Machines and their languages (G51MAL)
Lecture notes
Spring 2005

Thorsten Altenkirch
(revised by Henrik Nilsson)

May 11, 2005

Contents

1 Introduction

Most references refer to the course text [HMU01]. Please, note that the 2nd edition is quite different to the first one, which appeared in 1979 and is a classical reference on the subject.
I have also been using [Sch99] for those notes, but note that this book is written in German.
The on-line version of this text contains some hyperlinks to webpages that contain additional information.

1.1 Examples on syntax

In PR1 and PR2 you are learning the language JAVA. Which of the following programs are *syntactically correct*, i.e. will be accepted by the JAVA compiler without error messages?

Hello-World.java

```
public class Hello-World {

  public static void main(String argc[3]) {
      System:out.println('Hello World');
  }
}
```

A.java

```
class A {
  class B {
      void C () {
          {} ; {{}}
      }
  }
}
```

I hope that you are able to spot all the errors in the first program. It may be actually surprising but the 2nd (strange looking) program is actually correct. How do we know whether a program is syntactically correct? We would hope that this doesn't depend on the compiler we are using.

1.2 What is this course about?

1. Mathematical models of computation, such as:
   - Finite automata,
   - Pushdown automata,
   - Turing machines
2. How to specify formal languages?
   - Regular expressions
   - Context free grammars
   - Context sensitive grammars
3. The relation between 1. and 2.

1.3 Applications

- Regular expressions

  Regular expressions are a convenient ways to express patterns (i.e. for search). There are a number of tools that use regular expressions:

  - `grep` pattern matching program (UNIX)
  - `sed` stream editor (UNIX)
  - `lex` A generator for lexical analyzers (UNIX)

- Grammars for programming languages.

  The appendix of [GJSB00] contains a *context free grammar* specifying the syntax of JAVA.

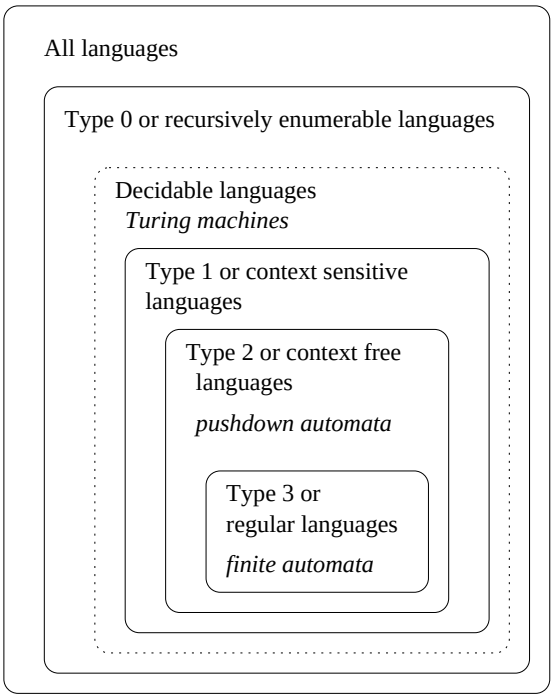
  YACC is a tool that can be used to generate a C program (a parser) from a grammar. Parser generators now also exist for other languages, like Java CUP for Java.

- Specifying protocols

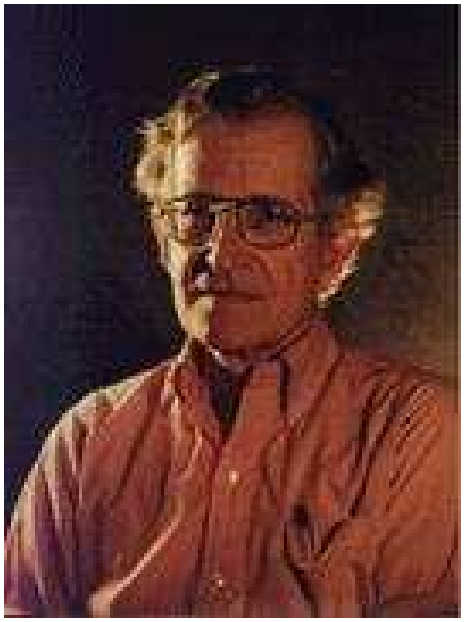
  See section 2.1 (pp 38 - 45) contains a simple example of how a protocol for electronic cash can be specified and analyzed using finite automata.

1.4 History

1.4.1 The Chomsky Hierarchy



Noam Chomsky introduced the *Chomsky hierarchy* that classifies grammars and languages. This hierarchy can be amended by different types of machines (or automata) that can recognize the appropriate class of languages.
Chomsky is also well known for his unusual views on society and politics.



1.4.2 Turing machines

Alan Turing (1912-1954) introduced an abstract model of computation, which we call Turing machines, to give a precise definition that problems can be solved by a computer. All the machines we are introducing can be viewed as restricted versions of Turing machines.
I recommend Andrew Hodges biography Alan Turing: the Enigma.

1.5 Languages

In this course we will use the terms *language* and *word* different than in everyday language:

- A **language** is a set of words.
- A **word** is a sequence of symbols.

This leaves us with the question: what is a symbol? The answer is: anything, but it has to come from an alphabet Σ that is a finite set. A common (and important) instance is $\Sigma = \{0, 1\}$.
More mathematically we say: Given an alphabet Σ we define the set Σ^* as set of words (or sequences) over Σ: the empty word $\epsilon \in \Sigma^*$ and given a symbol $x \in \Sigma$ and a word $w \in \Sigma^*$ we can form a new word $xw \in \Sigma^*$. These are all the ways elements on Σ^* can be constructed (this is called an *inductive definition*). E.g. in the example $\Sigma = \{0, 1\}$, typical elements of Σ^* are 0010, 00000000,ϵ. Note, that we only write ϵ if it appears on its own, instead of 00ϵ we just write 00. It is also important to realize that although there are infinitely many words, each word has a finite length.
An important operation on Σ^* is concatenation. Confusingly, this is denoted by an *invisible* operator: given $w, v \in \Sigma^*$ we can construct a new word $wv \in \Sigma^*$ simply by concatenating the two words. We can define this operation by

primitive recursion:

$$\begin{aligned} \epsilon v &= v \\ (xw)v &= x(wv) \end{aligned}$$

A language L is a set of words, hence $L \subseteq \Sigma^*$ or equivalently $L \in \mathcal{P}(\Sigma^*)$[1]. Here are some informal examples of languages:

- The set $\{0010, 00000000, \epsilon\}$ is a language over $\Sigma = \{0, 1\}$. This is an example of a finite language.
- The set of words with odd length over $\Sigma = \{1\}$.
- The set of words that contain the same number of 0s and 1s is a language over $\Sigma = \{0, 1\}$.
- The set of words that contain the same number of 0s and 1s modulo 2 (i.e. both are even or odd) is a language over $\Sigma = \{0, 1\}$.
- The set of palindromes using the English alphabet, e.g. words that read the same forwards and backwards like `abba`. This is a language over $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$.
- The set of correct Java programs. This is a language over the set of UNICODE "characters" (which correspond to numbers between 0 and $17 \cdot 2^{16} - 1$, less some invalid subranges, 1112062 valid encodings in all).
- The set of programs that, if executed on a Windows machine, prints the text "Hello World!" in a window. This is a language over $\Sigma = \{0, 1\}$.

2 Finite Automata

Finite automata correspond to a computer with a fixed finite amount of memory[2]. We will introduce *deterministic finite automata* (DFA) first and then move to *nondeterministic finite automata* (NFA). An automaton will accept certain words (sequences of symbols of a given alphabet Σ) and reject others. The set of accepted words is called the language of the automaton. We will show that the class of languages that are accepted by DFAs and NFAs is the same.

[1] Given a set A, $\mathcal{P}(A)$ is the *powerset* of A; that is, the set of all possible subsets of A. For example, if $A = \{a, b\}$, then $\mathcal{P}(A) = \{\, \emptyset, \{a\}, \{b\}, \{a, b\} \,\}$. The number of elements $|A|$ of a set A, its *cardinality*, and the number of elements in its power set are related by $|\mathcal{P}(A)| = 2^{|A|}$. Hence powerset.

[2] However, that does not mean that finite automata are a good model of general purpose computers. A computer with n bits of memory has 2^n possible states. That is an absolutely enormous number even for very modest memory sizes, say 1024 bits or more, meaning that describing a computer using finite automata quickly becomes infeasible. We will encounter a better model of computers later, the *Turing Machines*. Conceptually, they have infinite memory, which may seem overly optimistic, but what that really means is that we consider the memory to be large enough for the task at hand. Exactly how much memory that is needed becomes a separate, orthogonal, issue, a separation of concerns that has proved to be very fruitful. See also Hopcroft *et al.* [HMU01, p. 315] for a deeper discussion of this.

2.1 Deterministic finite automata

2.1.1 What is a DFA?

A *deterministic finite automaton* (DFA) $A = (Q, \Sigma, \delta, q_0, F)$ is given by:

1. A finite set of states Q,
2. A finite set of input symbols Σ,
3. A transition function $\delta \in Q \times \Sigma \to Q$,
4. An initial state $q_0 \in Q$,
5. A set of accepting states $F \subseteq Q$.

As an example consider the following automaton

$$\mathrm{D} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_\mathrm{D}, q_0, \{q_2\})$$

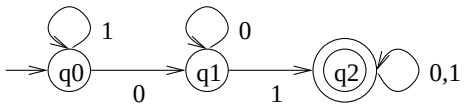
where

$$\delta_\mathrm{D} = \{((q_0, 0), q_1), ((q_0, 1), q_0), ((q_1, 0), q_1), ((q_1, 1), q_2), ((q_2, 0), q_2), ((q_2, 1), q_2)\}$$

The DFA may be more conveniently represented by a transition table:

	0	1
$\to q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_2	q_2

The table represents the function δ, i.e. to find the value of $\delta(q, x)$ we have to look at the row labelled q and the column labelled x. The initial state is marked by an $\to$ and all final states are marked by $*$.
Yet another, optically more inspiring, alternative are transition diagrams:



There is an arrow into the initial state and all final states are marked by double rings. If $\delta(q, x) = q'$ then there is an arrow from state q to q' that is labelled x. We write Σ^* for the set of words (i.e. sequences) over the alphabet Σ. This includes the empty word that is written ϵ. I.e.

$$\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$$

2.1.2 The language of a DFA

To each DFA A we associate a language $L(A) \subseteq \Sigma^*$. To see whether a word $w \in L(A)$ we put a marker in the initial state and when reading a symbol forward the marker along the edge marked with this symbol. When we are in an accepting state at the end of the word then $w \in L(A)$, otherwise $w \notin L(A)$.

In the example above we have that $0 \notin L(\mathrm{D})$,$101 \in L(\mathrm{D})$ and $110 \notin L(\mathrm{D})$. Indeed, we have

$$L(\mathrm{D}) = \{w \mid w \text{ contains the substring } 01\}$$

To be more precise we give a formal definition of $L(A)$. First we define the extended transition function $\hat{\delta} \in Q \times \Sigma^* \to Q$. Intuitively, $\hat{\delta}(q, w) = q'$ if starting from state q we end up in state q' when reading the word w. Formally, $\hat{\delta}$ is defined by primitive recursion:

$$\hat{\delta}(q, \epsilon) = q \tag{1}$$
$$\hat{\delta}(q, xw) = \hat{\delta}(\delta(q, x), w) \tag{2}$$

Here xw stands for a non empty word whose first symbol is x and the rest is w. E.g. if we are told that $xw = 010$ then this entails that $x = 0$ and $w = 10$. w may be empty, i.e. $xw = 0$ entails $x = 0$ and $w = \epsilon$.
As an example we calculate $\hat{\delta}_{\mathrm{D}}(q_0, 101) = q_1$:

$$\begin{array}{rcll}
\hat{\delta}_{\mathrm{D}}(q_0, 101) & = & \hat{\delta}_{\mathrm{D}}(\delta_{\mathrm{D}}(q_0, 1), 01) & \text{by (2)} \\
& = & \hat{\delta}_{\mathrm{D}}(q_0, 01) & \text{because } \delta_{\mathrm{D}}(q_0, 1) = q_0 \\
& = & \hat{\delta}_{\mathrm{D}}(\delta_{\mathrm{D}}(q_0, 0), 1) & \text{by (2)} \\
& = & \hat{\delta}_{\mathrm{D}}(q_1, 1) & \text{because } \delta_{\mathrm{D}}(q_0, 0) = q_1 \\
& = & \hat{\delta}_{\mathrm{D}}(\delta_{\mathrm{D}}(q_1, 1), \epsilon) & \text{by (2)} \\
& = & \hat{\delta}_{\mathrm{D}}(q_2, \epsilon) & \text{because } \delta_{\mathrm{D}}(q_1, 1) = q_2 \\
& = & q_2 & \text{by (1)}
\end{array}$$

Using $\hat{\delta}$ we may now define formally:

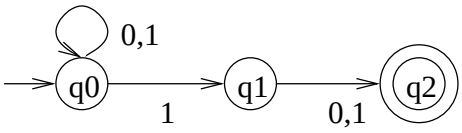
$$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

Hence we have that $101 \in L(\mathrm{D})$ because $\hat{\delta}_{\mathrm{D}}(q_0, 101) = q_2$ and $q_2 \in F_{\mathrm{D}}$.

2.2 Nondeterministic Finite Automata

2.2.1 What is an NFA?

Nondeterministic finite automata (NFA) have transition functions that may assign several or no states to a given state and an input symbol. They accept a word if there is *any possible* transition from the one of initial states to one of the final states. It is important to note that although NFAs have a non-determistic transition function, it can always be determined whether or not a word belongs to its language ($w \in L(A)$). Indeed, we shall see that every NFA can be translated into an DFA that accepts the same language.
Here is an example of an NFA C that accepts all words over $\Sigma = \{0, 1\}$ s.t. the symbol before the last is 1.



A *nondeterministic finite automaton* (NFA) $A = (Q, \Sigma, \delta, S, F)$ is given by:

1. A finite set of states Q,
2. A finite set of input symbols Σ,
3. A transition function $\delta \in Q \times \Sigma \to \mathcal{P}(Q)$,
4. A set of initial state $S \subseteq Q$,
5. A set of accepting states $F \subseteq Q$.

The differences to DFAs are to have start states instead of a single one and the type of the transition function. As an example we have that

$$C = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_C, \{q_0\}, \{q2\})$$

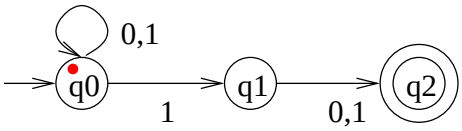
where δ_C so given by

δ_C	0	1
$\to q_0$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

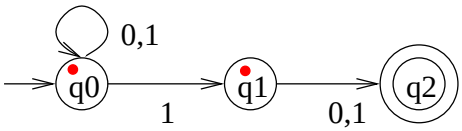
Note that we diverge slightly from the definition in the book, which uses a single initial state instead of a set of initial states. Doing so means that we can avoid introducing ϵ-NFAs (see [HMU01], section 2.5).

2.2.2 The language accepted by an NFA

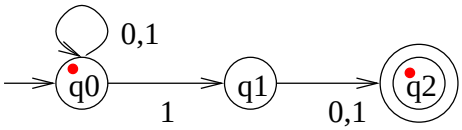
To see whether a word $w \in \Sigma^*$ is accepted by an NFA ($w \in L(A)$) we may have to use several markers. Initially we put one marker on each initial state. Then each time when we read a symbol we look at all the markers: we remove the old marker and put markers at all the states that are reachable via an arrow marked with the current input symbol (this may include the state that was marked in the previously). Thus we may have to use several marker but it may also happen that all markers disappear (if no appropriate arrows exist). In this case the word is not accepted. If at the end of the word any of the final states has a marker on it then the word is accepted.
E.g. consider the word 100 (which is not accepted by C). Initially we have



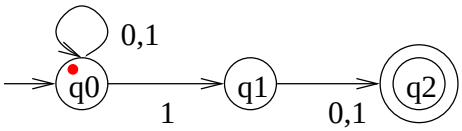
After reading 1 we have to use two markers because there are two arrows from q_0 that are labelled 1:



Now after reading 0 the automaton has still got two markers, one of them in an accepting state:



However, after reading the 2nd 0 the second marker disappears because there is no edge leaving q_2 and we have:



which is not accepting because no marker is in the accepting state.
To specify the extended transition function for NFAs we use an generalisation of the union operation $\cup$ on sets. We define $\bigcup$ to be the union of a (finite) set of sets:

$$\bigcup\{A_1, A_2, \ldots A_n\} = A_1 \cup A_2 \cup \cdots \cup A_n$$

In the special cases of the empty sets of sets and a one element set of sets we define:

$$\bigcup\emptyset = \emptyset \qquad \bigcup\{A\} = A$$

As an example

$$\bigcup\{\{1\}, \{2,3\}, \{1,3\}\} = \{1\} \cup \{2,3\} \cup \{1,3\} = \{1,2,3\}$$

Actually, we may define $\bigcup$ by comprehension, which also extends the operation to infinite sets of sets (although we don't need this here)

$$\bigcup B = \{x \mid \exists A \in B.x \in A\}$$

We define $\hat{\delta} \in \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ with the intention that $\hat{\delta}(S, w)$ is the set of states that is marked after having read w starting with the initial markers given by S.

$$\hat{\delta}(P, \epsilon) = P \tag{3}$$

$$\hat{\delta}(P, xw) = \hat{\delta}(\bigcup\{\delta(q, x) \mid q \in P\}, w) \tag{4}$$

As an example we calculate $\hat{\delta}_C(q_0, 100)$ which is $\{q_0\}$ as we already know from

playing with markers.

$$\begin{aligned}
\hat{\delta}_C(\{q_0\}, 100) &= \hat{\delta}_C(\bigcup\{\delta_C(q,1) \mid q \in \{q_0\}\}, 00) && \text{by (4)}\\
&= \hat{\delta}_C(\delta_C(q_0,1), 00)\\
&= \hat{\delta}_C(\{q_0, q_1\}, 00)\\
&= \hat{\delta}_C(\bigcup\{\delta_C(q,0) \mid q \in \{q_0, q_1\}\}, 0) && \text{by (4)}\\
&= \hat{\delta}_C(\delta_C(q_0,0) \cup \delta_C(q_1,0), 0)\\
&= \hat{\delta}_C(\{q_0\} \cup \{q_2\}, 0)\\
&= \hat{\delta}_C(\{q_0, q_2\}, 0)\\
&= \hat{\delta}_C(\bigcup\{\delta_C(q,0) \mid q \in \{q_0, q_2\}\}, \epsilon) && \text{by (4)}\\
&= \hat{\delta}_C(\delta_C(q_0,0) \cup \delta_C(q_2,0), 0)\\
&= \hat{\delta}_C(\{q_0\} \cup \emptyset, \epsilon)\\
&= \{q_0\} && \text{by (3)}
\end{aligned}$$

Using the extended transition function we define the language of an NFA as

$$L(A) = \{w \mid \hat{\delta}(S, w) \cap F \neq \emptyset\}$$

This shows that $100 \notin L(C)$ because

$$\hat{\delta}(\{q_0\}, 100) = \{q_0\} \cap \{q_2\} = \emptyset$$

2.2.3 The subset construction

DFAs can be viewed as a special case of NFAs, i.e. those for which the the there is precisely one start state $S = \{q_0\}$ and the transition function returns always one-element sets (i.e. $\delta(q, x) = \{q'\}$ for all $q \in Q$ and $x \in \Sigma$).
Below we show that for every NFA we can construct a DFA that accepts the same language. This shows that NFAs aren't more powerful than DFAs. However, in some cases NFAs need a lot fewer states than the corresponding DFA and they are easier to construct.
Given an NFA $A = (Q, \Sigma, \delta, S, F)$ we construct the DFA

$$D(A) = (\mathcal{P}(Q), \Sigma, \delta_{D(A)}, S, F_{D(A)})$$

where

$$\begin{aligned}
\delta_{D(A)}(P, x) &= \bigcup\{\delta(q, x) \mid q \in P\}\\
F_{D(A)} &= \{P \in \mathcal{P}(Q) \mid P \cap F \neq \emptyset\}
\end{aligned}$$

The basic idea of this construction (*the subset construction*) is to define a DFA whose states are *sets of states* of the NFA. A final state of the DFA is a set that contains at least a final state of the NFA. The transitions just follow the active set of markers, i.e. a state $P \in \mathcal{P}(Q)$ corresponds to having markers on all $q \in S$ and when we follow the arrow labelled x we get the set of states that are marked after reading x.
As an example let us consider the NFA C above. We construct a DFA $D(C)$

$$D(C) = (\mathcal{P}(\{q_0, q_1, q_2\}, \{0, 1\}, \delta_{D(C)}, \{q_0\}, F_{D(C)})$$

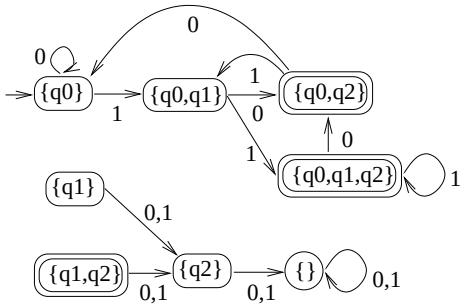
with $\delta_{D(C)}$ given by

$\delta_{D(C)}$	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_1\}$	$\{q_2\}$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
$*\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1\}$
$*\{q_1, q_2\}$	$\{q_2\}$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$

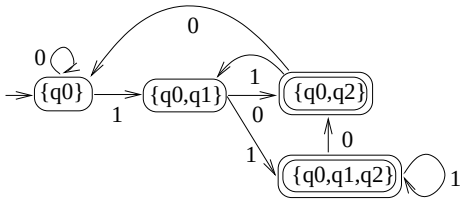
and $F_{D(C)}$ is the set of all the states marked with $*$ above,i.e.

$$F_{D(C)} = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$$

Looking at the transition diagram:



we note that some of the states $(\emptyset, \{q_1\}, \{q_2\}, \{q_1, q_2\})$ cannot be reached from the initial state, which means that they can be omitted without changing the language. Hence we obtain the following automaton:



We still have to convince ourselves that the DFA $D(A)$ accepts the same language as the NFA A, i.e. we have to show that $L(A) = L(D(A))$.
As a lemma we show that the extended transition functions coincide:

Lemma 2.1

$$\hat{\delta}_{D(A)}(P, w) = \hat{\delta}_A(P, w)$$

The result of both functions are sets of states of the NFA A: for the left hand side because the extended transition function on NFAs returns sets of states and for the right hand side because the states of $D(A)$ are sets of states of A.
Proof: We show this by *induction over the length of the word* w, let's write $|w|$ for the length of a word.

$|w| = 0$ Then $w = \epsilon$ and we have

$$\begin{array}{rcll} \hat{\delta}_{D(A)}(P, \epsilon) & = & P & \text{by (1)} \\ & = & \hat{\delta}_A(P, \epsilon) & \text{by (3)} \end{array}$$

$|w| = n + 1$ Then $w = xv$ with $|v| = n$.

$$\begin{array}{rcll} \hat{\delta}_{D(A)}(P, xv) & = & \hat{\delta}_{D(A)}(\delta_{D(A)}(P, x), v) & \text{by (2)} \\ & = & \hat{\delta}_A(\delta_{D(A)}(P, x), v) & \text{ind.hyp.} \\ & = & \hat{\delta}_A(\bigcup\{\delta_A(q, x) \mid q \in P\}, v) & \text{by construction} \\ & = & \hat{\delta}_A(P, xv) & \text{by (4)} \end{array}$$

□

We can now use the lemma to show

Theorem 2.2

$$L(A) = L(D(A))$$

Proof:

$$\begin{array}{rl} & w \in L(A) \\ \iff & \text{Definition of } L(A) \text{ for NFAs} \\ & \hat{\delta}_A(S, w) \cap F \neq \emptyset \\ \iff & \text{Lemma 2.1} \\ & \hat{\delta}_{D(A)}(S, w) \cap F \neq \emptyset \\ \iff & \text{Definition of } F_{D(A)} \\ & \hat{\delta}_{D(A)}(S, w) \in F_{D(A)} \\ \iff & \text{Definition of } L(A) \text{ for DFAs} \\ & w \in L_{D(A)} \end{array}$$

□

Corollary 2.3 *NFAs and DFAs recognize the same class of languages.*

Proof: We have noticed that DFAs are just a special case of NFAs. On the other hand the subset construction introduced above shows that for every NFA we can find a DFA that recognizes the same language. □

3 Regular expressions

Given an alphabet Σ a language is a set of words $L \subseteq \Sigma^*$. So far we were able to describe languages either by using set theory (i.e. enumeration or comprehension) or by an automaton. In this section we shall introduce *regular expressions* as an elegant and concise way to describe *languages*. We shall see that the languages definable by regular expressions are precisely the same as those accepted by deterministic or nondeterministic finite automata. These languages

are called *regular languages* or (according to the Chomsky hierarchy) Type 3 languages.
As already mentioned in the introduction regular expressions are used to define patterns in programs such as `grep`. `grep` gets as an argument a regular expression and then filters out all those lines from a file that match the regular expression, where matching means that the line contains a substring that is in the language assigned to the regular expression. It is interesting to note that even in the case when we search for a specific word (this is a special case of a regular expresion) programs like `grep` are more efficient than a naive implementation of word search.
To find out more about `grep` have a look at the UNIX manual page and play around with `grep`. Note that the syntax `grep` uses is slightly different from the one we use here. `grep` also use some convenient shorthands that are not relevant for a theoretical analysis of regular expressions because they do not extend the class of languages.

3.1 What are regular expressions?

Given an alphabet Σ (e.g. $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$), the *syntax* (i.e., *form*) of regular expressions over Σ is defined inductively as follows:

1. $\emptyset$ is a regular expression.
2. ϵ is a regular expression.
3. For each $x \in \Sigma$, $\mathbf{x}$ is a regular expression. E.g. in the example all small letters are regular expressions[3].
4. If E and F are regular expressions then $E + F$ is a regular expression.
5. If E and F are regular expressions then EF (i.e. just one after the other) is a regular expression.
6. If E is a regular expression then E^* is a regular expression.
7. If E is a regular expression then (E) is a regular expression.

These are all regular expressions.
Here are some examples for regular expressions:

- ϵ
- $\mathbf{hallo}$
- $\mathbf{hallo} + \mathbf{hello}$
- $\mathbf{h(a + e)llo}$
- $\mathbf{a}^*\mathbf{b}^*$
- $(\epsilon + \mathbf{b})(\mathbf{ab})^*(\epsilon + \mathbf{a})$

[3] We use boldface to emphasize the difference between the *symbol* `a` and the regular expression $\mathbf{a}$ that, as we will see, denotes the one-word language $\{\mathtt{a}\}$. Symbols and words are typeset using a type-writer font in this and the next section (and on occasion later on as well) to further emphasize the difference.

As in arithmetic, there are some conventions regarding how to read regular expressions:

- $*$ binds stronger than sequence and $+$. E.g. we read $\mathbf{ab}^*$ as $\mathbf{a}(\mathbf{b}^*)$. We have to use parentheses to enforce the other reading $(\mathbf{ab})^*$.
- Sequencing binds stronger than $+$. E.g. we read $\mathbf{ab} + \mathbf{cd}$ as $(\mathbf{ab}) + (\mathbf{bc})$. To enforce another reading we have to use parentheses as in $\mathbf{a}(\mathbf{b} + \mathbf{c})\mathbf{d}$.

3.2 The meaning of regular expressions

We now know what regular expressions are but what do they mean?
For this purpose, we shall first define an operation on languages called *the Kleene star*. Given a language $L \subseteq \Sigma^*$ we define

$$L^* = \{w_0 w_1 \dots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n. w_i \in L\}$$

Intuitively, L^* contains all the words that can be formed by concatenating an arbitrary number of words in L. This includes the empty word since the number may be 0.
As an example consider $L = \{\mathtt{a}, \mathtt{ab}\} \subseteq \{\mathtt{a}, \mathtt{b}\}^*$:

$$L^* = \{\epsilon, \mathtt{a}, \mathtt{ab}, \mathtt{aab}, \mathtt{aba}, \mathtt{aaab}, \mathtt{aaba}, \dots\}$$

Note that we use the same symbol for this operation as in Σ^*. However, there is a subtle difference: Σ is a set of *symbols* whereas $L \subseteq \Sigma^*$ is a set of *words*, i.e. a language.
Alternatively (and more abstractly) one may describe L^* as the least language (with respect to $\subseteq$) that contains L and the empty word and is closed under concatenation:

$$w \in L^* \wedge v \in L^* \implies wv \in L^*$$

In the previous section, we defined the *syntax* of regular expressions, their *form*. We now proceed to define the *semantics* of regular expressions; i.e., what they *mean*, what language a regular expression stands for. To each regular expression E over Σ we assign a language $L(E) \subseteq \Sigma^*$. We do this by *induction over the definition of the syntax*:

1. $L(\emptyset) = \emptyset$
2. $L(\epsilon) = \{\epsilon\}$
3. $L(\mathbf{x}) = \{x\}$ where $x \in \Sigma$.
4. $L(E + F) = L(E) \cup L(F)$
5. $L(EF) = \{wv \mid w \in L(E) \wedge v \in L(F)\}$
6. $L(E^*) = L(E)^*$
7. $L((E)) = L(E)$

Subtle points: in 1 the symbol $\emptyset$ is used both as a regular expression (as in $L(\emptyset)$) and as the empty set ($\emptyset = \{\}$), i.e. the empty language. Similarly, ϵ in 2 is used in two ways: both as a regular expression and as a word (the *empty* word). In 6, the $*$-operator is used both to construct regular expressions and as an operation on languages. Such *overloading* of notation is quite common. Which alternative is meant only becomes clear from the context; there is no generally agreed mathematical notation[4] to make this difference explicit.
Let us now calculate what the examples of regular expressions from the previous section mean, i.e. what are the langauges they define:

ϵ

$$L(\epsilon) = \{\epsilon\}$$

By 2.

hallo

Let's just look at $L(\mathbf{ha})$. We know from 3:

$$L(\mathbf{h}) = \{\mathtt{h}\}$$
$$L(\mathbf{a}) = \{\mathtt{a}\}$$

Hence by 5:

$$\begin{aligned} L(\mathbf{ha}) &= \{wv \mid w \in \{\mathtt{h}\} \wedge v \in \{\mathtt{a}\}\} \\ &= \{\mathtt{ha}\} \end{aligned}$$

Continuing the same reasoning we obtain:

$$L(\mathbf{hallo}) = \{\mathtt{hallo}\}$$

hallo + hello

From the previous point we know that:

$$L(\mathbf{hallo}) = \{\mathtt{hallo}\}$$
$$L(\mathbf{hello}) = \{\mathtt{hello}\}$$

Hence by using 4 we get:

$$\begin{aligned} L(\mathbf{hallo} + \mathbf{hello}) &= \{\mathtt{hallo}\} \cup \{\mathtt{hello}\}\} \\ &= \{\mathtt{hallo}, \mathtt{hello}\} \end{aligned}$$

h(a + e)llo

Using 3 and 4 we know

$$L(\mathbf{a} + \mathbf{e}) = \{\mathtt{a}, \mathtt{e}\}$$

[4] This is different in programming, e.g. in JAVA we use "..." to signal that we mean things literally.

Hence using 5 we obtain:

$$\begin{aligned} L(\mathbf{h}(\mathbf{a}+\mathbf{e})\mathbf{llo}) &= \{uvw \mid u \in L(\mathbf{h}) \wedge v \in L(\mathbf{a}+\mathbf{e}) \wedge w \in L(\mathbf{llo})\} \\ &= \{uvw \mid u \in \{\mathtt{h}\} \wedge v \in \{\mathtt{a},\mathtt{e}\} \wedge w \in \{\mathtt{llo}\}\} \\ &= \{\mathtt{hallo},\mathtt{hello}\} \end{aligned}$$

$\mathbf{a}^*\mathbf{b}^*$

Let us introduce the following notation:

$$w^i = \underbrace{ww\ldots w}_{i \text{ times}}$$

Now using 6 we know that

$$\begin{aligned} L(\mathbf{a}^*) &= \{w_0 w_1 \ldots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n. w_i \in L(\mathbf{a})\} \\ &= \{w_0 w_1 \ldots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n. w_i \in \{\mathtt{a}\}\} \\ &= \{\mathtt{a}^n \mid n \in \mathbb{N}\} \end{aligned}$$

and hence using 5 we conclude

$$\begin{aligned} L(\mathbf{a}^*\mathbf{b}^*) &= \{uv \mid u \in L(\mathbf{a}^*) \wedge v \in L(\mathbf{b}^*)\} \\ &= \{uv \mid u \in \{\mathtt{a}^n \mid n \in \mathbb{N}\} \wedge v \in \{\mathtt{b}^m \mid m \in \mathbb{N}\}\} \\ &= \{\mathtt{a}^n\mathtt{b}^m \mid m,n \in \mathbb{N}\} \end{aligned}$$

I.e. $L(\mathbf{a}^*\mathbf{b}^*)$ is the set of all words that start with a (possibly empty) sequence of $\mathtt{a}$'s followed by a (possibly empty) sequence of $\mathtt{b}$'s.

$(\epsilon+\mathbf{b})(\mathbf{ab})^*(\epsilon+\mathbf{a})$

Let's analyze the parts:

$$\begin{aligned} L(\epsilon+\mathbf{b}) &= \{\epsilon,\mathtt{b}\} \\ L((\mathbf{ab})^*) &= \{(\mathtt{ab})^i \mid i \in \mathbb{N}\} \\ L(\epsilon+\mathbf{b}) &= \{\epsilon,\mathtt{b}\} \end{aligned}$$

Hence, we have

$$L((\epsilon+\mathbf{b})(\mathbf{ab})^*(\epsilon+\mathbf{a})) = \{u(\mathtt{ab})^i v \mid u \in \{\epsilon,\mathtt{b}\} \wedge i \in \mathbb{N} \wedge v \in \{\epsilon,\mathtt{a}\}\}$$

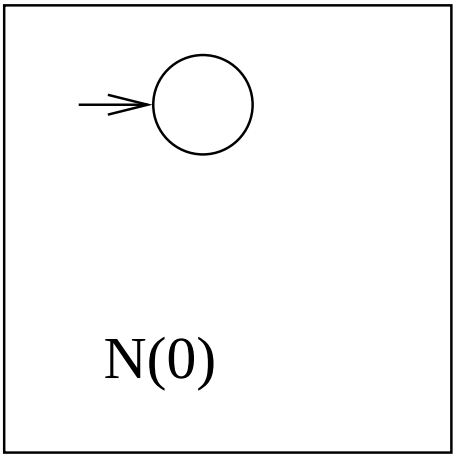
In english: $L((\epsilon+\mathbf{b})(\mathbf{ab})^*(\epsilon+\mathbf{a}))$ is the set of (possibly empty) sequences of interchanging $\mathtt{a}$'s and $\mathtt{b}$'s.

3.3 Translating regular expressions into NFAs

Theorem 3.1 *For each regular expression E we can construct ab NFA $N(E)$ s.t. $L(N(E)) = L(E)$, i.e. the automaton accepts the language described by the regular expression.*

Proof:
We do this again by induction on the syntax of regular expressions:

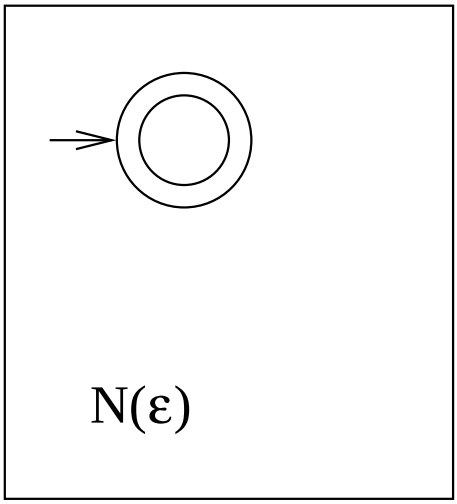
1. $N(\emptyset)$:



which will reject everything (it has got no final states) and hence

$$\begin{aligned} L(N(\emptyset)) &= \emptyset \\ &= L(\emptyset) \end{aligned}$$

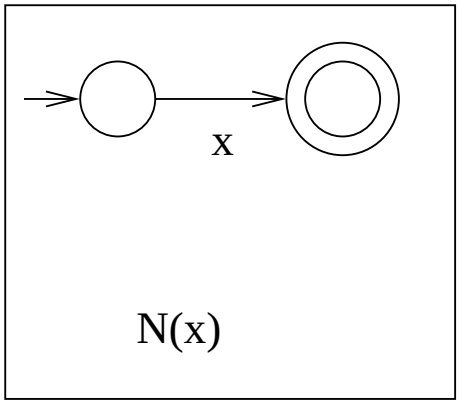
2. $N(\epsilon)$:



This automaton accepts the empty word but rejects everything else, hence:

$$\begin{aligned} L(N(\epsilon)) &= \{\epsilon\} \\ &= L(\epsilon) \end{aligned}$$

3. $N(\mathbf{x})$:

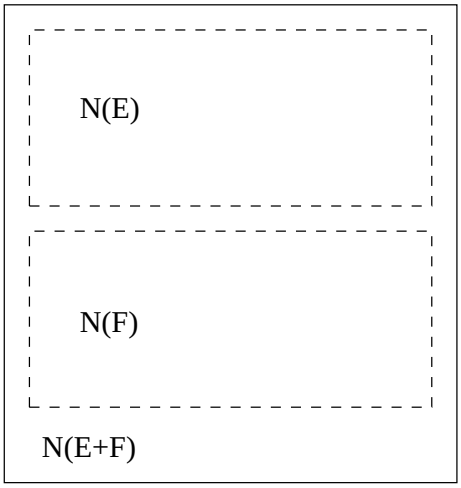


This automaton only accepts the word x, hence:

$$\begin{aligned} L(N(\mathbf{x})) &= \{\mathrm{x}\} \\ &= L(\mathbf{x}) \end{aligned}$$

4. $N(E + F)$:

   We merge the diagrams for $N(E)$ and $N(F)$ into one:



   I.e. given

$$\begin{aligned} N(E) &= (Q_E, \Sigma, \delta_E, S_E, F_E) \\ N(F) &= (Q_F, \Sigma, \delta_F, S_F, F_F) \end{aligned}$$

   Now we use the disjoint union operation on sets (see the MCS lecture

notes [Alt01], section 4.1)

$$\begin{aligned} Q_{E+F} &= Q_E + Q_F \\ \delta_{E+F}((0,q),x) &= \{(0,q') \mid q' \in \delta_E(q,x)\} \\ \delta_{E+F}((1,q)),x &= \{(1,q') \mid q' \in \delta_F(q,x)\} \\ S_{E+F} &= S_E + S_F \\ F_{E+F} &= F_E + F_F \end{aligned}$$

$$N(E+F) = (Q_{E+F}, \Sigma, \delta_{E+F}, S_{E+F}, F_{E+F})$$

The disjoint union just signals that we are not going to identify states, even if they accidently happen to have the same name.

Just thinking of the game with markers you should be able to convince yourself that

$$L(N(E+F)) = L(N(E)) \cup L(N(F))$$

Moreover to show that

$$L(N(E+F)) = L(E+F)$$

we are allowed to assume that

$$\begin{aligned} L(N(E)) &= L(E) \\ L(N(F)) &= L(F) \end{aligned}$$

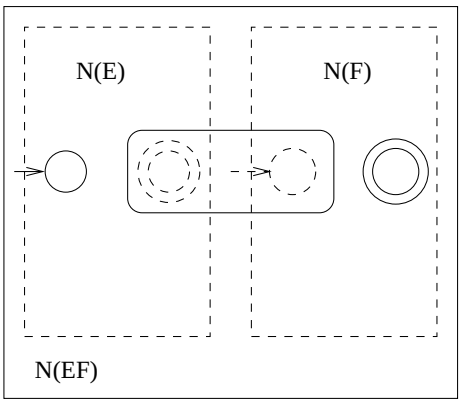
that's what is meant by *induction over the syntax of regular expressions.*

Now putting everything together:

$$\begin{aligned} L(N(E+F)) &= L(N(E)) \cup L(N(F)) \\ &= L(E) \cup L(F) \\ &= L(E+F) \end{aligned}$$

5. $N(EF)$:

   We want to put the two automata $N(E)$ and $N(F)$ in series. We do this by *connecting* the final states of $N(E)$ with the initial states of $N(F)$ in a way explained below.



In this diagram I only depicted one initial and one final state of each of the automata although they may be several of them.

Here is how we construct $N(EF)$ from $N(E)$ and $N(F)$:

$$N(E) = (Q_E, \Sigma, \delta_E, S_E, F_E)$$
$$N(F) = (Q_F, \Sigma, \delta_F, S_F, F_F)$$

- The states of $N(EF)$ are the disjoint union of the states of $N(E)$ and $N(F)$:
$$Q_{EF} = Q_E + Q_F$$
- The transition function of $N(EF)$ contains all the transitions of $N(E)$ and $N(F)$ (as for $N(E+F)$) and for each state q of $N(E)$ that has a transition to a final state of $N(E)$ we add a transition with the same label to all the initial states of $N(F)$:

$$\begin{aligned}\delta_{EF}((0,q),x) &= \{(0,q') \mid q' \in \delta_E(q,x)\} \\ &\quad \cup \{(1,q'') \mid \exists q'.q' \in F_E \wedge q' \in \delta_E(q,x) \wedge q'' \in S_E\} \\ \delta_{EF}((1,q)) &= \{(1,q') \mid q' \in \delta_F(q))\}\end{aligned}$$

- The initial states of $N(EF)$ are the initial states of $N(E)$, and the initial states of $N(F)$ **if** there is an initial state of $N(E)$ that is also a final state:
$$S_{EF} = \{(0,q) \mid q \in S_E\} \cup \{(1,q) \mid q \in S_F \wedge S_E \cap F_E \neq \emptyset\}$$
- The final states of $N(EF)$ are the final states of $N(F)$:
$$F_{EF} = \{(1,q) \mid q \in F_F\}$$

We now set
$$N(EF) = (Q_{EF}, \Sigma, \delta_{EF}, S_{EF}, Z_{EF})$$

I hope that you are able to convince yourself that

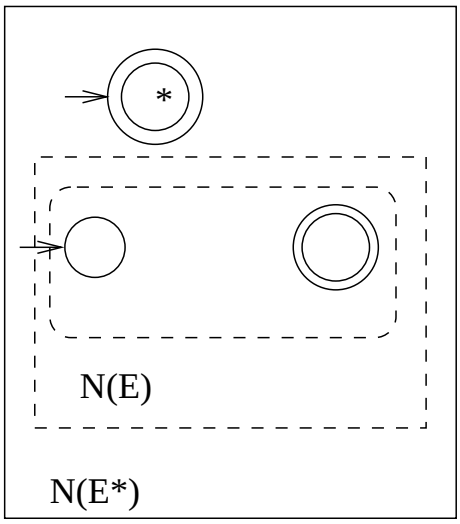
$$L(N(EF)) = \{uv \mid u \in L(N(E)) \wedge v \in L(N(F))$$

and hence we can reason

$$\begin{aligned}L(N(EF)) &= \{uv \mid u \in L(N(E)) \wedge v \in L(N(F)) \quad = \{uv \mid u \in L(E) \wedge v \in L(F) \\ &= L(EF)\end{aligned}$$

6. $N(E^*)$:

We construct $N(E^*)$ from $N(E)$ by merging initial and final states of $N(E)$ in a way similar to the previous construction and we add a new state $*$ that is initial and final.



Given

$$N(E) = (Q_E, \Sigma, \delta_E, S_E, F_E)$$

we construct $N(E^*)$.

- We add one extra state $*$:

$$Q_{E^*} = Q_E + \{*\}$$

- $N_E{}^*$ inherits all transitions form N_E and for each state that has an arrow to the final state labelled x we also add an arrow to all the initial states labelled x.

$$\begin{aligned}\delta_{E^*}((0,q),x) =& \{(0,q') \mid q' \in \delta_E(q,x)\} \\ & \cup \{(0,q') \mid \delta_E(q,x) \cap F_E \neq \emptyset \wedge q' \in S_E\}\end{aligned}$$

- The initial states of $N(E^*)$ are the initial states of $N(E)$ and $*$:

$$S_{E^*} = \{(0,q) \mid q \in S_E\} \cup \{(1,*)\}$$

- The final states of N_{E^*} are the final states of N_E and $*$:

$$F_{E^*} = \{(0,q) \mid q \in F_E\} \cup \{(1,*)\}$$

We define

$$N(E^*) = (Q_{E^*}, \Sigma, \delta_{E^*}, S_{E^*}, F_{E^*})$$

We claim that

$$L(N(E^*)) = \{w_0 w_1 \dots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n. w_i \in L(N(E))\}$$

since we can run through the automaton an arbitrary number of times. The new state $*$ allows us also to accept the empty sequence. Hence:

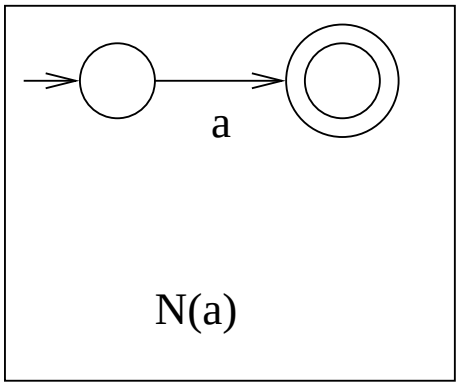
$$\begin{aligned} L(N(E^*)) &= \{w_0 w_1 \dots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n.w_i \in L(N(E))\} \\ &= L(N(E))^* \\ &= L(E)^* \\ &= L(E^*) \end{aligned}$$

7. $N((E)) = N(E)$

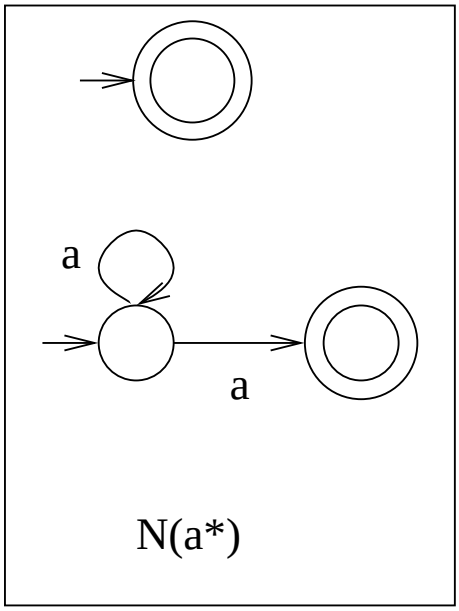
   I.e. using brackets does not change anything.

□

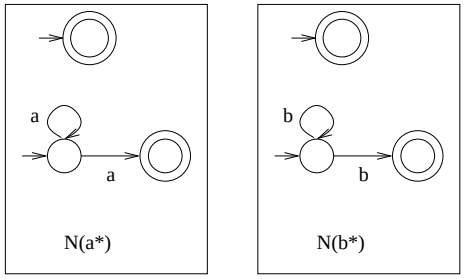
As an example we construct $N(\mathrm{a}^*\mathrm{b}^*)$. First we construct $N(\mathbf{a})$:



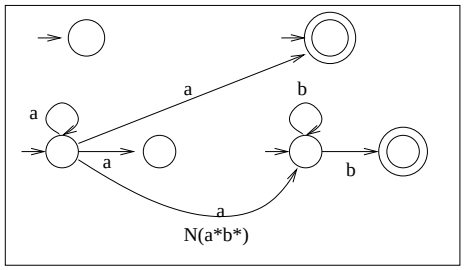
Now we have to apply the $*$-construction and we obtain:



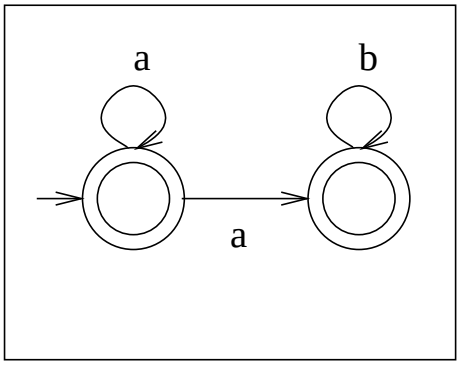
$N(b^*)$ is just the same and we get



and now we have to serialize the two automata and we get:



Now, you may observe that this automaton, though correct, is unnecessary complicated, since we could have just used



However, we shall not be concerned with minimality at the moment.

3.4 Summing up ...

From the previous section we know that a language given by regular expression is also recognized by a NFA. What about the other way: Can a language recognized by a finite automaton (DFA or NFA) also be described by a regular expression? The answer is yes:

Theorem 3.2 (Theorem 3.4, page 91) *Given a DFA A there is a regular expression $R(A)$ that recognizes the same language $L(A) = L(R(A))$.*

We omit the proof (which can be found in the [HMU01] on pp.91-93). However, we conclude:

Corollary 3.3 *Given a language $L \subseteq \Sigma^*$ the following is equivalent:*

1. *L is given by a regular expression.*

2. *L is the language accepted by an NFA.*

3. *L is the language acceped by a DFA.*

Proof: We have that 1. $\Longrightarrow$ 2 by theorem 3.1. We know that 2. $\Longrightarrow$ 3. by2.2 and 3. $\Longrightarrow$ 1. by 3.2. □
As indicated in the introduction: the languages that are characterized by any of the three equivalent conditions are called *regular languages* or *type-3-languages*.

4 Showing that a language is not regular

Regular languages are languages that can be recognized by a computer with finite (i.e. fixed) memory. Such a computer corresponds to a DFA. However, there are many languages that cannot be recognized using only finite memory. A simple example is the language

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$

i.e. the language of words that start with a number of 0s followed by the **same** number of 1s. Note that this is different from $L(0^*1^*)$, which is the language of words of a sequences of 0s followed by a sequence of 1s but not necessarily of the same length. (We know this to be regular because it is given by a regular expression.)
Why can L not be recognized by a computer with fixed finite memory? Assume we have 32 Megabytes of memory, that is we have $32*1024*1024*8 = 268435456$ bits. Such a computer corresponds to an enormous DFA with $2^{268435456}$ states (imagine you have to draw the transition diagram). However, the computer can only count until $2^{268435456}$ if we feed it any more 0s in the beginning it will get confused! Hence, you need an unbounded amount of memory to recognize n.
We shall now show a general theorem called *the pumping lemma* that allows us to prove that a certain language is not regular.

4.1 The pumping lemma

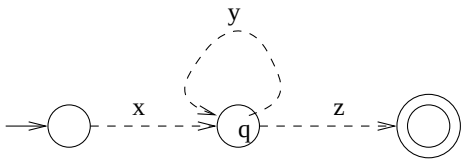
Theorem 4.1 *Given a regular language L, then there is a number $n \in \mathbb{N}$ such that all words $w \in L$ that are longer than n ($|w| \geq n$) can be split into three words $w = xyz$ s.t.*

1. *$y \neq \epsilon$*

2. *$|xy| \leq n$*

3. *for all $k \in \mathbb{N}$ we have $xy^k z \in L$.*

Proof: For a regular language L there exists a DFA A s.t. $L = L(A)$. Let us assume that A has got n states. Now if A accepts a word w with $|w| \geq n$ it must have visited a state q twice:



We choose q s.t. it is the first cycle, hence $|xy| \leq n$. We also know that y is non empty (otherwise there is no cycle).
Now, consider what happens if we feed a word of the form xy^iz to the automaton, i.e. s instead of y it contains an arbitrary number of repetitions of y, including the case $i = 0$, i.e. y is just left out. The automaton has to accept all such words and hence $xy^iz \in L$

□

4.2 Applying the pumping lemma

Theorem 4.2 *The language $L = \{0^n1^n \mid n \in \mathbb{N}\}$ is not regular.*

Proof: Assume L would be regular. We will show that this leads to contradiction using the pumping lemma.
Now by the pumping lemma there is an n such that we can split each word that is longer than n such that the properties given by the pumping lemma hold. Consider $0^n1^n \in L$, this is certainly longer than n. We have that $xyz = 0^n1^n$ and we know that $|xy| \leq n$, hence y can only contain 0s, and since $y \neq \epsilon$ it must contain at least one 0. Now according to the pumping lemma $xy^0z \in L$ but this cannot be the case because it contains at least one 0 less but the same number of 1s as 0^n1^n.
Hence, our assumption that L is regular must have been wrong.

□

It is easy to see that the language

$$\{1^n \mid n \text{ is even}\}$$

is regular (just construct the appropriate DFA or use a regular expression). However what about

$$\{1^n \mid n \text{ is a square}\}$$

where by saying n is a square we mean that is there is an $k \in \mathbb{N}$ s.t. $n = k^2$. We may try as we like there is no way to find out whether we have a got a square number of 1s by only using finite memory. And indeed:

Theorem 4.3 *The language $L = \{1^n \mid n \text{ is a square}\}$ is not regular.*

Proof: We apply the same strategy as above. Assume L is regular then there is a number n such we can split all longer words according to the pumping lemma. Let's take $w = 1^{n^2}$ this is certainly long enough. By the pumping lemma we know that we can split $w = xyz$ s.t. the conditions of the pumping lemma hold. In particular we know that

$$1 \leq |y| \leq |xy| \leq n$$

Using the 3rd condition we know that

$$xyyz \in L$$

that is $|xyyz|$ is a square. However we know that

$$\begin{aligned}
n^2 &= |w| \\
&= |xyz| \\
&< |xyyz| && \text{since } 1 \leq |y| = |xyz| + |y| \\
&\leq n^2 + n && \text{since } |y| \leq n \\
&< n^2 + 2n + 1 \\
&= (n+1)^2
\end{aligned}$$

To summarize we have

$$n^2 < |xyyz| < (n+1)^2$$

That is $|xyyz|$ lies between two subsequent squares. But then it cannot be a square itself, and hence we have a contradiction to $xyyz \in L$.
We conclude L is not regular. □
Given a word $w \in \Sigma^*$ we write w^R for the word read backwards. I.e. $\texttt{abc}^R = \texttt{bca}$. Formally this can be defined as

$$\begin{aligned}
\epsilon^R &= \epsilon \\
(xw)^R &= w^R x
\end{aligned}$$

We use this to define the language of even length palindromes

$$L_{\text{pali}} = \{ww^R \mid w \in \Sigma^*$$

I.e. for $\Sigma = \{\texttt{a}, \texttt{b}\}$ we have $\texttt{abba} \in L_{\text{pali}}\}$. Using the intuition that finite automata can only use finite memory it should be clear that this language is not regular, because one has to remember the first half of the word to check whether the 2nd half is the same word read backwards. Indeed, we can show:

Theorem 4.4 *Given $\Sigma = \{\texttt{a}, \texttt{b}\}$ we have that L_{pali} is not regular.*

Proof: We use the pumping lemma: We assume that $L_{\texttt{pali}}$ is regular. Now given a pumping number n we construct $w = \texttt{a}^n\texttt{bba}^n \in L_{\text{pali}}$, this word is certainly longer than n. From the pumping lemma we know that there is a splitting of the word $w = xyz$ s.t. $|xy| \leq n$ and hence y may only contain $\texttt{a}$'s and since $y \neq \epsilon$ at least one. We conclude that $xz \in L_{\text{pali}}$ where $xz = \texttt{a}^m\texttt{bba}^n$ where $m < n$. However, this word cannot be a palindrome since only the first half contains any $\texttt{a}$'s.
Hence our assumption L_{pali} is regular must be wrong. □
The proof works for any alphabet with at least 2 different symbols. However, if Σ contains only one symbol as in $\Sigma = \{1\}$ then L_{pali} is the language of an even number of 1s and this is regular $L_{\text{pali}} = (11)^*$.

5 Context free grammars

We will now introduce *context free grammars* (CFGs) as a formalism to define languages. CFGs are more general than regular expressions, i.e. there are more languages definable by CFGs (called *type-2-languages*). We will define the corresponding notion of automata, the *push down automata* (PDA) later.

5.1 What is a context-free grammar?

A context-free grammar $G = (V, \Sigma, S, P)$ is given by

- A finite set V of variables or nonterminal symbols.
- A finite set Σ of symbols or terminal symbols. We assume that the sets V and Σ are disjoint.
- A start symbol $S \in V$.
- A finite set $P \subseteq V \times (V \cup T)^*$ of productions. A production (A, α), where $A \in V$ and $\alpha \in (V \cup T)^*$ is a sequence of terminals and variables, is written as $A \to \alpha$.

As an example we define a grammar for the language of arithmetic expressions over a (using only $+$ and $*$), i.e. elements of this language are $a + (a * a)$ or $(a + a) * (a + a)$. However words like $a + +a$ or $)(a$ are not in the language. We define $G = (\{E, T, F\}, \{(,), a, +, *\}, E, P)$ where P is given by:

$$\begin{aligned} P = \{&E \to T \\ &E \to E + T \\ &T \to F \\ &T \to T * F \\ &F \to a \\ &F \to (E)\} \end{aligned}$$

To save space we may combine all the rules with the same left hand side, i.e. we write

$$\begin{aligned} P = \{&E \to T \mid E + T \\ &T \to F \mid T * F \\ &F \to a \mid (E) \end{aligned}$$

5.2 The language of a grammar

How do we check whether a word $w \in \Sigma^*$ is in the language of the grammar? We start with the start symbol E and use one production $V \to \alpha$ to replace the nonterminal symbol by the α until we have no nonterminal symbols left. This

is called *a derivation*. I.e. in the example G:

$$\begin{aligned} E &\underset{G}{\Rightarrow} E+T \\ &\underset{G}{\Rightarrow} T+T \\ &\underset{G}{\Rightarrow} F+T \\ &\underset{G}{\Rightarrow} a+T \\ &\underset{G}{\Rightarrow} a+F \\ &\underset{G}{\Rightarrow} a+(E) \\ &\underset{G}{\Rightarrow} a+(T) \\ &\underset{G}{\Rightarrow} a+(T*F) \\ &\underset{G}{\Rightarrow} a+(F*F) \\ &\underset{G}{\Rightarrow} a+(a*F) \\ &\underset{G}{\Rightarrow} a+(a*a) \end{aligned}$$

Note that $\underset{G}{\Rightarrow}$ here stands for the relation *derives in one step in grammar* G and has nothing to do with implication. When the grammar used it is clear from the context, it is conventional to drop the subscript G and simply use $\Rightarrow$, meaning *derives in one step*. In the example above we have always replaced the leftmost nonterminal symbol. This is called a *leftmost* derivation. However, this is not necessary: in general, we are free to pick any non-terminal for replacement. An alternative would be to always pick the rightmost one, which results in a *rightmost* derivation. Or we could pick whichever nonterminal seems more convenient to expand. The symbols $\underset{\text{lm}}{\Rightarrow}$ and $\underset{\text{rm}}{\Rightarrow}$ are sometimes used to indicate leftmost and rightmost derivation steps, respectively.
Given any grammar $G = (V, \Sigma, S, P)$ we define the relation *derives in one step in grammar* G as follows:

$$\begin{aligned} \underset{G}{\Rightarrow} \quad &\subseteq \quad (V \cup T)^* \times (V \cup T)^* \\ \alpha V \gamma \underset{G}{\Rightarrow} \alpha\beta\gamma \quad &\iff \quad V \to \beta \in P \end{aligned}$$

The relation *derives in grammar* G is defined as[5]:

$$\begin{aligned} \overset{*}{\underset{G}{\Rightarrow}} \quad &\subseteq \quad (V \cup T)^* \times (V \cup T)^* \\ \alpha_0 \overset{*}{\underset{G}{\Rightarrow}} \alpha_n \quad &\iff \quad \alpha \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \dots \alpha_{n-1} \underset{G}{\Rightarrow} \alpha_n \end{aligned}$$

This includes the case $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ because n can be 0.
We now say that the language of a grammar $L(G) \subseteq \Sigma^*$ is given by all words (over Σ) that can be derived in any number of steps, i.e.

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

A language that can be given by a context-free grammar is called a context-free language (CFL).

[5] $\overset{*}{\underset{G}{\Rightarrow}}$ is the *transitive-reflexive closure* of $\underset{G}{\Rightarrow}$.

5.3 More examples

Some of the languages that we have shown not to be regular are actually context-free.
The language $\{0^n 1^n \mid n \in \mathbb{N}\}$ is given by the following grammar

$$G = (\{S\}, \{0, 1\}, S, \{S \to \epsilon \mid 0S1\})$$

Also the language of palindromes

$$\{ww^R \mid w \in \{\mathrm{a}, \mathrm{b}\}^*\}$$

turns out to be context-free;

$$G = (\{S\}, \{\mathrm{a}, \mathrm{b}\}, S, \{S \to \epsilon \mid \mathrm{a}S\mathrm{a} \mid \mathrm{b}S\mathrm{b}\})$$

We also note that

Theorem 5.1 *All regular languages are context-free.*

We don't give a proof here - the idea is that regular expressions can be translated into (special) context-free grammars, i.e. $\mathrm{a}^*\mathrm{b}^*$ can be translated into

$$G = (\{A, B\}, \{\mathrm{a}, \mathrm{b}\}, \{A \to aA \mid B, B \to bB \mid \epsilon)$$

(Extensions of) context free grammars are used in computer linguistics to describe natural languages. As an example consider

$$\Sigma = \{\text{the}, \text{dog}, \text{cat}, \text{that}, \text{bites}, \text{barks}, \text{catches}\}$$

we use the grammar $G = (\{S, N, NP, VI, VT, VP\}, \Sigma, S, P)$ where

$$\begin{aligned} S &\to NP \;\; VP \\ N &\to \text{cat} \mid \text{dog} \\ NP &\to \text{the } N \mid NP \text{ that } VP \\ VI &\to \text{barks} \mid \text{bites} \\ VT &\to \text{bites} \mid \text{catches} \\ VP &\to VI \mid VT \;\; NP \end{aligned}$$

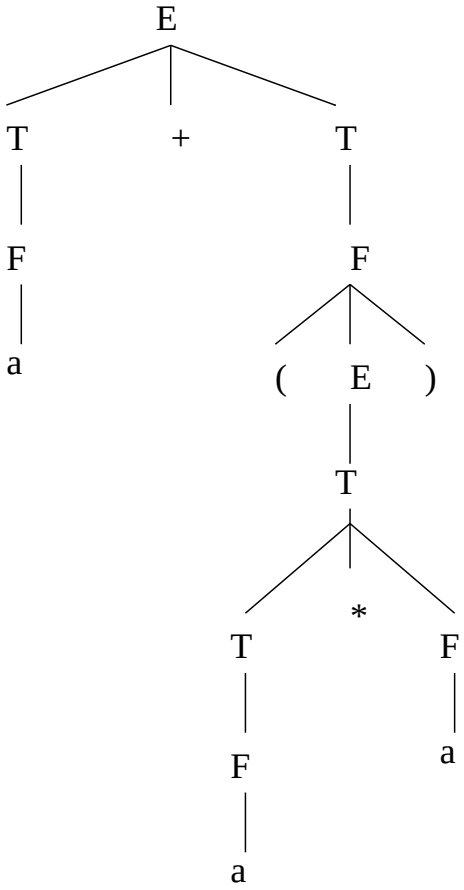
that allows us to derive interesting sentences like

> the dog that catches the cat that bites barks

An important example for context-free languages is the syntax of programming languages. We have already mentioned appendix of [GJSB00] that uses a formalism slightly different from the one introduced here. Another example is the toy language used in the compilers course [Nil05].
Note that not all syntactic aspects of programming languages are captured by the context free grammar, i.e. the fact that a variable has to be declared before used and the type correctness of expressions are not captured. However, at least in the area of programming languages, it is common practice to use the notion of "syntactically correct" in the more limited sense of "conforming to the context-free grammar of the language". Aspects such as type-correctness are considered separately.

5.4 Parse trees

With each derivation we also associate a *derivation tree* that shows the structure of the derivation. As an example consider the tree associated with the derivation of $a + (a * a)$ given before:



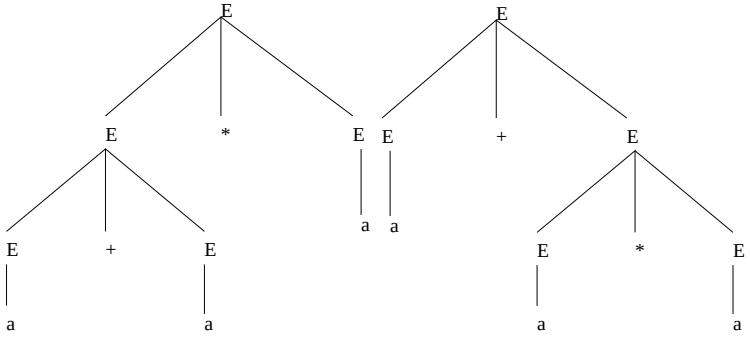
The top of the tree (called its root) is labelled with start symbol, the other *nodes* are labelled with nonterminal symbols and the leaves are labelled by terminal symbols. The word that is derived can be read off from the leaves of the tree. The important property of a parse tree is that the ancestors of an internal node correspond to a production in the grammar.

5.5 Ambiguity

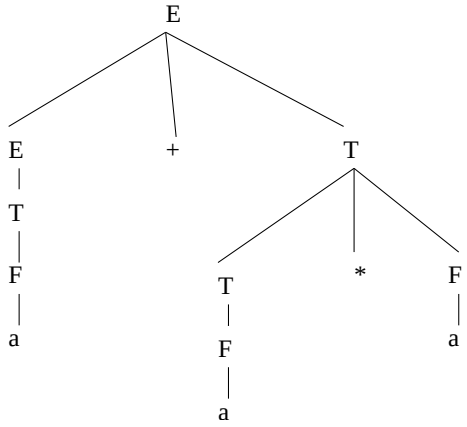
We say that a grammar G is *ambiguous* if there is a word $w \in L(G)$ for which there is more than one parse tree, or, equivalently, more than one leftmost derivation or more than one rightmost derivation. This is usually a bad thing because it entails that there is more than one way to interpret a word (i.e. it leads to semantical ambiguity).
As an example consider the following alternative grammar for arithmetical expressions: We define $G' = (\{E\}, \{(,), a, +, *\}, E, P')$ where P' is given by:

$$P' = \{E \to E + E \mid E * E \mid a \mid (E)\}$$

This grammar is shorter and requires only one variable instead of 4. Moreover it generates the same language, i.e. we have $L(G) = L(G')$. But it is ambigous: Consider $a + a * a$ we have the following parse trees:



Each parse tree correspond to a different way to read the expression, i.e. the first one corresponds to $(a + a) * a$ and the second one to $a + (a * a)$. Depending on which one is chosen an expression like $2 + 2 * 3$ may evaluate to 12 or to 8. Informally, we agree that $*$ binds more than $+$ and hence the 2nd reading is the intended one [6].
This is actually achieved by the first grammar which only allows the 2nd reading:



6 Pushdown Automata

We will now consider a new notion of automata *Pushdown Automata* (PDA). PDAs are finite automata with a stack, i.e. a data structure that can be used to store an arbitrary number of symbols (hence PDAs have an infinite set of states) but which can be only accessed in a *last-in-first-out* (LIFO) fashion. The languages that can be recognized by PDA are precisely the context free languages.

[6] Strictly speaking, this is just a natural and often convenient convention, since it implies that the meaning of the whole can be understood in terms of the meaning of the parts in the obvious way implied by the structure of the tree. However, sometimes, in real compilers, it might be necessary or convenient to parse things in a way that does not reflect the semantics as directly. But then the parse tree is usually transformed later into a simplified, internal version (an *Abstract Syntax Tree*), the structure of which reflects the intended semantics as described here.

6.1 What is a Pushdown Automaton?

A Pushdown Automaton $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is given by the following data

- A finite set Q of states,
- A finite set Σ of input symbols (the alphabet),
- A finite set Γ of stack symbols,
- A transition function

  $$\delta \in Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_{\text{fin}}(Q \times \Gamma^*)$$

  Here $\mathcal{P}_{\text{fin}}(A)$ are the finite subsets of a set, i.e. this can be defined as

  $$\mathcal{P}_{\text{fin}}(A) = \{X \mid X \subseteq A \wedge X \text{is finite.}\}$$

  Thus, PDAs are in general nondeterministic, since they may have a choice of transitions from any state. However, there are always only finitely many choices.
- An initial state $q_0 \in Q$,
- An initial stack symbol $Z_0 \in \Gamma$,
- A set of final states $F \subseteq Q$.

As an example we consider a PDA P that recognizes the language of even length palindromes over $\Sigma = \{0, 1\}$ — $L = \{ww^R \mid w \in \{0, 1\}^*\}$. Intuitively, this PDA pushes the input symbols on the stack until it *guesses* that it is in the middle and then it compares the input with what is on the stack, popping of symbols from the stack as it goes. If it reaches the end of the input precisely at the time when the stack is empty, it accepts.

$$P_0 = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_0, \#, \{q_2\})$$

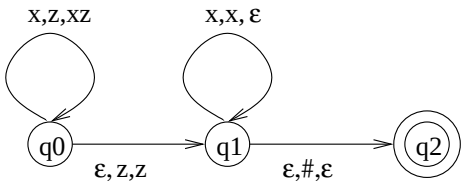
where δ is given by the following equations:

$$\begin{array}{lcll}
\delta(q_0, 0, \#) & = & \{(q_0, 0\#)\} & \\
\delta(q_0, 1, \#) & = & \{(q_0, 1\#)\} & \\
\delta(q_0, 0, 0) & = & \{(q_0, 00)\} & \\
\delta(q_0, 1, 0) & = & \{(q_0, 10)\} & \\
\delta(q_0, 0, 1) & = & \{(q_0, 01)\} & \\
\delta(q_0, 1, 1) & = & \{(q_0, 11)\} & \\
\delta(q_0, \epsilon, \#) & = & \{(q_1, \#)\} & \\
\delta(q_0, \epsilon, 0) & = & \{(q_1, 0)\} & \\
\delta(q_0, \epsilon, 1) & = & \{(q_1, 1)\} & \\
\delta(q_1, 0, 0) & = & \{(q_1, \epsilon)\} & \\
\delta(q_1, 1, 1) & = & \{(q_1, \epsilon)\} & \\
\delta(q_1, \epsilon, \#) & = & \{(q_2, \epsilon)\} & \\
\delta(q, w, z) & = & \emptyset \qquad \text{everywhere else} &
\end{array}$$

To save space we may abbreviate this by writing:

$$\begin{array}{lcll} \delta(q_0, x, z) & = & \{(q_0, xz)\} & \\ \delta(q_0, \epsilon, z) & = & \{(q_1, z)\} & \\ \delta(q_1, x, x) & = & \{(q_1, \epsilon)\} & \\ \delta(q_1, \epsilon, \#) & = & \{(q_2, \epsilon)\} & \\ \delta(q, x, z) & = & \emptyset & \text{everywhere else} \end{array}$$

where $q \in Q, x \in \Sigma, z \in \Gamma$. We obtain the previous table by expanding all the possibilities for q, x, z.
We draw the transition diagram of P by labelling each transition with a triple x, Z, γ with $x \in \Sigma, Z \in \Gamma, \gamma \in \Gamma^*$:



6.2 How does a PDA work?

At any time the state of the computation of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is given by:

- the state $q \in Q$ the PDA is in,
- the input string $w \in \Sigma^*$ that still has to be processed,
- the contents of the stack $\gamma \in \Gamma^*$.

Such a triple $(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$ is called an Instantaneous Description (ID). We define a relation $\underset{P}{\vdash} \subseteq \text{ID} \times \text{ID}$ between IDs that describes how the PDA can change from one ID to the next one. Since PDAs in general are nondeterministic this is a relation (not a function), i.e. there may be more than one possibility. There are two possibilities for $\underset{P}{\vdash}$:

1. $(q, xw, z\gamma) \underset{P}{\vdash} (q', w, \alpha\gamma)$ if $(q', \alpha) \in \delta(q, x, z)$
2. $(q, w, z\gamma) \underset{P}{\vdash} (q', w, \alpha\gamma)$ if $(q', \alpha) \in \delta(q, \epsilon, z)$

In the first case the PDA reads an input symbol and consults the transition function δ to calculate a possible new state q' and a sequence of stack symbols α that replaces the currend symbol on the top z.
In the second case the PDA ignores the input and silently moves into a new state and modifies the stack as above. The input is unchanged.

Consider the word 0110 — what are possible sequences of IDs for P_0 starting with $(q_0, 0110, \#)$?

$$\begin{array}{lll}
(q_0, 0110, \#) & \underset{P_0}{\vdash} (q_0, 110, 0\#) & 1.\ \text{with } (q_0, 0\#) \in \delta(q_0, 0, \#) \\
& \underset{P_0}{\vdash} (q_0, 10, 10\#) & 1.\ \text{with } (q_0, 10) \in \delta(q_0, 1, 0) \\
& \underset{P_0}{\vdash} (q_1, 10, 10\#) & 2.\ \text{with } (q_1, 1) \in \delta(q_0, \epsilon, 1) \\
& \underset{P_0}{\vdash} (q_1, 0, 0\#) & 1.\ \text{with } (q_1, \epsilon) \in \delta(q_1, 1, 1) \\
& \underset{P_0}{\vdash} (q_1, \epsilon, \#) & 1.\ \text{with } (q_1, \epsilon) \in \delta(q_1, 0, 0) \\
& \underset{P_0}{\vdash} (q_2, \epsilon, \epsilon) & 2.\ \text{with } (q_2, \epsilon) \in \delta(q_1, \epsilon, \#)
\end{array}$$

We write $(q, w, \gamma) \overset{*}{\underset{P}{\vdash}} (q', w', \gamma)$ if the PDA can move from (q, w, γ) to (q', w', γ') in a (possibly empty) sequence of moves. Above we have shown that

$$(q_0, 0110, \#) \overset{*}{\underset{P_0}{\vdash}} (q_2, \epsilon, \epsilon).$$

However, this is not the only possible sequence of IDs for this input. E.g. the PDA may just guess the middle wrong:

$$\begin{array}{lll}
(q_0, 0110, \#) & \underset{P_0}{\vdash} (q_0, 110, 0\#) & 1.\ \text{with } (q_0, 0\#) \in \delta(q_0, 0, \#) \\
& \underset{P_0}{\vdash} (q_1, 110, 0\#) & 2.\ \text{with } (q_1, 0) \in \delta(q_0, \epsilon, 0)
\end{array}$$

We have shown $(q_0, 0110, \#) \overset{*}{\underset{P_0}{\vdash}} (q_1, 110, 0\#)$. Here the PDA gets stuck as there is no state after $(q_1, 110, 0\#)$.
If we start with a word that is not in the language L (like 0011) then the automaton will always get stuck before reaching a final state.

6.3 The language of a PDA

There are two ways to define the language of a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ($L(P) \subseteq \Sigma^*$) because there are two notions of acceptance:

Acceptance by final state

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \gamma) \wedge q \in F\}$$

That is the PDA accepts the word w if there is any sequence of IDs starting from (q_0, w, Z_0) and leading to (q, ϵ, γ), where $q \in F$ is one of the final states. Here it doesn't play a role what the contents of the stack are at the end.

In our example the PDA P_0 would accept 0110 because $(q_0, 0110, \#) \overset{*}{\underset{P_0}{\vdash}} (q_2, \epsilon, \epsilon)$ and $q_2 \in F$. Hence we conclude $0110 \in L(P_0)$.

On the other hand since there is no successful sequence of IDs starting with $(q_0, 0011, \#)$ we know that $0011 \notin L(P_0)$.

Acceptance by empty stack

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)\}$$

That is the PDA accepts the word w if there is any sequence of IDs starting from (q_0, w, Z_0) and leading to (q, ϵ, ϵ), in this case the final state plays no role.

If we specify a PDA for acceptance by empty stack we will leave out the set of final states F and just use $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$.

Our example automaton P_0 also works if we leave out F and use acceptance by empty stack.

We can always turn a PDA that uses one acceptance method into one that uses the other. Hence, both acceptance criteria specify the same class of languages.

6.4 Deterministic PDAs

We have introduced PDAs as nondeterministic machines that may have several alternatives how to continue. We now define Deterministic Pushdown Automata (DPDA) as those that never have a choice.
To be precise we say that a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is deterministic (is a DPDA) iff

$$|\delta(q, x, z)| + |\delta(q, \epsilon, z)| \leq 1$$

Remember, that $|X|$ stands for the number of elements in a finite set X.
That is: a DPDA may get stuck but it has never any choice.
In our example the automaton P_0 is not deterministic, e.g. we have $\delta(q_0, 0, \#) = \{(q_0, 0\#)\}$ and $\delta(q_0, \epsilon, \#) = \{(q_1, \#)\}$ and hence $|\delta(q_0, 0, \#)| + |\delta(q_0, \epsilon, \#)| = 2$.
Unlike the situation for finite automata, there is in general no way to translate a nondeterministic PDA into a deterministic one. Indeed, there is no DPDA that recognizes the language L! **Nondeterministic PDAs are more powerful than deterministic PDAs.**
However, we can define a similar language L' over $\Sigma = \{0, 1, \$\}$ that can be recognized by a deterministic PDA:

$$L' = \{w\$w^R \mid w \in \{0, 1\}^*\}$$

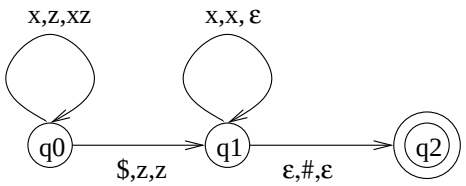
That is L' contains palindroms with a marker \$ in the middle, e.g. $01\$10 \in L'$.
We define a DPDA P' for L':

$$P' = (\{q_0, q_1, q_2\}, \{0, 1, \$\}, \{0, 1, \#\}, \delta', q_0, \#, \{q_2\})$$

where δ' is given by:

$$\begin{array}{lcll}
\delta'(q_0, x, z) & = & \{(q_0, xz)\} & x \in \{0, 1\},\ z \in \{0, 1, \#\} \\
\delta'(q_0, \$, z) & = & \{(q_1, z)\} & z \in \{0, 1, \#\} \\
\delta'(q_1, x, x) & = & \{(q_1, \epsilon)\} & x \in \{0, 1\} \\
\delta'(q_1, \epsilon, \#) & = & \{(q_2, \epsilon)\} & \\
\delta'(q, x, z) & = & \emptyset & \text{everywhere else}
\end{array}$$

Here is its transition graph:



We can check that this automaton is deterministic. In particular the 3rd and 4th line cannot overlap because # is not an input symbol.
Different to PDAs in general the two acceptance methods are not equivalent for DPDAs — acceptance by final state makes it possible to define a bigger class of langauges. Hence, we shall always use acceptance by final state for DPDAs.

6.5 Context free grammars and push-down-automata

Theorem 6.1 *For a language $L \subseteq \Sigma^*$ the following two statements are equivalent:*

1. *L is given by a CFG G, $L = L(G)$.*
2. *L is the language of a PDA P, $L = L(P)$.*

To summarize: Context Free Languages (CFLs) can be described by a Context Free Grammar (CFG) and can be processed by a pushdown automaton.
We will he only show how to construct a PDA from a grammar - the other direction is shown in [HMU01] (6.3.2, pp. 241).
Given a CFG $G = (V, \Sigma, S, P)$, we define a PDA

$$P(G) = (\{q_0\},\ \Sigma,\ V \cup \Sigma,\ \delta,\ q_0,\ S)$$

where δ is defined as follows:

$$\begin{array}{lcll} \delta(q_0, \epsilon, A) & = & \{(q_0, \alpha) \mid A \to \alpha \in P\} & \text{for all } A \in V \\ \delta(q_0, a, a) & = & \{(q_0, \epsilon)\} & \text{for all } a \in \Sigma. \end{array}$$

We haven't given a set of final states because we use acceptance by empty stack. Yes, we use only one state!
Take as an example

$$G = (\{E, T, F\},\ \{\texttt{(}, \texttt{)}, \texttt{a}, \texttt{+}, \texttt{*}\},\ E,\ P)$$

where the set of productions P are given by

$$\begin{array}{lcl} E & \to & T \mid E\,\texttt{+}\,T \\ T & \to & F \mid T\,\texttt{*}\,F \\ F & \to & \texttt{a} \mid \texttt{(}\,E\,\texttt{)} \end{array}$$

we define

$$P(G) = (\{q_0\},\ \{\texttt{(}, \texttt{)}, \texttt{a}, \texttt{+}, \texttt{*}\},\ \{E, T, F, \texttt{(}, \texttt{)}, \texttt{a}, \texttt{+}, \texttt{*}\},\ \delta,\ q_0,\ E)$$

where

$$\begin{array}{rcll}
\delta(q_0, \epsilon, E) & = & \{(q_0, T),\ (q_0, E\texttt{+}T)\} & \\
\delta(q_0, \epsilon, T) & = & \{(q_0, F),\ (q_0, T\texttt{*}F)\} & \\
\delta(q_0, \epsilon, F) & = & \{(q_0, \texttt{a}),\ (q_0, \texttt{(}E\texttt{)})\} & \\
\delta(q_0, \texttt{(}, \texttt{(}) & = & \{(q_0, \epsilon)\} & \\
\delta(q_0, \texttt{)}, \texttt{)}) & = & \{(q_0, \epsilon)\} & \\
\delta(q_0, \texttt{a}, \texttt{a}) & = & \{(q_0, \epsilon)\} & \\
\delta(q_0, \texttt{+}, \texttt{+}) & = & \{(q_0, \epsilon)\} & \\
\delta(q_0, \texttt{*}, \texttt{*}) & = & \{(q_0, \epsilon)\} & \\
\delta(q, x, z) & = & \emptyset & \text{everywhere else}
\end{array}$$

How does the $P(G)$ accept `a+(a*a)`?

$$\begin{array}{rcl}
(q_0,\ \texttt{a+(a*a)},\ E) & \vdash & (q_0,\ \texttt{a+(a*a)},\ E\texttt{+}T) \\
& \vdash & (q_0,\ \texttt{a+(a*a)},\ T\texttt{+}T) \\
& \vdash & (q_0,\ \texttt{a+(a*a)},\ F\texttt{+}T) \\
& \vdash & (q_0,\ \texttt{a+(a*a)},\ \texttt{a+}T) \\
& \vdash & (q_0,\ \texttt{+(a*a)},\ \texttt{+}T) \\
& \vdash & (q_0,\ \texttt{(a*a)},\ T) \\
& \vdash & (q_0,\ \texttt{(a*a)},\ F) \\
& \vdash & (q_0,\ \texttt{(a*a)},\ \texttt{(}E\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a*a)},\ E\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a*a)},\ T\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a*a)},\ T\texttt{*}F\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a*a)},\ F\texttt{*}F\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a*a)},\ \texttt{a*}F\texttt{)}) \\
& \vdash & (q_0,\ \texttt{*a)},\ \texttt{*}F\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a)},\ F\texttt{)}) \\
& \vdash & (q_0,\ \texttt{a)},\ \texttt{a)}) \\
& \vdash & (q_0,\ \texttt{)},\ \texttt{)}) \\
& \vdash & (q_0,\ \epsilon,\ \epsilon)
\end{array}$$

Hence $\texttt{a+(a*a)} \in L(P(G))$.
This above example illustrates the general idea:

$$\begin{array}{rcl}
w \in L(G) & \iff & S \stackrel{*}{\Rightarrow} w \\
& \iff & (q_0, w, S) \stackrel{*}{\vdash} (q_0, \epsilon, \epsilon) \\
& \iff & w \in L(P(G))
\end{array}$$

The automaton we have constructed is very non-deterministic: Whenever we have a choice between different rules the automaton may silently choose one of the alternatives.

7 How to implement a recursive descent parser

A parser is a program that processes input defined by a context-free grammar. The translation given in the previous section is not very useful in the design of such a program because of the non-determinism. Here I show how for a certain class of grammars this non-determinism can be eliminated and using the example of arithmetical expressions I will show how a JAVA-program can be constructed that parses and evaluates expressions.

7.1 What is a LL(1) grammar ?

The basic idea of a *recursive descent parser* is to use the current input symbol to decide which alternative to choose. Grammars that have the property that it is possible to do this are called LL(1) grammars.
First we introduce an end marker \$, for a given $G = (V, \Sigma, S, P)$ we define the augmented grammar $G^{\$} = (V', \Sigma', S', P')$ where

- $V' = V \cup \{S'\}$ where S' is chosen s.t. $S' \notin V \cup \Sigma$,
- $\Sigma' = \Sigma \cup \{\$\}$ where \$ is chosen s.t. $\$ \notin V \cup \Sigma$,
- $P' = P \cup \{S' \to S\$\}$

The idea is that

$$L(G^{\$}) = \{w\$ \mid w \in L(G)\}$$

Now for each nonterminal symbol $A \in V' \cup \Sigma'$ we define

$$\text{First}(A) = \{a \mid a \in \Sigma \wedge A \stackrel{*}{\Rightarrow} a\beta\}$$
$$\text{Follow}(A) = \{a \mid a \in \Sigma \wedge S' \stackrel{*}{\Rightarrow} \alpha A a \beta\}$$

i.e. First(A) is the set of terminal symbols with which a word derived from A may start and Follow(A) is the set of symbols that may occur directly after A. We use the augmented grammar to have a marker for the end of the word.
For each production $A \to \alpha \in P$ we define the set Lookahead($A \to \alpha$) that is the set of symbols that indicate that we are in this alternative.

$$\text{Lookahead}(A \to B_1 B_2 \ldots B_n) = \bigcup \{\text{First}(B_i) \mid \forall 1 \leq k < i . B_k \stackrel{*}{\Rightarrow} \epsilon\} \cup \begin{cases} \text{Follow}(A) & \text{if } B_1 B_2 \ldots B_k \stackrel{*}{\Rightarrow} \epsilon \\ \emptyset & \text{otherwise} \end{cases}$$

We now say **a grammar G is LL(1)**, iff for each pair $A \to \alpha, A \to \beta \in P$ with $\alpha \neq \beta$ it is the case that Lookahead($A \to \alpha$) $\cap$ Lookahead($A \to \beta$) $= \emptyset$

7.2 How to calculate First and Follow

We have to determine whether $A \stackrel{*}{\Rightarrow} \epsilon$. If there are no ϵ-production we know that the answer is always negative, otherwise

- If $A \to \epsilon \in P$ we know that $A \stackrel{*}{\Rightarrow} \epsilon$.
- If $A \to B_1 B_2 \ldots B_n$ where all B_i are nonterminal symbols and for all $1 \leq i \leq n$: $B_i \stackrel{*}{\Rightarrow} \epsilon$ then we also know $A \stackrel{*}{\Rightarrow} \epsilon$.

We calculate First and Follow in a similar fashion:

- First$(a) = \{a\}$ if $a \in \Sigma$.
- If $A \to B_1 B_2 \dots B_n$ and there is an $i \leq n$ s.t. $\forall 1 \leq k < i . B_k \stackrel{*}{\Rightarrow} \epsilon$ then we add First(B_i) to First(A).

And for Follow:

- \$ $\in$ Follow(S) where S is the original start symbol.
- If there is a production $A \to \alpha B \beta$ then everything in First(β) is in Follow(B).
- If there is a production $A \to \alpha B \beta$ with $\beta \stackrel{*}{\Rightarrow} \epsilon$ then everything in Follow(A) is also in Follow(B).

7.3 Constructing an LL(1) grammar

Let's have a look at the grammar G for arithmetical expressions again. $G = (\{E, T, F\}, \{(,), a, +, *\}, E, P)$ where

$$\begin{aligned} P = \{ & E \to T \mid E + T \\ & T \to F \mid T * F \\ & F \to a \mid (E) \end{aligned}$$

We don't need the Follow-sets in the moment because the empty word doesn't occur in the grammar. For the nonterminal symbols we have

$$\begin{aligned} \text{First}(F) &= \{\texttt{a}, \texttt{(}\} \\ \text{First}(T) &= \{\texttt{a}, \texttt{(}\} \\ \text{First}(E) &= \{\texttt{a}, \texttt{(}\} \end{aligned}$$

and now it is easy to see that most of the Lookahead-sets agree, e.g.

$$\begin{aligned} \text{Lookahead}(E \to T) &= \{\texttt{a}, \texttt{(}\} \\ \text{Lookahead}(E \to E + T) &= \{\texttt{a}, \texttt{(}\} \\ \text{Lookahead}(T \to F) &= \{\texttt{a}, \texttt{(}\} \\ \text{Lookahead}(T \to T * F) &= \{\texttt{a}, \texttt{(}\} \\ \text{Lookahead}(F \to a) &= \{\texttt{a}\} \\ \text{Lookahead}(F \to (E)) &= \{\texttt{(}\} \end{aligned}$$

Hence the grammar G **is not LL(1)**.
However, luckily there is an alternative grammar G' that defines the same language: $G' = (\{E, E', T, T', F\}, \{(,), a, +, *\}, E, P')$ where

$$\begin{aligned} P' = \{ & E \to TE' \\ & E' \to \texttt{+}TE' \mid \epsilon \\ & T \to FT' \\ & T' \to \texttt{*}FT' \mid \epsilon \\ & F \to a \mid \texttt{(}E\texttt{)} \end{aligned}$$

Since we have ϵ-productions we do need the Follow-sets.

$$\begin{aligned}
\text{First}(E) = \text{First}(T) = \text{First}(F) &= \{\texttt{a}, \texttt{(}\} \\
\text{First}(E') &= \{\texttt{+}\} \\
\text{First}(T') &= \{\texttt{*}\} \\
\text{Follow}(E) = \text{Follow}(E') &= \{\texttt{)}, \texttt{\$}\} \\
\text{Follow}(T) = \text{Follow}(T') &= \{\texttt{+}, \texttt{)}, \texttt{\$}\} \\
\text{Follow}(F) &= \{\texttt{+}, \texttt{*}, \texttt{)}, \texttt{\$}\}
\end{aligned}$$

Now we calculate the Lookahead-sets:

$$\begin{aligned}
\text{Lookahead}(E \to TE') &= \{\texttt{a}, \texttt{(}\} \\
\text{Lookahead}(E' \to \texttt{+}TE') &= \{\texttt{+}\} \\
\text{Lookahead}(E' \to \epsilon) &= \text{Follow}(E') = \{\texttt{)}, \texttt{\$}\} \\
\text{Lookahead}(T \to \texttt{+}FT') &= \{\texttt{a}, \texttt{(}\} \\
\text{Lookahead}(T' \to \texttt{*}FT') &= \{\texttt{*}\} \\
\text{Lookahead}(T' \to \epsilon) &= \text{Follow}(T') = \{\texttt{+}, \texttt{)}, \texttt{\$}\} \\
\text{Lookahead}(F \to a) &= \{\texttt{a}\} \\
\text{Lookahead}(F \to (E)) &= \{\texttt{(}\}
\end{aligned}$$

Hence the grammar G' **is LL(1)**.

7.4 How to implement the parser

We can now implement a parser - one way would be to construct a deterministic PDA. However, using JAVA we can implement the parser using recursion - here the internal JAVA stack plays the role of the stack of the PDA.
First of all we have to separate the input into *tokens* that are the terminal symbols of our grammar. To keep things simple I assume that tokens are separated by blanks, i.e. one has to type

```
( a + a ) * a
```

for `(a+a)*a`. This has the advantage that we can use `java.util.StringTokenizer`.
In a real implementation tokenizing is usually done by using finite automata.
I don't want to get lost in java details - in the main program I read a line and produce a tokenizer:

```
String line=in.readLine();
st = new StringTokenizer(line+" $");
```

The tokenizer `st` and the current token are static variables. I implement the convenience method `next` taht assigns the next token to `curr.`

```
static StringTokenizer st;
static String curr;

static void next() {
```

```
        try {
            curr=st.nextToken().intern();
        } catch( NoSuchElementException e) {
            curr=null;
        }
    }
```

We also implement a convenience method `error(String)` to report an error and terminate the program.
Now we can translate all productions into methods using the Lookahead sets to determine which alternative to choose. E.g. we translate

$$E' \rightarrow +TE' \mid \epsilon$$

into (using `E1` for E' to follow JAVA rules):

```
    static void parseE1() {
        if (curr=="+") {
            next();
            parseT();
            parseE1();
        } else if(curr==")" || curr=="$" ) {
        } else {
            error("Unexpected :"+curr);
        }
```

The basic idea is to

- Translate each occurrence of a non terminal symbol into a test that this symbol has been read and a call of `next()`.
- Translate each nonterminal symbol into a call of the method with the same name.
- If you have to decide between different productions use the lookahead sets to determine which one to use.
- If you find that there is no way to continue call `error()`.

We initiate the parsing process by calling `next()` to read the first symbol and then call `parseE()`. If after processing `parseE()` we are at the end marker, then the parsing has been successful.

```
        next();
        parseE();
        if(curr=="$") {
            System.out.println("OK ");
        } else {
            error("End expected");
        }
```

The complete parser can be found at
http://www.cs.nott.ac.uk/~txa/g51mal/ParseE0.java.
Actually, we can be a bit more realistic and turn the parser into a simple evaluator by

- Replace `a` by any integer. I.e. we use

  ```
  Integer.valueOf(curr).intValue();
  ```

  to translate the current token into a number. JAVA will raise an exception if this fails.

- Calculate the value of the expression read. I.e. we have to change the method interfaces:

  ```
  static int parseE()
  static int parseE1(int x)
  static int parseT()
  static int parseT1(int x)
  static int parseF()
  ```

  The idea behind `parseE1` and `parseT1` is to pass the result calculated so far and leave it to the method to incorporate the missing part of the expression. I.e. in the case of `parseE1`

  ```
  static int parseE1(int x) {
      if (curr=="+") {
          next();
          int y = parseT();
          return parseE1(x+y);
      } else if(curr==")" || curr=="$" ) {
          return x;
      } else {
          error("Unexpected :"+curr);
          return x;
      }
  }
  ```

Here is the complete program with evaluation
http://www.cs.nott.ac.uk/~txa/g51mal/ParseE.java.
We can run the program and observe that it handles precedence of operators and brackets properly:

```
[txa@jacob misc]$ java ParseE
3 + 4 * 5
OK 23
[txa@jacob misc]$ java ParseE
( 3 + 4 ) * 5
OK 35
```

7.5 Beyond hand-written parsers — use parser generators

The restriction to LL(1) has a number of disadvantages: In many case a natural (and unambiguous) grammar like G has to be changed. There are some cases where this is actually impossible, i.e. although the language is deterministic there is no LL(1) grammar for this.
Luckily, there is a more powerful approach, called LR(1). LL(1) proceeds from top to bottom, when we are looking at the parse tree, hence this is called top-down parsing. In contrast LR(1) proceeds from bottom to top, i.e. it tries to construct the parse tree from the bottom upwards.
The disadvantage with LR(1) and the related approach LALR(1) (which is slightly less powerful but much more efficient) is that it is very hard to construct LR-parsers by hand. Hence there are automated tools that get the grammar as an input and produce a parser as the output. One of the first of those *parser generators* was YACC for C. Nowadays one can find parser generators for many languages such as JAVA CUP for Java [Hud99] and Happy for Haskell [Mar01]. However, there are also very serious tools based on LL(k) parsing technology, such as ANTLR (ANother Tool for Language Recognition) [Par05], that overcome some of the problems of basic LL parsing and that provides as much automation as any other parser generator tool. It is true that a grammar still may have to be changed a little bit to parseable, but that is true for LR parsing too. In practice, once one become familiar with a tool, it is not that hard to write a grammar in such a way so as to avoid the most common pitfalls from the outset. Additionally, most tools provide mechanics such as declarative specification of disambiguation rules that allows grammars to be written in a natural way without worrying too much about the details of the underlying parsing technology. Today, when it comes to choosing a tool, the underlying parsing technology is probably less important than other factors such as tool quality, supported development languages, feature set, etc.

8 Turing machines and the rest

A Turing machine (TM) is a generalization of a PDA that uses a tape instead of a stack. Turing machines are an abstract version of a computer - they have been used to define formally what is *computable*. There are a number of alternative approaches to formalize the concept of computability (e.g. called the λ-calculus, or μ-recursive functions, ...) but they can all shown to be equivalent. That this is the case for any reasonable notion of computation is called the *Church-Turing Thesis*.
On the other side there is a generalization of context free grammars called phrase structure grammars or just grammars. Here we allow several symbols on the left hand side of a production, e.g. we may define the context in which a rule is applicable. Languages definable by grammars correspond precisely to the ones that may be accepted by a Turing machine and those are called *Type-0-languages* or the *recursively enumerable languages* (or *semidecidable languages*)
Turing machines behave different from the previous machine classes we have seen: they may run forever, without stopping. To say that a language is accepted by a Turing machine means that the TM will stop in an accepting state for each word that is in the language. However, if the word is not in the language the

Turing machine may stop in a non-accepting state or loop forever. In this case we can never be sure whether the given word is in the language - i.e. the Turing machine doesn't decide the word problem.
We say a language is *recursive* (or *decidable*), if there is a TM that will always stop. There are *type-0-languages* that are not recursive — the most famous one is the *halting problem*. This is the language of encodings of Turing machines that will always stop.
There is no type of grammars that captures all recursive languages (and for theoretical reasons there cannot be one). However there is a subset of recursive languages that are called *context-sensitive languages*. They are given by *context-sensitive grammars*: grammars where the left hand side of a production is always shorter than the right hand side. Context sensitive languages on the other hand correspond to linear bounded TMs, these are those TMs that use only a tape whose length can be given by a linear function over the length of the input.

8.1 What is a Turing machine?

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is given by the following data

- A finite set Q of states,
- A finite set Σ of symbols (the alphabet),
- A finite set Γ of tape symbols s.t. $\Sigma \subseteq \Gamma$. This is the case because we use the tape also for the input.
- A transition function

  $$\delta \in Q \times \Gamma \to \{\text{stop}\} \cup Q \times \Gamma \times \{\text{L}, \text{R}\}$$

  The transition function defines how the function behaves if is in state q and the symbol on the tape is x. If $\delta(q, x) = \text{stop}$ then the machine stops otherwise if $\delta(q, x) = (q', y, d)$ the machines gets into state q', writes y on the tape (replacing x) and moves left if $d = \text{L}$ or right, if $d = \text{R}$
- An initial state $q_0 \in Q$,
- The blank symbol $B \in \Gamma$ but $B \notin \Sigma$. In the beginning only a finite section of the tape containing the input is not blank.
- A set of final states $F \subseteq Q$.

In [HMU01] the transition function is defined without the stop option as $\delta \in Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$. However they allow δ to be undefined, which correspond to our function returning stop.
This defines deterministic Turing machines, for non-deterministic TMs we change the transition function to

$$\delta \in Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L}, \text{R}\})$$

Here stop corresponds to δ returning an empty set. As for finite automata (and unlike for PDAs) there is no difference in the strength of deterministic or non-deterministic TMs.

As for PDAs we define instantaneous descriptions ID for Turing machines. We have $\text{ID} = \Gamma^* \times Q \times \Gamma^*$ where $(\gamma_L, q, \gamma_R) \in \text{ID}$ means that the TM is in state Q and left from the head the non-blank part of the tape is γ_L and starting with the head itself and all the non-blank symbols to the right is γ_R.
We define the next state relation $\underset{M}{\vdash}$ similar as for PDAs:

1. $(\gamma_L,\, q,\, x\gamma_R) \underset{M}{\vdash} (\gamma_L y,\, q',\, \gamma_R)$ if $\delta(q, x) = (q', y, R)$
2. $(\gamma_L z,\, q,\, x\gamma_R) \underset{M}{\vdash} (\gamma_L,\, q',\, zy\gamma_R)$ if $\delta(q, x) = (q', y, L)$
3. $(\epsilon,\, q,\, x\gamma_R) \underset{M}{\vdash} (\epsilon,\, q',\, By\gamma_R)$ if $\delta(q, x) = (q', y, L)$
4. $(\gamma_L,\, q,\, \epsilon) \underset{M}{\vdash} (\gamma_L y,\, q',\, \epsilon)$ if $\delta(q, B) = (q', y, R)$
5. $(\gamma_L z,\, q,\, \epsilon) \underset{M}{\vdash} (\gamma_L,\, q',\, zy)$ if $\delta(q, B) = (q', y, L)$
6. $(\epsilon,\, q,\, \epsilon) \underset{M}{\vdash} (\epsilon,\, q',\, By)$ if $\delta(q, B) = (q', y, L)$

The cases 3 to 6 are only needed to deal with the situation of having reached the end of the non-blank part of the tape.
We say that a TM M accepts a word if it goes into an accepting state, i.e. the language of a TM is defined as

$$L(M) = \{w \in \Sigma^* \mid (\epsilon, q_0, w) \overset{*}{\underset{M}{\vdash}} (\gamma_L, q', \gamma_R) \wedge q' \in F\}$$

I.e. the TM stops automatically if it goes into an accepting state. However, it may also stop in a non-accepting state if δ returns stop - in this case the word is rejected.
A TM M decides a language if it accepts it and it never loops (in the negative case).
To illustrate this we define a TM M that accepts the language $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ — this is a language that cannot be recognized by a PDA or be defined by a CFG.
We define $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ by

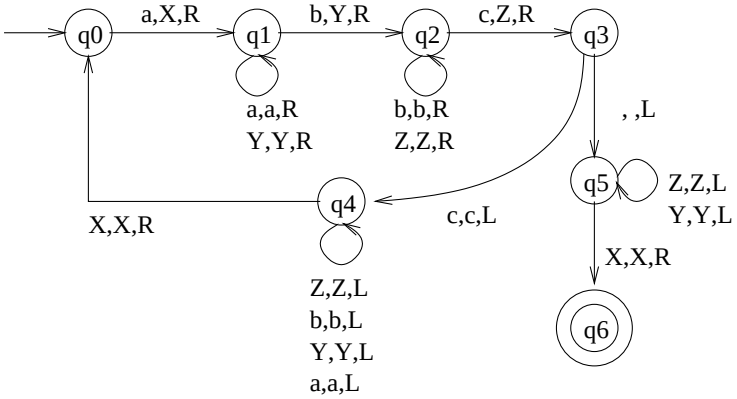
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathrm{c}\}$
- $\Gamma = \Sigma \cup \{X, Y, Z, \sqcup\}$

- δ is given by

$$\begin{aligned}
\delta(q_0, \sqcup) &= (\sqcup, q_6, \mathrm{R}) \\
\delta(q_0, \mathtt{a}) &= (X, q_1, \mathrm{R}) \\
\delta(q_1, \mathtt{a}) &= (\mathtt{a}, q_1, \mathrm{R}) \\
\delta(q_1, Y) &= (Y, q_1, \mathrm{R}) \\
\delta(q_1, \mathtt{b}) &= (Y, q_2, \mathrm{R}) \\
\delta(q_2, \mathtt{b}) &= (\mathtt{b}, q_2, \mathrm{R}) \\
\delta(q_2, Z) &= (Z, q_2, \mathrm{R}) \\
\delta(q_2, \mathtt{c}) &= (Z, q_3, \mathrm{R}) \\
\delta(q_3, \sqcup) &= (\sqcup, q_5, \mathrm{L}) \\
\delta(q_3, \mathtt{c}) &= (\mathtt{c}, q_4, \mathrm{L}) \\
\delta(q_4, Z) &= (Z, q_4, \mathrm{L}) \\
\delta(q_4, \mathtt{b}) &= (\mathtt{b}, q_4, \mathrm{L}) \\
\delta(q_4, Y) &= (Y, q_4, \mathrm{L}) \\
\delta(q_4, \mathtt{a}) &= (\mathtt{a}, q_4, \mathrm{L}) \\
\delta(q_4, X) &= (X, q_0, \mathrm{R}) \\
\delta(q_5, Z) &= (Z, q_5, \mathrm{L}) \\
\delta(q_5, Y) &= (Y, q_5, \mathrm{L}) \\
\delta(q_5, X) &= (X, q_6, \mathrm{R}) \\
\delta(q, x) &= \text{stop} \qquad \text{everywhere else}
\end{aligned}$$

- $q_0 = q_0$
- $B = \sqcup$
- $F = \{q_6\}$

The machine replaces an `a` by X (q_0) and then looks for the first `b` replaces it by Y (q_1) and looks for the first `c` and replaces it by a Z (q_2). If there are more `c`'s left it moves left to the next `a` (q_4) and repeats the cycle. Otherwise it checks whether there are no `a`'s and `b`'s left (q_5) and if so goes in an accepting state (q_6).
Graphically the machine can be represented by the following transition diagram, where the edges are labelled by (read-symbol,write-symbol,move-direction):



E.g. consider the sequence of IDs on `aabbcc`:

$$\begin{aligned}
(\epsilon, q_0, \texttt{aabbcc}) &\vdash (\texttt{X}, q_1, \texttt{abbcc}) \\
&\vdash (\texttt{Xa}, q_1, \texttt{bbcc}) \\
&\vdash (\texttt{XaY}, q_2, \texttt{bcc}) \\
&\vdash (\texttt{XaYb}, q_2, \texttt{cc}) \\
&\vdash (\texttt{XaYbZ}, q_3, \texttt{c}) \\
&\vdash (\texttt{XaYb}, q_4, \texttt{Zc}) \\
&\vdash (\texttt{XaY}, q_4, \texttt{bZc}) \\
&\vdash (\texttt{Xa}, q_4, \texttt{YbZc}) \\
&\vdash (\texttt{X}, q_4, \texttt{aYbZc}) \\
&\vdash (\epsilon, q_4, \texttt{XaYbZc}) \\
&\vdash (\texttt{X}, q_0, \texttt{aYbZc}) \\
&\vdash (\texttt{XX}, q_1, \texttt{YbZc}) \\
&\vdash (\texttt{XXY}, q_1, \texttt{bZc}) \\
&\vdash (\texttt{XXYY}, q_2, \texttt{Zc}) \\
&\vdash (\texttt{XXYYZ}, q_2, \texttt{c}) \\
&\vdash (\texttt{XXYYZZ}, q_3, \epsilon) \\
&\vdash (\texttt{XXYYZ}, q_5, \texttt{Z}_\sqcup) \\
&\vdash (\texttt{XXYY}, q_5, \texttt{ZZ}_\sqcup) \\
&\vdash (\texttt{XXY}, q_5, \texttt{YZZ}_\sqcup) \\
&\vdash (\texttt{XX}, q_5, \texttt{YYZZ}_\sqcup) \\
&\vdash (\texttt{X}, q_5, \texttt{XYYZZ}_\sqcup) \\
&\vdash (\texttt{XX}, q_6, \texttt{YYZZ}_\sqcup)
\end{aligned}$$

We see that M accepts `aabbcc`. Since M never loops it does actually decide L.

8.2 Grammars and context-sensitivity

Grammars $G = (V, \Sigma, S, P)$ are defined as context-free grammars before with the only difference that there may be several symbols on the left-hand side of a production, i.e. $P \subseteq (V \cup T)^+ \times (V \cup T)^*$. Here $(V \cup T)^+$ means that at least one symbol has to present. The relation derives $\underset{G}{\Rightarrow}$ (and $\overset{*}{\underset{G}{\Rightarrow}}$) is defined as before

$$\begin{aligned}
\underset{G}{\Rightarrow} \quad &\subseteq \quad (V \cup T)^* \times (V \cup T)^* \\
\alpha\beta\gamma \underset{G}{\Rightarrow} \alpha\beta'\gamma \quad &\iff \quad \beta \to \beta' \in P
\end{aligned}$$

and as before the language of G is defined as

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

We say that a grammar is context-sensitive (or type 1) if the left hand side of a production is at least as long as the right hand side. That is for each $\alpha \to \beta \in P$ we have $|\alpha| \leq |\beta|$

Here is an example of a context sensitive grammar: $G = (V, \Sigma, S, P)$ with $L(G) = \{\{a^n b^n c^n \mid n \in \mathbb{N} \wedge n \geq 1\}$. where

- $V = \{S, B, C\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathrm{c}\}$
- $$\begin{aligned} P = \{ & S \to aSBC \\ & S \to aBC \\ & aB \to ab \\ & CB \to BC \\ & bB \to bb \\ & bC \to bc \\ & cC \to cc\} \end{aligned}$$

We present without proof:

Theorem 8.1 *For a language $L \subseteq \Sigma^*$ the following is equivalent:*

1. *L is accepted by a Turing machine M, i.e. $L = L(M)$*
2. *L is given by a grammar G, i.e. $L = L(G)$*

Theorem 8.2 *For a language $L \subseteq \Sigma^*$ the following is equivalent:*

1. *L is accepted by a Turing machine M, i.e. $L = L(M)$ such that the length of the tape is bounded by a linear function in the length of the input, i.e. $|\gamma_L| + |\gamma_R| \leq f(x)$ where $f(x) = ax + b$ with $a, b \in \mathbb{N}$.*
2. *L is given by a context sensitive grammar G, i.e. $L = L(G)$*

8.3 The halting problem

Turing showed that there are languages that are accepted by a TM (i.e. type 0 languages) but that are undecidable. The technical details of this construction are quite involved but the basic idea is quite simple and is closely related to Russell's paradox, which we have seen in MCS.
Let's fix a simple alphabet $\Sigma = \{0, 1\}$. As computer scientist we are well aware that everything can be coded up in bits and hence we accept that there is an encoding of TMs in binary. I.e. given a TM M we write $\lceil M \rceil \in \{0, 1\}^*$ for its binary encoding. We assume that the encoding contains its length s.t. we know when subsequent input on the tape starts.
Now we define the following language

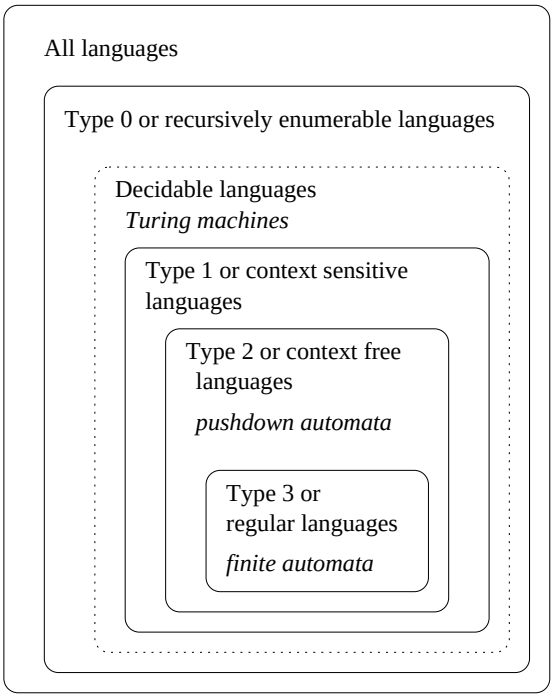
$$L_{\text{halt}} = \{\lceil M \rceil w \mid M \text{ halts on input } w.\}$$

It is easy (although the details are quite daunting) to define a TM that accepts this language: we just simulate M and accept if M stops.
However, Turing showed that there is no TM that decides this language. To see this let us assume that there is a TM H that decides L. Now using H we construct a new TM F that is a bit obnoxious: F on input x runs H on xx. If H says yes then F goes into a loop otherwise (H says no) F stops.

The question is what happens if I run F on $\lceil F \rceil$? Let us assume it terminates, then H applied to $\lceil F \rceil \lceil F \rceil$ returns yes and hence we must conclude that F on $\lceil F \rceil$ loops??? On the other hand if F with input $\lceil F \rceil$ loops then H applied to $\lceil F \rceil \lceil F \rceil$ will stop and reject and hence we have to conclude that F on $\lceil F \rceil$ will stop?????
This is a contradiction and hence we must conclude that our assumption that there is a TM H that decides L_{halt} is false. We say L_{halt} is undecidable.
We haven shown that a Turing machine cannot decide whether a program (for a Turing machine) halts. Maybe we could find a more powerful programming language that overcomes this problem? It turns out that all computational formalisms (i.e. programming languages) that can actually be implemented are equal in power and can be simulated by each other — this observation is called the *Church-Turing thesis* because it was first formulated by Alonzo Church and Alan Turing in the 30ies.

8.4 Back to Chomsky

At the end of the course we should have another look at the Chomsky hierarchy, which classifies languages based on sublasses of grammars, or equivalently by different types of automata that recognize them



We have worked our way from the bottom to the top of the hierarchy: starting with finite automata, i.e. computation with fixed amount of memory via pushdown automata (finite automata with a stack) to Turing machines (finite automata with a tape). Correspondigly we have introduced different grammatical formalisms: regular expressions, context-free grammars and grammars.
Note that at each level there are languages that are on the next level but not on the previous: $\{a^n b^n \mid n \in \mathbb{N}\}$ is level 2 but not level 3; $\{a^n b^n c^n\}$ is level 1

but not level 2 and the Halting problem is level 0 but not level 1.
We could have gone the other way: starting with Turing machines and grammars and then introduce restrictions on them. I.e. Turing machines that only use their tapes as a stack, and Turing machines that never use the tape apart for reading the input. Again correspondingly we can define restrictions on the grammar sise: first introduce context-free grammars and then grammars where all productions are of the form $A \to aB$ or $A \to a$, with A, B non-terminal symbols and a, b are terminals. These grammars correspond precisely to regular expressions (I leave this as an exercise).
I believe that Chomsky introduced his herarchy as a classification of grammars and that the relation to automata was only observed a bit later. This is maybe the reason why he introduced the Type-1 level, which is not so interesting from an automata point of view (unless you are into computational complexity, i.e. resource use - here linear use of memory). It is also the reason why on the other hand the decidable languages do not constitute a level: there is no corresponding grammatical formalism (we can even prove this).

References

[Alt01] Thorsten Altenkirch. Mathematics for computer scientists (g51mcs) — lecture notes. http://www.cs.nott.ac.uk/∼txa/g51mcs/notes.pdf, 2001. 20

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2nd edition edition, 2000. 3, 30

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition edition, 2001. 2, 6, 9, 24, 37, 45

[Hud99] Scott E. Hudson. Cup parser generator for java. http://www.cs.princeton.edu/∼appel/modern/java/CUP/, 1999. 44

[Mar01] Simon Marlow. Happy: The parser generator for haskell. http://www.haskell.org/happy/, 2001. 44

[Nil05] Henrik Nilsson. Compilers (g52cmp) — lecture notes. http://www.cs.nott.ac.uk/∼nhn/G52CMP, 2005. 30

[Par05] Terence Parr. ANTLR: Another tool for language recognition. http://www.antlr.org/, 2005. 44

[Sch99] Uwe Schöning. *Theoretische Informatik — kurzgefaßt*. Spektrum - Akademischer Verlag, 3. Auflage edition, 1999. 2