# Formal Languages and Logics

Jacobs University Bremen
Herbert Jaeger

Course Number 320211

# Lecture Notes

*Version from Oct 18, 2017*

# 1. Introduction

## 1.1 What these lecture notes are and aren't

This lecture covers material that is relatively simple, extremely useful, superbly described in a number of textbooks, and taught in almost precisely the same way in hundreds of university courses. It would not make much sense to re-invent the wheel and re-design this lecture to make it different from the standards. Thus I will follow closely the classical textbook of Hopcroft, Motwani, and Ullman for the first part (automata and formal languages) and, somewhat more freely, the book of Schoening in the second half (logics). Both books are available at the IRC. These books are not required reading; the lecture notes that I prepare are a fully self-sustained reference for the course. However, I would recommend to purchase a personal copy of these books all the same for additional and backup reading because these are highly standard reference books that a computer scientist is likely to need again later in her/his professional life, and they are more detailed than this lecture or these lecture notes.

There are several sets of course slides and lecture notes available on the Web which are derived from the Hopcroft/Motwani/Ullman book. At http://www-db.stanford.edu/~ullman/ialc/win00/win00.html, you will find lecture notes prepared by Jeffrey Ullman himself.

Given the highly standardized nature of the material and the availablity of so many good textbooks and online lecture notes, these lecture notes of mine do not claim originality. In fact, many of the examples I use are taken from the Hopcroft/Motwani/Ullman book, and I will not explicitly mention this in the text. I also occasionally copy from the ACS1 lecture notes written by Andreas Jacoby in Spring 2002, without giving the reference. The main purpose of these (fully self-contained) lecture notes is to make the purchase of a textbook unnecessary and to establish a reference of what contents will be asked for in the exams.

## 1.2 What automata and formal languages are and are good for

The first part of this lecture will be concerned with *formal languages*. One of the most convenient and most widely used ways to specify a formal language is through *automata.* In this first introduction, we will give an informal sketch of the most simple type of an automaton, which will in turn give you a first idea of what a formal language is.

What are "automata"? They are mathematical formalisms that can be interpreted intuitively as simple computational machines.

What do they "do"? Typically they process "words", that is, finite symbol sequences.

**Example:** The simplest kind of automaton is a *finite automaton* (FA). Physically, think of a FA as a device that can switch between a finite number of different *states* (they might be realised electronically or mechanically), with the switching events being triggered by symbols that are read in into the FA. More abstractly, such a FA can be represented by a directed graph, whose nodes ("vertices") correspond to the states. They are connected by arrows which are labelled by the symbols that the FA can process. Fig. 1.1 gives a simple example (from the Hopcroft book):

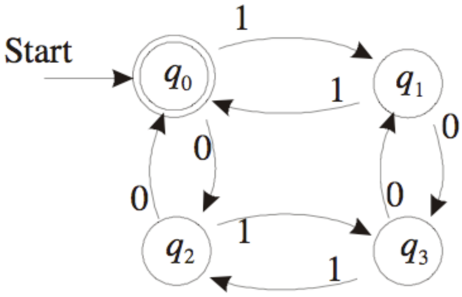


**Fig. 1.1:**A graph representation of a finite automaton. It has 4 states $q_0$, $q_1$, $q_2$, $q_3$, of which $q_0$ is both the starting state (indicated by "start" arrow) and the final accepting state (indicated by double circle).

The FA from Fig. 1.1 starts in the *starting state* $q_0$. It now reads in a word made from symbols {0,1}, for instance the word "0110". The symbols are read in one by one. Each newly read-in symbol induces a state transition according to the transition arrows' labels. Thus, the word 0110 takes the FA through the state sequence $q_0$, $q_2$, $q_3$, $q_2$, $q_0$. Because the last state in this sequence is $q_0$, the *accepting state*, the FA *accepts* the input 0110. It would *reject* any input word that does not drive it through a state sequence ending in the accepting state. For instance, the word 10 would bring the FA to $q_3$, which is not an accepting state, the word 10 would be rejected. The set of words accepted by this FA is the *language* of this automaton. (Can you describe this language in intuitive terms?)

This is just an informal first contact to give an idea of the "flavour" of automata. We will later re-introduce FAs rigorously, and learn about more complicated (and more powerful) automata types. Here are some characteristic features of the theory of automata and languages:

- The task of automata is to accept/reject input words.
- Thus, an automaton defines a formal language, that is, the set of accepted words.
- Finite automata are the most simple automata.
- More complicated automata can be derived from FA by adding further mechanisms, for instance a stack memory (then you get "pushdown automata") or a tape memory (then you get "Turing machines"), or one can associate output actions with states (then one gets input-output-devices), or one can introduce randomness (then one gets non-deterministic or probabilistic automata)
- The basic use of automata is as algorithms that "decide" languages, that is, for any query word carry out a computation (run the automaton on that word) whose result is a yes – no answer (via accept – reject) whether the query word belongs to the language.
- The theory of automata and formal languages investigates how certain classes of formal languages correspond to certain types of automata.

Applications of the theory of automata and formal languages:

- Computer science:
    - Syntax checking of computer programs, lexical analysis of computer programs
    - Software for scanning large bodies of text (e.g., web page collections) for key words or key patterns

- Mechanical and electrical engineering: Design and implement control devices in mechano-electrical machines, from laundering machines to cellphones to nuclear power plants (control chips often implement tens to hundreds to thousands of FAs – note that the basic forking of labelled arrows leaving a FA state has the logical structure of a case distinction)
- Chemistry, biochemistry: Models of chemical reaction chains, folding states of enzymes.
- Information theory, theoretical physics: measuring the complexity of measurement series
- Cognitive science, robotics: modeling and implementing decision-making ("action selection") sequences

All in all, automata models and formal languages provide one of the most basic and most widely useful tools that computer science and mathematics offers, comparable to elementary calculus or elementary linear algebra.

## 1.3 What a logic is and is good for

What is a "logic"? A mathematical language that can describe non-mathematical and mathematical facts in mathematical "propositions".

What can you "do" with a logic? You can formally derive ("deduce") new propositions from ones that you already have, thereby answering interesting questions about the reality you are describing.

You already know a simple kind of logic, Boolean logic. Boolean logic is quite restricted in its descriptive possibilities, you can only point to complete real-life facts and claim that they are false or true. In this lecture we will be concerned with another logic, *first order logic* (FOL). FOL can describe the internal structure of real-life facts; FOL can name individual things, actions, properties. A famous example, which is in fact as old as it appears, is illustrated in Fig. 1.2.

Important points:

- FOL can point to objects (Socrates), properties (human, mortal), actions (not in this example)
- Using FOL, one can describe almost anything one wants; FOL is a very *expressive* logic. For instance, FOL is powerful enough to capture all of mathematics (within a certain set-theoretic framework) – in contrast, with Boolean logic you could not express mathematical statements as simple as "1 + 1 = 2".
- Within the world of logics, one can derive new propositions (the *conclusions*, here: *mortal Socrates*) from given ones (the *premises*, here: $\forall x$ (*human x* $\rightarrow$ *mortal x*) and *human Socrates*). Such derivations are *mathematical proofs*. (the $\Downarrow$ in Fig. 1.2.).
- If a logic is correct, the conclusions one can prove are true (= hold in the real world) whenever the premises are true. This is a miraculous fact – think about it!

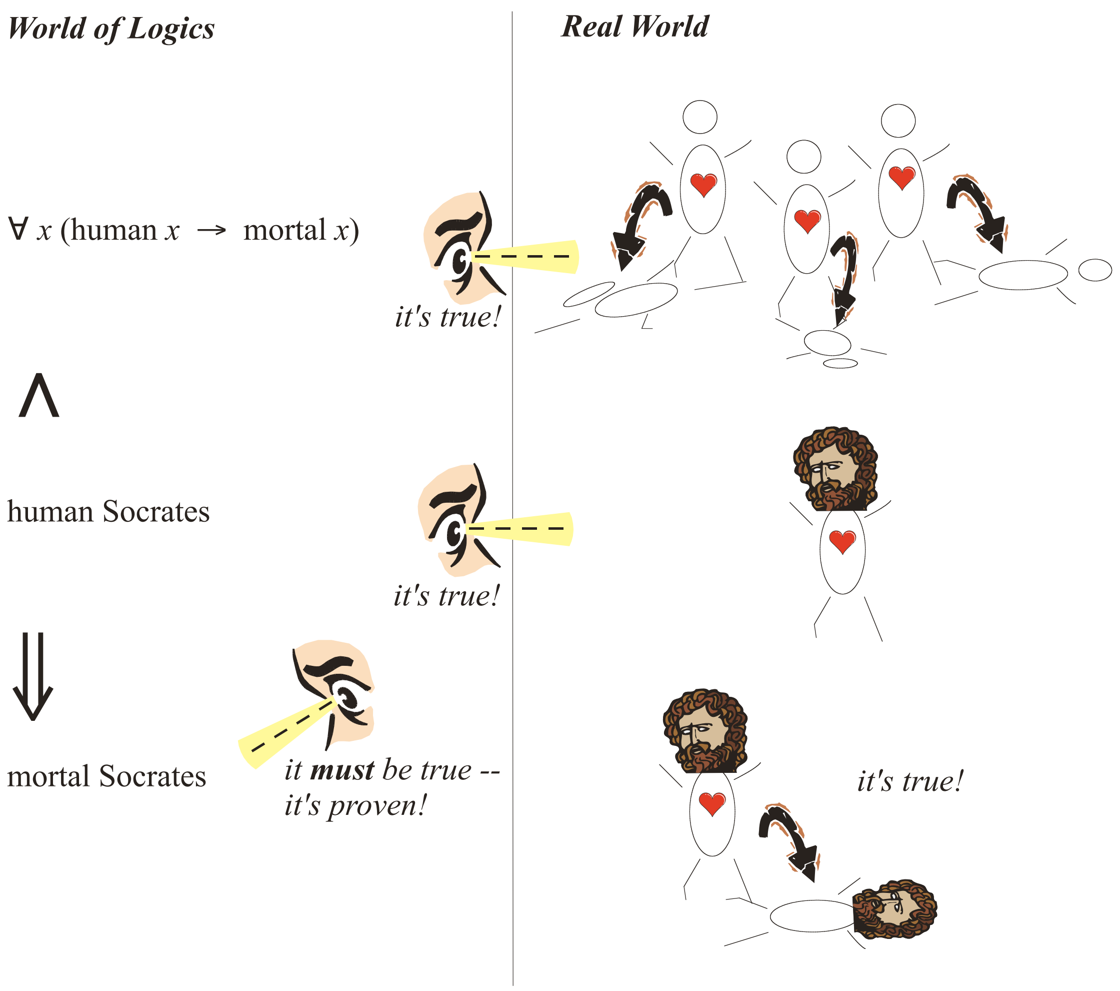


**Fig. 1.2:** The miracle of logics: Purely formal derivations are true in the real world!

Applications of Boolean logic:

- In basic EE & CS: for all binary switching systems, -- that is, for pretty much all of digital electronics from simple logical gate arrays to DSP chips to complete computers.
- In Artificial Intelligence (AI): some real-world planning problems can be cast in Boolean logic and solved mechanically by the rules of Boolean logic. In the advanced lecture on Computation and Complexity (in your third year) you will learn that in fact, Boolean logic is in a sense universal: every computational problem that can be solved out by "trying out solutions" (technically, every problem that lies in class NP) can be translated into an equivalent Boolean logic problem.

Applications of FOL:

- In basic CS: program verification
- In Artificial Intelligence: FOL is the basis of most AI systems, like expert systems, high-level robot control systems, intelligent user interfaces, speech understanding systems, etc.
- In database systems: deductive databases

### 1. 4 Basic textbooks (for additional and backup reading, not required):

Hopcroft, John E., Motwani, Rajeev, Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, 2 (Addison Wesley) IRC: QA267 .H56 2003

Schoening, Uwe: *Logic for Computer Scientists (Progress in Computer Science and Applied Logic, Vol 8)*, (Birkhauser) IRC: QA9 .S363 1989

Michal Walicki: *Introduction to Logic.* Online lecture notes, obtainable from the author's website at http://www.ii.uib.no/~michal/ . *This is a nicely written introduction to propositional and predicate logic, apparently intended for math students, but well-explained. It contains an interesting historical introduction section.* I put a copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/IntroToLogicWalicki.pdf.

# Part 1: Formal languages

## 2. Basic notions and two different kinds of infinity

An **alphabet** Σ is a finite nonempty set of symbols, for instance $\Sigma = \{0, 1\}$ or

$$\Sigma = \{a_i \mid i = 1, ..., n\}.$$

A **word** $w$ over Σ is a finite sequence of symbols from Σ. ε denotes the **empty word**. For two words $w$ and $v$, $wv$ is the concatenated word. For a word $w$ and a symbol $x$, $wx$ is the word $w$ with $x$ appended at the end; $xw$ is $w$ with $x$ prepended. $| w |$ is the length of $w$, $| \varepsilon | = 0$. Furthermore, we use the following notations for certain sets of words:

1. $\Sigma^k$: set of all words over Σ of length $k$.
2. $\Sigma^0 = \{ \varepsilon\}$
3. $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup$ ..., the set of all non-empty words over Σ.
4. $\Sigma^* = \Sigma^0 \cup \Sigma^+$, the set of all words over Σ.

Any subset $L$ of $\Sigma^*$ is a **language** over Σ. If you want to emphasize that you are dealing with the abstract theory of such languages, say "formal language" instead of just "language".

Since by definition, the alphabet Σ is finite, we can *enumerate* all words over Σ. That is, we can order all words by sorting them in a right-infinite sequence. This ordering can be done in many ways, but the most standard ordering is the *alphabetical enumeration*. For instance, for the binary alphabet $\Sigma = \{0, 1\}$, the set $\Sigma^*$ of all words over Σ is sorted by alphabetical enumeration like this:

ε, 0, 1, 00, 01, 10, 11, 000, 001, ...

In the general case of a generic alphabet $\Sigma = \{a_i \mid i = 1, ..., n\}$, the alphabetical enumeration of $\Sigma^*$is achieved as follows. Assume that the symbols from Σ are ordered, e.g. by their indices: in $\Sigma = \{a_i \mid i = 1, ..., n\}$, $a_i$ comes before $a_j$ if $i < j$ (just as the letters of the English alphabet are ordered, with "a" first and "z" last). Then enumerate $\Sigma^*$ via

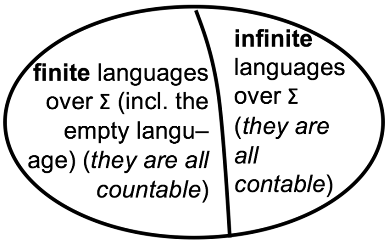
$$\varepsilon, a_1, a_2, ..., a_n, a_1a_1, a_1a_2, ..., a_1a_n, a_2a_1, ... ... a_na_n, a_1a_1a_1, a_1a_1a_2, ...$$

Any set is called *countable* if it has at most as many elements as there are natural numbers. In mathematical rigor, a set $S$ is defined to be countable if there exists an *injective* map $i$: S → ℕ (remember: a map $f$: $A \rightarrow B$ from a set $A$ to a set $B$ is called injective if for every $b \in B$ there exists at most one $a \in A$ such that $f(a) = b$). The set $\{0, 1\}^*$ can be seen to be countable by understanding the alphabetical enumeration as a map from $i$: $\{0, 1\}^* \rightarrow$ ℕ:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $\{0, 1\}^*$ : | ε, | 0, | 1, | 00, | 01, | 10, | 11, | 000, | 001, ... |
| map $i$ : | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| ℕ : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 ... |

Finite sets are always countable. But there are also many sets that are infinite and countable — one calls such sets *infinite(ly) countable*. $\{0, 1\}^*$ is an example. Another example of a countable infinite set is ℕ itself, or the set of all even integers.

So the set of all languages over Σ splits into two subsets, as in the following diagram:


**finite** languages over Σ (incl. the empty langu–age) (*they are all countable*)
**infinite** languages over Σ (*they are all contable*)

Any subset of a countable set is countable. Therefore the following implication follows:

$\Sigma^*$ is infinite countable ⇒ any language over Σ is countable.

This is a good moment to briefly explain how "sizes" of sets are handled mathematically. The technical term for the size of a set $S$ is called its *cardinality*. It is denoted by vertical bars: $|S|$ is the cardinality of $S$. For finite sets, the cardinality is just the number of its elements, for instance $|\{a, b, c\}| = 3$. The cardinality of infinite sets is measured by so-called *cardinal numbers*, an advanced concept of set theory that we will not further explore. But there is a workaround which is good enough for us. We will characterize cardinalities of infinite sets by equating them with the cardinality of known infinite sets. In particular, $|S| = |\mathbb{N}|$ means that $S$ has the same cardinality as the set of natural numbers ("infinitely countable"); $|S| = |\mathbb{R}|$ means that $S$ has the same size as the set of the reals. In general, a statement of the form $|A| = |B|$ means that $A$ has the same cardinality as $B$. The rigorous definition for "same cardinality" goes like this:

**Definition 2.1** Two sets $A$, $B$ are said to have the same cardinality, written $|A| = |B|$, if there exists a bijective map $f$: $A \rightarrow B$.

Infinite cardinalities need some time to get used to. For instance, the set $\mathbb{N} = \{0, 1, 2, ...\}$ of natural numbers has the same cardinality as the set $\mathbb{E} = \{0, 2, 4, ...\}$ of even natural numbers, that is, | $\mathbb{N}$ | = | $\mathbb{E}$ | – although maybe you would feel that there are twice as many naturals as even naturals. Also with a little thinking you will find that there are as many rational numbers as there are naturals: | $\mathbb{N}$ | = | $\mathbb{Q}$ |. (Recall that the rational numbers are the numbers that can be written as a signed ratio of two natural numbers).

We now turn to a not-so-easy question: how many languages exist over Σ? There exist as many languages over Σ as there exist subsets of $\Sigma^*$. Because $\Sigma^*$ has as many elements as $\mathbb{N}$ has, this boils down to the question of how many subsets $\mathbb{N}$ has. This question is easily answered: $\mathbb{N}$ has as many subsets as there are real numbers between 0 and 1. This is because every subset of $\mathbb{N}$ can be specified by a right-infinite binary string. For instance, the string 01100... would express a subset of $\mathbb{N}$ containing 2, 3,... but not 1, 4, 5,... . But every such string also specifies a real number in the interval [0 1], if you think of the string as the binary digit representation of a real number (prepend "0." to the string, for instance interpret the string 01100... as the binary represented number 0.01100...). We skipped over a little technical difficulty here: a string that ends in an infinite sequence of all 1's represents the same real number as the same string with the trailing ones replaced by all 0's and the previous 0 switched to 1.

Remember that there exist as many real numbers altogether as there exist reals between 0 and 1. (This is so because the open unit interval can be mapped 1-1 to all of $\mathbb{R}$). Summing up, we find:

- $\mathbb{R}$ has as many elements as [0 1]
- [0 1] has as many elements as there are right-infinite binary strings
- There are as many right-infinite binary strings as there are subsets of $\mathbb{N}$
- $\mathbb{N}$ has as many subsets as $\Sigma^*$ has
- There are as many subsets of $\Sigma^*$ as there are languages over Σ.

Altogether, there are | $2^{\mathbb{N}}$ | = | $\mathbb{R}$ | many languages over Σ. ($2^{\mathbb{N}}$ is a mathematical way to denote all right-infinite binary strings.) This is a *huge* number of languages... it is *uncountable*. There is no way of enumerating all languages over Σ!

Examples of formal languages:

1. English (if English is cast as a formal language, what would be Σ, what would be *L*? there are many reasonable choices! For instance, Σ might be the set of letters of the English alphabet – but it might also be the set of English words. And *L* might be the set of English words [makes sense if Σ is the standard English alphabet], but in computational linguistics, *L* is more often taken to be the set of grammatically correct English sentences, or even texts!) **Remark:** historically, one of the origins of the theory of formal languages came from linguistics. Noam Chomsky (check him out on wikipedia), linguist, philosopher, politician – one of the great intellectual figures of the 2nd half of the last century – developed the foundations of the formal language theory in order to achieve a mathematically rigorous account of natural language grammars.
2. The set of syntactically correct programs of the programming language C
3. ∅, the empty language
4. { ε}, the language containing the empty word (note that ∅ ≠ { ε})
5. The language $\{0^n1^n \mid n \geq 0\}$

6. The language $L$ which is equal to {1} if the Riemann hypothesis is true, else is equal to {0}.
7. The language WINCHESS = $\{w \mid w$ codes a winning strategy for white in chess$\}$ (may be the empty language! nobody knows). **Remark.** This and the previous example indicate that complicated and deep *problems* can be cast as *language decision problems.* If we possessed some algorithm to *decide*, for some word *w,* whether it codes a winning strategy for chess, then (in principle, not accounting for computation times) we could compute a winning strategy for white. How? One would "just" have to generate all words $w_1, w_2, \ldots$ , and for each of them decide whether it is in this miracle language WINCHESS. If a winning strategy exists at all, it corresponds to some word $w_i$, and will eventually be found … in due time…
8. The language $L_f = \{(x, y) \mid y = f(x)\}$ over the alphabet $\Sigma = \{0,1,(,), ","\}$, where $f$ is some numerical function on the integers and $x$, $y$ are binary strings representing integers. **Remark.** Just as solving problems can be cast as language decision, so can *function computation*. How? Well, in order to compute $f(x)$ for some $x$, "simply" generate all words $(x, y_1)$, $(x, y_2)$, … and decide for each of them whether it is in the language $L_f$. For one (and only one) $y_i$ it will hold that $y_i = f(x)$, and this will be found.

**A grand preview.** The remarks in examples 7. and 8. indicate that deciding languages is a very fundamental operation. This is indeed very true. In the course "Computability and Complexity" you will learn to understand (and analyse) essentially *everything that computers can do* as language decision problems.

## 3. Finite automata, regular expressions and regular languages

In the beginning of this course we will investigate a particularly simple class of formal languages, the *regular* languages. They are a subclass of the context-free languages, which are a subclass of the context-sensitive languages, which are included in the recursive languages, which are a subclass of the recursively enumerable languages, ... that's about it. There are several ways to define and analyse regular languages. One of them is to describe them through finite automata like the one we have met in the Introduction.

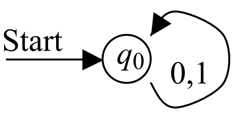
### 3.1 Finite automata

**Aside:** if $M$, $K$ are sets, then $M \times K = \{(m, k) \mid m \in M, k \in K\}$ denotes the set of all pairs of elements of $M$ and $K$.

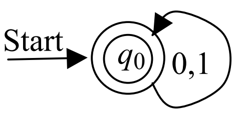
**Def. 3.1** A **deterministic finite automaton (DFA)** (or simply a **finite automaton** for short) $A$ is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, ..., q_n\}$ is a finite set of *states*, $\Sigma$ is the alphabet of *input symbols*, $\delta$: $Q \times \Sigma \rightarrow Q$ is the *transition function*, $q_0$ is the *start state*, and $F \subseteq Q$ is the set of *accepting states.*

**Examples 3.1**. A convenient format to depict DFAs graphically is their *transition graph*. States are denoted by circles (the vertices of the graph), $\delta$ makes for labelled arrows (labels = symbols from $\Sigma$ which determine the state transitions), the starting state is distinguished by a "start" arrow, and all accepting states are marked by a double circle (there may be none!). In the Introduction we already saw a state transition diagram. Here are three more (for all, $\Sigma = \{0,1\}$):

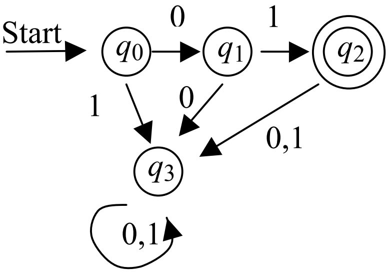
(i)



(ii)

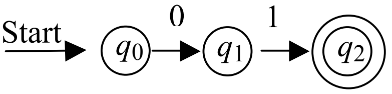


(iii)



The DFA (i) accepts no word, (ii) accepts all words, and (iii) accepts exactly the word 01. In the design of (iii) a standard trick was used: state $q_3$ serves as a "dead" state from which there is no escape – all "unwanted" word continuations are channelled into this dead state. Such dead states may be omitted from a transition graph, that is, (iii) could also be rendered like

(iii')



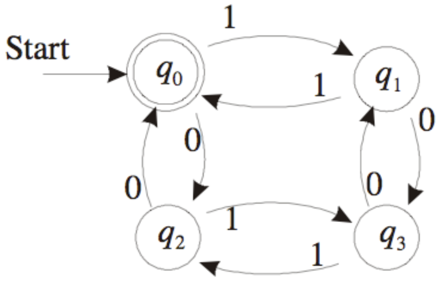
Thus, the characteristic property of transition graphs of DFAs is that for every state and symbol there is *at most one* arrow leaving that state with that symbol.

**Def. 3.2** Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Define the **extended transition function** $\hat{\delta} : Q \times \Sigma^* \to Q$ by induction over words through $\forall q \in Q : \hat{\delta}(q, \varepsilon) = q$ and $\forall q \in Q\ \forall w \in \Sigma^*\ \forall x \in \Sigma : \hat{\delta}(q, wx) = \delta(\hat{\delta}(q, w), x)$. Use notation $\delta(q, w)$ for $\hat{\delta}(q, w)$.

**Def. 3.3** A DFA $A$ **accepts** a word $w$ if $\delta(q_0, w) \in F$. If $\delta(q_0, w) \notin F$, the word is **rejected**. The set of words $L(A) = \{w \in \Sigma^* \mid A \text{ accepts } w\}$ is the **language accepted by** $A$. Any language that is the language accepted by some DFA is a **regular language**. The set of all regular languages is denoted by $\mathcal{REG}$.

Another convenient representation, in tabular form (for use in computers, if you wish to simulate a DFA on your machine) is the **transition table**, which contains in its rows a tabular form of δ. Format: the left column lists all states, the right columns indicate into which next state the DFA is taken by reading in a symbol. The starting state is singled out by an arrow, the accepting states are marked by a *.

**Example 3.2**. of a transition table (right panel, for our introductory DFA):



| | | 0 | 1 |
|---|---|---|---|
| → | $q_0$* | $q_2$ | $q_1$ |
| | $q_1$ | $q_3$ | $q_0$ |
| | $q_2$ | $q_0$ | $q_3$ |
| | $q_3$ | $q_1$ | $q_2$ |

**Notation:** it is sometimes convenient to write $(q, x, p)$ to denote $\delta(q,x) = p$, and call $(q, x, p)$ *a transition*. In other words, the transition function δ is interpreted as the set $\delta = \{(q, x, p) \mid \delta(q,x) = p\}$, that is, by *an edge-labelled graph*.

**Aside:** if $M$ is a set, then Pot($M$) denotes the **power set** of $M$, the set of all subsets of $M$. For a finite set $M$ of size $n$, Pot($M$) has size $2^n$.

**Def. 3.4** A **nondeterministic finite automaton** (**NFA**) is defined like a DFA, except that the transition function reads $\delta$: $Q \times \Sigma \to \mathrm{Pot}(Q)$, that is, assigns a subset of $Q$ to each state-symbol pair. Note that $\delta(q,a) = \varnothing$ is perfectly possible, that is, a transition from state $q$ triggered by a symbol $a$ may lead nowhere.

**Def. 3.5** The **extended transition function for a NFA** $\hat{\delta}: Q \times \Sigma^* \to Pot(Q)$ is defined inductively through

$$\forall q \in Q: \ \hat{\delta}(q, \varepsilon) = \{q\}$$

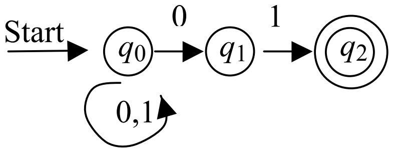
and

$$\forall q \in Q \ \forall w \in \Sigma^* \ \forall x \in \Sigma: \ \hat{\delta}(q, wx) = \bigcup_{p \in \hat{\delta}(q,w)} \delta(p, x).$$

Again, we use the notation $\delta(q, w)$ for $\hat{\delta}(q, w)$.

**Def. 3.6** A NFA $A$ **accepts** a word $w$ if $\delta(q_0, w) \cap F \neq \varnothing$. The language $L(A)$ of a NFA is the set of all words accepted by $A$.

**Example 3.3**. The following NFA accepts all the words that end with 01. Note that there are two "0" arrows leaving $q_0$, which would be forbidden in a DFA.



Intuitively, a NFA accepts a word $w$ if there exists *some* state sequence that (on input of $w$) ends in an accepting state. Note that NFAs are (i) good for human design: it is typically much easier to write down an NFA for a given language than a DFA, but (ii) NFAs are bad for computer implementations because of the non-determinism: if a computer program would have to emulate an NFA, the program wouldn't know which transition options it should take when a non-deterministic situation is encountered. Fortunately, it is possible to automatically transform any (maybe human-specified) NFA into an equivalent (computer-runnable) DFA. This construction is the underlying idea used in the proof of the following proposition.

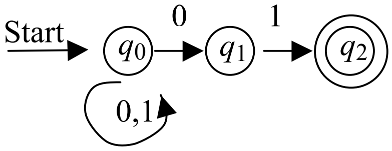
**Proposition 3.1.** The languages accepted by DFAs are the languages accepted by NFAs.

**Proof, general comment**: This proposition is of the form "set A equals set B". Almost invariably, when proving statements of this form, one shows two things: (i) A $\subseteq$ B, and (ii) B $\subseteq$ A.

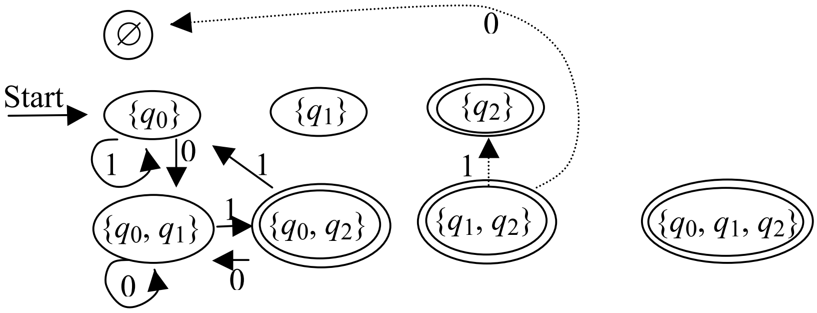
**Proof, main idea:** (i) Showing that the set of languages accepted by DFAs is a subset of the set of languages accepted by NFAs is trivial, because DFAs *are* NFAs (the definition of NFAs is a *generalization* of the definition of DFAs). The difficult part is to show (ii) that the set of languages accepted by NFAs is a subset of the set of languages accepted by DFAs. For this one has to start with some NFA $A = (Q_N, \Sigma, \delta_N, q_{0N}, F_N)$ and construct from it a DFA $A' = (Q_D, \Sigma, \delta_D, q_{0D}, F_D)$ that accepts the same language. The crucial idea is to take as the *states* of $A'$ all *subsets* of states of $A$, that is, put $Q_D = \mathrm{Pot}(Q_N)$ ("**subset construction**"). Further details: put $q_{0D} = \{ q_{0N} \}$, $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \varnothing\}$, and for all $S \in Q_D = \mathrm{Pot}(Q_N)$, $x \in \Sigma$,

$$\delta_D (S, x) = \bigcup\nolimits_{q_N \in S} \delta_N (q_N, x) . \qquad (3.1)$$

**Proof, example:** The subset construction will become immediately plausible through an example. We take the NFA from example 3.3, redrawn here for convenience:



And here is the corresponding NFA obtained from the subset construction:



Notes:

1. The states $q_i$ that occur inside the state circles of the subset DFA are from the nondeterministic NFA, therefore they should more correctly be written $q_{iN}$. The $N$ subscript is omitted for readability.
2. According to $q_{0D} = \{ q_{0N} \}$, the start state is the one that contains (as a set) only $q_{0N}$.
3. The accepting states of the subset DFA are all subsets of NFA states that contain some accepting state of the NFA (here this is only $q_{2N}$), according to $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \varnothing\}$.
4. To illustrate how it is determined which arrows leave a given state of the DFA, let us consider the state $S = \{q_{0N}, q_{2N} \}$, and apply the rule $\delta_D (S, x) = \bigcup\nolimits_{q_N \in S} \delta_N (q_N, x)$. For $x =$

0, we have to check to which NFA states one can go from either $q_{0N}$ or $q_{2N}$. We find that we can reach $q_{0N}$ and $q_{1N}$ from $q_{0N}$ and no other state from $q_{2N}$. Therefore, what we can reach altogether is $\{q_{0N}, q_{1N}\}$. This yields the "0" arrow from $\{q_{0N}, q_{2N}\}$ to $\{q_{0N}, q_{1N}\}$ in the diagram. Similarly, the "1" arrow is determined by considering what states we can reach in the NFA from either $q_{0N}$ or $q_{2N}$. From $q_{0N}$ a 1 leads to $q_{0N}$ and from $q_{2N}$ a 1 leads nowhere, so we determine in the DFA a "1" arrow from $\{q_{0N}, q_{2N}\}$ to $\{q_{0N}\}$.

5. Seen strictly, in a DFA for every state $q$ and every symbol $x$ there must be an arrow labelled with $x$ leaving $q$. For instance, from $\{q_{1N}, q_{2N}\}$ we get the dotted arrows marked in the diagram. However, the state $S = \{q_{1N}, q_{2N}\}$ cannot be reached from the starting state in the subset DFA. Generally, in the diagram I omitted all such irrelevant transitions that cannot be reached from the start state, which leads to the numerous seemingly "orphan" states in the diagram.

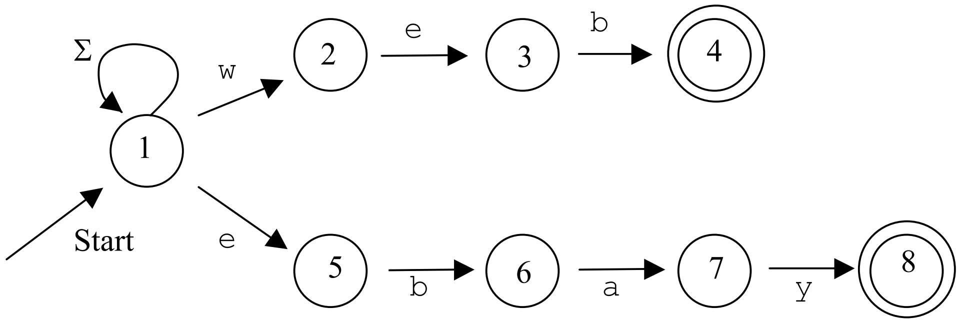
**Proof, formal:** The formal proof has to show that the language accepted by our subset DFA is the same language as the one accepted by the original NFA. We proceed *by induction on the length of words*. This is a very common type of proof in the theory of formal languages, and I suggest that you digest this proof thoroughly. We show for every word $w$ that

$$\hat{\delta}_N(q_{0N}, w) = \hat{\delta}_D(q_{0D}, w). \tag{3.2}$$

[Be careful that you understand this formula. $\hat{\delta}_N(q_{0N}, w)$ is the *set* of NFA states that can be reached from the starting state via $w$. $\hat{\delta}_D(q_{0D}, w)$ is the single *state* of the DFA that is deterministically reached from the DFA starting state via $w$. However, due to the subset construction, this single DFA state corresponds to a *set* of NFA states – namely, to $\hat{\delta}_N(q_{0N}, w)$.]
[Furthermore, make sure that you understand that if we have shown (3.2), then we are done. Because, $w$ is accepted by the NFA ⇔ $\hat{\delta}_N(q_{0N}, w)$ *contains* some accepting state ⇔ $\hat{\delta}_D(q_{0D}, w)$ *is* an accepting state ⇔ $w$ is accepted by the DFA.]
Induction basis: $|w| = 0$, i.e., $w = \varepsilon$. Then $\hat{\delta}_N(q_{0N}, \varepsilon) = \{q_{0N}\} = \hat{\delta}_D(q_{0D}, w)$.
Induction step: assume that (3.2) has been shown for all $|w| \leq n$. Let $v = wx$ be a word of length $n+1$. Then $\hat{\delta}_N(q_{0N}, wx) = \bigcup_{p_N \in \hat{\delta}_N(q_{0N}, w)} \delta_N(p_N, x)$, which by induction is equal to $\bigcup_{p_N \in \hat{\delta}_D(q_{0D}, w)} \delta_N(p_N, x)$, which by (3.1) is equal to $\delta_D(\hat{\delta}_D(q_{0D}, w), x) = \hat{\delta}_D(q_{0D}, wx)$. □

The exponential blowup of number of states through the subset construction may become necessary in some cases: there are NFA $A$ with $n$ states where the smallest equivalent DFA needs $2^n$ states. An example that comes close in badness is the language $\{w \in \{0,1\}^* \mid |w| \geq n$ and the $n$-th symbol from the last is a $1\}$. However, in many practical examples a DFA equivalent to a given NFA has about the same number of states.

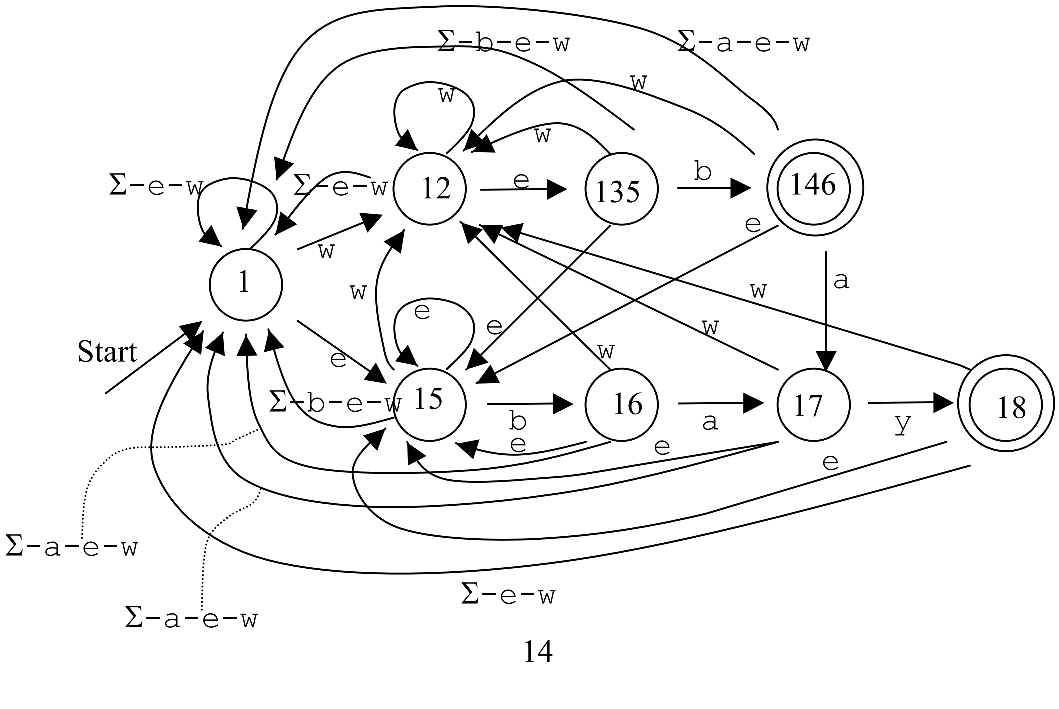
**Example 3.4.** A common task in text processing is to collect from a text occurrences of some keywords. Assume that you want to browse a text for keywords `web` and `ebay`. If you set out on the task of writing such a keyword recognizer naively, you may easily end up with an unecessarily costly algorithm. Using DFAs and NFAs, there is a way to detect keywords in a very fast way. We will now learn how one can systematically construct DFAs that read in a stream of symbols such that whenever one of a set of keywords has been read, the DFA is in an accepting state. It is easy to design an NFA for the `web`-and-`ebay`-task:



However, this being a nondeterministic device, it does not lend itself directly for use as an automated keyword search algorithm. Happily, we can transform NFAs of this type into an equivalent DFA *that has the same number of states* (note that we can certainly transform it to *some* equivalent DFA by the subset construction). Here is the recipe (only for the case where the keywords start with different symbols; the general case needs some adjustments that you can find in the HMU book):

1. The NFA start state $q_0$ becomes the DFA start state $\{q_0\}$, as usual.
2. For every other NFA state $p$ create a DFA state that consists of $q_0$, $p$ and possibly some others. These others are determined as follows. Suppose $p$ is reachable in the NFA via a word $a_1a_2...a_m$. Then include into the corresponding DFA state every other NFA state that can be reached in the NFA through any suffix $a_ja_{j+1}...a_m$ of $a_1a_2...a_m$.
3. The accepting DFA states are the ones that correspond to the accepting states of the NFA.
4. Determine the transitions as in the subset construction.

Comment: This DFA is the one that one would obtain from the subset construction, weeding out all inaccessible DFA states. The 4-step procedure above is a cheap shortcut to the subset construction which works for NFAs of the "word branching" kind as in the `web`-and-`ebay`-task. See the transition graph of the resulting DFA:



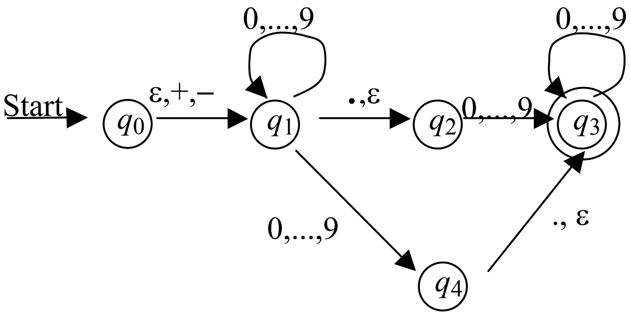
If you sit and think this over for a while, you will find that and how this principle functions. The DFA obtained can be turned into an algorithm by implementing its transition table. When reading in a text, at each symbol only one line of this transition table has to be evaluated; whenever one of the keywords has been completed, the DFA will be in an accepting state. Note that in this application, the DFA reads in an endless stream and not, like in our standard usage, finite words.

This construction is one member of the large family of *string search algorithms*, which come in many variants and many of which involve the construction of DFAs. Check out http://en.wikipedia.org/wiki/String_searching_algorithm if you want to get an overview impression.

For a human designer, NFAs are mostly easier to design than DFAs because they have fewer restrictions. By a further generalisation we arrive from NFAs at ε-NFAs, which are again even more convenient for hand-design.

**Def. 3.7** A **nondeterministic finite automaton with ε-transitions** (ε-**NFA**) is defined like a NFA, except that the transition function reads $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$, that is, from each state $q$ one can "jump" to other states without reading in a symbol.

**Example 3.5.** Here is an ε-NFA[1] designed to accept decimal numbers with an optional + or – sign and an optional decimal point, such that either the digit string before the point or after it may be empty (if a decimal point is written), but not both:



Defining the extended transition function needs to take care of the possibility that ε-transitions may follow each other in a line in the transition graph, such that one can "tunnel" from the beginning of an ε-chain to its end in a single step of the extended transition function. To accomodate this circumstance, one has to consider the transitive closure of ε-transitions, i.e., intuitively, "ε-chains". Technically, we do not introduce ε-chains but the ε-*closure* of a state. The ε-closure ECLOSE($q$) of a state $q$ is $\{q\}$ joined with the set of states accessible from $q$ through any sequence of ε-transitions:

**Def. 3.8** Let $q$ be a state in an ε-NFA. Then ECLOSE($q$) is the smallest set of states satisfying (i) $q \in$ ECLOSE($q$) and (ii) if $p \in$ ECLOSE($q$) and $\delta(p,\varepsilon) = r$, then $r \in$ ECLOSE($q$).

**Def. 3.9** The **extended transition function** for an ε-NFA $\hat{\delta}: Q \times \Sigma^* \rightarrow 2^Q$ is inductively defined through $\forall q \in Q: \hat{\delta}(q,\varepsilon) =$ ECLOSE($q$) and

[1] Mostly taken from the Hopcroft et al book, with improvements suggested by Razvan Pascanu.

$\forall q \in Q\ \forall w \in \Sigma^*\ \forall x \in \Sigma: \hat{\delta}(q, wx) = \bigcup_{q' \in \hat{\delta}(q,w)} \bigcup_{q'' \in \delta(q',x)}$ ECLOSE($q''$). Again, you may use the sloppy notation $\delta(q, w)$ for $\hat{\delta}(q, w)$.

**Def. 3.10** A ε-NFA $A$ **accepts** a word $w$ if $\delta(q_0, w) \cap F \neq \emptyset$. The language $L(A)$ of a NFA is the set of all words accepted by $A$.

**Proposition 3.2.** The set of languages accepted by DFAs is the set of languages accepted by ε-NFAs.

**Proof, main idea:** Essentially, repeat the subset construction from the proof of Prop. 3.1: $Q_D = \mathrm{Pot}(Q_E)$, $q_{0D}$ = ECLOSE($q_{0E}$), $F_D = \{S \in Q_D \mid S \cap F_E \neq \emptyset\}$, and for all $S \in Q_D$, $x \in \Sigma$,

$$\delta_D (S, x) = \bigcup_{q_E \in S} \bigcup_{q'_E \in \delta_E(q_E, x)} \text{ECLOSE}(q'_E).$$

By induction on the length of words $w$ show that $\hat{\delta}_E(q_{0E}, w) = \hat{\delta}_D(q_{0D}, w)$ for all words. □

**Thinkie.** Example 3.4 showed how to construct an „online detector" for a finite set of words, that is, for any finite language. It is also possible to create a similar „online detector" for any regular language. That is, if $L = L(A)$ for some DFA $A$, it is possible to construct a deterministic transition machine which, when reading in a (possibly infinite) symbol sequence $a_1\ a_2$ ..., passes through an accepting state at time $n$ whenever a suffix of the input sequence $a_1\ a_2\ ...\ a_n$ is in $L(A)$. Hint: use ε-transitions to transform $A$ into an „ε-transition-online detector", then transform this into a DFA.

The automata presented so far are very simple machines: They accept a word over Σ as input and either accept it or reject it, i.e., their output is "Yes" or "No". Often these automata are simply called "acceptors" to distinguish them from more complicated automata that output more than just Yes or No.[2]

There are two important classes of such automata with output: Moore and Mealy automata. Both have a translation function τ instead of final states. Their function is to *translate* words over Σ to words over O, instead of accepting/rejecting words over Σ:

**Def. 3.11.** Moore and Mealy DFA are tuples $(Q, \Sigma, \mathrm{O}, \delta, q_0, \tau)$ where $Q$, Σ, δ and $q_0$ are as for DFAs, O is the **output alphabet** and τ is a **translation function**

$\tau: Q \to \mathrm{O}$ (Moore-DFA)
$\tau: Q \times \Sigma \to \mathrm{O}$ (Mealy-DFA).

The difference between Moore and Mealy automata is that in a Moore-DFA the output only depends on the current state whereas in a Mealy-DFA it also depends on the current input. The extended transition function of Moore and Mealy DFAs is defined in the same way as for DFAs without output.

Corresponding to the extended transition function, the extended translation functions are defined as follows:

[2] The following part about Moore and Mealy machines is taken from a version of this script written by Florian Rabe.

**Def. 3.12.** The **extended translation function** $\hat{\tau}: Q \times \Sigma^* \to O^*$ is defined by induction on the input word:

$\hat{\tau}(q, \varepsilon) = \varepsilon$ and
$\hat{\tau}(q, wx) = \hat{\tau}(q, w)\, \tau(\delta(q, wx))$ (Moore-DFA) or
$\hat{\tau}(q, wx) = \hat{\tau}(q, w)\, \tau(\delta(q, w), x)$ (Mealy-DFA).

Moore and Mealy machines give rise to word-to-word translations $T: \to O^*$ by putting $T(w) = \hat{\tau}(q_0, w)$.

Moore and Mealy machines are equivalent in the sense that any word-to-word translation $T: \Sigma^* \to O^*$ which can be realized by a Moore machine can also be realized by a Mealy machine, and vice versa (mildly challenging exercise!).

Different textbooks may have slight differences in defining Moore / Mealy machines. For instance, one also finds variants of Moore machines where the extended transition function is defined from the induction basis condition $\hat{\tau}(q, \varepsilon) = \tau(q)$.

## 3.2 Regular expressions

Regular expressions (for short, regexps, singular: regex) provide an alternative way to define the regular languages. The idea behind regexps is to *construct* the regular languages by language-construction operations. The general format of a *language-construction operator C* is $L = C(L_1,..., L_n)$, that is, such an operator takes some argument languages $L_1,..., L_n$ and transforms them into a target language $L$. To obtain the regular languages, we need three such operators:

1. The union of two languages, $C(L_1, L_2) = L_1 \cup L_2$;
2. the concatenation of two languages: $C(L_1, L_2) = L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$;
3. the closure (or "star", or "Kleene closure") of a language $L$:

$$C(L) = L^* = \bigcup_{i \geq 0} L^i \text{, where } L^0 = \{\varepsilon\}, \quad L^1 = L, \text{ and } L^i = LL...L \ (i \text{ copies) for } i > 1.$$

That is, $L^*$ contains all words that are concatenations of words of $L$ (including the empty concatenations $\varepsilon$).

The regular languages are all the languages that one can obtain through iterated (nested) applications of these constructors, starting from some very simple basis languages. The notation used to describe both these basis languages and the nested applications of our constructors is *regular expressions*. They are defined together with the languages they denote:

**Def. 3.13**. (Regular expressions and their denoted languages). Let $\Sigma$ be an alphabet. A regular expression $E$ over $\Sigma$ denotes a language $L(E)$ over $\Sigma$. The syntax of regular expressions is defined inductively together with the languages they denote:

1. The constants $\boldsymbol{\varepsilon}$ and $\boldsymbol{\varnothing}$ are regular expressions, with $L(\varepsilon) = \{ \varepsilon\}$ and $L(\varnothing) = \varnothing$.
2. For each $a \in \Sigma$, $\boldsymbol{a}$ is a regular expression, with $L(\boldsymbol{a}) = \{a\}$.
3. If $E$ and $F$ are regular expressions, then $(E + F)$ is a regular expression, with $L((E + F)) = L(E) \cup L(F)$.
4. If $E$ and $F$ are regular expressions, then $(EF)$ is a regular expression, with $L((EF)) = L(E)\, L(F)$.
5. If $E$ is a regular expression, then $(E^*)$ is a regular expression, with $L((E^*)) = (L(E))^*$.

An extension of regexps is *regular expressions with language variables*. The set of language variables is $\Lambda = \{ \boldsymbol{L}_i \mid i = 1, 2, 3, ...\}$. A $\Sigma$-*interpretation* of the language variables is a mapping $I$: $\Lambda \rightarrow \mathrm{Pot}(\Sigma^*)$, that is, every language variable is assigned to some (not necessarily regular) language over $\Sigma$. If a regex $E$ contains language variables, the language denoted by the regex depends on the $\Sigma$-interpretation. We then speak of the language denoted by the regex *relative to* the $\Sigma$-interpretation, and write $L_I(E)$. If we replace the basis rules 1. and 2. by the following:

1'. the constant $\boldsymbol{\varnothing}$ is a regular expression, with $L(\varnothing) = \varnothing$,

2'. for every *language variable* $\boldsymbol{L}_i$, and $\Sigma$-*interpretation* $I$, $\boldsymbol{L}_i$ is a regular expression, with $L_I(\boldsymbol{L}_i) = I(\boldsymbol{L}_i)$,

we speak of *regular expressions with language variables*.

Precedence rules for saving parentheses: * has highest precedence, then concatention, then +.

*Notes*

1. The rules 1 and 2 provide the basis languages over $\Sigma$.
2. We distinguish between the empty language $\varnothing$ and the regex $\boldsymbol{\varnothing}$ that denotes it. The latter is in boldface (hard to see, I admit – on my computer screen the difference is clear, but in my printouts, there is no difference). Likewise, $\boldsymbol{\varepsilon}$ (boldface) is the regex denoting the language $\{\varepsilon\}$ and $\boldsymbol{a}$ is the regex denoting the singleton language $\{a\}$.
3. Our definition of regexps and the languages they denote is our first encounter with a very common way of handling things in the theory of languages, the theory of computation, and logics. Namely, one simultaneously introduces a *syntax* of certain formulae and their *semantics*, that is, what the formulae denote. A clean syntax-semantics distinction is one of the healthiest and most powerful things to have – we'll learn to appreciate this fully in the logic part of this lecture.
4. Our definition is inductive. We start with some basis expressions (and what they denote) and introduce further rules that allow us to build new expressions (and things they denote) from already given parts.
5. The syntax of regular expressions varies. Other authors and books and programming languages often use other syntaxes.
6. Regexps with language variables may denote languages that are not regular, depending on the $\Sigma$-interpretation.
7. UNIX has a much extended syntax that contains many additional convenience features (see http://www.grymoire.com/Unix/Regular.html for a nice little tutorial). The expressive power (what languages can be denoted) is however not increased by the additional syntactical features. A possible source of confusion is that what we called regular expressions is actually called a "pattern" or "search pattern" in the UNIX world; what they call "regular expressions" is a mixture of search patterns and wrapup command syntax that

tells UNIX what you want to do with the search patterns (e.g., use them to replace hits found by the search pattern with something else). Here are some (not all) components of UNIX search pattern syntax:

- . wildchart for regexp $\boldsymbol{s}_1 + \boldsymbol{s}_2 + \ldots + \boldsymbol{s}_m$, where $\Sigma = \{s_1, \ldots, s_m\}$ is the UNIX symbol set (except the end-of-line symbol). The UNIX search pattern " . " thus just denotes any single symbol.
- $[a_1\ a_2\ \ldots\ a_n]$ for regex $\boldsymbol{a}_1 + \boldsymbol{a}_2 + \ldots + \boldsymbol{a}_n$.
- [A – Z] for ASCII symbols with codenumbers from A to Z. Example: [\- \+ \. 0–9] matches any symbol from the collection -, +, ., 0, 1, ..., 9. Note that the backslash must be used if one wants to refer to a symbol that has a special function in the UNIX regex syntax: \. denotes the symbol " . ", whereas . (without backslash) denotes any symbol.
- $[\hat{}a_1\ a_2\ \ldots\ a_n]$ matches any symbol except $a_1\ a_2\ \ldots\ a_n$.
- | for our regex operator +
- ? means "zero or one of". $R$? in UNIX is same as the regex $\varepsilon + R$
- + means "at least one of". $R$+ in UNIX is same as the regex $RR^*$
- $\{n\}$ "$n$ of". $R\{5\}$ is same as $RRRRR$.
- $\{min, max\}$ "at most $max$, at least $min$ of". $R\{2,4\}$ is same as $RR + RRR + RRRR$.

**Example 3.6.** Here are two regexps denoting the language of all alternating 01 strings:

1. $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
2. $(\varepsilon+1)(01)^*(\varepsilon+0)$

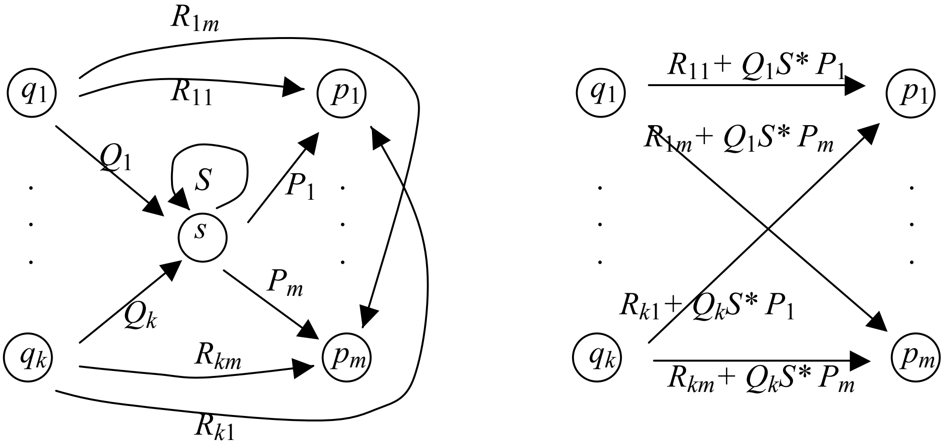
Note that here we use bracket saving conventions and are sloppy by not writing the 0 and 1 and ε symbols in boldface (which would have been the proper way). This sloppyness is common practice.

We claimed that the languages denoted by regexps are the regular languages (which we defined through DFAs in Subsection 3.1). Now we prove this fact. Again, we have the situation that we want to prove equality of two sets (the set $\mathcal{A}$ of languages denoted by regexps and the set $\mathcal{B}$ of regular languages). Thus, we must prove two directions of inclusion: $\mathcal{A} \subseteq \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{A}$. We put this into two separate propositions, starting with the second inclusion.

**Proposition 3.3:** For each ε-NFA $A$, there is a regex $E$ with $L(A) = L(E)$.

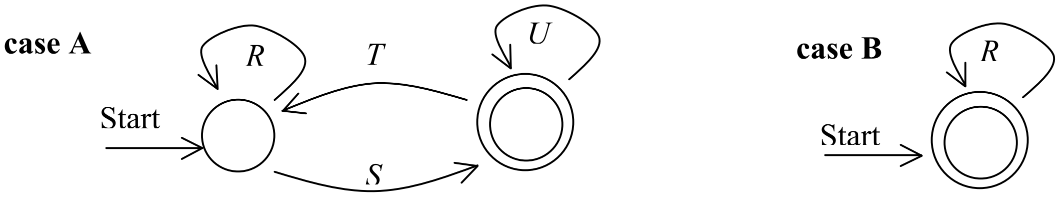
**Idea of proof:** We work with a generalized version of ε-NFA, where transition arrows can be labelled by regexps. In such *generalized NFAs*, a transition from one state to another along an arc labelled by a regex $E$ occurs whenever a string is read in that is from $L(E)$. A path along arcs labelled by regexps $E_1, \ldots, E_k$ corresponds to reading in any word from the language $L(E_1 \ldots E_k)$. The language accepted by such an automaton is the set of words $w$ which lies in some language of the form $L(E_1 \ldots E_k)$, where $E_1, \ldots, E_k$ are labels of a path from the starting state to some accepting state. Note that the ε-NFA $A$ from which we start can be considered a generalized NFA: ε-arcs in the original $A$ can be considered to be labelled by the regex $\boldsymbol{\varepsilon}$, $a$-arcs can be considered to be labelled by regex $\boldsymbol{a}$. Assume that $A$ has only one accepting state. We describe a method whereby we can transform any generalized NFA (that has more than two states and only one accepting state) into an equivalent generalized NFA that has one state less (and one accepting state). Using this method, we can stepwise transform $A$ into a generalized NFA $A'$ that has only one or two states. If we have that, we can directly give a regex for the language accepted by $A'$. If $A$ has more than one accepting states, say $q_1$, ..., $q_n$, we consider $n$ copies $A_i$ of $A$ where in the $i$-th copy only $q_i$ is accepting. The language accepted by the original $A$ is the union of the languages accepted by the $A_i$. We then get $n$ regexps for the languages of the $A_i$, which by applying the + constructor give the desired regex for the language of $A$.

**Proof (sketch).**.We skip the exercise to formally introduce generalized NFAs and proceed directly to describing the method for reducing the number of states in a generalized NFA $B$ (which has only one accepting state) by one. Let $s$ be a state in $B$ that is neither the starting nor the accepting state. We want to eliminate $s$. Let $q_1$,...,$q_k$ be the predecessors of $s$ and $p_1$, ..., $p_m$ its successors. ($q_i$ is called a predecessor if there is an arc from $q_i$ to $s$; $p_j$ is called a successor if there is an arc from $s$ to $p_j$. Some $q_i$ may coincide with some $p_j$, but that is of no concern.) The figure below (left) shows the generic situation for $s$ waiting to be eliminated. (Note that in a generalized NFA we can assume without loss of generality that between any two states $q$ and $p$, there is at most one arc – because if there were several, labelled with different regexps, then we could merge them and label the merged arc with the "+" of all the previous regexps.)



The right diagram in the figure shows the situation after eliminating $s$. All pathways from the $q_i$ to the $p_j$ (the $R_{ij}$ in the original diagram) have been augmented by the possible transitions that went through $s$. It is clear that the generalized NFA $B'$ obtained by deleting $s$ in this manner accepts the same language.

Starting from one of the $A_i$ (remember: that's the original ε-NFA with only the $i$-th accepting state left declared as accepting, and re-interpreted as generalized NFA), by repeated state elimination we arrive at either a two-state generalized NFA $A_i$* (case A) with the accepting state different from the starting state, or a one-state generalized NFA $A_i$* (right, case B) where the accepting and starting state coincide.



In case A, the regex $E_i = (R + SU^*T)^* \, SU^*$ obviously denotes the language accepted by $A_i^*$. In case B, we can put $E_i = R^*$. Altogether we have found a regex $E_i$ denoting the language accepted by $A_i$.

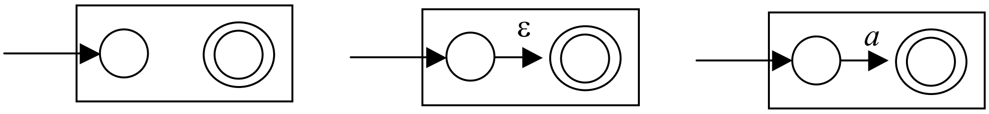
Finally, the regex denoting the language accepted by our original ε-NFA $A$ is described by $E_1 + ... + E_n$. □

Note: if $A$ has more than one accepting state, we don't have to duplicate all the work for all the reductions of the $A_i$. We can economize by deleting from $A_1$ first all those states that are not accepting in $A$, and use the intermediate reduced generalized NFA as a starting point for reducing the $A_2$,..., $A_k$.

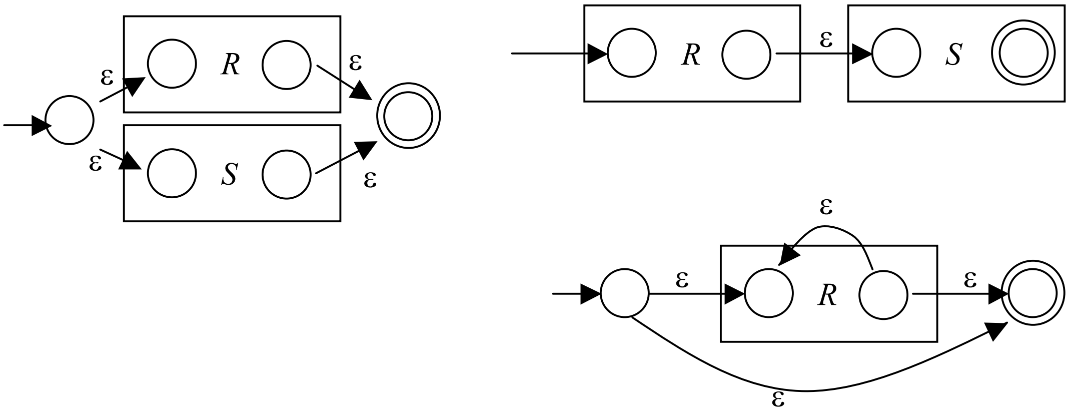
**Proposition 3.4:** for every regex $F$, there is an ε-NFA accepting $L(F)$. This ε-NFA can be constructed such that it has a single accepting state, no arcs leading back into the start state and no arcs out of the accepting state.

**Idea of proof:** re-implement the rules 1-5 of the inductive definition of regexps (Def. 3.13) in ε-NFA "hardware". For the basis rules 1 and 2, directly provide equivalent ε-NFAs. For the constructive rules 3-5, describe methods to build more complex ε-NFAs from already given ones.

**Proof, sketch:** The figure below gives three ε-NFAs for the empty language denoted by ∅, for the language {ε} denoted by **ε**, and for the singleton language denoted by ***a*** (from left to right):



The next figure shows how from ε-NFAs for regexps $R$ and $S$ one can build an ε-NFAs for $R$+$S$ (left), for $RS$ (right upper) and for $R$* (right lower):



The graphics should be self-explaining. □

## 3.3 Algebraic laws for regular expressions

For any regex, there exist infinitely many different ones that denote the same language (consider a trivial example like $\boldsymbol{a}$, $\boldsymbol{a} + \boldsymbol{a}$, $\boldsymbol{a} + \boldsymbol{a} + \boldsymbol{a}$ , ... – these are obviously equivalent.) It is desirable to perform transformations on regexps, for instance in order to simplify a regex or to show that two regexps are equivalent (transform one into the other). Such "algebraic" transformation rules serve the same purposes as say, the ordinary algebraic transformation rules or the rules that you know from transforming set theoretic or Boolean expressions. Here is a collection of them:

**Proposition 3.5** (algebraic laws for regexps)**:** Define two regexps $E$, $F$ to be *equivalent*, written $E = F$, if $L(E) = L(F)$. The following algebraic laws hold for equivalence of regexps:

1. $E + F = F + E$ and $(E + F) + G = E + (F + G)$ [communtativity and associativity for union]
2. $\varnothing + E = E + \varnothing = E$ [identity law for union]
3. $\Sigma^* + E = E + \Sigma^* = \Sigma^*$ [annihilator law for union]
4. $(EF)G = E(FG)$ [associativity for concatenation]
5. $\boldsymbol{\varepsilon}E = E\boldsymbol{\varepsilon} = E$ [identity law for concatenation]
6. $\varnothing E = E\varnothing = \varnothing$ [annihilator law for concatenation]
7. $E(F + G) = EF + EG$ and $(F + G)E = FE + GE$ [distributive law]
8. $E + E = E$ [idempotence law for union]
9. $E^{**} = E^*$ and $\varnothing^* = \boldsymbol{\varepsilon}$ and $\boldsymbol{\varepsilon}^* = \boldsymbol{\varepsilon}$ and $EE^* = E^*E = E^+$ and $E^+ + \boldsymbol{\varepsilon} = E^*$ [various laws concerning closure]

**Proof:** All these statements can easily be proven, using the standard approach for proving equality of sets: show set inclusion in both directions. As an example, we prove the first equality in 7. First we show that $L(E(F + G)) \subseteq L(EF + EG)$. Let $w \in L(E(F + G))$. Then $w = uv$, with $u \in L(E)$ and $v \in L(F + G)$, that is, $v \in L(F)$ [case 1] or $v \in L(G)$ [case 2]. Case 1 implies that $w = uv \in L(EF)$, case 2 implies $w \in L(EG)$. So we have $w \in L(EF)$ or $w \in L(EG)$, that is, $w \in L(EF) \cup L(EG)$, that is, $w \in L(EF + EG)$. Secondly we show that $L(EF + EG) \subseteq L(E(F + G))$. If $w \in L(EF + EG)$, then $w \in L(EF)$ or $w \in L(EG)$. That is, $w = uv$ with $v \in L(F)$ or $v \in L(G)$. This implies that $w = uv$ with $v \in L(F+G)$, from which we conclude $w \in L(E(F + G))$. □

So far, we have only investigated regexps that contained no language variables. Regexps *with* language variables can denote languages that are not regular if the languages denoted by the language variables are not regular. Trivial example: if the language denoted by the language variable $L_1$ is not regular, then $L_1$ is itself a regex (with language variables) which trivially denotes a non-regular language. Things become more interesting when we consider more complicated regexps $E$, $E'$ with language variables. First we define what it means for two such expressions to be equivalent:

**Definition 3.14** (equivalence of regexps with language variables). Let $E$, $E'$ be two regexps with at most language variables $\boldsymbol{L}_1, ..., \boldsymbol{L}_n$ (that is, $E$ and $E'$ *may* contain these variables, but don't have to). Fix some alphabet $\Sigma$. Then $E$ and $E'$ are $\Sigma$-*equivalent*, written $E =_\Sigma E'$, if for all $\Sigma$-interpretations $I$, the languages denoted by $E$ and $E'$ relative to $I$ are identical, that is, if $L_I(E) = L_I(E')$.

$\Sigma$-equivalence of regexps $E$, $E'$ with language variables can be reduced to equivalence of ordinary regexps through the following proposition:

**Proposition 3.6** (characterizing the languages denoted by regexps with language variables through ordinary regular languages): Let $E$ be a regex with at most language variables $\boldsymbol{L}_1, ..., \boldsymbol{L}_m$. Let $I$ be a $\Sigma$-interpretation. Let $\Sigma' = \{a_1, ..., a_m\}$ be an alphabet disjoint from $\Sigma$. Let $C$ be the regex over $\Sigma'$ that we obtain from $E$ by replacing each occurrence of $\boldsymbol{L}_i$ with symbol $\boldsymbol{a}_i$ ($i = 1, ..., m$). Then the following two statements hold:

1. Every word $w \in L_I(E)$ can be written as $w = w_1...w_k$, where $k$ depends on $w$ and each $w_j$ ($j = 1, ..., k$) is from the language $L_{i_j} = I(\boldsymbol{L}_{i_j})$ ($i_j \in \{1, ..., m\}$), such that, if we replace each subword $w_j$ by $a_{i_j}$, the word $a_{i_1}... a_{i_k}$ is in $L(C)$.
2. Conversely, if $a_{i_1} ... a_{i_k} \in L(C)$, then for any substitution $a_{i_j} \rightarrow w_j$ ($j = 1, ..., k$), where $w_j \in L_{i_j}$, we have $w_1 ...w_k \in L(E)$.

**Example 3.7.** (for part 1.) Let $\Sigma = \{0,1\}$. Let $E = \boldsymbol{L}_1+\boldsymbol{L}_2\boldsymbol{L}_1$. Then $C = \boldsymbol{a}_1+\boldsymbol{a}_2\boldsymbol{a}_1$. We consider an interpretation $I$ that assigns $L_1$ and $L_2$ to the following two languages:

$$I: \qquad L_1 \mapsto L(0^*) \qquad\qquad L_2 \mapsto L(1(01)^*)$$

One word $w \in L_I(E)$ is, for instance, $w = 10100$. Why is $w \in L_I(E)$? Well, it can be segmented as $w = w_1w_2 = 101|00$, where $w_1 = 101 \in L_I(\boldsymbol{L}_2)$ and $w_2 = 00 \in L_I(\boldsymbol{L}_1)$, so

$$w = w_1w_2 \in L_I(\boldsymbol{L}_2\boldsymbol{L}_1) \subseteq L_I(\boldsymbol{L}_1+\boldsymbol{L}_2\boldsymbol{L}_1) = L_I(E).$$

Now we apply part 1 of the proposition. We replace $w_1$ by $\boldsymbol{a}_2$ (because $w_1 \in L_I(\boldsymbol{L}_2)$) and $w_2$ by $\boldsymbol{a}_1$ (because $w_2 \in L_I(\boldsymbol{L}_1)$). The statement of part 1 tells us that $\boldsymbol{a}_2\boldsymbol{a}_1 \in L(C) = L(\boldsymbol{a}_1+\boldsymbol{a}_2\boldsymbol{a}_1)$, which is apparently not a "deep" insight. Actually, this proposition is very shallow – its unwieldiness results from the care one has to invest in getting the indexing notation right.

**Proof of 1.** The proof is an example of a standard proof technique in the theory of formal languages, namely, by *induction over the structure of regexps*. We first show that the statement holds for elementary regexps with langage variables derived by the basis rules 1', 2' (induction basis) and then proceed to show that the statement carries over from constitutents of regexps to more complex regexps derived by the construction rules 3, 4, 5 from Definition 3.13.

*Basis:*

1. Rule 1': $E = \varnothing$. The statement's premise is void and thereby the statement true.
2. Rule 2': $E = \boldsymbol{L}_1$. Let $w \in L_I(E)$, that is, $w \in I(\boldsymbol{L}_1)$. We put $w = w_1$ and get $C = \boldsymbol{a}_1$. Then $a_1 \in L(C)$.

*Induction:* we treat only rule 3 as an example. Assume that $E = (E' + E'')$, and let $w \in L_I(E)$, that is, $w \in L_I(E')$ or $w \in L_I(E'')$. Let $C'$ and $C''$ be the regexps over $\Sigma'$ obtained for $E'$ and $E''$ by substitution of language variables with symbols from $\Sigma'$. In the case $w \in L_I(E')$, we know by induction that $w$ can be written as $w = w_1' ... w_k'$, where each $w_j'$ ($j = 1, ..., k$) is from the language $L_{i_j} = I(\boldsymbol{L}_{i_j})$ ($i_j \in \{1, ..., m\}$), such that, if we replace each subword $w_j'$ by $a_{i_j}'$, the word $a_{i_1}' ... a_{i_k}'$ is in $L(C')$ and therefore also in $L(C') \cup L(C'') = L(C' + C'')$. Similarly, if $w \in L_I(E'')$, we can find $w = w_1'' ... w_l''$, such that the substitute $a_{i_1}'' ... a_{i_l}''$ is in $L(C'')$ and thus in $L(C' + C'')$. But $(C' + C'')$ is the the regex over $\Sigma \cup \Sigma'$ that we obtain from $(E' + E'')$ by replacing each occurrence of $\boldsymbol{L}_i$ with symbol $\boldsymbol{a}_i$ ($i = 1, ..., m$), and thus we are done.

The **proof of 2.** is carried out in a similar fashion. □

**Corollary 3.7** Let $\Sigma$ be an alphabet with at least two symbols and let $E$, $E'$ be two regexps with at most language variables $\boldsymbol{L}_1, ..., \boldsymbol{L}_m$. Let $C$, $C'$ be the substitute ordinary regexps defined as in proposition 3.6. Then $E =_\Sigma E'$ iff $L(C) = L(C')$.

**Proof.** "if": Let $L(C) = L(C')$. Let $I$ be a $\Sigma$-interpretation. By Proposition 3.6 (1), let $w_1 ... w_k \in L_I(E)$ such that $a_{i_1} ... a_{i_k} \in L(C) = L(C')$. By Proposition 3.6 (2), $w_1 ... w_k \in L_I(E')$. Thus, $L_I(E) \subseteq L_I(E')$. By a symmetric argument, we also get $L_I(E') \subseteq L_I(E)$ and thus $L_I(E) = L_I(E')$. Since $I$ was chosen arbitrarily, we find $E =_\Sigma E'$.
"only if": This is the only not-altogether-boring part in all of this ado about $E$'s and $C$'s. Let $E =_\Sigma E'$. Let $a_{i_1} ... a_{i_k} \in L(C)$. Because $\Sigma$ has at least two symbols, we can code the $\Sigma'$ symbols $a_1, ..., a_m$ by uniquely identifying words $v_1, ..., v_m \in \Sigma^k$ for some $k$. Consider an interpretation $I^*$ that assigns to every language variable $\boldsymbol{L}_i$ the language $\{v_i\}$. Then by Proposition 3.6 (2), $w = v_{i_1} ... v_{i_k} \in L_{I^*}(E)$. The unique segmentation of $w$ into subwords from the languages $\boldsymbol{L}_i$ is $w = v_{i_1} ... v_{i_k}$. Because $E =_\Sigma E'$, we have $w \in L_{I^*}(E')$ and thus by Proposition 3.6 (1) and the uniqueness of the segmentation we obtain $a_{i_1} ... a_{i_k} \in L(C')$. Thus, $L(C) \subseteq L(C')$. By symmetry, $L(C') \subseteq L(C)$ and therefore $L(C) = L(C')$.□

**Example 3.8.** Test whether $L_1+L_2L_1 =_\Sigma (L_1+L_2)\, L_1$. Our strategy gives us $C = a_1+a_2a_1$ and $C' = (a_1+a_2)a_1$. We see immediately that $a_1 \in L(C)$ but $\notin L(C')$, and thus the Corollary tells us that $L_1+L_2L_1 \neq_\Sigma (L_1+L_2)\, L_1$.

## 3.4 The pumping lemma for regular languages

The pumping lemma condenses a simple observation about DFAs into an immensely useful statement about a certain syntactical property that all regular languages have. Its main use is to show that some given language is not regular, by showing that the language in question does not have this property.

**Proposition 3.8** (pumping lemma): Let $L$ be a regular language. Then there exists a constant $n$ (depending on $L$), such that $\forall w \in L$, $|w| \geq n$, we can find a partition $w = xyz$, such that (1) $y \neq \varepsilon$, (2) $|xy| \leq n$, and (3) $\forall k \geq 0$, $xy^kz \in L$. In intuitive terms, every word $w$ from $L$ that exceeds $n$ in length can be "pumped" by replicating an inner part, such that the "pumped-up" words are also in $L$.

**Proof**: Because $L$ is regular, there exists some DFA $A = (Q, \Sigma, \delta, q_0, F)$ that accepts $L$. Let $A$ have $n$ states. Observe that in an accepting run of any word $w = x_1x_2...x_k$ of length at least $n$, at least one state (maybe the start state) must have been visited twice after $x_n$ has been read. Let $p$ be this state and let $x_ix_{i+1}...x_j$ be the subword before which the run went through $p$ and after which it went through $p$, that is, $\hat{\delta}(p, x_ix_{i+1}...x_j) = p$. Put $x = x_1x_2...x_{i-1}$ and $y = x_ix_{i+1}...x_j$. It is clear that the statement holds. □

**Example 3.9**. The language $L = \{0^m1^m \mid m \geq 0\}$ is not regular. Show this by the pumping lemma as follows. Assume that $L$ is regular. Let $n$ be the "pumping lemma constant" assured by the pumping lemma. Consider the word $w = 0^n1^n$. By the lemma, it can be split into $w = xyz$, with $|xy| \leq n$, that is, $xy$ is made only from 0's. By the lemma, $y$ is non-empty, say it is $0^k$, $k > 0$. Again by the lemma, also $0^{n+k}1^n \in L$ would follow, contradiction.

**Example 3.10**. The language $\{w \in \{0,1,(,)\} \mid$ the occurrences of "(" and ")" in $w$ are balanced$\}$ is not regular. This can be shown as in the previous example (pump the word $(^n)^n$ to obtain the contradiction).

*Note.* Generally speaking, any language whose words need some counting-and-remembering-counts scheme is not regular. Counting parentheses for balancing is an important example. The intuitive reason is that DFAs cannot maintain a "count" – they simply have no memory where they could store the count of the opened parentheses. As a consequence, programming languages (that is, the formal languages whose words are well-formed programs of some type, e.g. C) are not regular, because all programming languages need to count parentheses.

A consequence of the pumping lemma is that we can decide whether a regular language (that is given to us by a DFA) is empty, finite, or infinite:

**Proposition 3.9.** For a DFA $A$, it is decidable whether $L(A)$ is (i) non-empty, or (ii) infinite.

**Proof.** First determine a "pumping constant" $n$ from the DFA (take $n$ to be the number of states of $A$).

(i) We claim that $L(A)$ is (i) non-empty iff $A$ accepts a word of length $< n$ (and you can check them all with by exhaustive search). "if" is obvious. "only if": Suppose $A$ accepts a non-empty language. Let $w$ be an accepted word that has shortest length among all accepted words. Then $|w| < n$, for if $|w| \geq n$, then by the pumping lemma $w = xyz$ such that $y \neq \varepsilon$ and $xz \in L(A)$, contradicting the assumption that $w$ was shortest possible.

(ii) We claim that $L(A)$ is infinite iff $A$ accepts a word $w$ of length $l$, where $n \leq l < 2n$ (and you can check all such $w$ by exhaustive search). "if": $w$ of length $l$ can be pumped to infinitely many pumped-up versions, all in $L(A)$, by the pumping lemma. "only if": Assume that $L(A)$ is infinite. Then there exists some word $w \in L(A)$ of length $|w| \geq n$. If $|w| < 2n$, we are done. In the case $|w| \geq 2n$, using the PL we split $w$ into $w = xyz$, with $1 \leq |y| \leq n$, such that $xy \in L(A)$. We have thus reduced the length of $w$ by some length $L$ between 1 and $n$. As long as the shortened word $xy$ still has length $\geq 2n$, we can repeat this shortening by lengths between 1 and $N$. Clearly after a finite number of such shortenings, we obtain a word in $L(A)$ whose length $L$ satisfies $n \leq L < 2n$. □

## 3.5 Closure properties for regular languages

We now consider *closure properties* for regular languages. Generally, a closure property is of the form "if $L$ and $M$ are regular and $K$ is constructed from $L$ and $M$ by some constructor $C$, then $K$ is also regular". We say, the regular languages are *closed under* $C$. First we collect some elementary closure properties.

**Proposition 3.10.** Let $L$ and $M$ be regular languages over $\Sigma$. Then, the following languages are also regular:

(1) $L \cup M$,
(2) $L \cap M$,
(3) $L^c$ $[= \Sigma^* \setminus L]$,
(4) $L \setminus M$,
(5) $L^R = \{x_1 ... x_n \mid x_n ... x_1 \in L\}$,  %% superscript R stands for "reverse"
(6) $L^*$,
(7) $LM$.

**Proof:** (1), (6), (7) follow from regular expressions and their equivalence to regular languages.
(3) can be shown by exchanging accepting with non-accepting states in a DFA accepting $L$.
(2) follows from (1) and (3) by the set-theoretic rule $L \cap M = (L^c \cup M^c)^c$.
(4) follows from (1) and (2) by the set-theoretic rule $L \setminus M = (L \cap M^c)$.
(5) can be shown by a straightforward induction over the structure of regular expressions. □

**Definition 3.15** (homomorphisms and inverse homomorphisms on words).

1. Let $\Sigma_1$, $\Sigma_2$ be two alphabets. A *homomorphism* $h$ is any function $h$: $\Sigma_1 \to \Sigma_2^*$. Such $h$ induces another function (also called $h$) from $\Sigma_1^*$ to $\Sigma_2^*$, by putting $h(\varepsilon) = \varepsilon$ and $h(x_1...x_n) = h(x_1)... h(x_n)$. For $L \subseteq \Sigma_1^*$, $h(L) = \{h(w) \in \Sigma_2^* \mid w \in L\}$.
2. Let $h$: $\Sigma_1 \to \Sigma_2^*$ be a homomorphism. Then the *inverse homomorphism* is a function $h^{-1}$: $\Sigma_2^* \to \mathrm{Pot}(\Sigma_1^*)$ defined by $w \in h^{-1}(v)$ iff $h(w) = v$. For $M \subseteq \Sigma_2^*$,

$$h^{-1}(M) = \bigcup_{v \in M} h^{-1}(v) = \{w \in \Sigma_1^* \mid h(w) \in M\}.$$

**Proposition 3.11.** (closure of regular languages under homomorphisms and inverse homomorphisms). Let $h$: $\Sigma_1 \to \Sigma_2^*$ be a homomorphism. Then

1. if $L \subseteq \Sigma_1^*$ is regular, so is $h(L)$,
2. if $M \subseteq \Sigma_2^*$ is regular, so is $h^{-1}(M)$.

Proof. (1) Replace in an ε-NFA accepting $L$ each symbol label $x_i$ by $h(x_i)$ to obtain a generalized ε-NFA accepting $h(L)$ [revisit proof of Proposition 3.3 for the notion of a generalized ε-NFA]. (2) Let $A$ be a DFA accepting $M \subseteq \Sigma_2^*$. Re-use $A$ except for the transition function $\delta_A$, to create a DFA $B$ accepting $h^{-1}(M)$, by putting $\delta_B(q, x) = \hat{\delta}_A(q,h(x))$. Then $B$ accepts $w$ iff $A$ accepts $h(w)$, that is, $B$ accepts $h^{-1}(M)$. □

## 3.6 The Myhill-Nerode theorem and minimization of DFAs; decision properties of regular languages

So far, we have characterized the regular languages through a variety of automata, and through regular expressions. In this subsection we learn about yet another way of characterizing them – through abstract algebraic properties. The proof of the central theorem (Myhill-Nerode) will provide us with the insights necessary to transform some DFA in an equivalent DFA with the smallest possible number of states.
Recall: a binary relation $R$ on a set $X$ is an *equivalence relation* iff (1) $R$ is reflexive, that is $xRx$ for all $x \in X$; (2) $R$ is symmetric, that is, $xRy \Rightarrow yRx$ for all $x, y \in X$; (3) $R$ is transitive, that is, $xRy$ and $yRz \Rightarrow xRz$ for all $x, y, z \in X$. An equivalence relation $R$ on $X$ partitions $X$ into subsets $X_i$ (where $i \in I$) called the *equivalence classes* of $R$, defined by (1) for all $i \in I$, for all $x, y \in X_i$, it holds that $xRy$, (2) for all $i,j \in I$, $i \neq j$, for all $x \in X_i$, $y \in X_j$, it holds that not $xRy$. The number $|\, I \,|$ of equivalence classes is the *index* of $R$. If $C \subseteq X$ is an equivalence class containing $x$, we call $x$ a *representative* of $C$ and write $C = [x]_R$.

**Definition 3.16.** An equivalence relation $R$ on $\Sigma^*$ is *right invariant* if for all $u, v, w \in \Sigma^*$, it holds that $uRv \Rightarrow uwRvw$.

**Definition 3.17.** Let $A$ be a DFA with starting state $q_0$. Define a binary relation $R_A$ on $\Sigma^*$ by $uR_Av$ iff $\delta(q_0, u) = \delta(q_0, v)$. This is clearly an equivalence relation.

Obviously $R_A$ is right-invariant, has finite index, and $L(A)$ is the union of some of the equivalence classes of $R_A$ (namely those classes $[v]_{R_A}$ for which $\delta(q_0, v)$ leads into an accepting state). This insight is considerably refined in the following theorem.

**Proposition 3.12** (The Myhill-Nerode theorem; algebraic characterization of regular languages). Let $L \subseteq \Sigma^*$ be a language. The following three statements are equivalent:

1. $L \subseteq \Sigma^*$ is regular, that is, $L = L(A)$ for some DFA $A$.
2. $L$ is the union of some of the equivalence classes of a right-invariant equivalence relation $R$ on $\Sigma^*$ of finite index.
3. The equivalence relation $R_L$ on $\Sigma^*$ defined by:

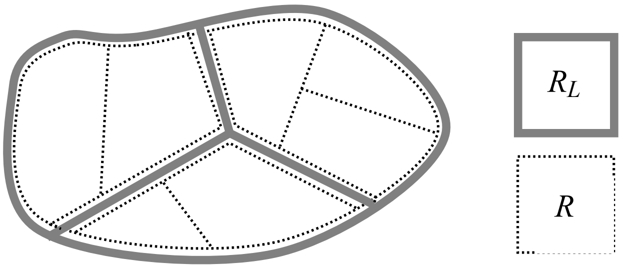
   $uR_Lv$ iff for all $w \in \Sigma^*$: $uw \in L \Leftrightarrow vw \in L$.

   is of finite index.

The proof of this proposition involves the construction of minimal automata, which is an important technique in its own right.

*Proof:* 1. $\Rightarrow$ 2: We saw that already.
2. $\Rightarrow$ 3: We show that any equivalence relation $R$ satisfying 2. is a refinement of $R_L$, that is, every equivalence class of $R$ is entirely contained in some equivalence class of $R_L$. Thus the

index of $R_L$ cannot be greater than the index of $R$ and therefore is finite. Assume $uRv$. Since $R$ is right invariant, for every $w$ in $\Sigma^*$ we have $uwRvw$. If $uw \in L$, then by 2. $vw \in L$ and vice versa. Therefore $uR_Lv$, so $R$ is a refinement of $uR_Lv$. The figure below illustrates an equivalence relation $R$ being a refinement of an equivalence relation $R_L$.



3. $\Rightarrow$ 1: We first show that $R_L$ is right invariant. Suppose that $uR_Lv$. For any $y \in \Sigma^*$ we must show that $uyR_Lvy$, that is, for any $z \in \Sigma^*$, $uyz \in L \Leftrightarrow vyz \in L$. But this follows from the definition of $R_L$ in 3., if we put $w = yz$.
We now construct a DFA $A_{\mathrm{MN}}$ accepting $L$. The trick is to use as states the equivalence classes of $R_L$ (of which there is only a finite number!). So $A_{\mathrm{MN}} = (Q, \Sigma, q_0, \delta, F) = (\{[w]_{R_L} \mid w \in \Sigma^*\}, \Sigma, q_0, \delta_{\mathrm{MN}}, F_{\mathrm{MN}})$. We put

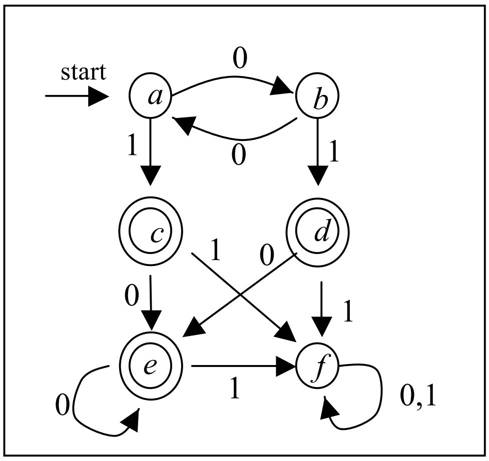
I. $q_0 = [\varepsilon]_{R_L}$,
II. for all $w \in \Sigma^*$, $a \in \Sigma$: $\delta_{\mathrm{MN}}([w]_{R_L}, a) = [wa]_{R_L}$,
III. $F_{\mathrm{MN}} = \{[u]_{R_L} \mid u \in L\}$.

We first have to show that this is well-defined. First we show that the definition of $\delta_{\mathrm{MN}}$ is independent of the choice of representative $w$ of the equivalence class $[w]_{R_L}$. So we have to show that if $wR_Lv$, then $[wa]_{R_L} = \delta_{\mathrm{MN}}([w]_{R_L}, a) = \delta_{\mathrm{MN}}([v]_{R_L}, a) = [va]_{R_L}$. But if $wR_Lv$, then $waR_Lva$ because $R_L$ is right-invariant, therefore $[wa]_{R_L} = [va]_{R_L}$. So $\delta_{\mathrm{MN}}$ is well-defined.

Next we show that the definition of $F_{\mathrm{MN}}$ is independent of the choice of representative $u$ of the equivalence class $[u]_{R_L}$, that is, if $u \in L$ and $v \in [u]_{R_L}$, then $v \in L$. This follows from 3. if we put $w = \varepsilon$.

Finally we have to show that $L = L(A_{\mathrm{MN}})$. This follows from $\hat{\delta}_{\mathrm{MN}}([\,\varepsilon]_{R_L}, w) = [w]_{R_L}$ (an easy induction over $|w|$) and III.

**Example 3.11.** Here is an example that illustrates the Myhill-Nerode theorem. Let $L$ be the language $0^*10^*$, that is, all words over $\{0,1\}$ that contain exactly one 1. $L$ is accepted, for instance, by the DFA $A$ shown in the following figure.



This DFA $A$ yields a right-invariant equivalence relation $R_A$ as follows. $R_A$ has six equivalent classes corresponding to the states of $A$. These equivalence classes each contain the words that lead from the starting state into that class. By visual inspection and a little thinking, we find that the six word sets (= languages) leading into the six states are the following:

| | | |
|---|---|---|
| $C_a$ = (00)* | $C_b$ = (00)*0 | $\underline{C_c = (00)^*1}$ |
| $\underline{C_d = (00)^*01}$ | $\underline{C_e = 0^*100^*}$ | $C_f$ = 0*10*1(0+1)* |

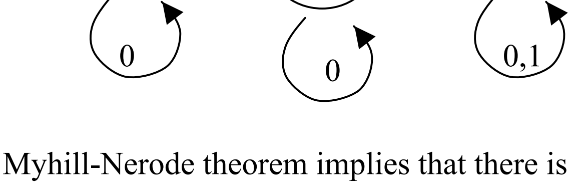
$L$ is the union of $C_c$, $C_d$ and $C_e$ (underlined).

If one thinks a little about the nature of $L$ (i.e., being all words with exactly one 1), one finds that the relation $R_L$ for $L$ has $wR_Lv$ if and only if either

$w$ and $v$ have no 1's [= equivalence class $D_1$],
$w$ and $v$ each have one 1 [= equivalence class $D_2$], or
$w$ and $v$ each have more than one 1 [= equivalence class $D_3$].

The classes $C_i$ refine the classes $D_j$ by $D_1 = C_a \cup C_b$, $D_2 = C_c \cup C_d \cup C_e$, and $D_3 = C_f$.

We may denote the three equivalence classes of $R_L$ by $D_1 = [\varepsilon]_{R_L}$= 0*, $D_2 = [1]_{R_L}$= 0*10*, $D_3 = [11]_{R_L}$= 0*10*1(0+1)*. The DFA $A_{\mathrm{MN}}$ we get according to the construction in the proof of the Myhill-Nerode theorem is the following:



The Myhill-Nerode theorem implies that there is an essentially unique minimal DFA for a given regular language:

**Proposition 3.13.** If $A$, $B$ are DFAs accepting $L$ and $A$ and $B$ both have minimal number of states (that is, no other DFA accepting $L$ has fewer states), then $A$ is identical to $B$ up to trivial renaming of states, that is, $A$ is *isomorphic* to $B$. Thus it makes sense to speak of *the* minimal DFA for $L$. The minimal DFA for $L$ is the DFA $A_{\mathrm{MN}}$ described in the proof of the Myhill-Nerode theorem.

**Proof.** In the proof of the Myhill-Nerode theorem we saw (in 2. ⇒ 3) that any DFA accepting $L$ creates, via its states, an equivalence relation that is a refinement of $R_L$. Thus the DFA $A_{\mathrm{MN}}$ has minimal state number. We show that any other DFA $B$ for $L$ with the same number of states as $A_{\mathrm{MN}}$ is isomorphic to $A_{\mathrm{MN}}$. We show how the states of $B$ can be identified with the states of $A_{\mathrm{MN}}$. Let $q$ be a state of $B$. There must some $w \in L$ such that $\delta_B(q_0, w) = q$, because otherwise $q$ would be *inaccessible* and could be removed from $B$, contradicting minimality of $B$. Identify $q$ with the state $[w]_{R_L}$ of $A_{\mathrm{MN}}$. This identification does not depend on $w$: if $v$ is another word such that $\delta_B(q_0, v) = q$, and we identify $q$ with $[v]_{R_L}$, then $[v]_{R_L} = [w]_{R_L}$ because obviously $wR_Lv$. Using this identification of states between $B$ and $A_{\mathrm{MN}}$, it is straightforward to see that also the transitions between corresponding states must correspond 1—1. □

The Myhill-Nerode theorem and Proposition 3.13 tell us that there exists an essentially unique minimal DFA, but don't inform us about how we can effectively transform any DFA into its minimal-state form. This gap is closed by the following

**"Table-filling" algorithm** to transform some DFA $A = (Q, \Sigma, \delta, q_0, F)$ into the minimal-state DFA $A_{\mathrm{MN}} = (\{[w]_{R_L} \mid w \in \Sigma^*\}, \Sigma, \delta_{\mathrm{MN}}, [\,\varepsilon]_{R_L}, F_{\mathrm{MN}})$.

**Preliminaries.** (i) On the states $Q$ define an equivalence relation by $p \equiv q$ iff for all $w \in \Sigma^*$, $\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$. Call $p$ and $q$ *distinguishable* if not $p \equiv q$.
(ii) Remove from $A$ all inaccessible states, if they exist.

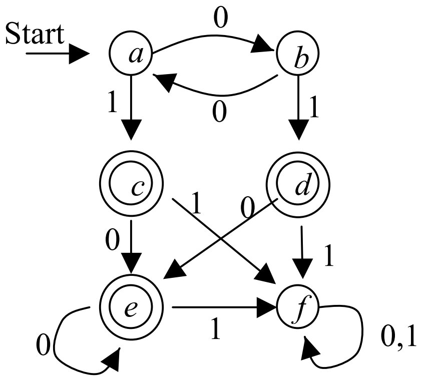
**Step 1:** For all pairs $(p, q)$ of different $p$, $q$ from $Q$, determine whether they are distinguishable, by the *table-filling method*. This is an iterative method. In iteration 1, mark all pairs $(p, q)$ as distinguishable where $p$ is accepting and $q$ is not, or vice versa. In iteration $n$, mark any previously unmarked pair $(p, q)$ as distinguishable if you find a symbol $a \in \Sigma$, such that for some previously found pair $(p', q')$ of distinguishable states you observe $\delta(p, a) = p'$ and $\delta(q, a) = q'$ (or same with $p'$, $q'$ exchanged). Stop if an iteration does not yield new pairs of distinguishables.

**Interpretation of step 1:** $p$, $q$ are distinguishable iff they can be reached by words that are not equivalent w.r.t. $R_L$ and therefore give rise to different states in $A_{\mathrm{MN}}$. If they are not distinguishable, they can simply be joined within $A$, and a union of pairwise indistinguishable states makes a state in $A_{\mathrm{MN}}$.

**Step 2:** Construct $A_{\mathrm{MN}}$ as follows. Define states $[q]_\equiv$ as the $\equiv$ equivalence classes found in step 1, where $q$ is some representative $A$-state for the equivalence class. Let $v$ be a word that leads into $q$. Rewrite $[q]_\equiv$ as $[v]_{R_L}$ and take it as a state of $A_{\mathrm{MN}}$. Establish $\delta_{\mathrm{MN}}$ by putting $\delta_{\mathrm{MN}}([v]_{R_L}, a) = [va]_{R_L}$, where $[va]_{R_L}$ corresponds to the equivalence class $[\hat{\delta}(q_0, va)]_\equiv$. Put $F_{\mathrm{MN}} = \{[v]_{R_L} \mid \hat{\delta}(q_0, v) \in F\}$. □

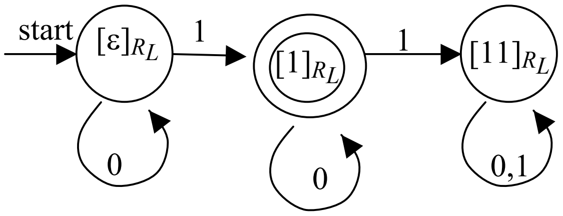
It can be easily shown that this method works correctly, that is, step 1 indeed constructs $R_L$ from $A$. The computational cost is dominated by step 1. A careful implementation (see HMU p. 159) needs O($|Q|^2$) pair checks altogether.

**Example 3.12.** Consider again the DFA from example 3.11, which I present again for convenience (left):



|  |  |  |  |  |  |
|---|---|---|---|---|---|
| *b* |  |  |  |  |  |
| *c* | $x_1$ | $x_1$ |  |  |  |
| *d* | $x_1$ | $x_1$ |  |  |  |
| *e* | $x_1$ | $x_1$ |  |  |  |
| *f* | $x_2$ | $x_2$ | $x_1$ | $x_1$ | $x_1$ |
|  | *a* | *b* | *c* | *d* | *e* |

The table on the right marks the state pairs found distinguishable in the first round of table filling by $x_1$, and in the second (and last) round by $x_2$. From that table we see that there are three classes of mutually indistinguishable states: $[a]_\equiv = \{a, b\} = [\varepsilon]_{R_L}$, $[c]_\equiv = \{c,d,e\} = [1]_{R_L}$, and $[f]_\equiv = \{f\} = [11]_{R_L}$. Carrying out step 2 gives the familiar minimal DFA:



Our ability to effectively construct a (unique) minimal DFA from any given DFA leads to a conceptually simple method for deciding whether two regular languages (described by automata or regexps) are the same:

**Proposition 3.14** (decidability of sameness of regular languages): Given given two regular languages $L$, $M$ (specified by DFAs, NFAs, ε-NFAs or regexps) it is decidable whether $L = M$.

**Proof:** First construct DFAs $A_L$ and $A_M$ for $L$ and $M$ (we know how to derive DFAs from NFAs or ε-NFAs by subset constructions, and we know how to obtain an ε-NFA from a regex). Then minimize the DFAs $A_L$ and $A_M$, obtaining $A_{L_{min}}$ and $A_{M_{min}}$. Then check whether $A_{L_{min}}$ and $A_{M_{min}}$ are identical (up to renaming of states). □

**Corollary 3.15**: If $L$, $L_1$, ..., $L_n$, $M$ are regular languages given by DFAs, NFAs, ε-NFAs or regexps, then it is decidable whether

1. $L = \emptyset$,
2. $w \in L$ (for any word $w$),
3. $L \subseteq M$,

4. $f(L_1, ..., L_n) = M$, where $f$ is some *effective* function that maps $n$ regular languages to regular languages, that is, $f$ takes descriptions of $L_1, ..., L_n$ in terms of automata or regexps and returns an automaton or regex.

**Proof:** 1. is a special case of Proposition 3.14. It can also decided by an application of the pumping lemma: it implies that if $A$ is an $k$-state DFA for $L$, then $L = \emptyset$ iff $A$ does not accept any word of length $< k$. Furthermore, it can be decided by checking whether in $A$ some accepting state is reachable from the starting state.
2. is also a special case of Proposition 3.14: decide whether $\{w\} \cap L = \{w\}$. Of course, a faster decision process is to check whether a DFA $A$ for $L$ accepts $w$.
3. is another special case of Proposition 3.14: decide whether $L \cap M = L$. (Use Proposition 3.10 to derive a DFA for $L \cap M$.)
4. is yet another special case of Proposition 3.14.

Note. You might find the claims of this corollary rather trivial. However, we will soon learn about classes of languages where it is not at all easy (or even possible) to decide, for instance, whether $L = \emptyset$.

**Things you must absolutely remember about regular languages until you retire, die, or quit computer science:**

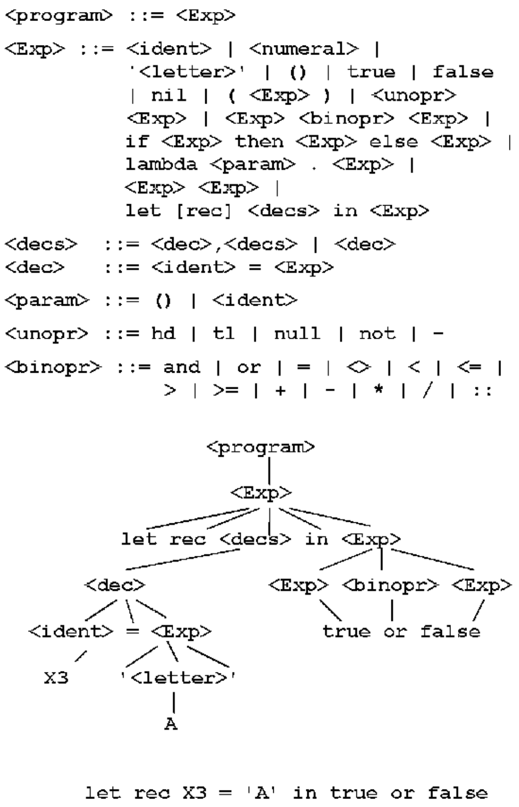
1. Regular languages can be defined concretely by various finite automata, regular expressions, and abstractly by the Myhill-Nerode theorem.
2. All concrete characterizations can effectively be transformed into each other.
3. Computer implementations often use DFAs to represent regular languages.
4. Regular languages are closed under a large variety of operations.
5. The pumping lemma helps to check whether some language is *not* regular.
6. Each regular language has a unique minimal DFA.
7. The unique minimal DFA can be effectively constructed, based on insights from the Myhill-Nerode theorem.
8. Unique minimal DFAs for regular languages are a case of a *unique normal form representation*. Whenever in mathematics some mathematical object has a unique normal form representation, a powerful theory with strong decidability results emerges.

# 4. Context-free languages and their grammars

Regular languages are very simple. The good side of this simplicity is that these languages are easy to analyze (e.g. in terms of closure or decidability properties). The downside is that many formal languages of practical interest are not regular – especially, programming languages. We saw that regular languages cannot "count" within words – therefore, any (programming) language that needs some counting/bookkeeping, e.g. for matching parentheses, cannot be regular. In this section we will generalize regular languages such that they include a "counting" mechanism, arriving at the *context-free* languages (CFLs). CFLs can be described by a generalization of DFAs, *pushdown automata*. However, this is not the most commonly used format to specify CFLs, so we will defer pushdown automata for a while. We start our treatment of CFLs by showing how they can be specified through *grammars*.

## 4.1 Basic definitions

**Example 4.1.** You are already familar with grammars for specifying programming languages from the General CS lecture. The grammars that you saw (likely in "Backus-Naur" notation) are actually grammars for context-free languages; and the grammars themselves are *context-free grammars* (CFGs). Here is a CFG for a toy programming language (the functional language L5 from http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/ ), a *parse tree* for a simple word ( = program) from that language, and the word itself at the bottom.



```
<program> ::= <Exp>

<Exp> ::= <ident> | <numeral> |
          '<letter>' | () | true | false
          | nil | ( <Exp> ) | <unopr>
          <Exp> | <Exp> <binopr> <Exp> |
          if <Exp> then <Exp> else <Exp> |
          lambda <param> . <Exp> |
          <Exp> <Exp> |
          let [rec] <decs> in <Exp>

<decs>   ::= <dec>,<decs> | <dec>
<dec>    ::= <ident> = <Exp>

<param> ::= () | <ident>

<unopr> ::= hd | tl | null | not | -

<binopr> ::= and | or | = | <> | < | <= |
             > | >= | + | - | * | / | ::
```

```
let rec X3 = 'A' in true or false
```

**Fig. 4.1:** A portion of a context-free grammar (rules for <ident> and <letter> are omitted), a parse tree, and its "yield".

**Definition 4.1.** (CFG) A *context-free grammar* (CFG) is a quadruple $G = (V, T, P, S)$, where

1. $V$ is a finite set of *variables* (also called *nonterminals*),
2. $T$ is a finite set of *terminals* (or *terminal symbols*, corresponding to the alphabet $\Sigma$ in the world of regular languages),
3. $S \in V$ is a *start symbol*, and
4. $P$ is a finite set of *productions* (also *rules*), each of the form $A \to \alpha$, where $A$ is a variable and $\alpha$ is a word from $(V + T)^*$ (that is, a string of variables and/or terminals, possibly the empty string $\varepsilon$). $A$ is called the *head* and $\alpha$ the *body* of the rule.

Shorthand notation for several rules $A \to \alpha_i$ with same head $A$: $A \to \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$.

Notational conventions: use

$A$, $B$, ... for variables,
$a$, $b$, ..., 0, 1, ..., +, -, ... (, ),... for terminals,
$u$, $v$, $w$, $x$, $y$, $z$ for strings of terminals,
$X$, $Y$, ... for symbols that are either variables or terminals,
$\alpha$, $\beta$, ... for strings containing variables and/or terminals.

If one has a string $\alpha$ made from terminals and variables, and one applies one production to one of the variables, obtaining a string $\beta$, one has *derived* $\beta$ from $\alpha$ via the grammar. This concept and its transitive closure are now formally introduced:

**Definition 4.2.** (derivations and sentential forms) For a CFG $G = (V, T, P, S)$, define a binary relation

$$\Rightarrow_G \subseteq (V + T)^* \times (V + T)^*$$

by $\alpha \Rightarrow_G \beta$ iff $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$, where $\alpha_1, \alpha_2 \in (V + T)^*$ and $A \to \gamma \in P$.

That is, $\alpha \Rightarrow_G \beta$ iff a single variable in $\alpha$ is replaced by a string according to a rule from $P$. Define

$$\Rightarrow^*_G \subseteq (V + T)^* \times (V + T)^*$$

by $\alpha \Rightarrow^*_G \beta$ iff there exists a sequence $\gamma_1, \gamma_2, ..., \gamma_n$, such that $n \geq 1$, $\alpha = \gamma_1$, $\beta = \gamma_n$, and $\gamma_i \Rightarrow_G \gamma_{i+1}$ for $i < n$.

Note that this includes $\alpha \Rightarrow^*_G \alpha$ as a special case ($n = 1$). If $G$ is understood, write $\alpha \Rightarrow \beta$ instead of $\alpha \Rightarrow_G \beta$. If $\alpha \Rightarrow^*_G \beta$ by $\gamma_1, \gamma_2, ..., \gamma_n$, then $\beta$ can be *derived* from $\alpha$, and $\alpha = \gamma_1 \Rightarrow_G \gamma_2 \Rightarrow_G ..., \Rightarrow_G \gamma_n = \beta$ is *a derivation* of $\beta$ from $\alpha$.

Any string $\alpha$ that can be derived from the start symbol $S$ is called a *sentential form*.

**Definition 4.3** (languages of variables and CFGs). For a CFG $G = (V, T, P, S)$ and a variable $A \in V$, let $L(A) = \{w \in T^* \mid A \Rightarrow^*_G w\}$ be the *language of A*. If $A = S$ is the start symbol, $L(S)$ is the language of $G$. If $L \subseteq T^*$ is a language of a CFG, we say that $L$ is a *context-free language*, or CFL.

**Definition 4.4** (leftmost / rightmost derivations). A derivation $\alpha = \gamma_1 \Rightarrow_G \gamma_2 \Rightarrow_G \; ..., \Rightarrow_G \; \gamma_n = \beta$ is a *leftmost* (*rightmost*) derivation, if in each step the leftmost (rightmost, respectively) variable in a string $\gamma_i$ is replaced by a rule to yield $\gamma_{i+1}$. We write $\alpha \Rightarrow*_{lm} \beta$ ($\alpha \Rightarrow*_{rm} \beta$, respectively) if there exists a leftmost (rightmost) derivation of $\beta$ from $\alpha$.

**Example 4.2.** (for a derivation). Consider the CFG with $V = \{E, I\}$ and $T = \{a, b, 0,1,+,*,(,)\}$, where $E$ is the start symbol, represented by the following rules:

1. $E \Rightarrow I$
2. $E \Rightarrow E + E$
3. $E \Rightarrow E * E$
4. $E \Rightarrow (E)$
5. $I \Rightarrow a$
6. $I \Rightarrow b$
7. $I \Rightarrow Ia$
8. $I \Rightarrow Ib$
9. $I \Rightarrow I0$
10. $I \Rightarrow I1$

This CFG describes simple arithmetic expressions ("*E*") made up from identifiers ("*I*"). A derivation for instance $a * (a + b00)$ is:

$$\begin{array}{lll}
E & \Rightarrow & \\
 & \Rightarrow & E * E \\
 & \Rightarrow & E * (E) \\
 & \Rightarrow & E * (E + E) \\
 & \Rightarrow & E * (E + I) \\
 & \Rightarrow* & I * (I + I) \\
 & \Rightarrow & I * (I + I0) \\
 & \Rightarrow & I * (I + I00) \\
 & \Rightarrow & I * (I + b00) \\
 & \Rightarrow* & a * (a + b00),
\end{array}$$

where at two places we put two identical $\Rightarrow$ derivations into a single $\Rightarrow*$.

**Definition 4.5** (parse trees and their yields). Let $G = (V, T, P, S)$ be a CFG. An ordered tree[3] $t$ whose nodes are labelled with symbols from $V \cup T \cup \{\varepsilon\}$ is a *parse tree* of $G$, if

1. each interior node and the root node is labelled with a variable symbol,
2. each leaf node is labelled with a variable, a terminal, or $\varepsilon$,
3. if an interior or the start node is labelled with $A$ and its children with $X_1, ..., X_n$ (from left to right), then $A \Rightarrow X_1...X_n$ is a production from $P$.

The *yield* of a parse tree is the word obtained by reading all its leaf symbols, from left to right (see example in Fig. 4.1). Note: the yield of a parse tree may contain variables.

[3] A tree is ordered if for each nonterminal node, there is an order defined on its children – that is, we can read the children "from left to right".

Parse trees are the most intuitive and "visually complete" form to represent derivations. Furthermore, they are the basic format in which the results of parsing programs are stored for subsequent use by compilers and interpreters. Each derivation leads to a parse tree in a natural way, and parse trees can be read as leftmost (or rightmost) derivations if they are traversed in an appropriate way. These elementary insights are collected in the following proposition:

**Proposition 4.1.** Given a CFG $G$, a variable $A$ and a word $\alpha$ of terminals and/or variables, the following are equivalent:

1) $A \Rightarrow^* \alpha$,
2) $A \Rightarrow^*_{lm} \alpha$,
3) $A \Rightarrow^*_{rm} \alpha$,
4) there is a parse tree with root $A$ and yield $\alpha$.

**Proof**: 2. ⇒ 1. and 3. ⇒ 1. are trivial. We sketch 1. ⇒ 4. and 4. ⇒ 2. (4. ⇒ 3. is symmetrical).

"1. ⇒ 4.": Given an $n$-step derivation $A \Rightarrow^* \alpha$, where the $i$-th step is $\alpha^i \Rightarrow_G \beta^i$ with $\alpha^i = \alpha_1{}^i A^i \alpha_2{}^i$ and $\beta^i = \alpha_1{}^i \gamma^i\ \alpha_2{}^i$, construct parse trees $t_i$ for the sentential form $\beta^i$ obtained after the $i$-th step of the derivation, by induction on the length of derivations. *Basis*: Put $t_1$ as the tree consisting only of root node $A$ and leaves from string $\beta^1$. It is clear that $t_1$ is a parse tree with yield $\beta^1$. *Induction*: Assume that $t_i$ is a parse tree for the sentential form $\beta^i$ obtained after the $i$-th step of the derivation. Consider the $i$+1-th derivation $\beta^i = \alpha^{i+1} \Rightarrow_G \beta^{i+1}$ with $\alpha^{i+1} = \alpha_1{}^{i+1} A^{i+1}\ \alpha_2{}^{i+1}$ and $\beta^{i+1} = \alpha_1{}^{i+1} \gamma^{i+1}\ \alpha_2{}^{i+1}$. By induction, $\alpha^{i+1}$ is the yield of $t_i$. That is, the $(|\alpha_1{}^{i+1}|+1)$-th leaf node is labelled by $A^{i+1}$. Attach $|\ \gamma^{i+1}|$ children labelled by the symbols from $\gamma^{i+1}$ to that node to obtain $t_{i+1}$. Clearly, $t_{i+1}$ is a parse tree for $\beta^{i+1}$.

"4. ⇒ 2.": Go through the variable nodes of the parse tree in a left-first, depth-first fashion and obtain a leftmost derivation. □

## 4.2 Ambiguity in CFGs

Consider the two parse trees of the string $a + a * a$ obtained using the grammar from example 4.2, shown in Fig. 4.2. The two trees are different, and represent two different binding-level structures: the left tree corresponds to a reading $(a + (a * a))$, the right tree to $((a + a) * a)$. (You should be familiar with the fact that nested list structures correspond 1-1 to ordered trees!). If used as an internal data structure for some arithmetic processor, the left parse tree would represent that * binds more strongly than + (desirable), while the right tree would imply that + binds more strongly than * (not desirable). It would be convenient (even necessary) to have grammars that do not allow these kinds of ambiguity, and yield nested list structures that are unique and correspond to the binding conventions made in the particular application (for instance, * binds more strongly than + in arithemtics). Remember that parse trees are the intermediate storage format (after parsing) of program code, and the order of execution of nested subcommands is determined by the tree structure. It is therefore of practical importance to ensure that the tree structure of parse trees does not violate the rules of precedence of algebraic (and other) operations.

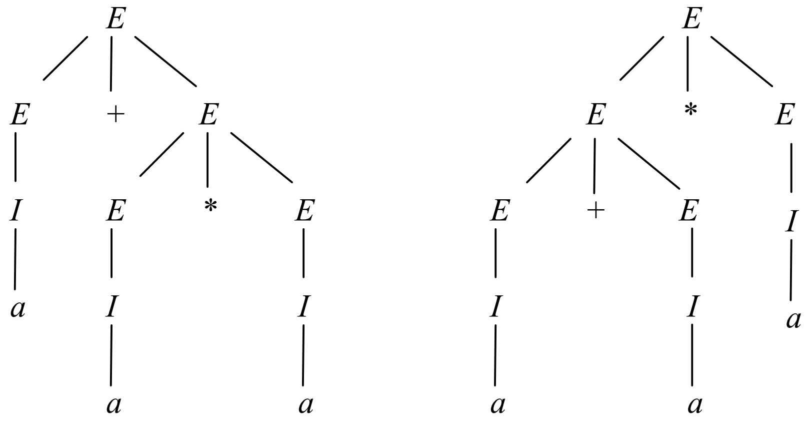


**Fig. 4.2:** Trees with yield $a + a * a$, demonstrating ambiguity of the grammar of Example 4.2.

**Definition 4.6** (ambiguity of a CFG): A CFG is *ambiguous* if there exists a word $w \in T^*$ which has two different parse trees. A CFG is *unambiguous* if it is not ambiguous.

There is no general method to decide whether a CFG is ambiguous, and there is no general method to remove ambiguity from a grammar. However, there are some tricks that one can try to enforce non-ambiguity on a grammar.

There are many ways of how ambiguity can arise. In our little example, there are two ways in which ambiguity becomes manifest:

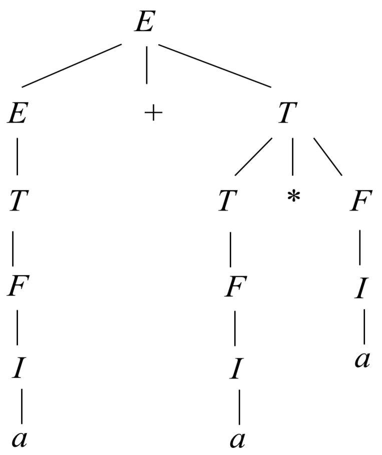
1. Precedence of different operators is not captured in the grammar – as in the example of Fig. 4.2.
2. Associative grouping of identical operators is ambiguous. For instance, we could derive $a + a + a$ in two ways, yielding parse trees similar to the ones in Fig. 4.2., corresponding to groupings $a + (a + a)$ and $(a + a) + a$.

These types of ambiguities are relevant for grammars of programming languages. They can be removed, obtaining grammars that are unambiguous. The main trick is to introduce auxiliary variables that code binding levels. For instance, if we introduce additional variables $F$ (= "factor") and $T$ (= "additive term"), we get a grammar that is equivalent (in the sense of describing the same language) to our grammar from Example 4.2., but is unambiguous:

**Example 4.3:** an unambiguous grammar for our little arithmetic expression grammar.

$$
\begin{array}{lll}
E & \to & T \mid E + T \\
T & \to & F \mid T * F \\
F & \to & I \mid (E) \\
I & \to & a \mid b \mid Ia \mid Ib \mid I0 \mid I1
\end{array}
$$

Here is a (in fact, *the*) derivation for $a + a * a$ with this unambiguous grammar:



Here the use of additional variables renders the parse tree representation of precedence of operators unique, and the fact that the two rules for introducing * and + can each be iterated only in a leftmost fashion makes the parse tree representation of associative grouping unambiguous (namely, left-associative). The fact that this grammar is unambiguous is not at all obvious. The HMU book gives an intuitive-level justification of this claim (pp. 208-209).

We have seen that sometimes ambiguity can be removed from a grammar. In fact, the causes of ambiguity that are present in programming languages are typically harmless and can be removed. However, this is not generally possible – uncurably ambiguous CFLs exist.

**Definition 4.7** (inherent ambiguity). A context-free language is *inherently ambiguous* if every grammar for it is ambiguous.

**Example 4.4:** The following context-free language is inherently ambiguous:

$$L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$$

The obvious grammar for $L$ is

$$\begin{array}{lll}
S & \rightarrow & AB \mid C \\
A & \rightarrow & aAb \mid ab \\
B & \rightarrow & cBd \mid cd \\
C & \rightarrow & aCd \mid aDd \\
D & \rightarrow & bDc \mid bc
\end{array}$$

It uses separate sets of productions to generate the two kinds of strings in $L$. All words of the form $a^n b^n c^n d^n$ have two different parse trees with this grammar, depending on whether the word is generated starting with $S \rightarrow AB$ or with $S \rightarrow C$. The proof that *every* grammar for this language is ambiguous is involved. The HMU book gives a handwaving argument (p. 213).

## 4.3 Connection to regular languages

It will take us a small effort only to show how the regular languages can be understood as a (proper) subclass of the CFLs. We will impose a restriction (right-linearity) on grammars and see that the languages derivable within this restricted class of grammars are the regular languages.

**Definition 4.8.** A CFG is called *right-linear* if each production body has at most one variable, and that is at the right end. That is, all productions are of the form $A \to wB$ or $A \to w$, where $w$ is a word of terminals.

**Proposition 4.2**. Every right-linear CFG defines a regular language.

**Proof.** Idea: identify variables with nonaccepting states of a generalized ε-NFA with arcs labelled by the $w$ from the productions of the grammar; add accepting states for each production of the form $A \to w$; starting state corresponds to start symbol. Formally: for a given right-linear CFG $(V, T, P, S)$ define a generalized ε-NFA $(Q, \Sigma, \delta, q_0, F)$, where

- $Q = V \cup Q_2 = V \cup \ \{q_p \mid p \in P, p \text{ is of form } A \to w\}$,
- $\delta$: $Q \times \Sigma^* \to 2^Q$ with $B \in \delta(A, w)$ iff $A \to wB \in P$ and $q_p \in \delta(A, w)$ iff $p = A \to w \in P$,
- $q_0 = S$,
- $F = Q_2$.

It is straightforward to show that this generalized ε-NFA accepts $L(G)$. □

**Proposition 4.3.** Every regular language $L$ is the language of a right-linear CFG.

**Idea of proof.** Consider a DFA for $L$, identify states $q$ of the DFA with variables $A_q$; identify transitions $\delta(q, a) = p$ with productions $A_q \to aA_p$; add productions $A_q \to a$ for all $\delta(q, a) = p$ where $p$ is accepting; add production $A_{q0} \to \varepsilon$ if starting state $q_0$ is accepting; define $A_{q0}$ as start symbol, where $q_0$ is the starting state. □
4.
An immediate consequence of this proposition is that every regular langage is context-free. However, not every context-free language is regular:

**Example 4.5.** The language $L = \{w \in \{0,1\}^* \mid w = w^R$, that is, $w$ is a palindrome$\}$ is not regular (Pumping lemma!). But $L$ is the language of the context-free grammar $G$ defined by the productions $S \Rightarrow \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$. We show that $w = w^R$ iff $w$ is in $L(G)$.

" ⇒": Induction over $|w|$. *Basis*: $|w| \le 1$. Then $w = \varepsilon$ or $w = 0$ or $w = 1$, which clearly are in $L(G)$. *Induction*: Assume all palindromes $u$, where $|\ u\ | \le n$, are in $L(G)$. Consider a palindrome $w$ of length $n$+1. It must be of form $0u0$ or $1u1$, with $u$ a palindrome of length $n$–1. By induction hypothesis, $u \in L(G)$, that is, $S \Rightarrow^* u$. From this, in case $w = 0u0$ infer derivation $S \Rightarrow 0S0 \Rightarrow^* 0u0 = w$; case $w = 1u1$ is similar.

"⇐": Induction over the number of steps of a derivation (a trick to remember!). *Basis*: $w$ can be derived in a single step. Then $w = \varepsilon$ or $w = 0$ or $w = 1$, that is, $w = w^R$. *Induction*: Suppose all $u$ that have derivations of length $\le n$ are palindromes. Let $w$ be derived in $n$+1 steps. Such a derivation must have form $S \Rightarrow 0S0 \Rightarrow^* 0u0 = w$ or $S \Rightarrow 1S1 \Rightarrow^* 1u1 = w$. Because $u$ is derived in $n$ steps, it must be a palindrome. Therefore $w$ is a palindrome, too.

## 4.4 Grammars at their most powerful: XML, XSL etc.

Most likely, you have inspected the HTML source code of a webpage once in a while. The most conspicuous feature of HTML code is the use of what one might call "labelled parentheses" of the form <a> ... </a> or <title> ... </title>. Such *tags* may be used in a nested fashion, for instance <sub> <b> ... <b/> </sub>.

HTML is only one member from a large and growing family of languages, all of which have the same "feel" of HTML with labelled tags. They are known as XML languages (eXtensible Markup Language). The XML FAQ at http://xml.silmaril.ie/ can serve as a nice introduction and overview. The official XML site is http://xml.coverpages.org/xml.html; here you can find information about history, standards, applications, and many more. The syntax of XML is presented (in Backus-Naur format) at http://www.w3.org/TR/REC-xml.

Here is an excerpt of an non-HTML XML text:

**Example 4.6:** (Document metadata.) In Fall 2003 I used Acrobat Distiller to create the pdf's from the Word versions of the exercise sheets for this lecture. When I opened such a pdf with Acrobat, then cklicked on `File -> Document Properties -> Document Metadata -> View Source`, I got the following text (here only a part of what you get is shown; the source is exercise sheet 1 from Fall 2003):

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
        xmlns:iX='http://ns.adobe.com/iX/1.0/'>

<rdf:Description about=''
  xmlns='http://ns.adobe.com/pdf/1.3/'
  xmlns:pdf='http://ns.adobe.com/pdf/1.3/'>
  <pdf:CreationDate>2004-09-08T16:14:07Z</pdf:CreationDate>
  <pdf:ModDate>2004-09-08T16:14:07Z</pdf:ModDate>
  <pdf:Producer>Acrobat Distiller 5.0 (Windows)</pdf:Producer>
  <pdf:Author>Herbert Jaeger</pdf:Author>
  <pdf:Creator>Acrobat PDFMaker 5.0 for Word</pdf:Creator>
  <pdf:Title>Exercises for ACS 1, Fall 2003</pdf:Title>
 </rdf:Description>
```

*... some similar blocks omitted ...*

```
 <rdf:Description about=''
      xmlns='http://purl.org/dc/elements/1.1/'
      xmlns:dc='http://purl.org/dc/elements/1.1/'>
  <dc:creator>Herbert Jaeger</dc:creator>
  <dc:title>Exercises for ACS 1, Fall 2003</dc:title>
 </rdf:Description>

</rdf:RDF>
```

This is an excerpt from the *document metadata* which Acrobat Distiller saves along with each pdf document it creates. (Today's Acrobat versions don't allow you to display the metadata in raw text format; you can only inspect them through a GUI. The website http://www.forensicswiki.org/wiki/Document_Metadata_Extraction contains links to numerous tools to extract the document metadata from a variety of filetypes). It contains various kinds of information about the creator of the document, its title, the software version used in creating it and much more. Document metadata is useful for libraries, bookselling companies, all kind of text databases, book search engines, and generally all institutions or persons or programs that wish to get an overview of some set of books, documents, texts. The important thing about this document metadata text is that it is not written in an arbitrary, Adobe-pdf-proprietary format. Document metadata only make sense if these metadata are independent of the specific format of the text. The metadata that Word saves with each Word document should be in the same format as the metadata that Amazon saves with each of its book records, and again the same that the British library uses, etc.

The common standard for document metadata is the `Resource Description Format` (RDF), which is chartered out at http://www.w3.org/RDF/. A CFG for RDF texts is given at http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/ (scroll down to Section 7).

Different groups of users/producers of text documents may have different desires about the text metadata. For instance, for Acrobat-generated pdf texts it may be reasonable to save, *in addition* to the general-purpose text metadata, some pdf-specific information – see the first block in the above RDF document. Thus, different groups of users may wish to define, within the RDF standards, a standard for their specific modules within a text metadata document. A recommended way of doing so is to create a CFG that specifies the syntax of such modules, and to make this CFG publicly available. In the world of XML, such grammars for particular types of XML documents, or for particular modules within them, are called *document type definitions* (DTD). A DTD is a CFG that specifies the structure of target documents. Documents composed according to a DTD can then be *automatically* processed for different purposes because they conform to a known grammar and are thereby parseable. Within the XML world, a rigorous syntax for DTDs has been developed (check http://www.w3.org/TR/REC-xml#NT-doctypedecl). We will learn more about DTDs and their grammar further below. DTDs are not the only way to formally specify XML document types. Other options are XML Schema (also known as XMD, "XML Schema Definition") – see https://en.wikipedia.org/wiki/XML_Schema_%28W3C%29 – or Relax NG – see https://en.wikipedia.org/wiki/RELAX_NG . All of these *schema languages for XML* are themselves grammars that allow the user to define XML document types. While DTDs use their own special grammar, XMD and Relax NG are specified themselves in an XML syntax.

The last block in the above document is a point in case. It conforms to the standards published by the "Dublin Core Metadata" standard. This appears to have become the most widely used text metadata standard for general library information uses. The DTD for Dublin Core Metadata has been published on the website of the Dublin Core Metadata Initiative (http://dublincore.org/about/ ) at http://dublincore.org/documents/2002/07/31/dcmes-xml/ .

Two basic concepts from the XML world should be explained at this point, *namespaces* and URIs. If some user group decides to define some type of XML documents, maybe by creating a DTD, they are free to choose the names of the tags to be used within their XML documents. Other user groups may happen to decide to use the same names for *their* XML documents. This would give rise to ambiguities when two XML texts of different sort are merged within a single embracing XML document – which is actually what happend in our RDF example, where one block conforms to an Acrobat XML standard and another to the Dublin Core standard. To preclude name clashes, *namespaces* are used. A user group that defines some XML text type (e.g., by publishing a DTD) also declares a "namespace" along with the DTD. All tag key words introduced by that group are unique only within that namespace – other XML document types defined by other groups have other namespaces. Namespaces are uniquely identified by *uniform resource identifiers*, URIs. URIs look like URLs (*uniform resource locators*), but does not necessarily point to any physical web location. In our RDF example, there appear two URIs for namespaces `http://ns.adobe.com/pdf/1.3/` and `http://purl.org/dc/elements/1.1/`. Within XML documents, the fact that some block adheres to some namespace is specified using the `xmlns` declarations that you find in our example. The syntactical similarity of URIs and URLs is certainly confusing. See http://www.rpbourret.com/xml/NamespacesFAQ.htm for more.

**Example 4.7:** (didactic toy example for DTDs from the HMU book). In order to illustrate DTDs, we present an artificial example from the HMU book. Scenario: computer vendors and web-based computer marketplaces want to make computer specifications readable by all kinds of application programs (databases, business calculation programs, website generators, ...). A DTD "PcSpecs" is agreed. It might look as follows (syntax slightly simplified):

```
<!DOCTYPE PcSpecs [
      <!ELEMENT PCS (PC*)>
      <!ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
      <!ELEMENT MODEL (\#PCDATA)>
      <!ELEMENT PRICE (\#PCDATA)>
      <!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
      <!ELEMENT MANF (\#PCDATA)>
      <!ELEMENT MODEL (\#PCDATA)>
      <!ELEMENT SPEED (\#PCDATA)>
      <!ELEMENT RAM (\#PCDATA)>
      <!ELEMENT DISK (HARDDISK | CD | DVD)>
      <!ELEMENT HARDDISK (MANF, MODEL, SIZE)>
      <!ELEMENT SIZE (\#PCDATA)>
      <!ELEMENT CD (SPEED)>
      <!ELEMENT DVD (SPEED)>
]>
```

Notes:

1. General (simplified) form of a DTD:

   <!DOCTYPE name-of-DTD [list of element definitions ]>

2. Each element definition is a generalized CFG production of the form

   <!ELEMENT element-name (body of rule)>,

   where the head of the rule is a variable name ("element-name" in XML terminology), and the body of the rule is a regular expression with a special Unix-like notation.

3. \#PCDATA (from "parsed character data") is an XML key word. It can be replaced in the target documents specified by the DTD essentially by any character string. It is the only terminal symbol in our example DTD.

Here is a sample document written according to the specs of our DTD:

```
<PCS>
      <PC>
            <MODEL>4560<\MODEL>
            <PRICE>$2295<\PRICE>
            <PROCESSOR>
                  <MANF>Intel<\MANF>
                  <MODEL>Pentium<\MODEL>
                  <SPEED>800Mhz<\SPEED>
            <\PROCESSOR>
            <RAM>256<\RAM>
            <DISK><HARDDISK>
                  <MANF>Maxtor<\MANF>
                  <MODEL>Diamond<\MODEL>
                  <SIZE>30.5Gb<\SIZE>
            <\HARDDISK><\DISK>
            <DISK><CD>
                  <SPEED>32x<\SPEED>
            <\CD><\DISK>
      <\PC>
      <PC>
      ...
      <\PC>
<\PCS>
```

**Example 4.8:** robot specification. This was my first real contact with the power of XML, and I was introduced to it when I was working at the Fraunhofer Institute for Autonomous Intelligent Systems (AIS) in a robotics group. In that group, many robots were designed, built, and programmed, and changes in robot layout and control program occurred daily. A robot is a complicated device, with many sensors, actuators, microprocessors and PC-like on-board computers. Besides the robot itself, a simulation program was always needed to test the robot control program in virtual reality, plus there was a detailed HTML-based documentation that was absolutely necessary to enable different people to work on the same robot. The various hardware, simulator and documentation components used a variety of programming languages: C, C++, Java, HTML, plus some extra scripting languages for remote monitoring protocols. There was only one way to keep all these various programs mutually consistent through the frequent changes done by several people: use a single basic XML document as *the* reference specification of the robot (including hardware configuration and control program), and derive all the various target program code *automatically* from that XML master blueprint[4]. This implies that in addition to the XML source document, there had to be *translation* programs for translating the reference specification into the various target programs. Writing these translation programs was only possible because the reference XML specification adhered to a known, rigorous grammar.

**Example 4.9:** mathML is an XML document standard for representing mathematical formulas for purposes of visualization and semantic processing. See http://xml.coverpages.org/mathML.html for more details and ask Prof. Michael Kohlhase for more detail, because he is responsible for organizing the mathML standards.

[4] In fact, at AIS we did not use XML, because the robot development activities started when XML was not fully established. Instead, we used a proprietary precursor/analogue of XML called APIECES – but if XML had been available earlier, we would have used XML.

**Example 4.10:** For some years, I used my own self-made content management system for my Jacobs homepage (now I have migrated to the University's standard Drupal CMS). This self-made system was based on an XML specification. There were several motivations. One was portability: if at one time I would be leaving Jacobs, I would wish to re-launch my personal webpages at my new place, re-using the content and only re-writing the layout. This *separation of contents from appearance* is one of the main motives for many XML applications: the contents is specified in an XML document, and the visual appearance is created by passing the XML source through a filter that generates the desired looks. Another reason is maintainability: the XML files that specify the contents of my pages are easier to read and change than complete HTML files, and on top, I only have a single XML file for each of the main parts that appear over and over again on all my pages. If, for instance, I wished to add an item to the navigation menu, I would simply add an entry in my XML file "leftmenu.xml", which looked like that:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<leftmenu>
      <item>
            <name>Welcome</name>
            <url>index.html</url>
      </item>
      <item>
            <name>Research</name>
            <url>research.html</url>
      </item>
      <item>
            <name>Teaching at Jacobs</name>
            <url>teaching.html</url>
      </item>
      <item>
            <name>Publications</name>
            <url>pubs.html</url>
      </item>
</leftmenu>
```

Note that I invented the tag identifiers (`leftmenu`, `item`, `name`, `url`) myself and specifically for use in `leftmenu.xml` – if there would be many other users of this type of document, I would cast the ad-hoc grammar that I use into a DTD and announce a namespace. I had similar XML documents for the other recurring portions of my webpages, like `topbar.xml` and `navbar.xml`.

In order to create the final HTML pages, I use a tool that is generally applicable for transforming XML sources into target documents of any kind. This tool is XSLT (Extensible Stylesheet Language Transformations), a programming language (itself adhering to XML standards!) for writing filters that transform XML documents into target documents, like HTML documents. Below is the XSLT file `leftbar.xsl` that I wrote in order to transform `leftbar.xml` into the html code that appears within my html webpages. This XSLT file contains some of the layout information that the ancient "IUB look" required (further layout information is added by working in a CSS stylesheet, an aspect that I skip here). Here is my `leftbar.xsl:`

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml" xmlns:xhtml="http://www.w3.org/1999/xhtml">
        <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" encoding="iso-8859-1"
          doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
          doctype-system="{concat($prefix,'xhtml/xhtml1-strict.dtd'}"/>
        <xsl:template name="leftmenu">
                <table width="170" border="0" cellspacing="0" cellpadding="0">
                        <br/>
                        <xsl:for-each select="document('leftmenu.xml')/leftmenu/item">
                                <tr>
                                        <td background="./pics/pixel.gif" height="30"
                                          width="14"/>
                                        <td background="./pics/pixel.gif" height="30"
                                          width="170">
                                                <a href="{url}">
                                                        <xsl:value-of select="name"/>
                                                </a>
                                        </td>
                                </tr>
                        </xsl:for-each>
                </table>
        </xsl:template>
</xsl:stylesheet>
```

There are a number of shareware, open-source XML editors and XSLT interpreters available. XML really develops its powers only in conjunction with XSLT or other "executing" software, which allows you to actually *do* something with your XML documents. By itself, an XML documents is not executable:

"*XML is not a programming language, so XML files don't `run' or `execute'. XML is a markup specification language and XML files are data: they just sit there until you run a program which displays them (like a browser) or does some work with them (like a converter which writes the data in another format, or a database which reads the data), or modifies them (like an editor). If you want to view or display an XML file, open it with an XML editor or an XML browser.*" [from http://xml.coverpages.org/FAQv21-200201.html ]

If you want to learn more about XSLT and XLM, I suggest you consult the online tutorials at the website of the W3C (World-Wide Web Consortium, the "official" consortium working out the standards for present and future Web-based technologies), e.g. http://www.w3schools.com/xsl/default.asp .

Today there are tens of thousands of XML applications, some hundreds of exemplary ones are exhibited at http://xml.coverpages.org/xml.html#applications (you should really visit this link, it is impressive – and impressively disorganized!).

XML is a subset of SGML. SGML is the Standard Generalized Markup Language – if you have 198 Swiss Franks to spare, you can buy a copy of this Standards document at http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16387 –, the international standard for defining descriptions of the structure of different types of electronic document. There is an SGML FAQ at http://lamp.man.deakin.edu.au/sgml/sgmlfaq.txt; the SGML Web pages are at http://xml.coverpages.org/.

**Example 4.11.** In the past, Microsoft Office programs (Word, Excel, Powerpoint) have been created and saved in a MS proprietary binary format (.doc, .xls, .ppt). Since some years, MS has changed this to an XML-based format (you recognize these by the file extensions .docx, .xlsx, .pptx). By default, MS office programs save XML files in compressed formats, so you don't see the XML ASCII if you load them into an editor, for example into emacs. You may however use the option "Save as XML document" in MS office applications, then the save file will be human-readable XML ASCII (open in an editor like emacs to see it; opening it in MS Word will not show the XML source but the rendered document).

> The design of most professional-level software and documentations systems nowadays includes at some point XML or SGML documents and document-type specifications. SGML / XML plus XSLT is one of the most important working tools in applied computer science.

## 4.5 Context-free languages and their automata

***Executive summary.*** The automata that accept CFLs are *pushdown automata* (PDAs). Essentially, they are NFA's equipped with an additional simple stack memory. Because they are non-deterministic, they cannot be used for compilers. Introducing *deterministic* pushdown automata (DPDAs), however, effectively reduces the set of recognizable languages. Fortunately, programming languages and markup languages can be recognized by DPDAs.

A PDA is essentially an ε-NFA augmented by a stack memory, i.e., a first-in, last-out ("FILA") device. The transitions of the ε-NFA are additionally determined by the top stack symbol, which is taken away. The action of a transition includes, as a side effect, the option to "push down" a word on top of the stack.

The formal definition of PDAs works along familiar lines: first fix what parts make up a PDA (the typical "*n-tuple*" definition we know from DFAs, NFAs etc.), then in some further definitions describe how PDAs actually process symbols and accept languages.

**Definition 4.9.** (Pushdown automaton, PDA). A PDA is a 7-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- $Q$ is a finite set of states,
- $\Sigma$ is a finite set of input symbols (corresponding to the terminals $T$ of CFGs),
- $\Gamma$ is a finite stack alphabet,
- $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathrm{Pot}_0(Q \times \Gamma^*)$ is the transition function ($\mathrm{Pot}_0$ denotes the finite subsets of a set: $\mathrm{Pot}_0(A) = \{B \subseteq A \mid |B| < \infty\}$,
- $q_0 \in Q$ is the start state,
- $Z_0 \in \Gamma$ is the start stack symbol,
- $F \subseteq Q$ is the set of accepting states.

Before we describe the processing principles of PDAs in formal terms, we'll see how they work by inspecting a simple example. In each operating cycle, the PDA (in state $q$) reads an input $a$ (or ε), also reads the top stack symbol $X$, and then uses the transition $\delta(q, a, X)$ to choose some next state $q'$ and stack "push word" $\alpha$ from $\delta(q, a, X) = \{..., (q', \alpha), ...\}$. If $\delta(q, a, X)$ is empty, the PDA halts in a dead end situation. Like with NFAs, an input word is accepted if there exists *some* sequence of operations that reads the entire input and ends in an accepting state.

**Example 4.11**. Consider the language $L_{wwR} = \{ww^R \mid w \in \{0, 1\}^+\}$. It can be accepted by a PDA with $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, Z_0\}$, $F = \{ q_2\}$, and $\delta$ containing the following transitions:

1. $\delta(q_0, a, X) = \{(q_0, aX), (q_1, aX)\}$ for $a = 0, 1$ and $X = 0, 1, Z_0$;
2. $\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$ for $a = 0, 1$;
3. $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$.

(For other arguments, $\delta(q, a, X) = \emptyset$. These need not be listed.) An accepting sequence of transitions for input $ww^R$ = 0110 is shown in Fig. 4.3.

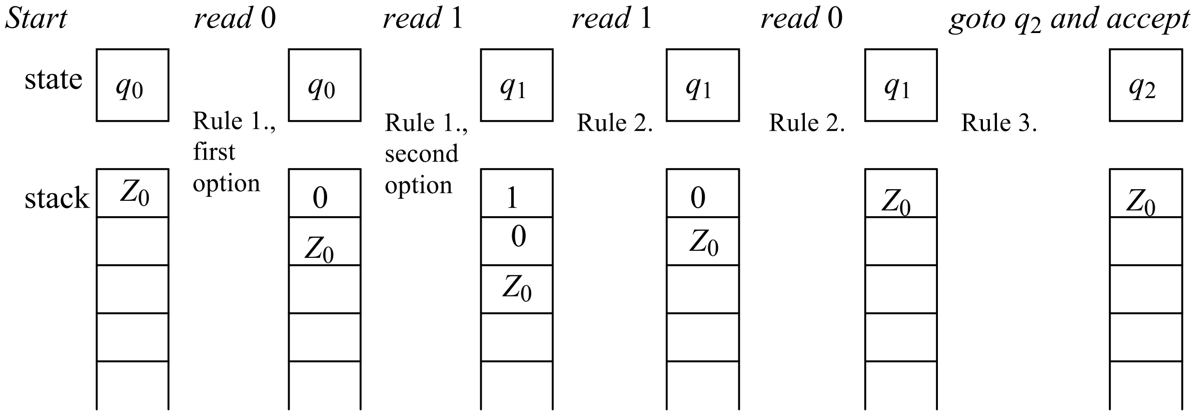


**Fig. 4.3.** An accepting run of the PDA for $L_{wwR}$.

Note that the PDA can simply delete the top stack symbol through transitions of the kind $\delta(q, a, X) = \{..., (q', \varepsilon), ...\}$.

The working principle behind this example is clear: while reading in the first half of $ww^R$, a "mirror image" of $w$ is written into the stack; then when the second part $w^R$ is read, it is matched against the stack, which is "eaten up" in the process. Note that after reading in $w$, the PDA has to "guess" that it should switch into the "matching mode". Only if it switches mode at the right time (after reading $w$), an altogether successful run can occur. But such a need to guess correctly is the hallmark of acceptance by a nondeterministic automaton – we saw the same occuring with NFAs.

**Notational conventions** for PDAs. We usally denote the following types of PDA symbols with the following types of variable symbols:

- $a$, $b$, ... : input symbols
- $q$, $p$, ... : states
- $w$, $u$, ..., $x$, $y$, ...: input words
- $X$, $Y$, .... : stack symbols
- $\alpha$, $\beta$, ... : stack words

A **graphical notation** for PDAs. In a similar way as with DFAs, we can graphically represent a PDA by a labelled state transition diagram. The arc labels $a, X / \alpha$ now contain an input symbol $a$, the top stack symbol $X$, and the stack push word $\alpha$. Here is our example PDA in graphical format:

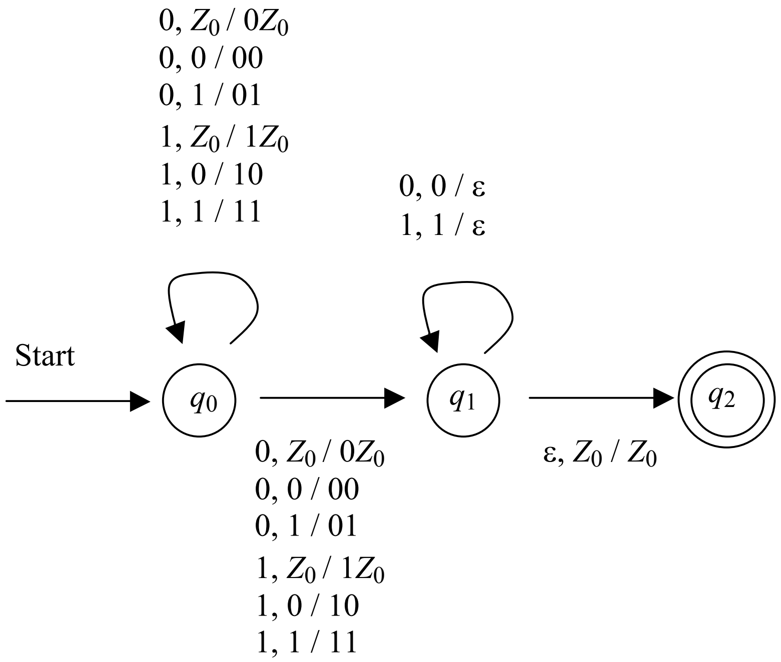


**Figure 4.4.** Graphical representation of the PDA from Example 4.11.

Now we go through the familiar routine of defining the workings of a PDA in rigorous terms.

**Definition 4.10.** (configuration of a PDA). A *configuration* (in the HMU book: *instantaneous description*) of a PDA is triple $(q, w, \gamma)$, where

- $q \in Q$ is the current state of the PDA,
- $w \in \Sigma^*$ is the remaining input word,
- $\gamma \in \Gamma^*$ is the word of current stack contents (first symbol in $\gamma$ = top of stack).

A configuration contains a complete "snapshot" of a PDA processing an input word.

**Definition 4.11.** (move of a PDA) For a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, all $q, q' \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$, $X \in \Gamma$, and $\alpha, \beta \in \Gamma^*$ define a binary relation $\vdash_P$ on the set of configurations by

$$(q, aw, X\beta) \vdash_P (q', w, \alpha\beta) \quad \text{iff} \quad (q', \alpha) \in \delta(q, a, X).$$

When $(q, aw, X\beta) \vdash_P (q', w, \alpha\beta)$, we say that $P$ may *move* from configuration $(q, aw, X\beta)$ to configuration $(q', w, \alpha\beta)$ in a single step. Note that moves are non-deterministic: there may be several, a single, or even no possible move(s) away from a current configuration $(q, aw, X\beta)$. As usual, define by $\vdash_P^*$ the transitive closure of $\vdash_P$ (that is, zero or any number of moves). We drop the subscript $P$ if the reference to $P$ is clear.

The PDA run of Fig. 4.3 corresponds to this sequence of configurations:

$(q_0, 0110, Z_0) \vdash_P (q_0, 110, 0Z_0) \vdash_P (q_1, 10, 10Z_0) \vdash_P (q_1, 0, 0Z_0) \vdash_P (q_1, \varepsilon, Z_0) \vdash_P (q_2, \varepsilon, Z_0)$

or

$(q_0, 0110, Z_0) \vdash_P^* (q_2, \varepsilon, Z_0)$.

**Definition 4.12.** (language accepted by a PDA). Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then the *language accepted by P by final state* is

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha), \text{ where } q \in F \text{ and } \alpha \in \Gamma^*\}.$$

**Definition 4.13**. The *language accepted by P by empty stack* is
w

$$N(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \varepsilon), \text{ where } q \in Q\}.$$

For PDAs accepting a language by empty stack we also use the empty set as set of accepting states, that is, write $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing)$.

Note that the working principle of PDAs is nondeterministic. A word $w$ is in $N(P)$ if *some* sequence of moves exists which leads from the initial configuration to a final state configuration (or empty stack configuration).

**Proposition 4.4.** (equivalence of empty-stack and final-state accepting PDAs) A language $L$ is accepted by some PDA by empty stack iff it is accepted by some PDA by final state.

**Sketch of proof of proposition 4.4:**

"⇒": Let $P_{\text{empty}} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing)$ accept $L$ by empty stack. Idea: construct PDA $P_{\text{final}}$ which accepts $L$ by final state. $P_{\text{final}}$ has an new stack start symbol $X_0$. $P_{\text{final}}$ essentially "simulates" $P_{\text{empty}}$. Before the simulated $P_{\text{empty}}$ starts, $P_{\text{final}}$ pushes $P_{\text{empty}}$ 's stack start symbol $Z_0$ on the stack, then starts the simulation of $P_{\text{empty}}$. $P_{\text{final}}$ needs a new starting state $p_0$ and a new accepting state $p_f$ to achieve this, as well as a few extra transitions. As soon as $P_{\text{empty}}$ empties its stack, $X_0$ becomes visible and $P_{\text{final}}$ goes into accepting state. To achieve this, for each state $q$ of $Q$, a transition $\delta(q, \varepsilon, X_0) = (p_f, \varepsilon)$ is added. See Fig. 4.5. for how this works. Note that this sketch is not a proof – it's the idea behind a proof. A full proof can be found in the HMU book.

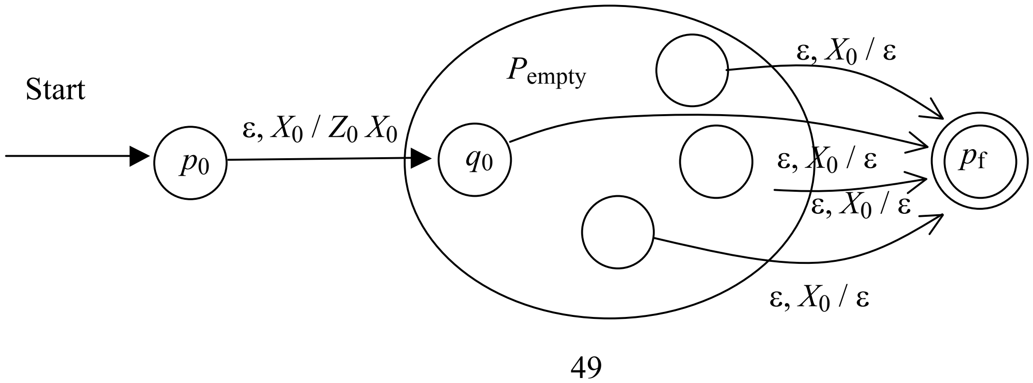

Start
$p_0$
$\varepsilon, X_0 / Z_0 X_0$
$P_{\text{empty}}$
$q_0$
$\varepsilon, X_0 / \varepsilon$
$p_f$
$\varepsilon, X_0 / \varepsilon$
$\varepsilon, X_0 / \varepsilon$
$\varepsilon, X_0 / \varepsilon$

**Figure. 4.5:** A final-state accepting PDA (complete figure) constructed from an empty-stack accepting PDA $P_{\text{empty}}$ (big oval).

"⇐": Let $P_{\text{final}} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ accept $L$ by final state. Idea: construct $P_{\text{empty}}$ from $P_{\text{final}}$ by "wrapping" $P_{\text{final}}$ into a PDA that empties the stack of $P_{\text{final}}$ when $P_{\text{final}}$ enters an accepting state. Details in Figure 4.6. □

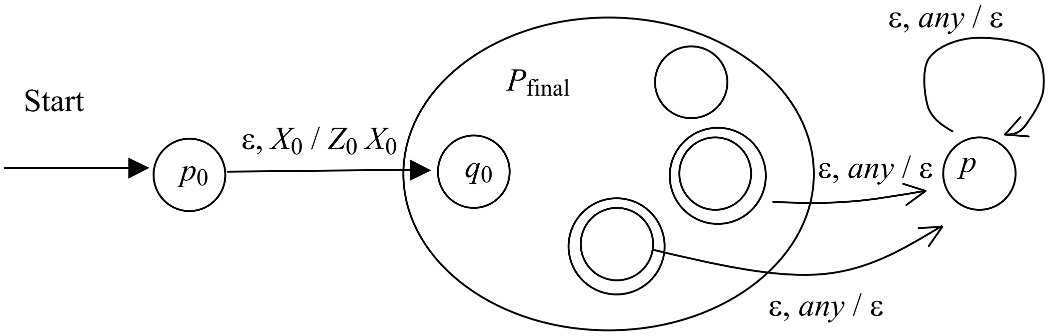


**Figure 4.6:** An empty-stack accepting PDA (complete figure) constructed from a final-state accepting PDA $P_{\text{final}}$ (big oval).

**Proposition 4.5.** (equivalence of CFGs and PDAs). A language $L$ is the language of some CFG iff it is accepted by some PDA by empty stack.

**Sketch of proof:**

"⇒": Let $L = L(G)$, where $G = (V, T, P, S)$ is a CFG. Idea: construct a PDA $P_{\text{empty}}$ that simulates all leftmost derivations in $G$. $P_{\text{empty}}$ has a single state $q$, word alphabet $T$, stack alphabet $V \cup T$, starting state $q$ (of course, because it's the only state), stack start symbol $S$. $P_{\text{empty}}$ has two types of transition rules. First type: a variable symbol $A$ is on top of stack, and a grammar rule $A \to \alpha$ exists in $P$. Then $P_{\text{empty}}$ has a transition rule that takes away $A$ from the stack and replaces it by $\alpha$. Second type: if $P_{\text{empty}}$ sees a terminal symbol $a$ on top of the stack, and at the same time can read $a$ as the next symbol in its input word, it may delete $a$ from the stack. Figure 4.7 shows $P_{\text{empty}}$.

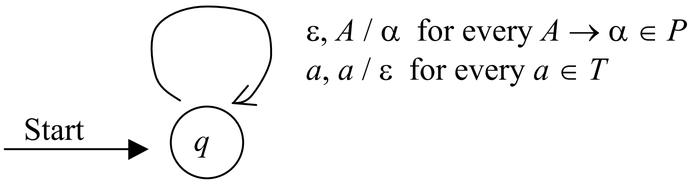

ε, $A$ / α  for every $A \to \alpha \in P$
$a$, $a$ / ε  for every $a \in T$
Start
$q$

**Figure 4.7:** A PDA that accepts a language given by a CFG by empty stack.

The proof needs now to be completed by showing that $L(G) \subseteq L(P_{\text{empty}})$ [induction on steps of a leftmost derivation] and $L(P_{\text{empty}}) \subseteq L(G)$ [induction on number of transition steps]. This can be found in the HMU book, page 238-241.

"$\Leftarrow$": Let $L = N(P_{\text{empty}})$, where $P_{\text{empty}} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, \varnothing)$. We construct a grammar $G = (V, T, P, S)$ for $L$. Main idea: for grammar variables we use complex symbols of the form $[qXp]$, which denote the following condition: "if $P_{\text{empty}}$ is in state $q$, and has $X$ on top of its stack, it can go through some sequence of transitions which effectively pop $X$ off the stack, after which it is in state $p$". Note that this sequence can be arbitrarily long and that during this sequence, new stack symbols may be pushed on top of $X$; however, at some time, whatever has been pushed on top of $X$ will eventually be removed again, and $X$ will finally be removed, too. – In addition of these compound symbols, the grammar variable alphabet also will include a special symbol $S$. Altogether, we put $V = \{ [qXp] \mid q, p \in Q, X \in \Gamma\} \cup \{S\}$ and $T = \Sigma$, and prepare the set $P$ of grammar rules as follows:

1. For all states $p$, $P$ contains the rule $S \to [q_0Z_0\,p]$. Comment: these rules describe the condition "there exists a sequence of PDA transitions that bring it from the starting configuration with $q_0$ and $Z_0$ to some state $p$, where the stack has been emptied, indicating acceptance".
2. Let $\delta(q, a, X)$ contain the pair $(r, Y_1Y_2...Y_k)$, where $a \in \Sigma \cup \{\varepsilon\}$ and $k \geq 0$. [If $k = 0$, then $(r, Y_1Y_2...Y_k) = (r, \varepsilon)$.]. Then for all sequences of states $r_1, r_2, ..., r_k$ add the production

   $$[qXr_k] \to a[rY_1r_1]\ [r_1Y_2r_2]\ ...\ [r_{k\text{-}1}Y_kr_k].$$

   [In the case $k = 0$ we add only $[qXr] \to a$.] This production says that one way to pop $X$ and being in state $r_k$ afterwards is to read $a$, then use some input and some transitions to pop $Y_1$, coming out in state $r_1$, the use some more input and some more transitions to pop $Y_2$, coming out in state $r_2$, etc., until finally $Y_k$ has been popped and the entire sequence comes out in state $r_k$.

The proof needs now to be completed by showing that $L(G) \subseteq L(P_{\text{empty}})$ [induction on number of leftmost derivations] and $L(P_{\text{empty}}) \subseteq L(G)$ [induction on number of transition steps]. This can be found in the HMU book, page 242-244. □

Our PDAs are non-deterministic devices and thus cannot work as an algorithm. In the world of regular languages and their automata, the nondeterministic NFAs had their deterministic equivalents (DFAs), which could be constructed from the NFAs by the subset construction. In the world of CFLs, we can likewise define deterministic PDAs, which we call DPDAs. But unfortunately, we lose some power. Not for every PDA there exists an equivalent DPDA. This is an encounter with a very general fact: if one enhances the power of some representation format – that is, if one generalizes some mathematical concept – some (tpyically nice) properties of the more restricted concept are lost.

**Definition 4.14**. A PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is *deterministic* if the following two conditions are met:

1. For all $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$: $|\delta(q, a, X)| \leq 1$.
2. If for some $a \in \Sigma$, $q \in Q$, $X \in \Gamma$: $|\delta(q, a, X)| = 1$, then $\delta(q, \varepsilon, X)$ is empty.

We denote deterministic PDAs by the abbreviation DPDA.

All regular languages have DPDAs. To see this, take a DFA *A* for a regular language *L* and construct a DPDA *P* that has *A* as its control unit and simply does not use its stack. *P* accepts *L* by final state.

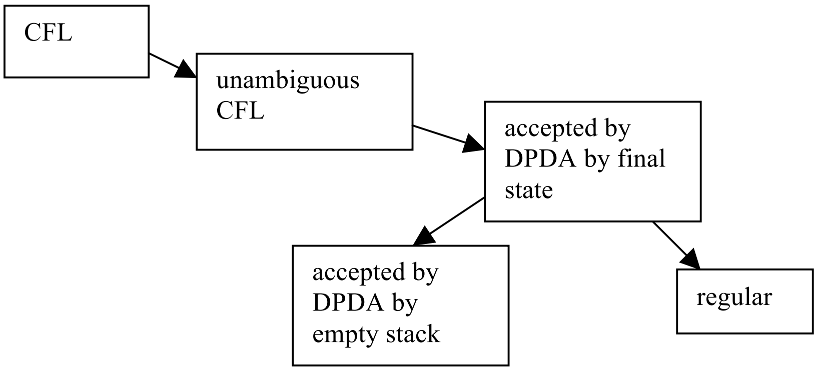
**Example 4.12.** There exist context-free languages that have DPDAs but are not regular. An example is $L = \{wcw^R \in \{0,1,c\}^* \mid w \in \{0,1\}^*\}$. The pumping lemma easily shows that *L* is not regular. Idea to construct a DPDA for *L*: while reading the first part *w* of an input word, copy it into the stack. When encountering *c*, switch (deterministically!) into a mode that reads-deletes the stack, comparing it to residual input $w^R$.

Thus, the languages acceptable by DPDAs are a proper superset of the regular languages.

There exist context-free languages that have no DPDA. An example is $L = \{ww^R \mid w \in \{0,1\}^*\}$. I found a proof in for this result in a classical paper by S. Ginsburg and S. Greibach[5] .

With DPDAs, acceptance by empty stack is not equivalent to acceptance by final state. To see this, consider the regular language 0*. It cannot be accepted by a DPDA by empty stack because in order to accept ε by empty stack, a DPDA must contain rules that on empty input lead to some sequence of ε-transitions that end in an empty stack. But the same PDA when started on a nonempty word *w*, has to go through the same sequence of ε-transitions initially (this follows from condition 2 of the definition of DPDAs). But then it has emptied its stack and cannot proceed even to read in just the first symbol of *w*. On the other hand, because 0* is regular, there exists a DPDA that accepts 0* by final state.

Each context-free language that is accepted by a DPDA (by final state or empty stack) has an unambiguous grammar (proof in HMU). Unfortunately, it is not true that the languages that have an unambiguous grammar are the languages accepted by DPDAs. All in all, we find the following proper inclusion hierarchy:


CFL
unambiguous CFL
accepted by DPDA by final state
accepted by DPDA by empty stack
regular

We witness that the world of CFLs is less transparent than the world of regular languages. We find non-equivalence of deterministic vs. non-deterministic automata, ambiguous vs. inherently ambiguous vs. unambiguous grammars (all non-decidable), and if we would deepen the treatment of CFLs, we would find more such pitfalls. But despite their not-so-

[5] Seymour Ginsburg, Sheila Greibach: Deterministic context free languages. Information and Control 9(6), 1966, 620-648

beautiful properties, CFLs are key for computer science, because programming languages fall into this class; furthermore, DPDAs are important for building parsers of programming languages, markup languages, database query languages, etc.

A more detailed treatment of DPDAs than can be found in the HMU book is given in the slides of a lecture given by Martin Fränzle from Osnabrück University; you can find them here: http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/folien28.10.pdf (originally retrieved from http://www.imm.dtu.dk/~mf/SprogModeller/folien28.10.pdf).

**Historical note.** The results reported in this section were all first found in the early 60ties, often independently by several researchers simultaneously. References in HMU page 253. The time was just ripe then.

## 4.6 Chomsky normal forms

This and the remaining subsections cover topics similar to the ones we treated for regular languages: normal forms, pumping lemma, closure properties. We will see that CFLs are more difficult to understand than regular languages, and have not as many "nice" properties. We start by presenting one of the normal forms for CFGs, the *Chomsky normal form* (CNF). A CFG will be said to be in *Chomsky normal form*, if all productions have the form $A \to BC$ or $A \to a$, that is, the head is replaced by two variables or by a single terminal.

Regular languages had a unique normal form representation: *the* minimal DFA. CFLs unfortunately have nothing similar. The best we can do is to transform CFGs into *a* (not: *the*) Chomsky normal form – for a given CFL, there may be several grammars in Chomsky normal form.

Note that with productions of the kind $A \to BC$ or $A \to a$ one can never derive ε. Therefore, only CFGs for languages that do not contain ε have CNFs. In the remainder of this subsection, we will see that every such language has a grammar in CNF.

Assume that we have a context-free grammar $G = (V, T, P, S)$, and that $\varepsilon \notin L(G)$. We arrive at an equivalent grammar in Chomsky normal form through a series of simplification and standardization steps: (1) eliminate ε-productions, (2) eliminate unit productions of the form $A \to B$, (3) eliminate useless symbols, (4) finally transform what we have "distilled" so far into a Chomsky normal form.

### Step 1: eliminate ε-productions

**Proposition 4.6** (eliminating ε-productions). Let $G = (V, T, P, S)$ be a CFG. Then we obtain a grammar $G' = (V, T, P', S)$ with $L(G') = L(G) \setminus \{\varepsilon\}$ as the result of the following two steps:

1) Detect all *nullable* variables $A$. A variable is called nullable if $A \Rightarrow^*_G \varepsilon$.
2) Initialize $P' =$ □.
3) Consider in turn all productions $A \to X_1X_2...X_k \in P$, $k \geq 1$, where $X_i \in T \cup V$. Let $m$ of the $X_i$ be nullable variables. Note that no such production is of the form $A \to \varepsilon$. For

each of the possible $2^m$ subset choices from the nullable $X_i$, create a new production from $A \to X_1X_2...X_{k'}$ where the chosen nullable $X_i$ are deleted in the body, and put this new production into $P'$. There is one exception: if $m = k$, do not put the production $A \to \varepsilon$ into $P'$. Note: the original production $A \to X_1X_2...X_k$ is also included into $P'$. Do this for all productions $A \to X_1X_2...X_k \in P$. Note: Productions $A \to \varepsilon \in P$ are thus not present in $P'$ any longer.

Proof of correctness: I would say it is obvious, but remember what I said in the lecture about the dangers of the word "obvious". □

**Inductive algorithm for finding the nullable variables.** Basis: all $A$ where $A \to \varepsilon \in P$ are nullable. Make NULL(1) the set of these $A$. Induction: Assume that a set NULL($n$) of nullable variables has been found in step $n$. Consider all productions $A \to B_1B_2... B_l$ where all $B_i$ are nullable variables in NULL($n$). Add such $A$ to NULL($n$) to obtain NULL($n$+1). Termination: stop when NULL($n$) = NULL($n$+1). – It remains to be shown that this algorithm is correct, that is, the final NULL($n$) is the set of nullable variables. It is obvious that NULL($n$) only contains nullable variables. To show that every nullable variable is in NULL($n$), perform an induction on the length of the shortest derivation $A \Rightarrow^*_G \varepsilon$. □

**Step 2: eliminate unit productions**

A production of the form $A \to B$, where $A$ and $B$ are grammar variables, is called a *unit production*. They are not necessary for building grammars and can be eliminated from a given grammar (although they may be useful, as for instance when we made our grammar for terms unambiguous, see example 6.10).

Terminology needed: Let $G = (V, T, P, S)$ be a CFG. If $A \Rightarrow^*_G B$, and all steps in this derivation are unit productions, we call $(A, B)$ a *unit pair* of $G$. Note that all $(A, A)$ are unit pairs.

**Proposition 4.7** (eliminating unit productions). Let $G = (V, T, P, S)$ be a CFG. Then we obtain a grammar $G' = (V, T, P', S)$ without unit productions if we carry out the following three steps.

1. Find all unit pairs of $G$.
2. Initialize $P'$ to $P$ minus all unit productions.
3. For each unit pair $(A, B)$, add to $P'$ all productions of the form $A \to \alpha$, where $B \to \alpha$ is a non-unit production in $P$.

Then $G'$ has no unit productions and $L(G) = L(G')$.

Proof of correctness: It is clear that $G'$ has no unit productions. $L(G) = L(G')$ is "almost obvious" if you think of parse trees. If $w \in L(G)$, then there exists a parse tree for $w$. Branches corresponding to sequences of unit productions $A_1 \to_G A_2 \to_G ... \to_G A_n \to_G \alpha$ with non-unit outcome $\alpha$ in this tree can be cut short by the single-step production $A_1 \to_{G'} \alpha$ from $P'$. Thus $w \in L(G')$. If conversely $w \in L(G')$ and we consider a parse tree for $w$ in $G'$, then obviously this parse tree can be turned into a parse tree for $w$ w.r.t. $G$, if we replace all productions of type $A_1 \to_{G'} \alpha$, which are not in $P$, by sequences $A_1 \to_G A_2 \to_G ... \to_G A_n \to_G \alpha$. Thus $w \in L(G)$. □

**Inductive algorithm for finding all unit pairs in a grammar.** Basis: Let PAIRS(1) be the set of all pairs of the form $(A, A)$. Induction: For all $(A, B)$ in PAIRS($n$) and all $B \rightarrow C$ in $P$, add $(A, C)$ to PAIRS($n$) to obtain PAIRS($n$+1). Termination: stop when PAIRS($n$) = PAIRS($n$+1).

It is clear that this algorithm yields exactly all unit pairs. □

**Step 3: eliminate useless symbols**

We define a symbol $X \in V \cup T$ to be *useful* if there exists a word $w \in L(G)$ with a derivation of the form $S \Rightarrow^* \alpha X\beta \Rightarrow^* w$, where $\alpha, \beta \in (V \cup T)^*$. If $X$ is not useful, it is *useless*.

Obviously, a symbol has to have two properties to be useful: it has to be *generating* and it has to be *reachable:*

We say that $X$ is generating if for some nonempty terminal word $u$, $X \Rightarrow^* u$. Note that every terminal symbol is generating by this definition. Notice furthermore that if $S \Rightarrow^* \alpha X\beta \Rightarrow^* w$, and $X$ is a variable, then $X \Rightarrow^* u$ for some *nonempty* subword $u$ of $w$ because we have eliminiated ε-productions in an earlier step.

$X$ is reachable if $S \Rightarrow^* \alpha X\beta$ for some $\alpha, \beta \in (V \cup T)^*$.

**Proposition 4.8** (eliminating useless symbols). Let $G = (V, T, P, S)$ be a CFG, with $L(G) \neq \emptyset$ and $\varepsilon$ □ $L(G)$. (Note that it can be decided whether $L(G) \neq \emptyset$ as a side-product of the *"Inductive algorithm for finding the generating symbols of a grammar G"* presented further below.) Let $G_1 = (V_1, T_1, P_1, S)$ be the grammar we obtain in the following two steps:

1) Delete from $G$ all nongenerating symbols and productions in which such symbols occur, to obtain $G_2 = (V_2, T_2, P_2, S)$. Note that $S$ must be generating because of $L(G) \neq \emptyset$.
2) Eliminate from $G_2$ all non-reachable symbols and productions in which such symbols occur, to obtain $G_1$.

Then $G_1$ has no useless symbols and $L(G_1) = L(G)$.

Proof: We first show that $G_1$ has no useless symbols. Suppose $X \in V_1 \cup T_1$. We know that $X \Rightarrow^*_G w$ for some $w \in T^*$ (mind that the subscript is $G$ and not $G_1$!). Because every symbol in the derivation $X \Rightarrow^*_G w$ is generating, we also know that $X \Rightarrow^*_{G2} w$. Because $X$ was not eliminated in the second step, we know that for some $\alpha, \beta \in (V_2 \cup T_2)^*$, $S \Rightarrow^*_{G2} \alpha X\beta$. Furthermore, every symbol used in this derivation is reachable, thus $S \Rightarrow^*_{G1} \alpha X\beta$. Furthermore we know that every symbol in $\alpha$ is reachable in $G_2$ (because of $S \Rightarrow^*_{G2} \alpha X\beta$) and generating in $G_2$ (because of $\alpha \in (V_2 \cup T_2)^*$). Therefore, for some terminal word $x \in T_2^*$, $\alpha \Rightarrow^*_{G2} x$. All symbols used in the derivation $\alpha \Rightarrow^*_{G2} x$ are also reachable in $G_2$, therefore $\alpha \Rightarrow^*_{G1} x$. Similarly, $\beta \Rightarrow^*_{G1} y$ for some $y \in T_2^*$. Assembling everything we get $S \Rightarrow^*_{G1} \alpha X\beta \Rightarrow^*_{G1} xwy$, and thus $X$ is not useless in $G_1$.

Next (and finally) we must show $L(G_1) \subseteq L(G)$ and $L(G) \subseteq L(G_1)$. The first is trivially true, because we stripped down $G$ to obtain $G_1$. The second is almost trivially true – because if $S \Rightarrow^*_G w$, then every symbol used in this derivation is generating in $G$ and therefore in $G_2$, and thus every symbol used in this derivation is reachable in $G_2$ and therefore in $G_1$. □

To round up this story we need algorithms to find the generating and reachable symbols in a grammar.

**Inductive algorithm for finding the generating symbols of a grammar** *G*. Basis: all terminal symbols are generating by definition. Make GEN(1) the set of terminal symbols. Induction: Assume that a set GEN($n$) of generating symbols has been found in step $n$. Consider all productions $A \to \alpha$, where $\alpha$ contains only symbols from GEN($n$). Add all variables $A$ that satisfy this requirement to GEN($n$) to obtain GEN($n$+1). Stop when GEN($n$) = GEN($n$+1) and put GEN($G$) = GEN($n$).

Proof of correctness: To show that GEN contains exactly the generating symbols of $G$, show that each symbol in GEN($G$) is generating (easy) and that every generating symbol $X$ of $G$ is in GEN($G$) (induction on length of generations $X \Rightarrow^*_G w$). □

**Algorithm for finding the reachable symbols of a grammar** *G*: Construct a directed graph whose nodes are all the terminal and variable symbols of $G$. For every rule $A \to \alpha$, for every symbol $X \in \alpha$, add an arc $A \to X$ to the graph. Then compute the set of all graph nodes that are reachable from $S$ – a known graph algorithm. The nodes in this set correspond to the reachable symbols.

Proof of correctness: obvious. □

Note: These two steps must be carried out in this order – an example where the reverse order fails can be found on page 256 in HMU.

We sum up our three "purification" steps:

**Proposition 4.9** If $G$ is a CFG for a language $L$ that contains at least one word other than ε, then there is another CFG $G'$ for the language $L \setminus \{\varepsilon\}$ that contains no ε-productions, no unit productions and no useless symbols.

Proof: Carry out the steps for eliminating ε-productions, unit productions, and useless symbols, in that order. After the first step, we have a grammar for $L \setminus \{\varepsilon\}$ that contains no ε-productions. If we then eliminate unit productions, we apparently don't re-introduce ε-productions, so afterwards we have a grammar for $L \setminus \{\varepsilon\}$ that contains no ε-productions and no unit productions. Finally, we eliminate useless symbols. As this method only eliminates symbols and rules, never re-introducing anything, we end up with a grammar for $L \setminus \{\varepsilon\}$ that contains no ε-productions, no unit productions and no useless symbols. □

**Theorem 4.10.** (Chomsky normal form). Let $L$ be a nonempty language not containing ε. Then there exists a grammar $G'$ for $L$ whose productions are all in one of two simple forms, namely

1. $A \to BC$, where $B$, $C$ are variables, or

2. $A \to a$, where $a$ is a terminal symbol.

Furthermore, $G'$ has no useless symbols. Such a grammar is said to be in *Chomsky normal form* (CNF).

Proof. By proposition 4.9 we know that $L$ has a grammar $G$ that contains no ε-productions, no unit productions and no useless symbols. Thus, all productions in $G$ are already of the desired form $A \to a$ or have a body of length 2 or more. We have to consider only the latter productions. We transform them into sets of equivalent productions of type 1. or 2., in two steps.

Step 1: arrange that all bodies of length 2 or more consist only of variables. This can be done as follows. If a terminal $a$ appears in some body, create a new variable $A_a$ , replace all occurrences of $a$ in rule bodies by $A_a$, and introduce a new rule $A_a \to a$.

Step 2: break productions with all-variable bodies of length 3 or more into sequences of productions of the form $A \to BC$. This can be done by introducing a new set of variables for each such production. For instance, replace $A \to BCDE$ by $A \to BC_1$, $C_1 \to CD_1$, $D_1 \to DE$.

Correctness: obvious. □

**Notes:**

1. A given language may have different grammars, all in CNF! Thus, the Chomsky normal form is not unique. This is bad news, because we saw earlier in this lecture that unique normal forms (= minimal DFAs) for regular languages entailed numerous decidability properties. We will soon see that in fact, many properties that can be decided for regular languages are undecidable for context-free languages.
2. The CNF is useful in various ways. Among other usages, it lays the grounds for a pumping lemma for CGL's (section 4.7), and it yields a test for deciding whether a word $w$ is in a context-free language $L$ (section 4.8).
3. There are other (non-unique) normal forms for context-free grammars. The most important one is the *Greibach* normal form for nonempty languages not containing ε. In a grammar that is in Greibach normal form, all productions are of the form $A \to a\alpha$, where $a$ is a terminal and α is a string of zero or more variables. The method to construct a Greibach normal form grammar is more complex than constructing Chomsky normal forms. One interesting consequence of the Greibach normal form is that in such grammars, a word of length $n$ has a derivation of length $n$, because each production introduces exactly one terminal symbol.

## 4.7 The pumping lemma for CFLs

*This subsection will be skipped in many years. It is included here for interested readers.*

A preparatory insight: parse trees in CNF grammars are binary trees (except at the branch tips where a unary branching leads into terminal leaves) and therefore the length of the longest branch gives a bound on the length of the terminal word:

**Proposition 4.11.** In a parse tree $t$ of a CNF grammar with yield $w$, where the length of the longest path in $t$ is $n$, it holds that $|w| \leq 2^{n-1}$. (Proof by induction over length of longest path in binary trees). Alternative statement: if the yield $w$ of $t$ has length $|w| \geq 2^{n-1}$, then $t$ must contain a path of length at least $n$. □

**Proposition 4.12** (Pumping Lemma for CFLs). Let $L$ be a CFL. Then there exists a constant $n$ such that every word $w \in L$ with $|w| \geq n$ can be written as $w = xuyvz$, with $x, u, y, v, z \in T^*$ such that

1. $|uyv| \leq n$,
2. $|uv| \geq 1$,
3. for all $i \geq 0$, $xu^iyv^iz \in L$.

**Proof (main idea):** first prove for finite $L$ (trivial – just choose $n$ greater than the length of longest word in $L$, statement of P.L. is trivially true). For infinite $L$, choose $n = 2^{|V|}$ and consider a parse tree for a word $w$ with $|w| \geq n$ in some CNF grammar for $L$. The tree must have a path $p$ of length at least $|V| + 1$ according to proposition 4.11. On this path, at least one variable $A \in V$ must occur twice. The rest of the argument formalises the geometrical idea that should become clear from Fig. 4.8. Note that the two dotted branches going downwards in the tree from the first occurrence of $A$ cannot both be empty, because a CNF does not contain unit productions; therefore, $u$, $v$ cannot both be empty. □

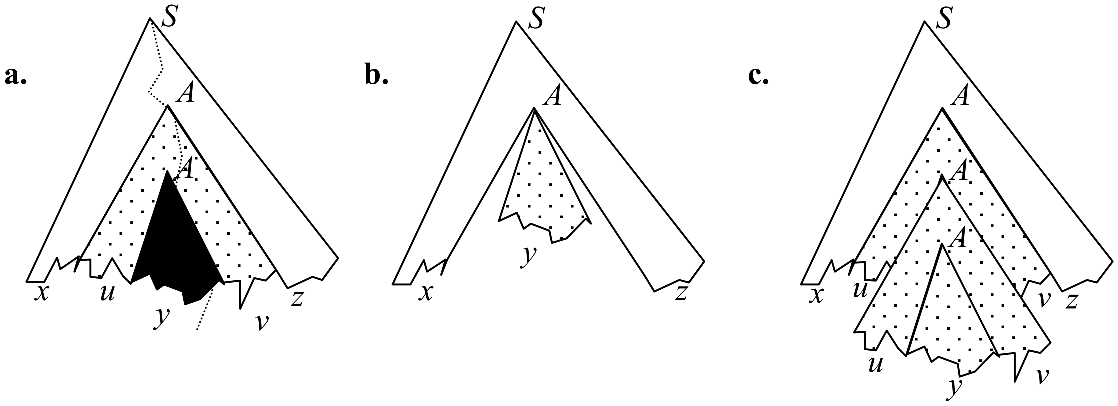


**Figure 4.8:** parse trees for *xyz* and *xuuyvvz* derived from parse tree for *xuyvz* according to pumping lemma for context-free languages.

The pumping lemma can be used to show that certain languages are not context-free. Example: $L = \{ww \mid w \text{ is in } \{0,1\}^*\}$ and $L = \{0^n1^n2^n \mid n \geq 1\}$ are not context-free.

There is a stronger version of the P.L., called Ogden's lemma:

**Proposition 4.13** (Ogden's Lemma for CFLs). Let $L$ be a CFL. Then there exists a constant $n$ such that in every word $w \in L$ with $|w| \geq n$, we can first arbitrarily select $n$ or more *distinguished positions*, that is, $n$ or more places within $w$. Then the word $w$ can be written as $w = xuyvz$, with $x, u, y, v, z \in T^*$ such that

1. $uyv$ covers at most $n$ distinguished positions,
2. $uv$ covers at least one distinguished position,
3. for all $i \geq 0$, $xu^iyv^iz \in L$.

The proof is actually the same as for the P.L., only that we pretend that the non-distinguished positions are not there in the parse tree we are using for our argument. □

With Ogden's lemma, one can prove more languages to be not context-free than with the P.L., or proofs become simpler. It's really helpful. For instance, the languages $\{0^i1^j2^k \mid k = \max\{i, j\}\}$ and $\{0^i1^i2^j \mid j \neq i\}$ can be shown to be not context-free.

## 4.8 Some closure and decision properties of CFLs

**Definition 4.15** (Substitution of symbols by languages).

1. Let $\Sigma = \{a_1, ..., a_n\}$ be an alphabet, and let $\Sigma_1, ..., \Sigma_n$ be alphabets and $L_1, ..., L_n$ languages over these alphabets. We write $L_i = s(a_i) = L_{a_i}$ [$i = 1, ..., n$] and say that $L_{a_i}$ is the language *substituted* for symbol $a_i$ by the substitution $s$.
2. If $w \in \Sigma^*$, $w = a^1...a^m$, then $s(w)$ is the language of all words $u$ of the form $u = u^1...u^m \in (\Sigma_1 \cup ... \cup \Sigma_n)^*$, that satisfy $u^j \in L_{aj}$ [for $j = 1, ..., m$].
3. If $L$ is a language over $\Sigma$, then $s(L) = \bigcup_{w \in L} s(w)$ is the language obtained from $L$ by the substitution $s$.

**Proposition 4.14** (Closure under substitution): If $L$ is a context-free langage over $\Sigma = \{a_1, ..., a_n\}$ and there is a substitution $s$ where the substitution languages $L_{a_i}$ are also context free, then $s(L)$ is context-free.

**Idea of proof:** combine a grammar for $L$ and grammars for the languages $L_{a_i}$ by first making all the variable sets of all these grammars disjoint, then identify the terminals $a_i$ of the grammar for $L$ with the starting symbols $S_i$ of the grammars for the languages $L_{a_i}$. □

Closure under substitution yields easy proofs for a number of other closure properties:

**Proposition 4.15** (Some elementary closure properties of CFLs). The context-free languages are closed under finite union, concatenation, Kleene star * and positive closure $^+$, and homomorphism.

**Proof:**

- Union: Let $L_1$ and $L_2$ be CFL's. Consider the CFL $L = \{1, 2\}$ [!] over $\Sigma = \{1, 2\}$. Then substitution $s(1) = L_1$, $s(2) = L_2$ does it.
- Concatenation: similar, use $L = \{12\}$.
- Kleene star *: similar, use $L = \{1\}^*$.
- Positive closure $^+$: similar, use $L = \{1\}^+$.

- Homomorphism: Let $L$ be a CFL over $\Sigma = \{a_1, ..., a_n\}$ , and $h: \Sigma \to (\Sigma')^*$ a homomorphism. Put $s(a_i) = \{h(a_i)\}$. Then $h(L) = s(L)$.

Note that the last case (homomorphisms) demonstrates a close similarity between homomorphisms of languages and substitution of symbols by languages. The only difference is that we defined homomorphisms to have a single target alphabet, whereas the target languages of "substitution of symbols by languages" may have different alphabets (not necessarily disjoint). One could also say, "homomorphisms are substitutions of symbols by languages where the target languages all have the same alphabet".

**Proposition 4.16** (closure under reversal). The CFL's are closed under reversal. □

Proof: reverse all bodies in rules of a grammar for a CFL to get a grammar for the reversed language. □

**Proposition 4.17** The CFL's are *not* closed under intersection and complement.

**Proof:** Intersection: the pumping lemma for CFLs can be used to show that $L = \{0^n1^n2^n \mid n \geq 1\}$ is not context-free. However, the languages $L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$ and $L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$ are context-free; they have simple grammars, for instance $L_1$ is generated by $S \to AB, A \to 0A1 \mid 01, B \to 2B \mid 2$. But $L$ is the intersection of $L_1$ with $L_2$. Thus, if CFL's were closed under intersection, $L$ must be context-free, too, which it isn't; therefore the CFL's cannot be closed under intersection. Complement: If the CFL's were closed under complement, they would also be closed under intersection, because intersection can be combined from complement and union. □

**Proposition 4.18** (Closure under intersection with a regular language). If $L$ is a CFL and $L'$ a regular language, then $L \cap L'$ is context-free.

Proof, idea: from a DFA $A'$ for $L'$ and a PDA $A$ for $L$ (accepting by final state) construct a PDA $B$ for $L \cap L'$ (likewise accepting by final state) by simulating $A'$ and $A$ in parallel, that is, $B$ has product states $(q',q)$, where $q'$ are states of $A'$ and $q$ are states of $A$, and the transition rules of $B$ act on these two state components independently according to the transition rules form $A'$ and $A$ (be a little careful here: we want a transition in the $A'$ part only if in the $A$ part a new symbol – not $\varepsilon$ – is processed); the accepting states in $B$ are the states $(q',q)$ where both $q'$ and $q$ are accepting in $A'$ and $A$, respectively. □

Next we turn to decision properties of CFLs. Decision problems for languages are of the general form,

- **given** a formal specification of a language (for instance via a grammar or an automaton), and
- **given** a language property (for instance, whether the language is empty or finite or contains some special word),
- **determine** by a deterministic procedure whether the specified language has this property.

The usual way to go about such decision problems is to proceed in two steps:

1. transform the given formal specification into some normal form,

2. use the normal form and a known algorithm that relies on this normal form to decide the property in question.

For CFLs, we know how any PDA can be transformed into a PDA that accepts by empty stack, how such a PDA can be transformed into a grammar, and grammars into Chomsky normal form. Thus, every type of formal specification that we are used to can be ultimately converted to CNF. For practical purposes, the runtime of conversion algorithms is of some interest. The HMU book discusses the various conversions, describing conversion algorithms of the following types and stating the time complexity:

- CFG to PDA that accepts by empty stack (complexity O($n$), where $n$ = description length of grammar),
- final-state-accepting PDA to empty-stack-accepting PDA (O($n$), where $n$ = description length of PD*A*),
- empty-stack-accepting PDA to final-state-accepting PDA (O($n$), where $n$ = description length of PDA)
- empty-stack-accepting PDA to CFG (O($n^3$), where $n$ = description length of PDA),
- CFG to CNF (O($n^2$), where $n$ = description length of grammar).

Some (few) elementary properties of context-free languages are decidable. Here they are:

**Proposition 4.19** It is decidable whether a CFL is empty.

A naive decision algorithm would first transform the given language specification into a grammar (if it is not already in grammar form) and re-use the algorithm for finding the generating symbols (cf. proof of Proposition 4.8) to find out whether the start symbol $S$ is generating. It is generating iff the language is non-empty. This naive algorithm has quadratic worst-case time complexity; HMU (p. 296) gives a more sophisticated algorithm that needs linear time. □

**Proposition 4.20** It is decidable whether a given word $w$ is in a context-free language $L$.

This is an important decision task – considering that compilers should check whether the program code they are about to process is syntactically correct.

A naive decision algorithm would first procure a CNF description of the language. Let $|w|$ = $n$. Any parse tree of $w$ (if it has one) has exactly $2n – 1$ non-leaf (variable) nodes. There are only finitely many parse trees at all in the grammar with that number of internal nodes. One may systematically generate them all and check whether one of them has yield $w$. Unfortunately, this algorithm has exponential time complexity and is therefore impractical. □

Because the membership decision task is of great practical importance, much work has been invested into exploring efficient decision algorithms. One of the most famous algorithms (famous because it came historically early and it is simple) is the *CYK* algorithm. It is named after J. Cocke, D. Younger and T. Kasami who independently discovered it in the mid-60ties.

**The CYK algorithm.** Let a context-free language $L$ be specified by a CNF grammar. Let $w = x_1...x_n$ be a word of length $n$ whose membership in $L$ has to be decided. In order to do this, for this given word $w$ construct a triangular table whose bottom line has length $n$, and whose entries $X_{ij}$ are arranged as indicated in the following figure that gives an example where $n$ = 4:

|  |  |  |  |
|---|---|---|---|
| $X_{14}$ |  |  |  |
| $X_{13}$ | $X_{24}$ |  |  |
| $X_{12}$ | $X_{23}$ | $X_{34}$ |  |
| $\underline{X_{11}}$ | $\underline{X_{22}}$ | $\underline{X_{33}}$ | $\underline{X_{44}}$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ |

Each $X_{ij}$ is a set of grammar variables, such that $A \in X_{ij}$ iff $A \Rightarrow^* x_i x_{i+1} \ldots x_j$, that is, $A$ can generate the substring of $w$ between (and including) positions $i$ and $j$. Clearly $w \in L$ iff $S \in X_{1n}$.

The sets $X_{ij}$ are constructed row-wise, bottom-up, inductively. Basis: In the first row, $X_{ii}$ must contain all variables $A$ such that $A \Rightarrow^* x_i$. Because the grammar is in CNF, this is equivalent to $A \rightarrow x_i$ being a production in the grammar, which can be checked in constant time for each $X_{ij}$. Induction: assume that all rows up to some row $k$ have been constructed. Let $X_{ij}$ be in row $k+1$, that is, $j - i + 1 = k + 1$. We compute $X_{ij}$ using the rows beneath it. Because our grammar is in CNF, it holds that $A \Rightarrow^* x_i x_{i+1} \ldots x_j$ iff there exists some $l$, with $i \leq l \leq j$, such that there exists a production $A \rightarrow BC$, and $B \Rightarrow^* x_i \ldots x_l$ and $C \Rightarrow^* x_{l+1} \ldots x_j$. In order to check for the existence of such pairs $B$, $C$ of variables, it suffices to inspect $X_{i'j'}$ computed in lower rows. Namely, for every $l$ with $i \leq l \leq j$, consider all $B$ in $X_{il}$; these are the $B$ that can generate $x_i \ldots x_l$. Similarly, $X_{(l+1)j}$ contains all $C$ with $C \Rightarrow^* x_{l+1} \ldots x_j$. Check for all pairs $B$, $C$ with $B \in X_{il}$ and $B \in X_{(l+1)j}$ whether $A \rightarrow BC$ is a production. This can be done in time proportional to $j - i$. □

Since every $X_{ij}$ can be computed in time at most O($n$), and there are $n^2/2$ such $X_{ij}$ to compute, the total time consumption of this algorithm is O($n^3$). Note that the grammar is fixed and enters the estimation of the runtime only as a constant!

The CYK algorithm can be cast into a form that is based on matrix multiplications. Using matrix multiplication algorithms whose cost is less than the O($n^3$) cost of the "naïve" matrix multiplication, the cost of CYK can be reduced to about O($n^{2.38}$). This is however a result of only theoretical value because the overhead hidden in the big-O is extremely large.

**Example 4.13.** Consider the CNF grammar

$S \rightarrow \quad AB \mid BC$
$A \rightarrow \quad BA \mid a$
$B \rightarrow \quad CC \mid b$
$C \rightarrow \quad AB \mid a$

To check whether *baaba* is in the language of this grammar, the table-filling algorithm constructs the following table and finds that $S$ is in the topmost table entry, thus *baaba* is in the language:

|  |  |  |  |
|---|---|---|---|
| $\{S,A,C\}$ |  |  |  |
| $\{\}$ | $\{S,A,C\}$ |  |  |
| $\{\}$ | $\{B\}$ | $\{B\}$ |  |
| $\{S,A\}$ | $\{B\}$ | $\{S,C\}$ | $\{S,A\}$ |

| | | | | |
|---|---|---|---|---|
| $\{B\}$ | $\{A,C\}$ | $\{A,C\}$ | $\{B\}$ | $\{A,C\}$ |
| $b$ | $a$ | $a$ | $b$ | $a$ |

**Proposition 4.21.** It is decidable whether a context-free language $L$ is finite.

A naive algorithm for checking finiteness would use the pumping lemma and the word membership decidability. Let $n$ be the pumping lemma constant. Decide whether $L$ contains a word of length $l$ with $n \leq l < 2n$. If yes, then $L$ is infinite, because any such word can be pumped to other words from $L$ in an infinite number of ways. If no, $L$ is finite. To see why, assume $L$ is infinite. Then let $w$ be a word of length $\geq 2n$, such that there exists no shorter word in $L$ of length $\geq 2n$. By the pumping lemma, we can write $w = xuyvz$ and we know that $xyz$ is in $L$, too. But the length of $xyz$ is shorter than the length of $w$, thus, $xyz$ must be shorter than $2n$. But $xyz$ must also be longer than $n$ due to the pumping lemma. Contradiction. □

The properties of emptiness, finiteness, word membership are the main properties that can be decided for CFLs. In contrast, the following properties cannot be decided:

- Is a CFG ambiguous?
- Is a CFG inherently ambiguous?
- Is the intersection of two CFLs empty?
- Are two CFGs generating the same language?
- Is a CFL equal to $\Sigma^*$?

Proving these indecidabilities needs techniques that you may learn in the lecture "Computability and Complexity".

# 5 Further types of formal languages

Grammars can be more general than the CFGs we are now familiar with. Here are two types of grammars that are more general:

**Definition 5.1:** An *unrestricted* grammar is a 4-tuple $G = (V, T, P, S)$, where $V$, $T$, $S$ are as in CFGs, but productions are of the form $\alpha \to \beta$, where $\alpha, \beta \in (V + T)^*$ are any strings of variables and / or terminals. Derivations in an unrestricted grammar proceed by replacing a substring $\alpha$ by a substring $\beta$ in sentential forms provided a production $\alpha \to \beta$ exists in $P$. More formally, for $\gamma_1, \gamma_2 \in (V + T)^*$ we have $\gamma_1 \Rightarrow_G \gamma_2$ iff $\gamma_1 = \delta_1\alpha\delta_2$, $\gamma_2 = \delta_1\beta\delta_2$, and $\alpha \to \beta \in P$.

Unrestricted grammars are also called *type-0 grammars*, *semi-Thue systems*, or *phrase structure grammars*.

The languages that can be generated by unrestricted grammars are the languages that can be generated by Turing machines (see lecture *Computability and Complexity*, next semester). This is tantamount to saying that these languages are the ones that can be generated by any computer program (on computers with infinite memory!). These languages are called the *recursively enumerable* languages.

The recursively enumerable languages form the largest class of languages that can be generated by finite means (grammars, automata, etc. are finite mathematical structures). The recursively enumerable languages have essentially no decidable properties. We will learn much more about them in *Computability and Complexity*.

**Definition 5.2:** An *context-sensitive* grammar is a 4-tuple $G = (V, T, P, S)$, where productions are of the form $\alpha \to \beta$, where $\alpha, \beta \in (V + T)^*$ are any strings of variables and / or terminals, and in addition to unrestricted grammars, we also have $| \alpha | \leq | \beta |$. Derivations are defined as usual.

Context-sensitive grammars are clearly more restricted than unrestricted grammars. However, they are extremely powerful and it is very difficult to find a language that is recursively enumerable but not context-sensitive. Virtually every formal language that one can reasonably invent is context-sensitive.

Unlike all types of languages considered so far, no natural automaton model for the context-sensitive languages is known, apart from a slightly awkward modification of Turing machines (restricting the length of the tape to a size proportional to the length of the input).

The context-sensitive grammars get their name from a normal form theorem stating that for each context-sensitive grammar there is an equivalent grammar whose productions have the form $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, where $A$ is a variable. The strings $\alpha_1$ and $\alpha_2$ are called the *context* of $A$. This also explains, finally, why the context-free languages are called context-free!

The context-sensitive grammars are mainly of historical interest because they appeared in the *Chomsky hierarchy* of languages, which is made from the regular, context-free, context-sensitive, and recursively enumerable languges. Originally this hierarchy was described by the linguist Noam Chomsky as modelling (portions of) natural languages.

**Modelling natural language.** Chomsky's influence on linguistics can hardly be over-estimated. His ideas have for many years determined the scientific view of a natural language as a set of "syntactically well-formed sentences" generated by, and fully described by, grammars. One could say, this perspective is *normative*, because it rests on the idea of syntactical *correctness*. The *Académie Française* and the German *Duden Verlag*, for instance, and other national institutions watching over the national language, would take this stance, too.

But in more recent years (let's say, since 25 years or so), other scientific perspectives on natural languages have emerged, which have widened the picture enormously. Here is a list of what I perceive as the most important other scientific (and engineering) approaches to natural languages.

**Language as a stochastic process.** Clearly, people when speaking make grammatical errors, jump in their thoughts, pronounce words in different ways, etc., etc., in a strange mixture of regularity and randomness. This calls for investigating the *statistics* of utterances, and modeling language as *stochastic sequence generation*. As a main workhorse, the formalism of *hidden Markov models* (HMMs) has established itself. It turned out that for computer-based speech processing systems (for instance, dictating programs or spoken word recognizers), the stochastic approach has been the most successful. For two decades (1990 – 2010), commercial systems for speech recognition have been built almost exclusively on HMMs. In the last few years however, HMM-based speech recognizers are becoming replaced by algorithms which are based on *neural networks*. These recognition systems likewise view language as a stochastic process.

**Language as carrier of information.** All of the approaches mentioned so far, including the grammar-based normative view, care little about the *meaning* of words, phrases and texts. They are syntax oriented. But of course, one cannot understand language without an account of *semantics.* The preferred mathematical tool for modelling this aspect is logic, which is all about the interplay between syntax and semantics – as we shall see in the remainder of this lecture. Computer-based systems for speech *understanding* (as opposed to mere *recognition*) are therefore mostly based on some logic formalism. An example would be automated translation systems – the source language message is first transformed into a logic-based representation of its semantic contents, from which then a target language message is derived. Note however that commercial automated translation systems (such as the one embedded in Google) do not do a semantic analysis of their input sentences. Instead, they rely on stochastic pattern matching (purely syntactic) and very large databases for training and retrieval of ready-made translation templates. In the last few years (since 2013 to be more precise), such "brute force" machine translation systems have been built on the basis of neural networks.

Modern scholarly linguistics is by and large a mixture of the grammar-based, logic-based, and stochastic-process-based approaches.

**Language as a nonlinear, self-organizing dynamical system.** Certain aspects of spoken language have a "dynamical" flavour, for instance intonation, rhythm, or singing. Such aspects are most naturally captured by the mathematical tools of nonlinear dynamical systems (ordinary and partial differential equations, artificial neural networks, coupled oscillators). It turned out that this kind of mathematical models also is most useful when modeling the neural dynamics in brains. Since language is undoubtedly produced by brains, there is a strong "natural" motivation to take this approach. The nonlinear dynamics approach can also explain in a very natural way certain phenomema of language learning in infants, for instance the sudden emergence of mastering new skill levels. However, as of today, the research programme of giving a comprehensive account of language in terms of dynamical systems (neural networks in particular) has not advanced very far.

**Language as a historically changing set of conventions.** Languages develop over time; one language may differentiate into new "descendent" languages, or become extinct, or altogether new languages may emerge from almost nothing (e.g., when an island becomes populated by people speaking many different native languages – then new *Creole* languages emerge in a highly organized, surprisingly fast and predictable fashion). In order to analyse language under this perspective, the relatively new field of *evolutionary linguistics* sees a language very much like a biological species which adapts to new ecological niches, differentiating into new species in the process. The main tool for this approach is computer simulations with artificial languages, which under various simulated evolutionary "pressures" can be demonstrated to develop new vocabularies and grammatical structures.

# Part 2: First-order predicate logic

Literature:

Uwe Schöning, Logic for Computer Scientists. Birkhäuser Verlag1989 (IRC: QA9 .S363 1989), Section 2. (Note: these ACS1 lecture notes are more detailed than the book).

H.-D. Ebbinghaus, J. Flum, W. Thomas, Mathematical logic. New York : Springer-Verlag, 1994 (IRC: QA9 .E2213 1994) (Note: this book is more detailed and more formal than these lecture notes).

## 6 Motivation

FOL is the standard mathematical formalism to describe facts – mathematical facts or others. Consider two examples, one from everyday life and one from mathematics:

| Fact: | FOL formula describing the fact: |
|---|---|
| "all humans are mortal" | $\forall x\ (\mathsf{Human}\ x \rightarrow \mathsf{Mortal}\ x)$<br>*read: "for all x, if x is human, then x is mortal"* |
| "every real number except 0 has a multiplicative inverse" | $\forall x\ ((\mathsf{Real}\ x \wedge \neg(x = 0)) \rightarrow \exists\, y\ (\mathsf{Real}\ y \wedge x \cdot y = 1))$<br>*read: "for all x, if x is a real number and x is not equal to 0, then there exists an y, such that y is a real number and* $x \cdot y = 1$*"* |

Like propositional logic, FOL uses the logical connectives $\wedge$, $\vee$ and $\neg$ (and derived ones like $\rightarrow$). But compared to propositional logic, FOL has the following additional ingredients:

- In FOL one can talk about *individual things* and denote them by variables (like $x$ or $y$) or by constant symbols (like 1).
- In FOL one can talk about *properties* of things, like being human or a real number. Such properties can be cast as formal symbols (e.g., Human or Mortal) and used in formulas.
- In FOL one can talk about *functions*, like the multiplication $\cdot$.
- In FOL one can state that something exists for which something holds, using expressions of the sort $\exists x$ (blah blah $x$ blah blah), and one can claim that something is true for all things, using $\forall x$ (blah blah $x$ blah blah). These are called *existential* and *universal* statements, respectively.

However, one cannot say everything in FOL. For instance, one cannot make existential or all statements about properties. It would be impossible to express in FOL: "I love all the qualities you have". More mundanely, it is impossible to express in FOL some important mathematical principles, for instance the principles of induction over natural numbers:

(*) $\forall P\,((P0 \wedge \forall\, n\,(Pn \rightarrow P(n{+}1))) \rightarrow \forall\, n\,(Pn))$

*read*: *"for all properties P, if 0 has it and if some n has it, then n+1 has it, too, then all n have this property"*

Statement (*) is an example of a *second-order* logic formula, where "second order" refers to the possibility to make existential or all statements about properties.

In fact, mathematics knows very many (infinitely many in fact) "logics". Their main difference lies in what they can express. Propositional logic is not very expressive, second-order logic is extremely expressive, and FOL lies somewhere in between. This should remind you of another hierarchy of expressiveness that by now you know very well: the Chomsky hierarchy of languages. You might compare Boolean logic to regular languages (both are about the simplest you can have), and FOL with the context-free languages (lying somewhere in the middle and being very useful). It might therefore seem that FOL is just a compromise and that there might be other nice compromise logics. However, that is not the case. FOL is a *superbly special* logic. The goal of the remaining lectures in this semester is to give you a thorough intuition about FOL and to make you appreciate the special role of FOL. For now, let it suffice to indicate that the special nature of FOL makes it *THE* logic for a host of applications:

- FOL is the language in which set theory is expressed. But set theory is the foundational theory of all mathematics; every mathematical structure can be re-cast as a set. Thus FOL is *THE* logic from which *mathematics* is buildt.
- FOL is the everyday language in maths. Except some rare cases (like the induction principle), almost everything that is stated in mathematical textbooks is stated in FOL (or some sloppy version thereof using words instead of symbols – but every mathematician is able to cast his/her statements in fully formal FOL). Thus FOL is *THE* logic for *mathematicians*.
- FOL is sufficient to express most practically relevant facts of real life. (You will rarely be meta-reasoning about properties; you typically make statements about things, and for all statements about things, FOL is enough). Thus, FOL is *THE* tool for Artificial Intelligence (AI). AI tools like "expert systems" or "knowledge bases" are written in variants of FOL. Check out *Watson_(computer)* and *Cyc* on Wikipedia to get an impression of how very large-scale FOL-fuelled knowledge bases can equal, or surpass, or be inferior to humans in interesting ways. Also take a look at http://www.youtube.com/watch?v=WFR3lOm_xhE and http://www.youtube.com/watch?v=d_yXV22O6n4 (links provided by Octavian Nasui).
- FOL is at the basis of a programming style called *logic programming*, a kind of declarative programming where a problem is "simply" described by FOL statements and the work of finding out how to do the necessary computations for solving the problem is left to the computer. The main language here is PROLOG. Logic programming is a preferred programming style in computer linguistics, where natural language input has to be processed both syntactically and semantically.
- Similarly, FOL is *THE* tool for other "knowledge-intensive" or "reasoning" applications in computer science, for instance for program verification tools or certain advanced databases (try Google on "deductive database"!).

You will see that this widespread use of FOL is not just a historical coincidence, but arises from some inbuilt properties of FOL which preciously single it out among all other logics.

# 7 Syntax and semantics of FOL

## 7.1 Syntax

The vocabulary of a FOL is adapted to particular domains. If you want to describe the world of natural numbers, you will need symbols like +, −, 0, 1, <, etc.; if you describe the domain of robot navigation, you will need symbols like goal, direction, landmark, etc. Depending on the target domain, there are different FOL vocabularies, hence FOL comes in many *languages*.

**Definition 7.1.** Each FOL *language* uses the following sorts of symbols:

1. Symbols which are shared by all FOL languages:
    a. variables $x_1, x_2, x_3, \ldots$ (for convenience we will also use $y$, $z$, and others),
    b. logical connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$,
    c. quantifiers $\exists$ and $\forall$,
    d. the identity symbol =, [6]
    e. brackets ( and ).
2. Domain-specific symbols:
    a. for each $n \geq 1$, a set (possibly empty) of $n$-ary predicate symbols,
    b. for each $n \geq 1$, a set (possibly empty) of $n$-ary function symbols,
    c. a set (possibly empty) of constant symbols.

Notes:

- When one talks about the *symbol set S* of a FOL, one typcially refers only to the collection of all its domain-specific symbols (the shared symbols are tacitly taken as granted). These domain-specific symbols are also called the *signature* of a FOL language.
- The symbol set *S* of a FOL language may be empty, finite, or infinite, even of arbitrary cardinality (so you could name every real number by a constant symbol!). Thus it is not called an "alphabet", because alphabets are by convention finite and non-empty.
- Another difference between logical symbols in *S* and the symbols we know from alphabets is that for each symbol in *S* there is declared a type (predicate, function, constant) and an arity, whereas the symbols we knew from formal language alphabets had none of these properties. To sum up: "alphabets" are just finite non-empty sets of different tokens, - that's it -, while a signature is a set (of arbitrary cardinality) of tokens with specified types and arities.
- You should think of constant symbols as names of (pointers to) individual things, for instance 0, my-laptop, Albert-Einstein.
- Unary predicate symbols denote properties of things, which the things may have or not have. Examples: is-human, is-blue, is-prime, is-greater-than-zero.

[6] Sometimes FOL is introduced without "=". If one wants to preclude misunderstandings, one may speak of "FOL with equality" or "FOL without equality".

- Binary predicate symbols denote relations between two things: for instance <, faster. In the pure syntax of FOL, prefix notation without brackets is used, for instance < 1 3 or is-faster cheetah snail.
- Predicate symbols of higher arity denote higher-order relations, as for instance in lies-between Vegesack Grohn Lesum.
- The scope of a relation symbol is implicitly understood because the arity of relation symbols is given with the symbol. However, in everyday use infix notation is common, e.g. 1 < 3, and in AI programming languages, bracketing is standard, for instance faster(leopard, snail).
- The words "predicate" or "relation" can be used interchangeably. Often "predicate" is used for unary relations, and "relation" for higher-arity ones, but this usage is not mandatory.
- Similarly, an *n*-ary function symbol is used with its arguments in prefix notation and without brackets in "pure" FOL, as in + 2 3 = 5. Again, you may use more convenient conventions if you like.

If the symbol set *S* is finite, a FOL language is actually a context-free language and could be specified by a grammar. However, this is not how FOL languages are specified in logics textbooks. Besites *S* being possibly infinite or empty, one reason for not using grammars is historical: the theory of FOL evolved in the decades around 1900, that is 50 years earlier than formal languages. The other reason has something to do with the fact that a FOL language is equipped with a *semantics*: the symbols *mean* something. The theory of formal languages, as we learnt to know it in the first part of this lecture, is solely concerned with the syntax of a formal language. In FOL languages, the additional aspects of semantics interact with the syntax. Therefore, the syntactical structures of FOL languages are introduced in an inductive fashion, *together with the semantics*.

In the theory of formal languages, the elements of a language are called *words*. We have learnt that this usage of "word" must be distinguished from our everyday interpretation of the word "word". For instance, the "words" of formal languages that specify programming languages are complete computer programs. The "words" of formal languages used by linguists to describe fragments of natural languages are utterances or texts. With FOL languages, the "words" are *expressions* or *formulas* like

$\forall x$ (Human $x \rightarrow$ Mortal $x$), or just
Human $x$, or
lies-between Vegesack Grohn Lesum, or
$\forall x$ ((Real $x \wedge \neg(x = 0)) \rightarrow \exists\, y$ (Real $y \wedge x \cdot y = 1$)).

We start our systematic introduction of the syntax of FOL by defining certain subwords that appear in the expressions from FOL languages. These subwords are called the *terms* of a FOL language. Semantically, the terms are those subwords within FOL expressions that denote individual things. We now define the syntax of terms, and then illustrate the semantics of terms with two examples. A precise definition of the semantics will be given later.

**Definition 7.2.** (Syntax of FOL terms with symbols from *S*). Given a symbol set *S*, the *terms* of the FOL language over *S* are defined inductively, as follows:

1. Each variable is an *S*-term.
2. Each constant from *S* is an *S*-term.

3. If $t_0, ..., t_{n-1}$ are $S$-terms, and $f$ is an $n$-ary function symbol from $S$, then $f\, t_0...t_{n-1}$ is an $S$-term.

**Remark.** For a finite signature $S$, we might cast this definition as a CFG. For instance, let $S = \{0, 1, +\}$ where 0 and 1 are constant symbols and + is a binary function. Then we could specify the $S$-terms by a CFG $G = (V, T, P, t)$ where

$V = \{t, \text{var}, \text{index}\}$
$T = \{x, {}_0, {}_1, 0, 1, +\}$

and we have the productions

$t \rightarrow \text{var} \mid 0 \mid 1 \mid +\, t\ t$
$\text{var} \rightarrow\ x\ {}_0\ \text{index} \mid\ x\ {}_1\ \text{index}$
$\text{index} \rightarrow {}_0\ \text{index} \mid {}_1\ \text{index} \mid \varepsilon$

**Example 7.1.** Let $S$ contain the constant symbols Anne, Stephen, the unary function symbol father-of, and the binary relation symbol is-the-father-of. Then the following are terms in the FOL language over $S$:

- Anne [semantics: this points to a particular person]
- Stephen [semantics: this denotes another particular person]
- $x_0$ [semantics: unspecified. Could be seen as a general "pointing device", like a finger, that might point to anything you want to.]
- father-of Anne [denotes the father of Anne]
- father-of father-of Anne [denotes the grandfather of Anne]
- father-of $x_0$ [denotes the father of somebody... we don't have this in natural language. Again, pointing to any person with the finger and saying, "the father of that one", would come close]

Note that the relation symbol is-father-of occurs nowhere in these examples. Terms cannot contain relation symbols.

**Example 7.2.** Let $S$ contain the constant symbol 0, the unary function $\sigma$ ("successor" function) and the binary function +. With these symbols we can specify natural numbers in a variety of ways. The following are examples of terms in the FOL language over $S$ which "point to" certain natural numbers:

- 0 [semantics: this symbol denotes the number 0]
- $x_1$ [semantics: again, think of this as if you were pointing to some natural number]
- $\sigma\, 0$ [denotes the successor of 0, that is, 1]
- $\sigma\, \sigma\, \sigma\, 0$ [denotes 3]
- $+\, \sigma\, 0\, \sigma\, \sigma\, \sigma\, 0$ [denotes 4]
- $+\, \sigma\, 0 + \sigma\, \sigma\, \sigma\, 0\, \sigma\, 0$ [denotes 5]

Our definition of the syntax of FOL is finished by specifying how one can build expressions from terms. Semantically, expressions are *statements of facts*. They can be true or false. Again, after the definition we give some informal examples.

**Definition 7.3** (Syntax of *S*-expressions; FOL language over *S*) Given a symbol set *S*, the *expressions* of the FOL language over *S* are defined inductively, as follows:

1. For *S*-terms $t_0$ and $t_1$, $t_0 = t_1$ is an *S*-expression.
2. For *S*-terms $t_0, ..., t_{n-1}$, and an *n*-ary relation symbol *R*, $Rt_0...t_{n-1}$ is an *S*-expression.
3. For an *S*-expression φ, ¬ φ is an *S*-expression.
4. For *S*-expressions φ and ψ, (φ ∧ ψ), (φ ∨ ψ), (φ → ψ), (φ ↔ ψ) are *S*-expressions.
5. For an *S*-expression φ and a variable *x*, ∃ *x* φ and ∀ *x* φ are *S*-expressions.

The set of *S*-expressions is denoted by $L^S$, that is, the first-order *language* over *S*.

**Note:** like in propositional logic, the implication → and the equivalence ↔ can be defined in terms of ∧ , ∨ and ¬:

$$(\varphi \rightarrow \psi) \;:\Leftrightarrow\; \neg\,(\varphi \wedge \neg\psi)$$
$$(\varphi \leftrightarrow \psi) \;:\Leftrightarrow\; ((\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi))$$

Including → and ↔ from the beginning is a matter of convenience.

**Example 7.3** (= example 7.1, continued). The following are *S*-expressions:

- Anne = Stephen [semantics: this states that Anne is Stephen – a false statement!]
- ¬ Anne = Stephen [semantics: this states that Anne is not the same thing as Stephen – a true statement!]
- is-father-of Stephen Anne [semantics: this states that Stephen is the father of Anne. Whether this is true or false depends on who Stephen and Anne are]
- Stephen = father-of Anne [same semantics as previous expression!]
- Stephen = Stephen [semantics: a tautology! always true independent of what individual is denoted by Stephen]
- ∃*x* *x* = father-of Anne [semantics: this claims that Anne has a father]
- ∀*x* (*x* = father-of Stephen → ¬ father-of *x* = Stephen)
  [this claims that Stephen cannot be his own grandfather]
- (∀*x* *x* = father-of Stephen → ¬ father-of *x* = Stephen)
  [this is syntactically correct, but its semantics is weird. It claims that if all individuals are Stephen's father, then whatever individual we may point to, its father is not Stephen. Note that the last *x* occurring in this expression is not "bound" by the ∀*x*; we would get a semantically equivalent expression by writing
  (∀*x* *x* = father-of Stephen → ¬ father-of *y* = Stephen).
  Although it makes no natural sense, the statement is true because
  ∀*x* *x* = father-of Stephen is false.]

**Example 7.4** (= example 7.2, continued). The following are *S*-expressions:

- $x = 0$ [this states that the thing we are pointing to by *x* is the number 0. This may be true or false, depending on what we are pointing at]
- $\forall x \neg\, x = \sigma\, x$ [this says that no number is its own successor, which is true]
- $\forall x\, (\neg\exists y\, x = \sigma\, y \ \rightarrow\ x = 0)$ [this says that 0 is the only number that is not the successor of another number]

Note that the syntax of *S*-expressions is maximally economical w.r.t. brackets. One is however allowed to add in extra brackets for better readability. For instance, instead of $\forall x \neg\, x = \sigma\, x$ one may also write $\forall x\, (\neg\, x \quad = \sigma\, x)$ or $\forall x\, (\neg(x = \sigma\, x))$ or $\forall x\, (\neg(x = \sigma\, (x)))$.

An important syntactical concept is that of the *scope* of quantifiers. For instance, in

$$\forall x\ \ (x = \text{father-of Stephen} \rightarrow \neg\ \text{father-of}\ x = \text{Stephen})$$

the last *x* lies in the scope of the all quantifier, whereas in

$$(\forall x\ \ x = \text{father-of Stephen} \rightarrow \neg\ \text{father-of}\ x = \text{Stephen})$$

the last *x* does not fall in the scope of the quantifier. To make this precise, we define when variables occur *free* (= not falling in the scope of any quantifier) vs. when they occur *bound* ( = falling in the scope of a quantifier).

**Definition 7.4** (free occurrence of variables in *S*-expressions). We inductively define the set free(φ) of variables that occur free in an expression φ. For any term *t*, let var(*t*) denote the variables that occur in *t*. Then define

$$\text{free}(t_0 = t_1) = \text{var}(t_0) \cup \text{var}(t_1)$$
$$\text{free}(Rt_0...t_{n\text{-}1}) = \text{var}(t_0) \cup ... \cup \text{var}(t_{n\text{-}1})$$
$$\text{free}(\neg\ \varphi) = \text{free}(\varphi)$$
$$\text{free}((\varphi * \psi)) = \text{free}(\varphi) \cup \text{free}(\psi) \qquad \text{for } * = \wedge, \vee, \rightarrow, \leftrightarrow$$
$$\text{free}(\exists x\ \varphi) = \text{free}(\varphi) \setminus \{x\}$$
$$\text{free}(\forall x\ \varphi) = \text{free}(\varphi) \setminus \{x\}$$

We say that *x* is *bound* by a quantifier in an expression $\forall x\ \varphi$ or $\exists x\ \varphi$ if *x* occurs free in φ.

Note that a variable may occur free in some subexpressions of an expression and bound in other subexpressions. For instance, in the expression

$$x + 1 = 2\ \wedge \forall x\ \ (x = 1 \rightarrow x > 0)$$

*x* occurs free in the first subexpression $x + 1 = 2$ and it occurs bound in the second subexpression; by Def. 7.4, *x* occurs free in the complete expression.

*S*-expressions which do not contain free variables are called *propositions*. Intuitively, they make verifiable statements about their topic domain, because without free variables, there is no situation-specific "pointing to things" required for understanding the meaning of the expression. For instance, the proposition

$$\forall x\ ((\text{Greek}\ x \wedge \neg\ \text{Socrates} = x) \rightarrow \text{wiser Socrates}\ x)$$

claims (verifiably... if you have a good test for wiseness...) that Socrates is the wisest of all Greeks.

### 7.2 Semantics of FOL

So far we have talked about the sematics of terms and expressions only in intuitive terms. Now we will give a rigorous account of FOL semantics. That a rigorous account of semantics is possible should wildly amaze you! – because, isn't the "stuff out there" *about* which one might wish to talk in a FOL language informal, messy, – just "worldly"? It is no wonder that it has taken philosophers and mathematicians more than 2000 years to figure out a convincing account of the semantics of logics, and the now generally agreed picture has emerged only in the first half of the last century. Landmark names connected with the devolpment of modern mathematical logic are Gotthold Frege (1848-1925), David Hilbert (1862-1943), Bertrand Russell (1872-1970), Kurt Gödel (1906-1978) and Alfred Tarski (1902-1983).

The only way to make the "stuff out there" accessible to a rigorous semantic analysis is to first procure an abstract, mathematical version of such stuff. The stuff from which the world is made – according to logicians and mathematicians – is sets.

The figure below tries to give an account of the three "kinds of world" involved in the game of logics and semantics.

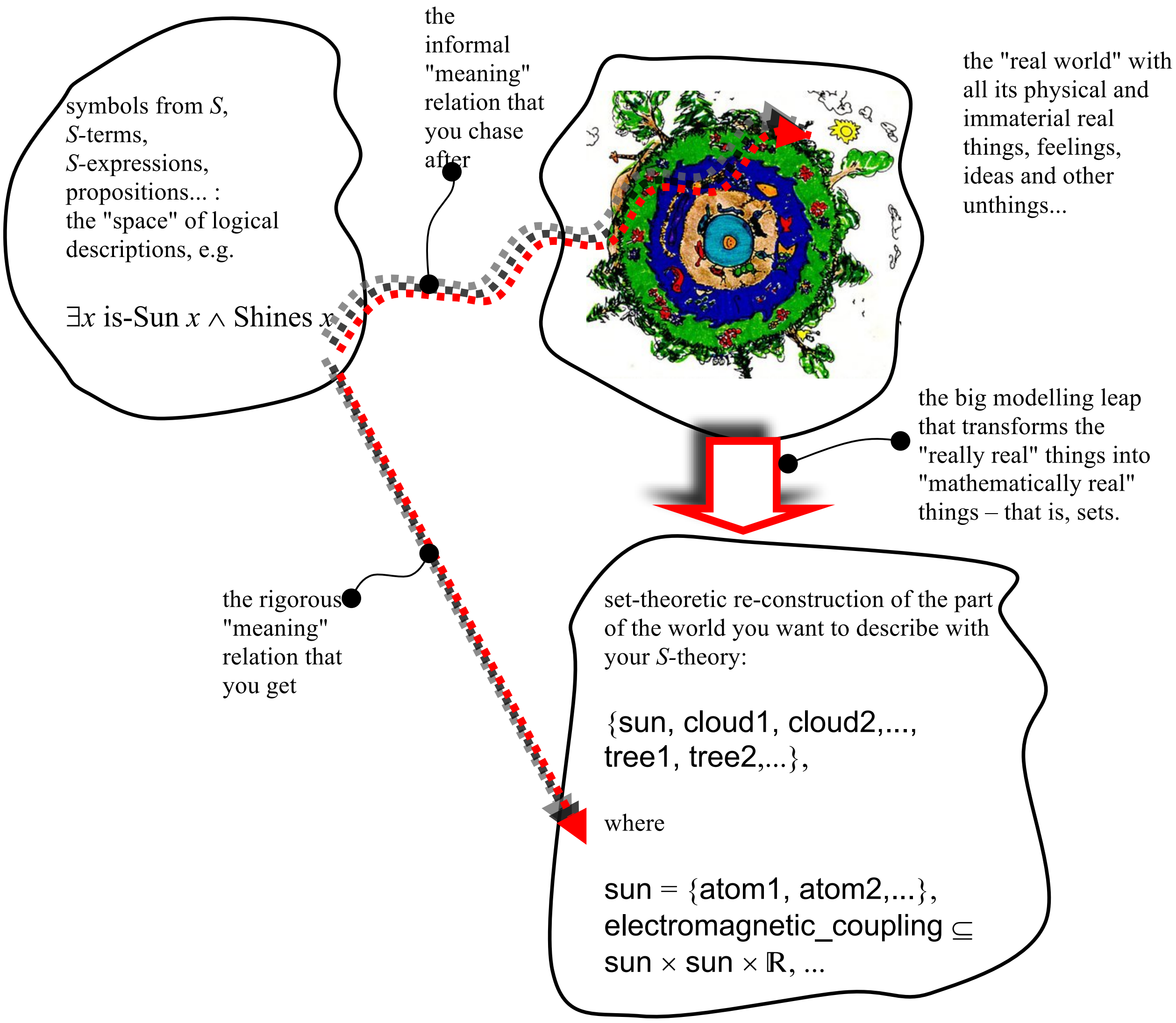

symbols from *S*, *S*-terms, *S*-expressions, propositions... : the "space" of logical descriptions, e.g.
$\exists x$ is-Sun $x \wedge$ Shines $x$
the informal "meaning" relation that you chase after
the "real world" with all its physical and immaterial real things, feelings, ideas and other unthings...
the big modelling leap that transforms the "really real" things into "mathematically real" things – that is, sets.
the rigorous "meaning" relation that you get
set-theoretic re-construction of the part of the world you want to describe with your *S*-theory:
{sun, cloud1, cloud2,..., tree1, tree2,...},
where
sun = {atom1, atom2,...}, electromagnetic_coupling $\subseteq$ sun $\times$ sun $\times$ $\mathbb{R}$, ...

**Figure 7.1** Three kinds of "stuff": The magic triangle of semantics. You *want* a rigorous semantic "meaning" relationship between logical expressions and the real things out there – this is impossible, because the world out there is just not made from math stuff (although that is open to philosophical dispute). So you first have to re-construct the world in the rigorous formalism of set theory, then you can establish a rigorous semantic relationship between logical expressions and the "set-things" in the reconstructed world. Believe it or not – it works. [world picture from http://www.kim-mckellar.com/round-world-card.html ]

It comes therefore as no surprise that modern logic could only blossom with the advent of modern set theory. In fact, set theory and logics are twins – since about the beginning of the last century they have been developed in close interaction, and good logicians must also be good set theorists and vice versa. In this subsection we will get a glimpse of how the world (at least the world of maths) can be construed as being made of sets.

In a first-order language we make statements about some given "domain of discourse", for instance the Anne, Stephen and fathership questions, or about natural numbers, or about the ancient community of Athens, or the environment of a mobile robot, or about any other fraction of reality. We want to obtain a precise notion of how and when a FOL proposition is true or false in the given domain. To that end, we first must formalize our vague notion of a "domain". Mathematicians (and AI systems, for that matter) have a very simple view of what they can be talking about. Essentially, a "domain of discourse" is a set (of all individual things within that domain). Properties and functions are defined as suitable subsets. The formal name for such a "domain of discourse" is *S-structure*.

**Definition 7.5** (*S*-structures).

Let $S$ be a set of predicate, function, and constant symbols. An $S$-structure $\mathcal{A}$ is a non-empty set $A$ (called the *domain* or the *carrier* of $\mathcal{A}$ – note that "domain" is here a technical term), wherein the following conditions are declared:

1. For each constant symbol $c$ of $S$, there is one fixed element of $A$, denoted by $c^{\mathcal{A}}$. For instance, if we talk about the real numbers, our signature $S$ will very likely contain the constant symbol 1 [a formal symbol from our language], and $1^{\mathcal{A}}$ then is the real number of unit size [a mathematical "thing"].
2. For each unary predicate symbol $P$ of $S$, there is a (possibly empty) subset $A_P$ of $A$, denoted by $P^{\mathcal{A}}$. For instance, when describing ancient Athens, is-dog$^{\mathcal{A}}$ is the set of all dogs in Athens. That is, in predicate logic we adhere to the so-called *extensional* interpretation of properties: a property "is" just the set of all things which have that property. In AI parlance one also speaks of the *class* of things that have the property.
3. For each binary predicate symbol $R$ of $S$, there is a (possibly empty) subset $A_P$ of $A \times A$, denoted by $R^{\mathcal{A}}$. That is, $R^{\mathcal{A}}$ is a set of pairs in $A$. Again, note the extensional perspective. For instance, when talking about the real numbers, $<^{\mathcal{A}}$ is the set of all pairs $(x, y)$ of real numbers where $x < y$. Predicates of higher arity are treated in a similar fashion.
4. For each unary function symbol $f$ of $S$, $f^{\mathcal{A}}$ is a unary function in $A$, that is, a function that takes arguments and values from $A$. Technically, one can define such a function again as a set of pairs: $f^{\mathcal{A}} = \{(x, y) \mid f^{\mathcal{A}}(x) = y\}$. Functions of higher arity are treated in a similar fashion.

Instead of $f^{\mathcal{A}}$ one often writes $f^{A}$ etc. In many important examples of (mathematical) structures, the signature $S$ is finite. A convenient and much-used way to write down such a structure is to put all its ingredients into a tuple, the domain first. For instance, the standard structure of arithmetics is described using the symbol set $S_{\mathrm{ar}} = \{+, \cdot, 0, 1\}$ and is denoted by

$$\mathcal{A}_{\mathrm{ar}} = (\mathbb{N}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}}).$$

If we want to describe facts related to integers being smaller or larger than others, the signature must be extended by < and the standard structure of arithmetics must be extended by $<^{\mathbb{N}}$. We then have another symbol set $S^{<}_{\mathrm{ar}} = \{<, +, \cdot, 0, 1\}$ and *another* structure

$$\mathcal{A}^{<}_{\mathrm{ar}} = (\mathbb{N}, <^{\mathbb{N}}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}}).$$

You may have sometimes wondered why in mathematical texts new entities (for instance, DFAs or grammars...) are introduced as "tuples". Now you know the answer: this is an echo from the foundations of maths where the structures of interest are – ... just that: tuples.

*Side remark:* the grouping of sets into a tuple yields a new set; so tuples are sets. As a simple example, consider two sets $A$ and $B$. Then, by the construction rules for tuple sets, the 2-tuple (or "ordered pair") $(A, B)$ is the set $\{\{A\}, \{A, B\}\}$. For instance, with $A = \{1\}$, $B = \{a, b\}$ we would get $(A, B) = \{\{\{1\}\}, \{\{1\},\{a, b\}\}\}$. Tuples of higher order are obtained by iteration, e.g. the 3-tuple $(A, B, C)$ is the nested pair $((A, B), C) = \{\{\{\{A\}, \{A, B\}\}\}, \{\{\{\{A\}, \{A, B\}\}\},\{C\}\}\}$.

Note that using FOL forces us to fix the matters that we want to describe beforehand: we can only argue "within" a signature $S$. If we want to extend it (for instance by starting to argue about <), we have to begin from scratch and start anew with an extended signature and structure. Previously obtained results of logical investigations carry over to the extended structure only if the new elements do not interfere with the old ones and are just "add-ons".

Sometimes, however, a new symbol can be *defined* within a FOL language $L^S$ and a particular $S$-structure, using the symbols from $S$. For instance, within $L^{S_{\mathrm{ar}}}$ and $\mathcal{A}_{\mathrm{ar}}$ we may define the constant symbol 2 by $\forall x\ (x = 2 \leftrightarrow x = 1 + 1)$, and will find that the only element of $\mathbb{N}$ in the structure $\mathcal{A}_{\mathrm{ar}}$ which satisfies this formula is the number 2. The structure $(\mathbb{N}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}})$ is thus essentially the same as the structure $(\mathbb{N}, +^{\mathbb{N}}, \cdot^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}}, 2^{\mathbb{N}})$. Another example: within $L^{S^{<}_{\mathrm{ar}}}$ and $\mathcal{A}^{<}_{\mathrm{ar}}$ the relation symbol $\geq$ can be defined by $\forall x\ \forall y\ (x \geq y \leftrightarrow \neg\, x < y)$. In such cases, we need not change the structure when we start using the new symbol, because wherever it is used it can be replaced by its definition. This is not always possible, of course. For instance, when we start with $S = \{+\}$ and $\mathcal{A} = (\mathbb{Z}, +^{\mathbb{Z}})$, there exists no FOL $S$-expression that could define the standard ordering relationship $<$ on $\mathbb{Z}$ (a nontrivial result). The structure $(\mathbb{Z}, +^{\mathbb{Z}})$ is thus in a fundamental sense different from the structure $(\mathbb{Z}, <^{\mathbb{Z}}, +^{\mathbb{Z}})$, at least as far as definability within FOL is concerned.

*Yet another, rather personal side remark.* The fact that the use of logics to describe fractions of reality is tied to an initial decision for a fixed symbol set makes it hard to use logics for describing *qualitative change*. For instance, one cannot use logic to describe evolutionary processes: as time passes on, new types of "things" (for instance, mammals) appear and old ones (e.g., dinosaurs) disappear. If one has mathematically described the world of dinosaurs as a structure, one cannot move the logical treatment forward in time: the new entities one wants to integrate into the picture (mammals) interfere with the old ones (dinosaurs) in many ways. This makes it very difficult to use logic to describe phenomena of learning, development and growth!

Now we have two interesting things in our hands: 1., $S$-expressions, that is statements in a formal language, and 2., $S$-structures, that is pieces of the world about which the $S$-expressions could start to talk. What remains is to formalize what it means that an $S$-expression "means something" in an $S$-structure.

We by now have an intuitive notion of what the symbols from $S$ "mean" (namely, $f$ "means" $f^A$ etc.) We now turn to the question of what the variables in an $S$-expression might "mean". This is the most technical part of FOL. Because variables are just that, *vary-able*s, there can be no unique "meaning" of a variable. A variable can denote anything – we said earlier that we can "point to" arbitrary things with a variable. Remember that technically we only admit variables of the form $x_0, x_1, \ldots$ (we used other symbols, like $x$ or $y$, just for convenience).

**Definition 7.6** (variable assignments). An *assignment* in an $S$-structure with domain $A$ is a mapping $\beta\colon \{x_0, x_1, \ldots\} \to A$.

Intuitively, an assignment specifies to what things variables point. After an assignment is given, variables work like constant symbols.

If we fix an assignment, we can interpret the (free) variables in $S$-expressions by the assignees, and we can proceed further toward defining what $S$-expressions "mean". The technical term for the combined interpretation of symbols from $S$ and variables under a fixed assignment is an $S$-interpretation:

**Definition 7.7** ($S$-interpretations). An *$S$-interpretation* $\mathcal{I}$ is a pair $(\mathcal{A}, \beta)$, where $\mathcal{A}$ is an $S$-structure and $\beta$ is an assignment.

Note that with $\mathcal{A}$ we know how to interpret symbols from $S$ and with $\beta$ we know how to interpret variables.

Often (namely, when we have to cope with variables unwantedly getting captured by quantifiers) we will have to rename variables. Renaming variables must be done carefully, because under a given assignment a variable is already "attached" to some element of $A$. To deal with this, we need a technical tool for adjusting assignments in one variable. We introduce the following notation:

**Definition 7.8** (re-assignment of a single variable). Let $\beta$ be an assignment, $a \in A$ and $x$ a variable. By $\beta\frac{a}{x}$ denote the assignment which acts like $\beta$ on all variables except $x$, and assigns $x$ to $a$:

$$\beta\frac{a}{x}(y) = \begin{cases} \beta(y) & for \quad x \neq y \\ a & for \quad x = y \end{cases}$$

If $\mathcal{I}$ is $(\mathcal{A}, \beta)$, let $\mathcal{I}\frac{a}{x}$ denote $(\mathcal{A}, \beta\frac{a}{x})$.

An interpretation assigns individual things (or "individuals" for short) to constants and variables. Constants and variables are the atomic versions of terms (compare Def. 7.2). We make this formal and extend it to composite terms:

**Definition 7.9** (interpretation of terms). Let $\mathcal{I} = (\mathcal{A}, \beta)$ be an $S$-interpretation.

1. For each variable $x$, $\mathcal{I}(x) = \beta(x)$.
2. For each constant symbol $c \in S$, $\mathcal{I}(c) = c^{\mathcal{A}}$.
3. For $n$-ary $f \in S$ and $S$-terms $t_0, ..., t_{n\text{-}1}$, $\mathcal{I}(f\, t_0 ... t_{n\text{-}1}) = f^{\mathcal{A}}\,(\mathcal{I}(t_0), ..., \mathcal{I}(t_{n\text{-}1}))$.

Now we have everything prepared to define in rigorous terms what it means for some $S$-expression $\varphi$ to "mean" something in an $S$-structure. The goal is to specify when a given $\varphi$ is true and when it is false. The truth or falseness of some $\varphi$ is relative to a given interpretation, because interpretations declare what the individual symbols occurring in $\varphi$ mean. Therefore, $\varphi$ can be true or false only relative to an interpretation $\mathcal{I}$. We say that $\mathcal{I}$ is a *model* of $\varphi$, or that $\mathcal{I}$ *satisfies* $\varphi$, or that $\varphi$ *holds* in $\mathcal{I}$, and write $\mathcal{I} \models \varphi$, if $\varphi$ is true relative to $\mathcal{I}$.

Technically, $\mathcal{I} \models \varphi$ is defined by induction over the structure of φ:

**Definition 7.10** (model relation between an interpretation and an *S*-expression or a set of *S*-expressions).

**a.** Let $\mathcal{A}$ be an *S*-structure. For all $\mathcal{I} = (\mathcal{A}, \beta)$ we define

$\mathcal{I} \models t_0 = t_1$ iff $\mathcal{I}(t_0) = \mathcal{I}(t_1)$

$\mathcal{I} \models Rt_0...t_{n-1}$ iff $R^{\mathcal{A}}\ \mathcal{I}(t_0) \ldots \mathcal{I}(t_{n-1})$
[that is, in extensional view, $(\mathcal{I}(t_0), ..., \mathcal{I}(t_{n-1})) \in R^{\mathcal{A}}$]

$\mathcal{I} \models \neg \varphi$ iff not $\mathcal{I} \models \varphi$

$\mathcal{I} \models (\varphi \wedge \psi)$ iff $\mathcal{I} \models \varphi$ and $\mathcal{I} \models \psi$

$\mathcal{I} \models \forall x\, \varphi$ iff for all $a \in A$, $\mathcal{I}\frac{a}{x} \models \varphi$

$\mathcal{I} \models \exists x\, \varphi$ iff for at least one $a \in A$, $\mathcal{I}\frac{a}{x} \models \varphi$

This definition can be extended by clauses dealing with ∨, →, and ↔ in an obvious manner.

**b.** If Φ is a set of *S*-expressions (possibly empty or infinite), we write $\mathcal{I} \models \Phi$ iff $\mathcal{I} \models \varphi$ for all $\varphi \in \Phi$.

Note that if φ is a *proposition*, the model relation does not depend on the assignment, that is, for $\mathcal{I}_1 = (\mathcal{A}, \beta_1)$ and $\mathcal{I}_2 = (\mathcal{A}, \beta_2)$, it holds that $\mathcal{I}_1 \models \varphi$ iff $\mathcal{I}_2 \models \varphi$. That is, for propositions φ we may write $\mathcal{A} \models \varphi$ instead of $\mathcal{I} \models \varphi$.

By a final twist, the model relation (which describes how a mathematical object is described by a formula) is turned into the *entailment* (or *implication*) relation (which describes how one formula or set of formulas entails another formula). Because the connection between the model and the entailment relation is so close, the same symbol ⊨ is used for both. I give the technical definition first and then some comments.

**Definition 7.11** (entailment) Let Φ be a set (empty, finite, or infinite) of *S*-expressions and φ an *S*-expression. Then Φ *entails* φ, written $\Phi \models \varphi$, iff every *S*-interpretation which is a model of Φ is also a model of φ. [More precisely: if every *S*-interpretation which is a model of every $\psi \in \Phi$, is also a model of φ.]

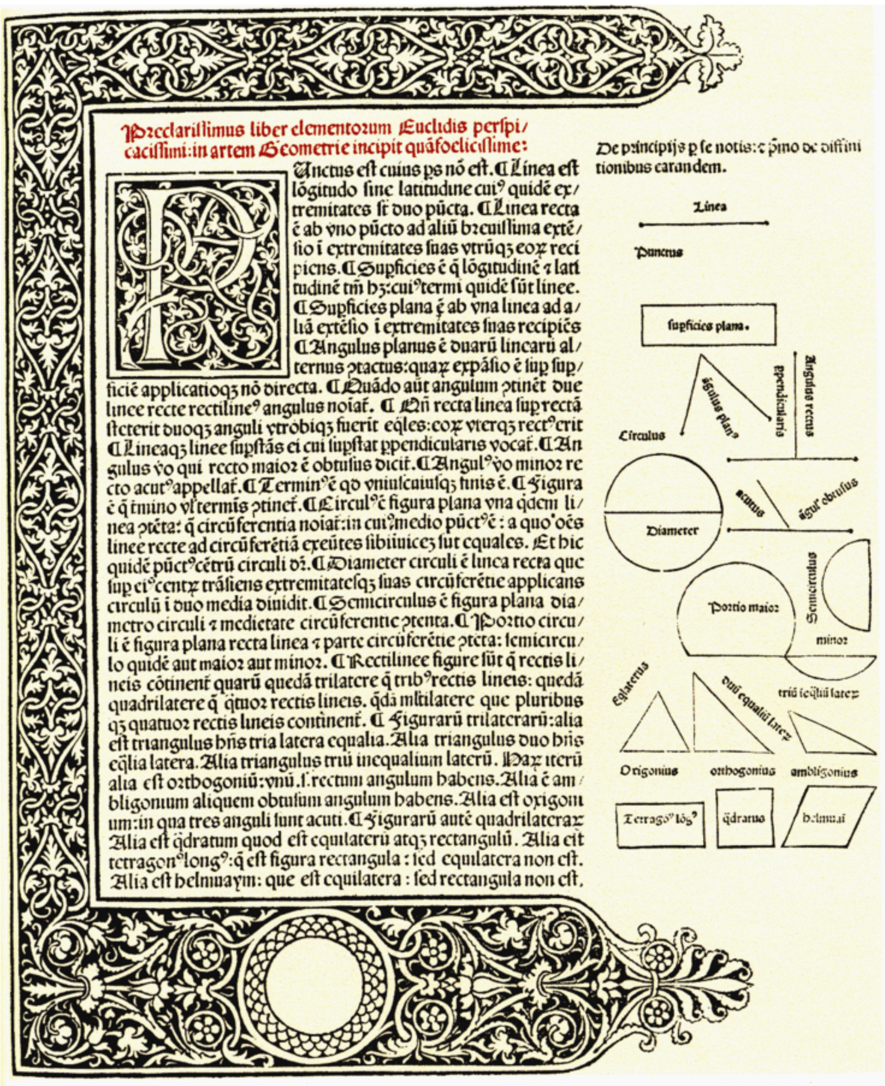
**Comment.** Definition 7.11 puts in a nutshell what today's mathematics considers the essence of mathematical truth. All mathematical statements are of the kind, "given Φ, it holds that φ". Here Φ usually comprises the set of assumptions that are provided as "givens" in the statement of a mathematical theorem, plus all the axioms of the mathematical area in which the theorem is embedded. φ is a statement which one claims follows from Φ. The claim is true, in the perspective of logics, if and only if $\Phi \models \varphi$. In intuitive terms, the essence of logical entailment could be expressed by

> " $\varphi$ follows from $\Phi$ if all mathematical objects for which $\Phi$ holds are also correctly described by $\varphi$".

Note that logical entailment $\Phi \models \varphi$ does not need the existence of a proof for $\varphi$ from $\Phi$. $\Phi \models \varphi$ holds, or does not hold, regardless of whether somebody has or has not already found a proof.

Logical entailment was first expressedly conceived in this way by Bernard Placidus Johann Nepomuk Bolzano in 1837, but he was far ahead of his time and remained largely ignored. Only after the formal appartus of formal logics was developed in the years 1880 – 1930, this conception was re-distilled and made explicit by Alfred Tarski in his work *The concept of truth in formalized languages* (original Polish publication 1933). Definition 7.11 is, in a sense, the most important definition of all mathematics, – in a sense it is even the definition of mathematics – because it describes what is a valid mathematical proof.

Mathematics is distinguished from other sciences, or even defined, by the fact that every result of mathematics must be *proven*. Hence the standard structure of mathematical texts as a succession of axioms, definitions, propositions, and proofs. This view on mathematics dates back (at least) to ancient Greece. With eternal beauty this nature of mathematics was displayed in Euclid's *Elements* (about 300 B.C.), a work which in 13 volumes developed a rigorous axiomatic treatment of the geometry and arithmetics of its time. The original version of this work is lost, but the book was traded down through history by copying and (starting in the middle ages) translations and printed editions. It is still available in print. A commented and animated online version is at http://aleph0.clarku.edu/~djoyce/java/elements/ .



Pꝛeclariſſimus liber elementoꝛum Euclidis perſpi/
caciſſimi: in artem Geometrie incipit quã foeliciſſime:

PUnctus eſt cuius ꝑs nõ eſt. ¶ Linea eſt
lõgitudo ſine latitudine cuiꝰ quidẽ ex/
tremitates ſũt duo pũcta. ¶ Linea recta
ẽ ab vno pũcto ad aliũ breuiſſima extẽ/
ſio i extremitates ſuas vtrũqꝫ eoꝝ reci/
piens. ¶ Supficies ẽ q̃ lõgitudinẽ ⁊ lati/
tudinẽ tm̃ hꝫ: cuiꝰ termi quidẽ ſũt linee.
¶ Supficies plana ẽ ab vna linea ad a/
liã extẽſio i extremitates ſuas recipiẽs
¶ Angulus planus ẽ duarũ linearũ al/
ternus ꝯtactus: quaꝝ expãſio ẽ ſup ſup/
ficiẽ applicatioqꝫ nõ directa. ¶ Quãdo aũt angulum ꝯtinẽt due
linee recte rectilineꝰ angulus noiat. ¶ Qñ recta linea ſup rectã
ſteterit duoqꝫ anguli vtrobiqꝫ fuerit eq̃les: eoꝝ vterqꝫ rectꝰ erit
¶ Lineaqꝫ linee ſupſtãs ei cui ſupſtat ꝑpendicularis vocat. ¶ An/
gulus vo qui recto maioꝛ ẽ obtuſus dicit. ¶ Angulꝰ vo minoꝛ re/
cto acutꝰ appellat. ¶ Terminꝰ ẽ qd vniuſcuiuſqꝫ finis ẽ. ¶ Figura
ẽ q̃ tmino vl termis ꝯtinet. ¶ Circulꝰ ẽ figura plana vna q̃dem li/
nea ꝯtẽta: q̃ circũferentia noiat: in cuiꝰ medio pũctꝰ ẽ: a quo oẽs
linee recte ad circũferẽtiã exeũtes ſibiinuicẽ ſut equales. Et hic
quidẽ pũctꝰ cẽtrũ circuli dr̃. ¶ Diameter circuli ẽ linea recta que
ſup eiꝰ centꝝ trãſiens extremitateſqꝫ ſuas circũferẽtie applicans
circulũ i duo media diuidit. ¶ Semicirculus ẽ figura plana dia/
metro circuli ⁊ medietate circũferentie ꝯtenta. ¶ Portio circu/
li ẽ figura plana recta linea ⁊ parte circũferẽtie ꝯtẽta: ſemicircu/
lo quidẽ aut maioꝛ aut minoꝛ. ¶ Rectilinee figure ſũt q̃ rectis li/
neis cõtinent quarũ quedã trilatere q̃ tribꝰ rectis lineis: quedã
quadrilatere q̃ q̈tuoꝛ rectis lineis. q̃dã mltilatere que pluribus
qꝫ quatuoꝛ rectis lineis continent. ¶ Figurarũ trilaterarũ: alia
eſt triangulus hñs tria latera equalia. Alia triangulus duo hñs
eq̃lia latera. Alia triangulus triũ inequalium laterũ. Haꝝ iterũ
alia eſt oꝛthogoniũ: vnũ .ſ. rectum angulum habens. Alia ẽ am/
bligonium aliquem obtuſum angulum habens. Alia eſt oxigoni/
um: in qua tres anguli ſunt acuti. ¶ Figurarũ autẽ quadrilateraꝝ
Alia eſt q̈dratum quod eſt equilaterũ atqꝫ rectangulũ. Alia eſt
tetragonꝰ longꝰ: q̃ eſt figura rectangula: ſed equilatera non eſt.
Alia eſt helmuaym: que eſt equilatera: ſed rectangula non eſt.

De pꝛincipijs p ſe notis: ⁊ p̃mo de diffini
tionibus earundem.

**Figure**: The first page of Euclid's Elements from the Latin edition by Campanus of Novara. Published by Erhard Ratdolt in 1482. From https://commons.wikimedia.org/wiki/Category:Elements_of_Euclid#/media/File:Euclid3a.gif

If you step back one bit further, you can think of $\Phi$ as a set of *axioms*, and the set of all $\varphi$ which are entailed by $\Phi$ as the *theory of* $\Phi$. For instance, take $\Phi$ as the axioms of group theory, then all $\varphi$ with $\Phi \vDash \varphi$ form the *theory of groups*. The systematic study of sets of axioms and their theories form the subject matter of a very active modern branch of mathematics called *model theory*. One main theme of model theory is to find out when a set $\Phi$ of axioms uniquely (up to isomorphims) determines the structures which are models of $\Phi$. It turns out that many well-known sets of axioms do *not* uniquely specify structures. For instance, there are "non-standard" models of the axioms of real numbers, which are not just exotisms but give rise to an interesting and useful *non-standard calculus*. Another famous example is that the classical axioms of geometry without the axiom of parallels ("two parallels never meet") has models in which parallels meet and other models in which they don't, so this particular axiom is independent of the others. Incidentally, as far as I know it is not clear whether in our universe parallels do meet – this question (whether the universe is "open", "closed", or "flat") still remains to be decided by astrophysics (?).

The next definition collects some basic semantical concepts of logics.

**Definition 7.12**

1. An $S$-expression $\varphi$ is *valid* or a *tautology* if it always holds, that is, if $\emptyset \models \varphi$. We also write $\models \varphi$ in this case.
2. An $S$-expression $\varphi$ is *invalid* or a *contradiction* if it never holds, that is, for all interpretations $\mathcal{I}$, $\mathcal{I} \models \varphi$ does not hold.
3. An $S$-expression $\varphi$ is *satisfiable* if there exists an interpretation $\mathcal{I}$ with $\mathcal{I} \models \varphi$.
4. Two $S$-expressions $\varphi$ and $\psi$ are *logically equivalent* if $\varphi \models \psi$ and $\psi \models \varphi$. This is also written as $\varphi \Leftrightarrow \psi$.

For later use we note the following little fact:

**Proposition 7.1:** For all $\Phi$ and all $\varphi$ it holds that $\Phi \models \varphi$ if $\Phi \cup \{ \neg\varphi \}$ is not satisfiable.

Proof: $\Phi \models \varphi$ iff every interpretation $\mathcal{I}$ with $\mathcal{I} \models \Phi$ is also a model of $\varphi$
iff there exists no interpretation $\mathcal{I}$ with $\mathcal{I} \models \Phi$ which is not a model of $\varphi$
iff there exists no interpretation $\mathcal{I}$ which is a model of $\Phi \cup \{ \neg\varphi \}$
iff $\Phi \cup \{ \neg\varphi \}$ is not satisfiable.

At this point I squeeze in some helpful hints for the everyday work with $\forall$ and $\exists$.

- $\forall$ can be expressed by $\exists$ and vice versa: $\forall\, x\, \varphi \Leftrightarrow \neg\exists\, x\, \neg\varphi$ and $\exists\, x\, \varphi \Leftrightarrow \neg\forall\, x\, \neg\varphi$. $\exists$ and $\forall$ are *dual* quantifiers.
- The all quantifier is mostly used to express statements of the form "All $x$ which satisfy $\varphi_1(x)$ also satisfy $\varphi_2(x)$ ... ", where $\varphi(x)$ is an $S$-expression with one free variable, namely, $x$. Example: for "all humans are mortal", put $\varphi_1(x) \equiv$ is-human $x$ and $\varphi_2(x) \equiv$ is-mortal $x$. This type of statement invariably leads to expressions of the form

$$\forall x\, (\varphi_1(x) \rightarrow \varphi_2(x)).$$

- Similarly, existential statements of the general form "there exists some $x$ of sort $\varphi_1(x)$ which satisfies $\varphi_2(x)$" invariably leads to expressions of the form

$$\exists x\, (\varphi_1(x) \wedge \varphi_2(x)).$$

# 8 A proof calculus for FOL

Logical entailment, that is $\Phi \models \varphi$, is a *semantic* concept, because it is defined using structures, that is, the "semantic denotation" of $\Phi$ and $\varphi$. Definition 7.11 tells us when a proof is correct, but does not help us to construct a proof in the first place. *The beauty and power of FOL lies in the fact that there is also a mechanism for finding such proofs (if they exist).* This mechanism takes the form of a "calculus", that is a purely mechanical-syntactical scheme to derive statements $\varphi$ from premises $\Phi$.

There are several ways of how one can specify the "mechanics" of a calculus, and different textbooks sometimes present different calculi. Here we present a specific kind of a calculus, namely, one whose rules are *sequent* rules. For an illustration of how sequent rules look and work, let's consider a type of argument that you often meet in proofs, namely, proving something by contradiction. Assume that you have some premises $\varphi_0, ..., \varphi_{n-1}$, from which you wish to deduce a consequence $\varphi$. You assume that $\varphi$ does not hold, and derive a contradiction from that assumption. This contradiction is revealed by that you can deduce some statement $\psi$ and also its negation $\neg\psi$ under the assumption that $\varphi$ does not hold. You conclude that $\varphi$ must hold. Formally, you may write this up in the following scheme of three sequences:

$$\begin{array}{l} \varphi_0 \ldots \varphi_{n-1} \;\; \neg\varphi \;\; \psi \\ \varphi_0 \ldots \varphi_{n-1} \;\; \neg\varphi \;\; \neg\psi \\ \hline \varphi_0 \ldots \varphi_{n-1} \;\; \varphi \end{array}$$

Explanation:

1. Each sequence claims that the last expression in the sequence follows from the first ones. For instance, the first sequence means, " $\psi$ follows from $\varphi_0, ..., \varphi_{n-1}$, and $\neg\varphi$", and the last sequence means "$\varphi$ follows from $\varphi_0, ..., \varphi_{n-1}$".
2. The ensemble of all three sequences is a *sequent rule.*
    - This sequent rule has two premises (the two lines above the bar) and one consequence (the line below).
    - The first premise is: "$\psi$ follows from $\varphi_0, ..., \varphi_{n-1}$, and $\neg\varphi$".
    - The second premise is: "$\neg\psi$ follows from $\varphi_0, ..., \varphi_{n-1}$, and $\neg\varphi$".
3. The consequence is: "$\varphi$ follows from $\varphi_0, ..., \varphi_{n-1}$".
4. That is, a sequent rule writes down premises (which are themselves claims that some expression follows from others), then draws a line, and then writes down the conclusion.
5. Each of the three lines is called a *sequent.* A sequent is a non-empty finite sequence of *S*-expressions, where the (single) last expression is entailed by all the previous ones. Note that this entailment "from left to right" within a sequent is the *semantic* entailment that we studied in the previous section. Each sequent therefore can be understood as a correct mathematical theorem!

In this way, numerous sequent rules can be written down which reflect basic types of mathematical argumentation. Here comes a basic list of such sequent rules. We will later see that these sequent rules are a *complete* set of rules, that is, *every* mathematical proof in FOL can be carried out using only these rules.

First we fix our notation: within a sequence, we may abbreviate a (possibly empty) starting sequence of assumptions by Γ. In this way, our sequent rule from above can be more concisely written as

$$\begin{array}{l} \Gamma\ \neg\varphi\ \ \psi \\ \Gamma\ \neg\varphi\ \ \neg\psi \\ \hline \Gamma\ \ \varphi \end{array}$$

There are other ways of specifying proof calculi for FOL, but we stick to sequent rules, because they are rather close matches of the way how mathematicians of flesh and blood actually work.

We now present all the rules of the FOL sequent calculus, discussing the first ones in more detail and speeding up as we gain experience.

**Antecedent rule** (Ant)

$$\begin{array}{l} \Gamma\ \ \varphi \\ \hline \Gamma'\ \ \varphi \end{array} \qquad \text{if } \Gamma \subseteq \Gamma'.$$

Comment: this rules simply states that if you add more assumptions, you can't prove less.

A crucial property of a rule is its *soundness*: if we use that rule in a proof, the proof should be sound, that is, the conclusion of the proof must follow from its premises. "Follow" here means logical entailment. That is, we must convince ourselves that conclusions $\Gamma'\ \ \varphi$ yielded by the Antecedent rule are indeed correct logical entailments, that is, we must show that $\Gamma' \vDash \varphi$ holds if $\Gamma \vDash \varphi$ holds:

*Proof of soundness* of (Ant): Assume that $\Gamma \vDash \varphi$. We must show that every model $\mathcal{I}$ of Γ' is also a model of φ. Let $\mathcal{I}$ be some model of Γ'. But if $\mathcal{I}$ is a model of Γ', then clearly $\mathcal{I}$ is also a model of Γ, because $\Gamma \subseteq \Gamma'$. Because of $\Gamma \vDash \varphi$, $\mathcal{I}$ is a model of φ.

**Premise rule** (Pre)

$$\begin{array}{l} \\ \hline \Gamma\ \ \varphi \end{array} \qquad \text{if } \varphi \in \Gamma.$$

Comment: (Pre) says that any φ follows from any set of assumptions which includes φ.

*Proof of soundness:* clear.

**Case distinction rule** (Cas)

$$\begin{array}{l} \Gamma\ \ \ \psi\ \ \varphi \\ \Gamma\ \neg\psi\ \ \varphi \\ \hline \Gamma\ \ \ \ \ \ \ \varphi \end{array}$$

Comment: this rule covers case distinction arguments: we conclude that $\varphi$ follows from $\Gamma$ if it follows from $\Gamma$ in the two cases when we assume $\psi$ and when we assume $\neg\psi$.

*Proof of soundness*: Assume that $\Gamma \cup \{\psi\} \vDash \varphi$ and $\Gamma \cup \{\neg\psi\} \vDash \varphi$. We have to show that $\Gamma \vDash \varphi$. Let $\mathcal{I}$ be any interpretation with $\mathcal{I} \vDash \Gamma$ (that is, $\mathcal{I} \vDash \xi$ for all $\xi \in \mathcal{I}$). Either it holds that $\mathcal{I} \vDash \psi$ or that $\mathcal{I} \vDash \neg\psi$. In the first case, because of $\Gamma \cup \{\psi\} \vDash \varphi$ one obtains $\mathcal{I} \vDash \varphi$. In the second case, one obtains $\mathcal{I} \vDash \varphi$ because of $\Gamma \cup \{\neg\psi\} \vDash \varphi$.

**Contradiction rule** (Con)

$$\begin{array}{ll} \Gamma & \psi \\ \Gamma & \neg\psi \\ \hline \Gamma & \varphi \end{array} \qquad \text{for any } \varphi.$$

Comment and sketch of proof of soundness: the two premises imply that $\Gamma$ is contradictory, that is, $\Gamma$ has no model. Therefore, $\Gamma \vDash \varphi$ holds for every *S*-expression $\varphi$.

**Rule of introducing $\vee$ in the antecedent** ($\vee$Ant)

$$\begin{array}{ll} \Gamma\ \varphi & \xi \\ \Gamma\ \psi & \xi \\ \hline \Gamma\ (\varphi \vee \psi) & \xi \end{array}$$

Comment: This rule says that if you can prove $\xi$ using additional assumption $\varphi$ or using additional assumption $\psi$, then you can prove $\xi$ using the additional assumption $(\varphi \vee \psi)$. Soundness proof is straightforward.

**Rules of introducing $\vee$ in the conclusion** ($\vee$Con)

$$\text{(a)} \quad \begin{array}{l} \Gamma\ \ \varphi \\ \hline \Gamma\ (\varphi \vee \psi) \end{array} \qquad\qquad \text{(b)} \quad \begin{array}{l} \Gamma\ \ \varphi \\ \hline \Gamma\ (\psi \vee \varphi) \end{array}$$

Comment: clear thing!

**Rule of introducing $\exists$ in the conclusion** ($\exists$Con)

$$\begin{array}{l} \Gamma\ \ \varphi\dfrac{t}{x} \\ \hline \Gamma\ \exists x\ \varphi \end{array}$$

Comments:

- $\varphi\frac{t}{x}$ denotes the *S*-expression φ', wherein all *free* occurrences of *x* in φ have been replaced by the term *t*. The exact definition of $\varphi\frac{t}{x}$ requires some care, because one must avoid that some variables in *t* are captured by quantifiers in φ. This may require variable renamings. We omit the technical treatment of defining $\varphi\frac{t}{x}$ in this lecture.
- This rule intuitively says that if we already have found an "example" *t* for *x* which makes φ follow from Γ, then also ∃*x* φ follows from Γ.
- The proof of soundness requires the apparatus of defining $\varphi\frac{t}{x}$ and we skip it.

**Rule of introducing ∃ in the assumption** (∃Ass)

$$\frac{\Gamma \quad \varphi\frac{y}{x} \quad \psi}{\Gamma \quad \exists x\varphi \quad \psi}$$

if *y* does not occur free somewhere in Γ ∃*x*φ ψ.

Comments: This rule is the most difficult to understand. Intuitively, the premise Γ $\varphi\frac{y}{x}$ ψ says that you have concluded ψ from Γ using an additional assumption $\varphi\frac{y}{x}$, that is, using φ with an "example" *y* used in φ at the place of *x*. Because *y* occurs nowhere else in Γ or ψ, nothing else is said about *y* at any other place in the proof and therefore nothing else can be relevant than its sheer existence. This leads to the consequence Γ ∃*x*φ ψ. The formal proof of soundness is rather involved and needs the full apparatus of variable substitution.

**Rule of reflexivity of** = (=)

$$\frac{}{t = t}$$

Needs no comment.

**Substitution rule for equality** (Sub)

$$\frac{\Gamma \qquad \varphi}{\Gamma \;\; x = t \quad \varphi\frac{t}{x}}$$

Comment: if previously you have shown that φ follows from Γ, you can conclude that $\varphi\frac{t}{x}$ follows from Γ if you additionally assume that *x* = *t*.

This finishes the list of all rules of our sequent calculus for FOL. Here's the complete list for your convenience:

(Ant)

$$\frac{\Gamma\ \ \varphi}{\Gamma'\ \varphi} \qquad \text{if } \Gamma \subseteq \Gamma'.$$

(Pre)

$$\frac{}{\Gamma\ \varphi} \qquad \text{if } \varphi \in \Gamma.$$

(Cas)

$$\frac{\begin{array}{lll}\Gamma & \psi & \varphi\\ \Gamma & \neg\psi & \varphi\end{array}}{\begin{array}{lll}\Gamma & & \varphi\end{array}}$$

(Con)

$$\frac{\begin{array}{ll}\Gamma & \psi\\ \Gamma & \neg\psi\end{array}}{\Gamma\quad \varphi} \qquad \text{for any } \varphi.$$

(∨Ant)

$$\frac{\begin{array}{ll}\Gamma\ \varphi & \xi\\ \Gamma\ \psi & \xi\end{array}}{\begin{array}{ll}\Gamma\ (\varphi \vee \psi) & \xi\end{array}}$$

(∨Con)

$$\text{(a)}\quad \frac{\Gamma\ \ \varphi}{\Gamma\ (\varphi \vee \psi)} \qquad \text{(b)}\quad \frac{\Gamma\ \ \varphi}{\Gamma\ (\psi \vee \varphi)}$$

for any ψ.

(∃Con)

$$\frac{\Gamma\ \ \varphi\frac{t}{x}}{\Gamma\ \exists x\ \varphi}$$

(∃Ass)

$$\frac{\Gamma\ \ \varphi\frac{y}{x}\ \ \psi}{\Gamma\ \ \exists x\varphi\ \ \psi} \qquad \text{if } y \text{ does not occur free in } \Gamma\ \exists x\varphi\ \psi.$$

(=)

$$\frac{}{t = t}$$

(Sub)

$$\frac{\Gamma \qquad\quad \varphi}{\Gamma\ \ x = t\ \ \varphi\frac{t}{x}}$$

Using this basic set of rules, other useful rules can be derived. As an example, we show how the *chain rule*

(Chain)

$$\frac{\begin{array}{lll}\Gamma & & \varphi\\ \Gamma & \varphi & \psi\end{array}}{\begin{array}{lll}\Gamma & & \psi\end{array}}$$

can be derived. I first give the formal derivation and then explain it.

Formal derivation of the chain rule:

| | | |
|---|---|---|
| 1. | $\Gamma \quad\quad \varphi$ | Premise |
| 2. | $\Gamma \quad \varphi \quad \psi$ | Premise |
| 3. | $\Gamma \quad \neg\varphi \; \varphi$ | (Ant) applied to 1. |
| 4. | $\Gamma \quad \neg\varphi \; \neg\varphi$ | (Pre) |
| 5. | $\Gamma \quad \neg\varphi \; \psi$ | (Con) applied to 3. and 4. |
| 6. | $\Gamma \quad\quad \psi$ | (Cas) applied to 2. and 5. |

Explanations:

- Each line in a formal derivation is a sequence of *S*-expressions.
- Each line must be justified. Either it is introduced as a premise (lines 1. and 2.) or it is derived by an application of one of our rules to previously written lines (remaining lines).
- The last line is what one wants as a conclusion line for the derived rule, here, the chain rule.

A crucial observation is that this formal derivation is a purely syntactical procedure which works mechanically on symbol strings. We do not need to consider interpretations, models or entailment to carry out such a formal derivation. It could be done by a machine which wouldn't have to understand anything about what the formulae *mean*.

However, this purely syntactical derivation of the chain rule guarantees that the chain rule is sound. Soundness is a semantic concept: it means that $\Gamma \vDash \psi$ if $\Gamma \vDash \varphi$ and $\Gamma \cup \{\varphi\} \vDash \psi$ . The soundness of the chain rule is inherited from the soundness of the rules (Ant), (Pre), (Con), (Cas) that we used in the derivation.

After having derived a rule like the chain rule from the basic rules, we may use the derived rules in further formal derivations. Dozens of other useful rules can be derived from our basic rules, and any textbook on logics contains several pages of them. Here is another derived rule which we will later use (can you explain in intuitive terms what this rule states?):

(DerivedRule2)

$$\frac{\Gamma \quad \forall x\, \varphi}{\Gamma \quad \varphi\frac{t}{x}} \qquad\qquad \text{for any } S\text{-term } t$$

The most important use of the rules of our sequent calculus (basic rules and derived rules) is not to derive more rules, but to derive proofs of "useful" mathematical statements. We sketch here an example from group theory.

First, we introduce the axioms of group theory as *S*-expressions. The basic concepts of group theory are the neutral element, which we will denote by *e*, and the group operation, which we denote by ∘. Thus, $S = \{e, \circ\}$, where *e* is a constant symbol and ∘ is a binary function symbol.

Here are the three axioms of elementary group theory, given as FOL *S*-expressions $\varphi_1$, $\varphi_2$, $\varphi_3$ (we use infix notation for better readability):

$\varphi_1$: $\forall x \forall y \forall z\ (x \circ y) \circ z = x \circ (y \circ z)$

$\varphi_2$: $\forall x\ x \circ e = x$

$\varphi_3$: $\forall x \exists y\ x \circ y = e$

We now want to prove the existence of a left inverse from these axioms, that is, we want to prove that

$$\{\varphi_1, \varphi_2, \varphi_3\} \models \forall x \exists y\ y \circ x = e.$$

Again, we proceed by writing down sequences in successive lines. The proof starts like follows:

| | | | |
|---|---|---|---|
| 1. | $\varphi_1\ \varphi_2\ \varphi_3$ | $\forall x\ x \circ e = x$ | (Pre) |
| 2. | $\varphi_1\ \varphi_2\ \varphi_3$ | $(y \circ x) \circ e = y \circ x$ | (DerivedRule2) applied to 1. with $t = y \circ x$ |
| | ... | | |

The proof now carries on for some 30 further lines which for illustration's sake are presented at the end of this section (taken from the German version of the book of Ebbinghaus/Flum/Thomas which is available at the IRC in the course materials for ACS 1). Each line is justified by a premise-free rule like (Pre) that allows us to write down certain lines, or by application of some rule to previously written down lines, as in the second line above we applied (DerivedRule2) to the first line. At the end, a final rule application gives us the desired conclusion

| | | |
|---|---|---|
| 34. | $\varphi_1\ \varphi_2\ \varphi_3$ | $\forall x \exists y\ y \circ x = e.$ |

We collect the essential points of this example in a definition of a formal proof:

**Definition 8.1** (formal proofs, derivations)

**(a)** Let $seq_1, ..., seq_n$ be a list of nonempty sequences of $S$-expressions, that is, $seq_i = \varphi_1 \ldots \varphi_{k_i}$, where $k_i > 0$, and where every sequence $seq_i$ can be either derived by applying a basic or derived rule to sequences that appear earlier in the list, or can be derived by applying a basic or derived rule which has no antecedent (such as rules (Pre) and (=)). Let $seq_n = \varphi_1 \ldots \varphi_{k_n-1}\ \psi$. The we say, that $\psi$ can be *derived*, or *formally proven*, from $\varphi_1 \ldots \varphi_{k_n-1}$, and denote this fact by $\{\varphi_1 \ldots \varphi_{k_n-1}\} \vdash \psi$.

**(b)** Let $\Phi$ be a set (possibly empty or infinite) of $S$-expressions, and $\psi$ an $S$-expression. Then we say that $\psi$ can be derived from $\Phi$, if there exists a finite subset $\{\varphi_1 \ldots \varphi_k\} \subseteq \Phi$ such that $\psi$ can be derived from $\varphi_1 \ldots \varphi_k$ in the sense of part (a).

In textbooks on group theory, the proof for the existence of a left-inverse takes a few lines only. Why do we need 30-something lines for our formal derivation? The reason is that in ordinary mathematical proofs, many "obvious" logical details are just tacitly filled in by the experienced intuition of human mathematicians. Formal proofs in FOL are much more detailed. In fact, they are so detailed that they become hard to understand (and to create) by human readers exactly because they are so replete with detail. FOL proofs reveal just how much logical fuel we humans pump into our proofs without being aware of it!

The positive aspect of the fact that FOL proofs contain *every* necessary detail is that machines (algorithms) can carry out such formal derivations. Machines don't have intuition and wouldn't know how to detect and omit "obvious" detail. Formal proof systems, like chess playing algorithms, replace insight by speed and memory space. Large parts of mathematics have been re-proven by proof algorithms which largely work like our sequent calculus, and it has been claimed that interesting new theorems have been found using such mechanical schemes. The problem with using automated proof systems to derive new theorems is that most of the time, these systems prove theorems which are (of course) correct, but uninteresting. If an automated proof generator happens to derive an "interesting" new theorem, it is difficult to become aware of that fact. Imagine that you receive a printout with a million true theorems of group theory, among which there is one which is both interesting and new. How would you find it?

The main use of automated proof systems is not to let them run on the loose to discover *something*, but to use them to verify a conjecture. That is, one already knows which conclusion you want to see as the result of the machine's working; it is not difficult to detect within a million formulae whether the one you are looking for is there.

One famous conjecture that has been proven with the assistance of an automated system is the four-color problem. This long-standing conjecture states that you can colorize any "political map" with only four colors such that no two adjacent nations get the same colour. Mathematicians had found that this question boils down to a case distinction of olympic measure: the problem dissociates into some 1000 different cases. A computer was successfully used to prove the conjecture for these cases in turn, and its programmer rose to fame.

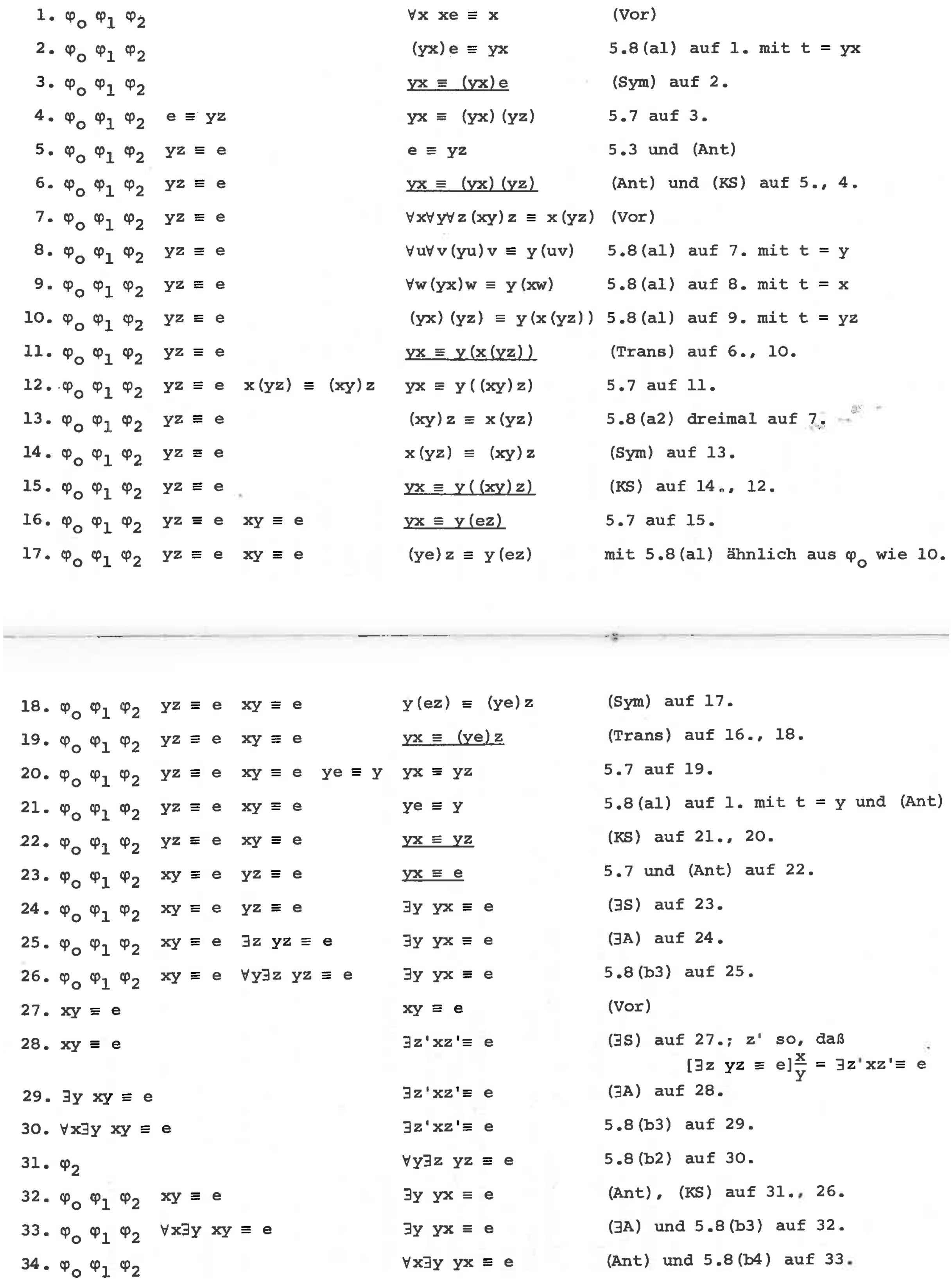
More mundane uses of formal proof systems are in program verification. The task is, given a computer program and a formal specification of its desired input-output behavior, show that the program always outputs the desired outputs. The formal verification of computer programs can be cast as a mathematical proof in FOL, and then handed over to a formal proof system. This is altogether a difficult, tedious, and computationally expensive labour, but numerous safety-critical applications require a formal program verification by law. Examples: the programs used for

- controlling spacecraft and missiles,
- autopilots,
- controlling nuclear power plants,
- prostheses control,
- robot control in high-power manufacturing robots (which would easily do a lot of damage when they go out of control),
- financial transactions,
- access systems (such as the transponder keys you use at Jacobs)

should be (and often are) verified.

Last, but not least, theorem proving algorithms are at the heart of the symbolic reasoning systems designed in the field of Artificial Intelligence. They span a wide range from expert systems, intelligent user interfaces, production line surveillance and fault monitoring systems, technical diagnosis systems, or speech understanding and translation systems. The largest, most famous (and most debated) of such systems is the CYC project, which was started in 1984 by Douglas Lenat. CYC's ultimate aim is to code all of human common-sense reasoning and everyday knowledge in a FOL-based format which allows to pose essentially arbitrary queries to this huge knowledge base. You should not miss to take a look at this project (e.g. visit its Wikipedia entry or the official homepage of the Cycorp company that today runs the project). Another large-scale project with a similar flavor is Wolfram Alpha (also at Wikipedia) – advertised by its creator as the predestined successor to Google.

Automated proof systems for FOL are typically not implemented using sequent calculi like the one we presented here. The sequent calculus is good for being hand-used by mathematicians, and it is good for an introductory lecture because its sequent rules are intuitively related to everyday mathematical arguments. However, for purposes of automated proofs carried out by computer programs, a rather different "calculus" is typically used, which is based on the idea of *resolution*. You can learn more about this in the book by Uwe Schöning (see course references), or you can try one of the many online courses for the programming language *Prolog* (list of such tutorials: http://www.freeprogrammingresources.com/prologtutr.html ). Prolog and its many dialects is *the* language for carrying out automated proofs. It is widely used in computational linguistics, artificial intelligence, and software verification. Section 10 (not mandatory for the course) of these lecture notes give a condensed intro to the mathematical foundations of resolution-based theorem provers.



| | | | |
|---|---|---|---|
| 1. | $\varphi_0\ \varphi_1\ \varphi_2$ | $\forall x\ xe \equiv x$ | (Vor) |
| 2. | $\varphi_0\ \varphi_1\ \varphi_2$ | $(yx)e \equiv yx$ | 5.8(a1) auf 1. mit t = yx |
| 3. | $\varphi_0\ \varphi_1\ \varphi_2$ | $\underline{yx \equiv (yx)e}$ | (Sym) auf 2. |
| 4. | $\varphi_0\ \varphi_1\ \varphi_2\ \ e \equiv yz$ | $yx \equiv (yx)(yz)$ | 5.7 auf 3. |
| 5. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $e \equiv yz$ | 5.3 und (Ant) |
| 6. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $\underline{yx \equiv (yx)(yz)}$ | (Ant) und (KS) auf 5., 4. |
| 7. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $\forall x \forall y \forall z (xy)z \equiv x(yz)$ | (Vor) |
| 8. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $\forall u \forall v (yu)v \equiv y(uv)$ | 5.8(a1) auf 7. mit t = y |
| 9. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $\forall w (yx)w \equiv y(xw)$ | 5.8(a1) auf 8. mit t = x |
| 10. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $(yx)(yz) \equiv y(x(yz))$ | 5.8(a1) auf 9. mit t = yz |
| 11. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $\underline{yx \equiv y(x(yz))}$ | (Trans) auf 6., 10. |
| 12. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ x(yz) \equiv (xy)z$ | $yx \equiv y((xy)z)$ | 5.7 auf 11. |
| 13. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $(xy)z \equiv x(yz)$ | 5.8(a2) dreimal auf 7. |
| 14. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $x(yz) \equiv (xy)z$ | (Sym) auf 13. |
| 15. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e$ | $\underline{yx \equiv y((xy)z)}$ | (KS) auf 14., 12. |
| 16. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e$ | $\underline{yx \equiv y(ez)}$ | 5.7 auf 15. |
| 17. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e$ | $(ye)z \equiv y(ez)$ | mit 5.8(a1) ähnlich aus $\varphi_0$ wie 10. |
| 18. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e$ | $y(ez) \equiv (ye)z$ | (Sym) auf 17. |
| 19. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e$ | $\underline{yx \equiv (ye)z}$ | (Trans) auf 16., 18. |
| 20. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e\ \ ye \equiv y$ | $yx \equiv yz$ | 5.7 auf 19. |
| 21. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e$ | $ye \equiv y$ | 5.8(a1) auf 1. mit t = y und (Ant) |
| 22. | $\varphi_0\ \varphi_1\ \varphi_2\ \ yz \equiv e\ \ xy \equiv e$ | $\underline{yx \equiv yz}$ | (KS) auf 21., 20. |
| 23. | $\varphi_0\ \varphi_1\ \varphi_2\ \ xy \equiv e\ \ yz \equiv e$ | $\underline{yx \equiv e}$ | 5.7 und (Ant) auf 22. |
| 24. | $\varphi_0\ \varphi_1\ \varphi_2\ \ xy \equiv e\ \ yz \equiv e$ | $\exists y\ yx \equiv e$ | (∃S) auf 23. |
| 25. | $\varphi_0\ \varphi_1\ \varphi_2\ \ xy \equiv e\ \ \exists z\ yz \equiv e$ | $\exists y\ yx \equiv e$ | (∃A) auf 24. |
| 26. | $\varphi_0\ \varphi_1\ \varphi_2\ \ xy \equiv e\ \ \forall y \exists z\ yz \equiv e$ | $\exists y\ yx \equiv e$ | 5.8(b3) auf 25. |
| 27. | $xy \equiv e$ | $xy \equiv e$ | (Vor) |
| 28. | $xy \equiv e$ | $\exists z' xz' \equiv e$ | (∃S) auf 27.; z' so, daß<br>$[\exists z\ yz \equiv e]\frac{x}{y} = \exists z' xz' \equiv e$ |
| 29. | $\exists y\ xy \equiv e$ | $\exists z' xz' \equiv e$ | (∃A) auf 28. |
| 30. | $\forall x \exists y\ xy \equiv e$ | $\exists z' xz' \equiv e$ | 5.8(b3) auf 29. |
| 31. | $\varphi_2$ | $\forall y \exists z\ yz \equiv e$ | 5.8(b2) auf 30. |
| 32. | $\varphi_0\ \varphi_1\ \varphi_2\ \ xy \equiv e$ | $\exists y\ yx \equiv e$ | (Ant), (KS) auf 31., 26. |
| 33. | $\varphi_0\ \varphi_1\ \varphi_2\ \ \forall x \exists y\ xy \equiv e$ | $\exists y\ yx \equiv e$ | (∃A) und 5.8(b3) auf 32. |
| 34. | $\varphi_0\ \varphi_1\ \varphi_2$ | $\forall x \exists y\ yx \equiv e$ | (Ant) und 5.8(b4) auf 33. |

**Figure 8.1.** The full deriviation of a simple group-theoretic fact (from Ebbinghaus/Flum/Thomas, Einführung in die mathematische Logik (Introduction to mathematical logic, english version available in the IRC).

# 9 The big theorems for FOL

In this section we will finally learn why FOL is such a "singular beauty" among all logics. In a nutshell, in FOL the semantic aspects (as expressed by $\models$) are in perfect balance with the syntactical aspects (expressed by $\vdash$). In fact, in FOL, and essentially *only* in FOL, semantics and syntax coincide. We first establish the easy direction of this equivalence:

**Proposition 9.1** (soundness of FOL sequent calculus). If $\Phi \vdash \varphi$, then $\Phi \models \varphi$.

Proof: This is a direct consequence of the soundness of our derivation rules.

In simple words, the soundness of the FOL sequent calculus means that if we (or a machine) carry out a mechanical proof in the syntactical sense of $\Phi \vdash \varphi$, then it is actually a sound proof in the semantic sense of $\Phi \models \varphi$.

Soundness is a property that we will of course require from any logic and formal proof system. There are hundreds of logics and associated proof systems; all are sound. By far the more precious property of FOL and the sequent calculus is that also the other direction holds: if some *S*-expression $\varphi$ is entailed by a set $\Phi$ of premises (that is if $\Phi \models \varphi$), then we can also formally derive this fact, that is, $\Phi \vdash \varphi$. This is called the *completeness* property of FOL:

**Theorem 9.2** (completeness of FOL sequent calculus). If $\Phi \models \varphi$, then $\Phi \vdash \varphi$.

Completeness of FOL was first shown by Gödel (by whom else!); the proof given in today's textbooks is due to Henkin. It is technically involved, but the basic idea is rather easy to grasp and very illuminating. I will outline its basic idea.

We first need some elementary concepts that concern the relationship between satisfiability (a semantic notion) and an anlog syntactic concept called *consistency*.

**Definition 9.1** A set $\Phi$ of *S*-expressions is called *inconsistent* if there exists some $\varphi$ such that $\Phi \vdash \varphi$ and $\Phi \vdash \neg\varphi$. If $\Phi$ is not inconsistent, it is *consistent*.

Here are two basic observations concerning consistency:

- If $\Phi$ is inconsistent, then *every* *S*-expression $\psi$ can be derived from it, that is $\Phi \vdash \psi$ for all $\psi$. This is a consequence of rule (Con).
- The following two statements are equivalent: (1) $\Phi$ is consistent, and (2) there exists an *S*-expression $\psi$ which cannot be derived from $\Phi$. Again, this can be shown essentially by using (Con).

We will also need the following little fact:

**Proposition 9.3.** If not $\Phi \vdash \varphi$, then $\Phi \cup \{ \neg\varphi \}$ is consistent.

Proof: Let not $\Phi \vdash \varphi$. Assume that $\Phi \cup \{ \neg\varphi \}$ is inconsistent. Then, for some suitable finite $\Gamma \subseteq \Phi$, there exists a derivation of the sequence $\Gamma\ \neg\varphi\ \varphi$. Continuing the derivation from that sequence, we find

$$\begin{array}{lll} & \ldots & \\ \Gamma & \neg\varphi\ \varphi & \\ \Gamma & \varphi\ \varphi & \text{(Pre)} \\ \Gamma & \varphi & \text{(Cas) applied to previous two lines} \end{array}$$

that is, $\Phi \vdash \varphi$, a contradiction.

Next we mention a property of consistent $\Phi$ which establishes a link between semantics and syntax:

**Proposition 9.4:** If $\Phi$ is satisfiable, then it is consistent.

Proof: Assume that $\Phi$ is inconsistent. Then for some $\varphi$, $\Phi \vdash \varphi$ and $\Phi \vdash \neg\varphi$. By soundness of FOL, it holds that $\Phi \vDash \varphi$ and $\Phi \vDash \neg\varphi$. Using the definition of $\vDash$, (Def. 7.10, case "$\neg$"), we may conclude that $\Phi$ is not satisfiable.

Now we are prepared to take a glimpse at the proof of the completeness theorem. We approach it sideways, by first showing that the statement of the completeness theorem follows from another, more accessible statement:

(*) Every consistent set $\Phi$ of $S$-expressions is satisfiable.

From (*) follows the completeness theorem, as follows. Assume that for some $\Phi$, $\varphi$ the completeness theorem does not hold, that is, $\Phi \vDash \varphi$ but not $\Phi \vdash \varphi$. Consider the set $\Phi \cup \{\neg\varphi\}$. It is not satisfiable because of $\Phi \vDash \varphi$. On the other hand, $\Phi \cup \{\neg\varphi\}$ is consistent by Prop 9.3. This contradicts (*). Therefore, from (*) the completeness theorem follows.

Thus, in order to prove the completeness theorem, it suffices to show that every consistent set $\Phi$ is satisfiable. That is, given some $\Phi$, we have to construct an interpretation $(\mathcal{A}, \beta)$ which is a model of $\Phi$, that is, $(\mathcal{A}, \beta) \vDash \Phi$. All that we have in our hands to construct such a model is $\Phi$ itself. The natural trick to construct the model $(\mathcal{A}, \beta)$, then, is *to construct it from the components of* $\Phi$ *itself*. The components of $\Phi$ are variable symbols, terms, subformulas and formulas occurring somewhere in $\Phi$. We have to assemble these components into a mathematical structure which is a model of $\Phi$. Here I can only demonstrate how this works with a simplistic example:

**Example** (for constructing a model $\mathcal{I} = (\mathcal{A}, \beta)$ of a given $\Phi$). Let $S = \{c, P\}$, where $c$ is a constant symbol and $P$ a unary predicate. Let $\Phi = \{c = x_1, Pc\}$. First we procure a carrier $A$ of our structure $\mathcal{A}$. Basically, we take $A$ to be the set of all variables and constant symbols. To distinguish the roles of a symbol as an element of our structure carrier vs. its role as a symbol within $\Phi$, we use underscores to indicate when we consider the symbol as an element of our carrier. So we start with a carrier $A_0 = \{\underline{c}, \underline{x}_1, \underline{x}_2,...\}$. This preliminary carrier $A_0$ will subsequently become modified to arrive at the final carrier $A$ in the end. Our interpretation $\mathcal{I}$ will map symbols and variables to their underscored correlates. Again we start with a preliminary version $\mathcal{I}_0$ of $\mathcal{I}$. We set $\mathcal{I}_0\,(c) = \underline{c}$, $\mathcal{I}_0\,(x_1) = \underline{x}_1$, $\mathcal{I}_0\,(x_2) = \underline{x}_2$, etc. However, this does not completely work out, because the formula $c = x_1$ requires that $\mathcal{I}_0\,(c)$ be the same individual as $\mathcal{I}_0\,(x_1)$. The trick that helps here is to join $\underline{c}$ and $\underline{x}_1$ into a set $\{\underline{c}, \underline{x}_1\}$. For the sake of notational uniformity we also put the remaining variables into singleton sets. That gives us $A = \{\{\underline{c}, \underline{x}_1\}, \{\underline{x}_2\}, \{\underline{x}_3\}, ...\}$ and $\mathcal{I}(c) = \mathcal{I}(\,x_1) = \{\underline{c}, \underline{x}_1\}$, $\mathcal{I}(x_2) = \{\underline{x}_2\}$, $\mathcal{I}(x_3) = \{\underline{x}_3\}$, etc. Finally, we declare the extension of $P$ according to the formula $Pc$, that is, $\mathcal{I}(P) = \{\{\underline{c}, \underline{x}_1\}\}$. It is clear that this $\mathcal{I}$ satisfies Φ.

[The general idea is to join all elements of $A_0$, whose pairwise equality is implied by Φ, into equivalence classes. If $S$ also contains function symbols, $A_0$ also contains all terms made from constants and variables using functions. The condition that Φ is consistent is needed to show that this kind of construction can actually be carried out in general. All the 12 basic rules of the sequent calculus are needed to show that this construction can always be achieved – this indicates that the full proof is tedious!].

Note the weird self-referentiality of this proof idea: the semantic "piece of mathematical world" that we need to fix as a model is made from the very stuff that formulae are made of! Logics is the art of not getting dizzy from shifting levels of descriptions until they meet!

By proving (*) we not only have shown the completeness theorem, but also the equivalence of the concepts of satisfiability and consistency (the other direction was given in Proposition 9.4). We burn the following GREAT FACT into that part of our brain that thinks about logics:

In FOL:

$\Phi \vdash \varphi \quad \Leftrightarrow \quad \Phi \models \varphi$

Φ is consistent ⇔ Φ is satisfiable

One consequence of the completeness theorem is that if Φ is enumerable, then the set $\Phi^{\models} = \{\varphi \mid \Phi \models \varphi\}$ of all of its entailments is enumerable (we earlier called this set the *theory* of Φ). Notice that this statement is not trivially true, because the signature $S$ may be uncountable. Specifically, if Φ is finite, the theory of Φ is enumerable. Thus, in principle, we could construct a Turing machine which would run infinitely long and generate on its output tape a list of all theorems of group theory. This machine would systematically apply the basic rules of the sequent calculus to the axioms of group theory and construct a complete derivation tree of all theorems of group theory. Unfortunately, as mentioned earlier, most of the derived "theorems" would be utterly uninteresting (for instance this machine would output all the formulae $x_1 = x_1$, $x_2 = x_2$, ...).

However, it is not generally true that the theory $\Phi^{\models}$ of some (even finite) Φ is *decidable*. Given Φ and some candidate φ, there is no general algorithm to decide whether $\Phi \models \varphi$. The undecidability of FOL was first shown by Gödel (again!); modern textbooks use a different and simpler proof which builds on the undecidability of the *halting problem*, a computational task from the theory of computability which we will get to know intimately in the lecture "Computability and Complexity".

The precious properties of FOL are not exhausted by the GREAT FACT. There are two other fundamental properties of FOL which are straightforward consequences of what we have already shown:

**Theorem 9.5** (Theorem of Löwenheim and Skolem). If $\Phi$ is satisfiable and countable, then $\Phi$ has a countable model.

This is actually a corollary to the proof of the completeness theorem (note that we constructed the carrier of our model from bits and pieces of $\Phi$, so if there are only countably many such bits and pieces, the model is also countable). In Section 10 we will get a closer view on the construction of countable models. The importance of the theorem of Löwenheim and Skolem lies in fact that countable models can (often) be "concretely" generated by some algorithm, which *enumerates* the components of the model. In contrast, uncountable models always contain items that cannot be individually described. Thus, in an abstract and admittedly handwaving way, the theorem of Löwenheim and Skolem implies something like, "the semantic underpinnings of FOL are within reach of efficient algorithmic methods". We will see this much more clearly in Section 10.

**Theorem 9.6** (Compactness theorem) $\Phi \models \varphi$ iff there exists a finite $\Phi_0 \subseteq \Phi$ with $\Phi_0 \models \varphi$.

The compactness theorem is an immediate consequence of the definition of $\vdash$.[remember: $\Phi \vdash \varphi$ iff there exists a finite $\Phi_0 \subseteq \Phi$ with $\Phi_0 \vdash \varphi$] and the completeness theorem.

The properties expressed in Theorems 9.5 and 9.6 single out FOL among all logics. We have seen earlier that logics can be [sometimes] ordered according to *expressiveness*: for instance, second-order predicate calculus is more expressive than FOL, which in turn is more expressive than propositional logic. Intuitively, a logic *L* is more expressive than another logic *L'*, if everything that can be said about mathematical structures within *L'* can also be expressed within *L*. A deep insight about FOL is expressed in a theorem that was first shown by Lindström in 1969: there exists no logic which is more expressive than FOL and which has the properties of Theorems 9.5 and 9.6.

Specifically, second-order predicate calculus is a very ill-behaved logic. It is *very much more* expressive than FOL, but has basically no properties that would make it amenable to computational exploits. Of course it is not complete (this was first shown by – well, you guess right!), not compact, and hasn't the Löwenheim-Skolem property.

This ends our all too brief introduction to FOL. Take home the following facts:

- There are many logics, which can be (partially) ordered according to their expressiveness.
- Every logic has two fundamental aspects: semantics (how mathematical structures are described) and syntax (how formal derivations are calculated).
- FOL together with the sequent calculus (or another type of proof calculus) is *complete*: everything that is true semantically can also be formally derived, and vice versa: $\models \Leftrightarrow \vdash$ .
- FOL is the language of
  - everyday maths
  - metamathematics and set theory
  - Artificial Intelligence
  - program verification
  - deductive databases
- FOL is the most expressive logic which still has the nice properties of completeness, compactness, and the Löwenheim-Skolem property.

- FOL is undecidable, so a mathematician's ingenuity cannot be completely replaced by a machine.

## 10 Herbrand theory – the basis for logic programming

The sequent *calculus* is a set of rules that one may apply to generate a result from some given premises. It is not an *algorithm*. From an algorithm one would require that it runs deterministically, that is, given the premises as input, one pushes a "run" button, and gets an output after a finite time. The reason that the sequent calculus does not operate in a deterministic-algorithmic way is that in the course of a derivation, when sequents $seq_1$, $seq_2$, … , $seq_n$ have already been derived, there is usually a large choice of rules that one can apply to create another sequent $seq_{n+1}$. Turning the sequent calculus into an algorithm amounts to specifying deterministic procedures to make this choice.

Note that also the grandmother of all calculi, "the" calculus (of derivatives and integrals), is a set of rules without a deterministic prescription of how to use them.

In practical applications, one desires of course an algorithm, not only a calculus. For instance, one would like to have the following:

- Mathematicians would like to have an *automated theorem prover*. This would be an algorithm which gets as input a finite set $\Phi$ of FOL premises, plus a statement $\varphi$, and the desired output would be a derivation $\Phi \vdash \varphi$.
- Database designers would like to have a *deductive database engine*. The database would be a finite (possibly large) set $\Phi$ of FOL statements (which taken together describe some interesting piece of the world, e.g. the customers of an insurance company). A *query* to the database would be some statement $\varphi$, and the desired behaviour of the deduction engine would be to generate a "yes" output if $\Phi \vdash \varphi$.
- Similarly, *expert systems* in Artificial Intelligence host a *world model* $\Phi$ which codes a (large) set of facts and laws of some knowledge domain (e.g. hypertension in medicine). In a typical case of using an expert system, the input has two components. The first component is another set $\Psi$ of formulae, which specify a specific *situation* (e.g., introduce a new patient and describe his symptoms). The second component is again a query $\varphi$, and the desired output is again a "yes" if if $\Phi \cup \Psi \vdash \varphi$. For instance, $\varphi$ could code the question, "does patient Peter have hypertension of degree 3 with a probability larger than 0.5".

Logic algorithms have indeed been developed for these and other application cases. They are not based on the sequent calculus however. The reason why we have used the sequent calculus in our introduction to the basics of FOL is that it is intuitive – it mimicks the way how humans carry out logical arguments. The same cannot be said of the methods for the mechanical verification of $\Phi \vDash \varphi$, which are used in "fielded" theorem provers, deductive databases, expert systems etc. These methods are based on constructing models for sets of *S*-expressions $\Phi$, very much in the spirit of the construction we peeked at in our proof sketch of the completeness theorem. These models are called *Herbrand* structures, after the French mathematician Jacques Herbrand (1908 – 1931).

The general structure of mechanical proofs (that is, algorithms) of a claim $\varphi$ from a set of premises $\Phi$ is as follows.

1. Input: $\Phi$, $\varphi$. Desired output: "yes" if $\Phi \models \varphi$. If not $\Phi \models \varphi$, then the algorithm need not terminate. (This is a consequence of the undecidability of FOL.)

2. We note that

   $\Phi \models \varphi$
   iff every model of $\Phi$ is a model of $\varphi$
   iff every model of $\Phi$ is not a model of $\neg\varphi$
   iff $\Phi$ □$\{\neg\varphi\}$ is unsatisfiable.

   In order to show that $\Phi \models \varphi$, it therefore suffices to show that $\Phi$ □$\{\neg\varphi\}$ is unsatisfiable.

3. A set $\Gamma$ of $S$-expressions is satisfiable iff it has a particular type of model, called Herbrand model. Thus, in order to show that $\Phi \models \varphi$, it suffices to show that $\Phi$ □$\{\neg\varphi\}$ has no Herbrand model.

4. A Herbrand model of a set $\Phi$ □$\{\neg\varphi\}$ can be constructed in a systematic way. If at some point this construction fails, one can conclude that $\Phi$ □$\{\neg\varphi\}$ is unsatisfiable, i.e., that $\Phi \models \varphi$. The algorithm will then output "yes".

This short sketch of how Herbrand-based theorem-provers operate is admittedly sketchy. We will now work out the details. In the remainder of this section I closely follow the treatment given in the book by U. Schöning[7], sections 2.2 and 2.4.

To begin with, I note that when FOL is used for practical applications, one usually employs a version of FOL which is a little weaker than the version that we have been studying so far. This difference is that this weaker version of FOL does not have the equality sign, "=", in its syntax (and correspondingly, no sequent rules that refer to "="). In fact, in many textbooks on logic, this FOL without equality is used as the standard version of FOL, and the extension by "=" is explicitly called, *first-order logic with equality*, and denoted $\mathrm{FOL}^{=}$. There are some philosophical / metamathematical issues connected to the question which version is preferable, which we will not further delve into[8]. We simply note that in practical applications one usually can make do without "=". For instance, the axioms of group theory, which we formulated in the previous section as follows, using "=":

$$\begin{array}{ll} \varphi_1: & \forall x \forall y \forall z\ (x \circ y) \circ z = x \circ (y \circ z) \\ \varphi_2: & \forall x\ x \circ e = x \\ \varphi_3: & \forall x \exists y\ x \circ y = e \end{array}$$

can also be coded without "=", if we introduce a ternary predicate $P$ and drop the function symbol $\circ$ from our signature. The idea is that $Pxyz$ stands for $x \circ y = z$. Then we can express

[7] Schoening, Uwe: *Logic for Computer Scientists (Progress in Computer Science and Applied Logic, Vol 8)*, (Birkhauser) IRC: QA9 .S363 1989

[8] You can find a discussion in chapter 11 of Michal Walicki: *Introduction to Logic.* Online lecture notes, obtainable from the author's website at http://www.ii.uib.no/~michal/ . Copy at http://minds.jacobs-university.de/sites/default/files/uploads/teaching/share/IntroToLogicWalicki.pdf.

associativity ($\varphi_1$), existence of right-neutral element ($\varphi_2$), and existence of right-inverse ($\varphi_3$), as follows:

$\varphi_1$': $\forall u \forall v \forall w \forall x \forall y \forall z\ ((Pxyu \square Pyzv) \rightarrow (Pxvw \leftrightarrow Puzw))$
$\varphi_2$': $\forall x\ Pxex$
$\varphi_3$': $\forall x \exists y\ Pxye$

In the remainder of this section, we will use FOL without equality.

### Equivalence w.r.t. satisfiability

The theorem provers that we are aiming at build on mechanisms that demonstrate that some finite collection $\Phi$ of *S*-expressions is unsatisfiable. A finite set $\Phi$ of *S*-expressions is equivalent to the single *S*-expression $\varphi$ that is the conjunction of all members of $\Phi$. Thus, we only need to design mechanisms that mechanically demonstrate the unsatisfiability of a single *S*-expression $\varphi$. We will introduce in the sequel some transformations that turn an *S*-expression $\varphi$ into another *S*-expression $\varphi$', which is satisfiable iff $\varphi$ is satisfiable. That is, the transformation does not preserve logical equivalence, but only satisfiability – which is enough for our purposes. We will call such transformations *sat-equivalance transformations.*

### "Existentializing away" free variables

If $\varphi$ is an *S*-expression that contains free variables $y_1, ..., y_n$ , then $\varphi' = \exists y_1 \ldots \exists y_n\ \varphi$ has no free variables and is satisfiably iff $\varphi$ is satisfiable (exercise!). "Existentializing away" free variables is thus a sat-equivalence transformation.

### Two simple normal forms.

To prepare the stage for a systematic, algorithmic treatment of FOL formulas, we introduce some standardizations for their syntax.

**Definition 10.1** An *S*-expression $\varphi$ is called *cleaned* if there is no variable that occurs in $\varphi$ both free and bound, and if all quantifiers in $\varphi$ are connected to different variables.

**Example.** $(\forall x \exists y\ (Pxfy \square \forall y\ Qxy) \square Rx)$ is not cleaned, but
$(\forall u \exists y\ (Pufy \square \forall z\ Qxz) \square Rx)$ is.

It should be clear that one can transform any *S*-expression into an equivalent one which is clean, by renaming all variables which are bound by quantifiers to new variables, working from the inside to the outside (e.g., in the example, first treat the $\forall y\ Qxy$ before the outer $\forall x \exists y\ (Pxfy \square \forall y\ Qxy)$).

**Definition 10.2**. An *S*-expression is in *prenex form* if it has the structure

$$Q_1 y_1\ Q_2 y_2 \ldots Q_n y_n\ F,$$

where $n \geq 0$, $Q_i \square \{\forall, \exists\}$, and $F$ is an *S*-expression that contains no quantifiers.

**Proposition 10.1** For every *S*-expression φ there is an equivalent *S*-expression φ' which is cleaned and in prenex form.

It is easy to give a constructive proof, i.e. an algorithm to transform φ to φ' in a sequence of steps, where every step preserves equivalence. We skip the details here and only remark that this transformation rests on two main insights:

- A formula of the kind $\neg Q_1 y_1\ Q_2 y_2 \ldots Q_n y_n\ F$ is equivalent to $\overline{Q}_1 y_1 \overline{Q}_2 y_2 .. \overline{Q}_n y_n \neg F$, where $\overline{Q}_i = \forall$ if $Q_i = \exists$ and $\overline{Q}_i = \exists$ if $Q_i = \forall$. This creates a mechanism to "trickle" negations out of blocks of quantifiers.
- A formula of the kind $(Q_1 y_1 .. Q_n y_n F \wedge Q_1' z_1 .. Q_k' z_k G)$ is equivalent to $Q_1 y_1 .. Q_n y_n Q_1' z_1' .. Q_k' z_k' (F \wedge G')$, where the variables $z_1' .. z_k'$ are new and *G*' is created from *G* by renaming bound occurrences of $z_1 ... z_k$ to $z_1' .. z_k'$. This equivalence affords a mechanism to pull out quantifiers from conjunctions.

In the following we use the abbreviation CPF for "cleaned and in prenex form".

## Skolemization, Skolem forms

The transformation that we now describe is called *Skolemization*, and the resulting φ' will be said to be in *Skolem form* or to be a *Skolem formula* (after Thoralf Albert Skolem (1887 - 1963), Norwegian mathematician who was productive in many fields of mathematics, but is best known for landmark contributions to logics and set theory in the years after World War 1). Skolemization is a sat-equivalence transformation.

Here is the Skolemization algorithm.

```
input: An S-expression $\varphi$ in CPF.
while $\varphi$ contains an existential quantifier do
  begin
    Let $\varphi$ have the form $\forall y_1 ... \forall y_n \exists z\, \chi$, where $\chi$ is in CPF and $n \geq 0$;
    Let $f$ be a new $n$-ary function symbol (i.e., it is not in the signature $S$ and
    was not used in this algorithm before);
    Assign $\varphi = \forall y_1 ... \forall y_n \chi \frac{f y_1 ... y_n}{z}$, that is, drop the quantification $\exists z$ and replace
    all occurrences of $z$ in $\chi$ by the term $f y_1 ... y_n$;
  end
```

Note that in this algorithm, when $n = 0$, *f* is a "0-ary" function symbol, which just means that it is a constant symbol.

This procedure preserves satisfiability. Again we skip the formal proof (which can be found, e.g., in the Schöning book), but illustrate with an example why this strange procedure functions.

Consider the statement "every husband has a wife", which might be formalized as

φ: □*x*□*y* (Husband *x* → isWifeOf *yx*)

Skolemization would turn this into

φ §': □*x* (Husband *x* → isWifeOf (*fx*)*x*),

where I have added an extra pair of brackets for readability. Now, assume that φ has a model, say 𝒜 = (*A*, Husband$^A$, isWifeOf$^A$). Since 𝒜 ⊨ φ, for every *h* □ Husband$^A$ there is some $w_h$ □ *A* such that ($w_h$, *h*) □ isWifeOf$^A$. Now define a new function $f^A$ on *A* such that for every *h* □ Husband$^A$, (*f*(*h*), *h*) □ isWifeOf$^A$ (on *a* □ Husband$^A$, *f* can be set to arbitrary values). Then the extended structure 𝒜' = (*A*, Husband$^A$, isWifeOf$^A$, $f^A$) is a model of φ §', which is thus satisfiable. For the converse, assume that φ §' has a model 𝒜' = (*A*, Husband$^A$, isWifeOf$^A$, $f^A$). Because 𝒜' ⊨ φ §', for all *h* □ Husband$^A$ it holds that (*f*(*h*), *h*) □ isWifeOf$^A$, that is, for every *h* □ Husband$^A$ there is some $w_h$ □ *A* (namely, *f*(*h*)) such that ($w_h$, *h*) □ isWifeOf$^A$. Thus, 𝒜' ⊨ φ §, that is, φ is satisfiable by a (Husband, isWifeOf, *f*)-structure, and therefore (because  does not contain *f*) by a (Husband, isWifeOf)-structure.

Purely intuitively speaking, Skolemization proceeds from "□*x*□*y*" to "for every *x* there is an *y*" to "one can pick an *y* for every *x*" to "there is a picking function *f* that returns an *y* = *f*(*x*) for every *x*" to "we can introduce a picking function $f^A$ in every model".

Note that if Skolemization is applied to a formula in CFP, then the resulting Skolem formula is in CFP, too – that is, it has the form

$$\forall y_1 \ldots \forall y_n \, \varphi$$

that is, it consists of a block of universal quantifiers followed by a "body" φ which is quantifier-free.

**Clause set form**

Assume that φ has no free variables, is in CFP, and is Skolemized. As a further standardization step, which will turn out convenient for automated model constructions for φ, is to transform the body of φ to conjunctive normal form (CNF), as we know it from propositional logic. Notice that the body of φ contains no quantifiers. The role of literals in propositional logic is here taken by predications $Py_1 \ldots y_n$ or negated predications $\neg Py_1 \ldots y_n$ . We use the symbol *L* for such literals. Re-using the (constructive) proof of the analog fact from propositional logic, it is straightforward to show that φ can be transformed into an equivalent φ', whose body is in CNF, that is, φ' has the form

φ': $$\forall y_1 \ldots \forall y_n \, ((L_1^1 \vee \ldots \vee L_{n_1}^1) \wedge \ldots \wedge (L_1^k \vee \ldots \vee L_{n_k}^k)).$$

Since we can assume that φ' contains no free variables (due to the "existentializing away" step), and since the order of a block of □ quantifiers does not matter, and since the order of □ or □ sequences does not matter either, all the information of φ' is contained in its *clause form*:

φ': $$\{\{L_1^1, \ldots, L_{n_1}^1\}, \ldots, \{L_1^k, \ldots, L_{n_k}^k\}\}.$$

**Example**

Consider the expression

$\varphi_1$: $(\neg\exists x(Pxz \vee \forall y\, Qxf(y)) \vee \forall y\, Pg(x,y)z)$.

Cleaning gives an equivalent

$\varphi_2$: $(\neg\exists x(Pxz \vee \forall y\, Qxf(y)) \vee \forall w\, Pg(x,w)z)$.

Existentializing away the free variable *z* yields the sat-equivalent

$\varphi_3$: $\exists z(\neg\exists x(Pxz \vee \forall y\, Qxf(y)) \vee \forall w\, Pg(x,w)z)$.

Bringing this into prenex form can be done, among others, by

$\varphi_4$: $\exists z\forall x\exists y\forall w((\neg Pxz \wedge \neg Qxf(y)) \vee Pg(x,w)z)$.

Skolemization yields

$\varphi_5$: $\forall x\forall w((\neg Pxa \wedge \neg Qxf(h(x))) \vee Pg(x,w)a)$,

where *a* is a new 0-ary function symbol (i.e., a constant symbol!) and *h* is a new unary function symbol. Bringing the body of $\varphi_5$ to CNF results in

$\varphi_6$: $\forall x\forall w((\neg Pxa \vee Pg(x,w)a) \wedge (\neg Qxf(h(x)) \vee Pg(x,w)a))$,

which in clause form looks like this:

$$\{\{\neg Pxa, Pg(x,w)a\}, \{\neg Qxf(h(x)), Pg(x,w)a)\}\}.$$

This (the form of $\varphi_6$ or its clause form) is the standardarized format which we will henceforth assume.

**Herbrand universe and Herbrand structures and Herbrand models**

The preparatory constructions carried out so far have brought us to a point where our original target, "prove that $\Phi \vdash \varphi$", has been translated to "show that a formula $\psi$, which has no free variables and is Skolemized, is unsatisfiable". (Recall: $\psi$ has been obtained from $\Phi$ □$\{\neg\varphi\}$ and is satisfiable-equivalent to $\Phi$ □$\{\neg\varphi\}$.) As announced in the introductory remarks to this section, this amounts to showing that $\psi$ has no model of a particular kind called a *Herbrand model.* We now proceed to describe Herbrand models of $\psi$.

Let $S(\psi)$ be the set of all constant, function and predicate symbols in $\psi$. Note that $\psi$ may contain function and/or constant symbols that were not contained in the original signature of $\Phi$ □$\{\neg\varphi\}$ due to the Skolemization. Furthermore note that *S* is finite since $\psi$ is finite. $\psi$ may have several, one or no Herbrand model, the latter case occuring iff $\psi$ is not satisfiable. If $\psi$ has Herbrand models, these all share the same carrier, called the Herbrand universe of $\psi$:

**Definition 10.3** Let ψ be a formula which is Skolemized. Then, intuitively, the *Herbrand universe* $D(\psi)$ is the set of all variable-free terms which can be built from the constituents of ψ. Formally, $D(\psi)$ is defined to be the smallest set satisfying the following conditions:

1. All constant symbols occuring in ψ are in $D(\psi)$. If $D(\psi)$ contains no constant symbol, choose a new constant symbol *a* and put it into $D(\psi)$.
2. If $f$ □ $S(\psi)$ is an *n*-ary function symbol, and $t_1, ..., t_n$ □ $D(\psi)$, then the term $ft_1...t_n$ is in $D(\psi)$.

**Example.** Consider the formulae

$$\psi: \quad \forall x \forall y \forall z\ Pxf(y)g(zx)$$
$$\chi: \quad \forall x \forall y\ Qcf(x)h(yb)$$

where *P*, *Q* are ternary relation symbols, *f* is a unary and *g*, *h* are binary function symbols, and *c*, *b* are constant symbols. Then

$$D(\psi) = \{a, fa, gaa, fgaa, ffa, gafa, gfaa, ...\}$$
$$D(\chi) = \{b, c, fc, fb, hcc, hcb, hbc, hbb, ffc, ffb, fhcc, fhcb, ...\}$$

For convenience, we introduce the following shorthand notation.
Let $S(\psi) = \{a_1, ..., a_k, f_1, ..., f_l, P_1, ..., P_m\}$, where $k, l, m \geq 0$ and the *a*, *f*, *P*'s are constant symbols, function symbols (of various arities) and predicate symbols (of various arities), respectively. Then use $\mathcal{A} = (A, S(\psi)^A)$ as a shorthand for any $S(\psi)$-structure $(A, a_1^A, ..., a_k^A, f_1^A, ..., f_l^A, P_1^A, ..., P_m^A)$.

**Definition 10.4** Let ψ be a Skolem formula. Any $S(\psi)$-structure $\mathcal{A} = (A, S(\psi)^A)$ which satisfies

1. $A = D(\psi)$,
2. for every *n*-ary $f$ □ $S(\psi)$, for all $t_1, ..., t_n$ □ $D(\psi)$ it holds that $f^A(t_1,...,t_n) = ft_1...t_n$,

is called a *Herbrand structure* for ψ.

Note that the carrier of a Herbrand structure, and the interpretation of all function symbols (and thus of all constant symbols, seen as 0-ary function symbols) is fixed by definition. Herbrand structures for ψ may differ from one another only in the interpretation of predicate symbols. Note further that Herbrand universes and structures are defined for Skolem formulas ψ, but it is not required that ψ has no free variables.

In Herbrand structures, the interpretation of terms from $D(\psi)$ is the identity function: $\mathcal{I}(t) = t$. Don't proceed with reading before you have fully appreciated this fact.

For later use we note the following obvious (if one has understood the preceding two definitions...) *transfer lemma*:

$$\mathcal{I}\frac{t}{x} \models \psi \ \text{ iff } \ \mathcal{I} \models \psi\frac{t}{x}$$

for every $t \,\square\; D(\psi)$ and every interpretation $\mathcal{I} = (\mathcal{A}, \beta)$ where $\mathcal{A}$ is a Herbrand structure for $\psi$. Here $\psi\frac{t}{x}$ denotes the formula that is obtained from $\psi$ if all occurrences of the free variable $x$ (if any) have been replaced by $t$.

If $\psi$ is a formula with no free variables, we call a Herbrand structure $\mathcal{A}$ for $\psi$ a *Herbrand model* of $\psi$ if $\mathcal{A} \models \psi$.

Here is the main result of this section, which lays the foundation for all modern algorithms for automated theorem proving, and the programming language PROLOG. It essentially states that if one wants to demonstrate the (non)existence of a model for some Skolem proposition, one can restrict the quest to Herbrand models.

**Theorem 10.2** Let $\psi$ be a proposition (i.e., without free variables) in Skolem form. Then $\psi$ is satisfiable iff $\psi$ has a Herbrand model.

**Proof.** "If": clear. "Only if": takes a little – but really only a little – effort. Let $\mathcal{A} = (A, S(\psi)^A)$ be a model of $\psi$, and let $\mathcal{I}_{\mathcal{A}}$ be the corresponding interpretation. If $\psi$ does not contain a constant symbol (special circumstance in definition 10.3), we extend $\mathcal{A}$ by adding a new constant symbol $c$, which is interpreted by an arbitrarily chosen $a \,\square\, A$, i.e., $c^A = a$. We now give a Herbrand structure $\mathcal{B} = (B, S(\psi)^B) = (D(\psi), S(\psi)^B)$. The carrier is $B = D(\psi)$ by definition of a Herbrand structure. It remains to specify the interpretation of predicate symbols. Let $P \,\square\, S(\psi)$ be an $n$-ary predicate symbol, and let $t_1, ..., t_n \,\square\; D(\psi)$. We put

$$(t_1, ..., t_n) \,\square\; P^B \qquad :\text{iff} \qquad (\mathcal{I}_{\mathcal{A}}(t_1), \ldots, \mathcal{I}_{\mathcal{A}}(t_n)) \,\square\; P^A.$$

We claim that $\mathcal{B} \models \psi$. To show this, we prove a stronger statement: For every variable-free Skolem formula $\chi$ which is made from the same symbols as $\psi$ (i.e., $S(\chi) = S(\psi)$), it holds that

if $\mathcal{A} \models \chi$, then $\mathcal{B} \models \chi$.

We prove this by induction on the number $n$ of all-quantifiers in $\chi$.

*Induction basis* $n = 0$: If $\chi$ has no all quantifiers (and hence no quantifiers at all), it immediately follows from the definition of $\mathcal{B}$ that $\mathcal{A} \models \chi$ iff $\mathcal{B} \models \chi$.

*Induction step:* Let $\chi$ have $n$ all-quantifiers, $\chi = \square x\; \xi$, where $\xi$ contains only $n - 1$ all-quantifiers. We can't apply the induction hypothesis on $\xi$ directly because it may contain $x$ free. But since $\mathcal{A} \models \chi$, and $\chi$ is Skolem, it holds that $\mathcal{I}_{\mathcal{A}}\frac{a}{x} \models \xi$ for all $a \,\square\, A$. Specifically, this holds for all $a \,\square\, A$ which are of the special form $a = \mathcal{I}_{\mathcal{A}}(t)$ for some $t \,\square\, D(\chi)$. In other words, for every $t \,\square\, D(\chi)$ it holds that $\mathcal{I}_{\mathcal{A}}$ $\models \xi$, hence $\mathcal{I}_{\mathcal{A}} \models \xi$ , that is, $\mathcal{A} \models \xi$ . By induction hypothesis (note that $\xi$ has no free variables), $\mathcal{B} \models \xi$ for every $t \,\square\, D(\chi)$.

Applying the transfer lemma, this implies $\mathcal{B}$ $\models \xi$ for every $t \,\square\, D(\chi)$, that is (since the carrier $B$ of $\mathcal{B}$ consist exactly of all $t \,\square\, D(\chi)$), $\mathcal{B} \models \square x\; \xi$.

*Note.* Herbrand models are obviously countable. The construction of such models yields the basic tools for the proof of the fundamental theorem of Löwenheim-Skolem mentioned in Section 9.

## Herbrand expansions

We now take another important step toward practical algorithms for demonstrating that some $\psi$ has no (Herbrand) model. Namely, we transform the statement "$\psi$ has a Herbrand model" to an equivalent statement of the kind "a certain set of formulas of propositional logic has a model" – this step will allow us to use computational methods, which originate in propositional logic, to bring to bear on the problem of FOL satisfiability.

The key idea lies in the following definition.

**Definition 10.5** Let $\psi = \forall y_1 ... \forall y_n \, \chi$ be a Skolem formula without free variables. Then the *Herbrand expansion* $E(\psi)$ is the following set of variable-free formulas:

$$E(\psi) = \{ \chi \frac{t_1}{y_1} ... \frac{t_n}{y_n} \mid t_1,..., t_n \, \square \, D(\psi)\}.$$

$E(\psi)$ is thus created by replacing the variables in $\chi$ with terms from the Herbrand universe $D(\psi)$, in all possible combinations.

**Example.** Consider $\psi = \forall x \forall y (Pxc \wedge \neg Pg(xy) f(c))$, where $P$ is a binary relation symbol, $g$ is a binary and $f$ a unary function symbol and $c$ a constant. Then the following are some members of $E(\psi)$:

$$(Pcc \wedge \neg Pg(cc)f(c)) \quad \text{using} \quad \frac{c}{x}\frac{c}{y}$$

$$(Pf(c)c \wedge \neg Pg(f(c)c)f(c)) \quad \text{using} \quad \frac{f(c)}{x}\frac{c}{y}$$

$$(Pcc \wedge \neg Pg(cf(c))f(c)) \quad \text{using} \quad \frac{c}{x}\frac{f(c)}{y}$$

$$(Pf(c)c \wedge \neg Pg(f(c)f(c))f(c)) \quad \text{using} \quad \frac{f(c)}{x}\frac{f(c)}{y}$$

$$(Pcc \wedge \neg Pg(cg(cc))f(c)) \quad \text{using} \quad \frac{c}{x}\frac{g(cc)}{y}$$

|

At this point we establish a connection to propositional logic: We treat the atomic subformulae (i.e., the predication statements of form $Rt_1 ... t_n$) which occur in $E(\psi)$ as propositional variables, to which we may assign truth values. In the example, the atomic subformulae are $Pcc$, $Pg(cc)f(c)$, $Pf(c)c$, $Pg(f(c)c)f(c)$, ... (in the textual order in which they appear in the listing above). Assigning truth values to each of these means to (partially) characterize the extension of $P$ in a Herbrand structure. For instance, setting the truth value of $Pcc$ to 1 means that $(c, c)$ □ $P^A$, or setting the truth value of $Pg(cc)f(c)$ to 0 means that

$(g(cc), f(c))$ □ $P^A$. Using the notation $\mathcal{I}_{\text{Boole}}$ for Boolean truth value interpreations, we would write $\mathcal{I}_{\text{Boole}}(Pcc) = 1$ or $\mathcal{I}_{\text{Boole}}(Pg(cc)f(c)) = 0$ to express this formally.

Interpreting atomic subformulae as Boolean variables, the Herbrand expansion can be seen as a set of Boolean expressions in a natural way. The above listing would become

$(X \wedge \neg Y)$
$(Z \wedge \neg U)$
$(X \wedge \neg V)$
$(Z \wedge \neg W)$
$(X \wedge \neg R)$
L

It turns out that the satisfiability of the FOL formula $\psi$ is equivalent to the Boolean satisfiability of the set of formulae $E(\psi)$, understood as Boolean expressions:

**Theorem 10.3** (Gödel, Herbrand, Skolem). Let $\psi$ be a variable-free Skolem formula. Then $\psi$ is satisfiable iff $E(\psi)$, understood as a set of Boolean expressions, is satisfiable in the sense of propositional logic, that is, iff there is a truth value assignment to the atomic subexpressions in $E(\psi)$ which makes all formulae in $E(\psi)$ true.

**Proof.** It suffices to show that $\psi$ has a Herbrand model iff $E(\psi)$, understood as a set of Boolean expressions, is satisfiable. Let $\psi = \forall y_1 \ldots \forall y_n\, \chi$ be a Skolem formula without free variables. Then the following equivalences hold:

there exists a Herbrand model $\mathcal{A}$ for $\psi$
iff there exists a Herbrand structure $\mathcal{A}$ such that for all $t_1, \ldots, t_n$ □ $D(\psi)$:
$\mathcal{A} \models \chi \frac{t_1}{y_1} \ldots \frac{t_n}{y_n}$
iff there exists a Herbrand structure $\mathcal{A}$ such that for all $\zeta$ □ $E(\psi)$: $\mathcal{A} \models \zeta$
iff $E(\psi)$ is satisfiable in the Boolean sense.

This theorem can be understood in the sense that one can represent the satisfiability conditions of a FOL formula by a (typically infinite) set of Boolean expressions.

At this point we recall without proof a fundamental theorem of Boolean logic, the *compactness theorem* for Boolean logic: A set $\Psi$ of Boolean expressions is satisfiable iff every finite subset of $\Psi$ is satisfiable.

The converse of this theorem is that a set $\Psi$ of Boolean expressions is unsatisfiable iff there exists some finite subset of $\Psi$ which is unsatisfiable.

Combining this with theorem 10.3, we get

**Proposition 10.4** (Herbrand) A variable-free Skolem formula $\psi$ is unsatisfiable iff there is a finite subset of its Herbrand expansion $E(\psi)$ which is unsatisfiable.

## The Gilmore algorithm

This proposition provides the key mechanism for all modern algorithms for theorem proving, all modern "inference machines". In its most basic form, this is the *Gilmore algorithm*. This algorithm takes as input a variable-free Skolem formula $\psi$ and after a finite time returns "yes" if $\psi$ is unsatisfiable (if $\psi$ is satisfiable, the Gilmore algorithm does not terminate). Let $\zeta_1, \zeta_2, \ldots$ be an effective enumeration of $E(\psi)$.

Here it is:

```
input: a variable-free Skolem formula $\psi = \forall y_1 \ldots \forall y_n \chi$ with $\zeta_1, \zeta_2, \ldots$ being an
effective enumeration of $E(\psi)$
$k = 0$;
while $\zeta_1$ □ ... □ $\zeta_k$ is satisfiable     ; this can be tested with any method
                                           ; from propositional logic, e.g. truth
                                            ; table checking
    $k = k + 1$;
end while;
output "yes"
```

This algorithm is inefficient in many respects and useful only for theoretical purposes. However, it can be refined and constrained such that practically useful versions arise. These refinements lead beyond the scope of this lecture and would be covered in a specialization lecture on logic programming. Here I can only point out the most decisive refinement and the most important constraint:

*Refinement:* Do not use "ground substitution" formulae from $E(\psi)$ – where a ground substitution is a substitution of all variables in $\psi$ by variable-free "ground terms", as we did it here – but proceed more defensively and substitute only as many variables in versions of | as are needed to cancel certain literals in a partially substituted clause representation of $E(\psi)$. This leads to *unification* mechanisms.

*Constraint:* A decisive constraint on $\psi$ is to require that all clauses in the clause representation of $\psi$ are *Horn clauses* – you should remember from Gen CS 1 that satisfiability of propositional Horn formulae can be tested efficiently. This constraint speeds up the test of checking whether (a unification-based version of) $\zeta_1$ □ ... □ $\zeta_n$ is satisfiable. This constraint is observed and exploited by the PROLOG programming language, which provides a direct way for a user to code "$\Phi \models \varphi$ ?" questions (queries) as program calls. The book of Schöning is by and large an introduction to PROLOG.

# 11 Take-home messages: the lecture in a nutshell

## Take-home message, top level

- Automata and languages are simple, transparent, universally useful: the "small coin" of computer science.
- Logics is deep, ancient, mystical – but technically not difficult – and very powerful: the "royal palace" of computer science.
- A computer scientist who does not know about the basic concepts from these fields will not be considered a computer scientist by his fellows.

## Take-home message, middle level

- There are many ways to describe formal languages: automata, regular expressions (only for regular languages), grammars (and there are yet others).
- Languages come in the Chomsky hierarchy: regular – context free – context-sensitive – unrestricted. The higher in that hierarchy, the more expressive the languages, but also the more difficult to analyze. Almost everything can be decided about regular languages, but about unrestricted languages, almost nothing can be decided.
- For practical purposes, regular and context-free languages are most important. Regular languages appear everywhere in programs, like other basic programming constructs or data types (you need them for any interesting pattern matching). Context-free languages are needed to describe, design and parse programming languages – and XML, the contents technology of the coming decades.
- Standard tasks you encounter when working with languages in applications:
    - pattern matching (searching patterns defined by regular expressions)
    - word membership decision (trivial for regular languages, CYK algorithm and derivatives thereof for general context-free languages, deterministic PDAs for practically used programming languages)
    - parsing (building a structured tree representation of an input word)
    - translation (compiling, transforming XML documents into target code)

  We only covered the first two, the others would typically be treated in a lecture on "compilers and languages".
- Logics come in a hierarchy too, e.g. Boolean logic is less expressive than FOL than second-order logic.
- There are always two aspects of a logics: syntax and semantics. The deep question about any logical system (except Boolean logics which is too simplistic to make this interesting) is the clarification of the relationship between syntax and semantics.
- The semantics is concerned with the question which "real" things are "meant" by a logical expression. To make this analysis accessible to rigorous methods, the "real" things are assumed to be sets. Thus, logics and set theory are twin disciplines.
- The interesting thing about syntax in logics is not parsing the structure of expressions, but to develop calculi and other algorithms that can manipulate logical expressions in such a way that proofs result – that is, that you can mechanically derive logically valid consequences from given premises.
- In FOL, syntax and semantics coincide, that is, FOL is complete: $\models \Leftrightarrow \vdash$ .

- However, FOL is not decidable: there is no general algorithm that can return a yes-or-no answer to the question whether $\Phi \vDash \varphi$.

## Take-home message, fine-grained level (exam checklist)

Note: here I list not all the things we treated in the lecture, only a (large) subset of the most essential items. The listed things you really should know and master in the final exam. In some years, not all of this material will be covered, and/or other material will be additionally covered, depending on classroom dynamics. Thus, this list is not a 1-1 coverage of the final exam topics; it is an condensed review of the backbone topics in these lecture.

- General definitions of symbols, words, languages; how many symbols an alphabet may have, how many languages over an alphabet exist and how many words a language may have (cardinality issues)
- Definition of DFA or NFA or ε-NFA (collectively called "finite automata") and accepted languages
- Represent finite automata by graph or table
- Construct finite automata for a given language
- Subset construction to transform NFA to DFA
- Definition of regexps (standard)
- Algebraic laws for regexps
- Regexps with language variables
- Transform regex to finite automaton and vice versa
- Applying pumping lemma to show that a given language is not regular: I expect immaculate proofs here...
- Closure properties of regular languages
- Myhill-Nerode theorem and table-filling algorithm for obtaining minimal DFA
- Decidability properties of regular languages
- CFGs: definition; derivations; parse trees; language of a CFG
- Right-linear CFGs and regular languages
- Ambiguity
- The wonderful world of XML
- PDAs: definition; graphical notation; representing PDA by specifying δ; language accepted by PDA by final state or empty stack (without proof)
- Deterministic PDAs
- Chomsky normal form and how to transform a grammar to it
- Pumping lemma for CFLs
- Closure and decidability properties for CFLs; CYK algorithm
- Definition of unrestricted grammars and how they are used to derive words
- Syntax of FOL: inductive definition of terms and expressions; you should be able to write syntactically 100% correct FOL expressions in the rigorous syntax
- Ability to express simple real-world or mathematical statements in FOL
- Free occurrence of variables; definition of what a proposition is
- *S*-structures: tuple representation and intuitive interpretation
- Variable assignment: definition
- *S*-interpretations
- The model and the entailment relation ⊨: definition
- Ability to specify concrete *S*-structures that are or are not models of a given proposition

- Satisfiability: definition
- dual nature of ∀ and ∃
- Sequent rules: what a sequence means semantically, what an entire sequent rule means semantically
- Correctness of sequent rules: definition and ability to show correctness for simple rules
- Role of our 11 base rules within FOL
- Formal proofs (derivations) in FOL: definition; ability to carry out such proofs in simple cases
- Completeness theorem: statement and awareness of its importance
- Undecidability of FOL: statement and awareness of its importance