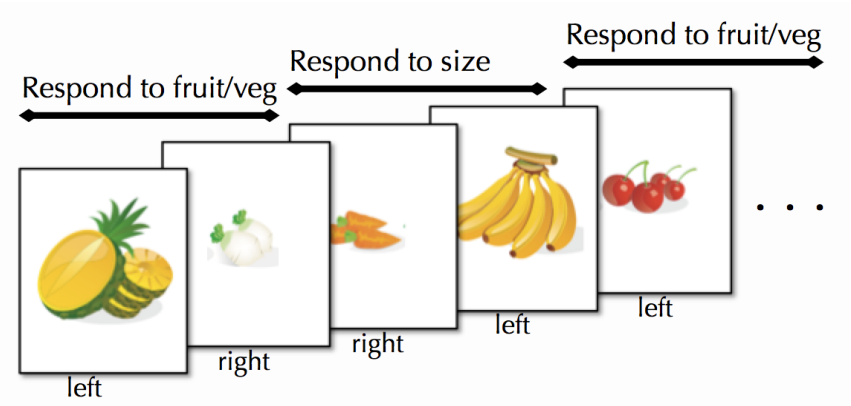
If we run the count-learn model, nothing will happen (in fact, the current version will crash). But luckily another model can supply all the necessary operators. So, also load `semantic-global.prims`, and now try to run count-learn. It will now run successfully, and will learn associations between the semantic operators, and the count goal.

Just semantic-global and count-learn can hardly go wrong. Load in a couple of other models, so that the right operators have some competition from wrong operators.

Example of Far Transfer

Karbach and Kray have shown in an experiment that training on task-switching transfers to several other control tasks, among which the Stroop task. You can load in both models, and check the overlap. The key aspect to modeling transfer here is not so much that there is a large overlap between the two tasks (which there isn't), but that task switching, at least this particular version, trains a particular skill that is useful for getting better at Stroop.

In the particular version of task switching, subjects have to keep track of what the task is themselves: there is no external cue. The stimuli consist of either pictures of fruit or vegetables, which can be either small or large. For the first two stimuli, subjects have to respond to fruit/vegetable, the next two large/small, the next two fruit/vegetable, etc. The structure of this version of task switching forces a strategy the prepares for the upcoming stimulus (other versions of task switching also allow a more reactive policy, so training on them may not always have the desired result).



In the model this means that after a response has been made on an item, it is not sufficient to wait for the next item: it has to prepare for what to do with that next item. But it should only prepare once.

To prepare, the model will set up a second goal that it will retract when it is done preparing. Setting up the second goal is done with the following operator, which activates when the model is looking at the fixation cross:

```
operator choose-next-task {
    V1 = fixation
    G2 = nil
    ==>
    prepare-next -> G2
}
```

The prepare-next goal now has to decide what the task will be. While waiting during the first fixation cross, the model sets the task to foodtask (in WM1), the count to one (WM2), puts nil in G2 to indicate that it is prepared for the task, and that waits for the stimulus.

```
define goal prepare-next {
    operator set-first-task {
        V1 = fixation
        WM1 = nil
        ==>
        foodtask -> WM1
        one -> WM2
        nil -> G2
        wait -> AC1
    }
```

After the task has been carried out (you can check the model yourself to see whether you can figure out how it works), the model needs to check what it has to do next. It there for uses the `choose-next-task` operator again that reinstates the `prepare-next` goal. The following operator, which is also part of the prepare-next goal, starts determining the next task:

```
    operator determine-next-task-retrieve-count {
        V1 = fixation
        RT1 = nil
        WM1 <> nil
        ==>
        count-fact->RT1
        WM2->RT2
    }
```

Whereas the task switching model has no choice whether to prepare, the Stroop model does have a choice. The idea is that the model can just wait for a stimulus to appear, or that it can prepare by being ready to just focus on the color and ignore the word. If the model just attends the stimulus, both the word and the color will be put in slots in the input buffer, but if the focus is on color-only, only the color will be represented. In the case of a conflict trial and a regular attend action, spreading activation will both increase and decrease activation of the response, while in a congruent trial spreading activation only increases activation. But if the focus is just on color, the difference disappears.

Preparation in the Stroop model is done by the following operators:

```
    operator prepare {
        "Prepare for the upcoming stimulus"
        V1 = fixation
        G2 = nil
        ==>
        focuscolor -> G2
    }

    define goal focuscolor {
    operator attendjustcolor {
```

```
        V1 = stim
        V2 = nil
        ==>
        attendcolor -> AC1
    }
}
```

While watching the fixation cross, the model sets up a second goal to focus on just the color when the stimulus appears. This goal has just a single operator, `attendjustcolor`, which carries out the `attendcolor` action that will but the color of the ink in `V2`.

The preparation strategy competes with the more default `just-wait` strategy:

```
    operator just-wait(activation=1.5) {
    "Just wait for the stimulus"
        V1 = fixation
        ==>
        wait -> AC1
    }


    operator attend(activation=1.5) {
        V1 = stim
        V2 = nil
        ==>
        attend -> AC1
    }
```

The `attend` action will attend both the color of the ink (which will appear in `V2`) and the identity of the word (which appears in `V3`). If both attributes are attended, the identity of the word will interfere with the color of the ink, but if only the color of the ink is attended, interference will be absent.

The `(activation=1.5)` addition to more default operators means they have a higher base-level activation than the `prepare` operator, so they would normally win the competition most of the time. However, the `prepare` operator is identical to the `choose-next-task` operator from task-switching . Training on task-switching will therefore increase the activation of that operator, making it more likely that it will be chosen after switching to Stroop.

You can try this out by running this by hand and observing the choice of strategy (and the resulting latencies), or by running the provided batch script (`taskswitchstrooptransfer.bprims`). A simple R-script is provided to extract the results out of the data file.

Assignment

Your goal is to make a model of the Zbrodoff task. From ACT-R's unit 4:

The following data were obtained by N. J. Zbrodoff on judging alphabetic arithmetic problems. Participants were presented with an equation like A + 2 = C and had to respond yes or no whether the equation was correct based on counting in the alphabet – the preceding equation is correct, but B + 3 = F is not.

She manipulated whether the addend was 2, 3, or 4 and whether the problem was true or false. She had 2 versions of each of the 6 kinds of problems (3 addends x 2 responses) each with a different letter (a through f). She then manipulated the frequency with which problems were studied in sets of 24 trials:

- In the Control condition, each of the 2, 3 and 4 addend problems occurred twice.

Each participant saw problems based on one of the three conditions. There were 8 repetitions of a set of 24 problems in a block (192 problems), and there were 3 blocks for 576 problems in all. The data presented below are in seconds to judge the problems true or false based on the block and the addend. They are aggregated over both true and false responses:

```
Control Group (all problems equally frequently)
           Two    Three    Four
Block 1   1.840   2.460   2.820
Block 2   1.210   1.450   1.420
Block 3   1.140   1.210   1.170
```

Your assignment is to write the whole model, but with some transfer from a given model, `addition.prims`. That model does addition by counting, similar to your unit 1 exercise, but it yields addition-facts as a result. The assumption is that this model is one of the building blocks for the zbrodoff model. (zbrodoff-start.prims will give you start).

There are several things you can do. One is to write a stand-alone zbrodoff model. If you run that with the appropriate number of trials (see for a bprims script below that does that), you will find that the model is too slow in block 1, because it has to start all the way from scratch.

```
repeat 5
reset
run zbrodoff block1c 192
run zbrodoff block2c 192
run zbrodoff block3c 192
```

However, with some prerequisite skills, for example by training the model on addition first, you may get closer to the data.

An alternative is to let the zbrodoff model discover (some of) the relevant operators itself, after learning addition first, and possibly other tasks (something with instance retrieval that fits).