

Machines and their languages (G51MAL)

Lecture notes

Spring 2003

Thorsten Altenkirch

April 23, 2004

Contents

1	Introduction	3
1.1	Examples on syntax	3
1.2	What is this course about?	4
1.3	Applications	4
1.4	History	5
1.4.1	The Chomsky Hierarchy	5
1.4.2	Turing machines	6
1.5	Languages	6
2	Finite Automata	8
2.1	Deterministic finite automata	8
2.1.1	What is a DFA?	8
2.1.2	The language of a DFA	9
2.2	Nondeterministic Finite Automata	10
2.2.1	What is an NFA?	10
2.2.2	The language accepted by an NFA	10
2.2.3	The subset construction	12
3	Regular expressions	15
3.1	What are regular expressions?	15
3.2	The meaning of regular expressions	16
3.3	Translating regular expressions to NFAs	18
3.4	Summing up ...	25
4	Showing that a language is not regular	27
4.1	The pumping lemma	27
4.2	Applying the pumping lemma	28
5	Context free grammars	30
5.1	What is a context-free grammar?	30
5.2	The language of a grammar	30
5.3	More examples	31
5.4	Parse trees	33
5.5	Ambiguity	34

6	Pushdown Automata	35
6.1	What is a Pushdown Automaton?	35
6.2	How does a PDA work?	36
6.3	The language of a PDA	37
6.4	Deterministic PDAs	38
6.5	Context free grammars and push-down-automata	39
7	How to implement a recursive descent parser	41
7.1	What is a LL(1) grammar ?	41
7.2	How to calculate First and Follow	41
7.3	Constructing an LL(1) grammar	42
7.4	How to implement the parser	43
7.5	Beyond LL(1) - use LR(1) generators	46
8	Turing machines and the rest	47
8.1	What is a Turing machine?	47
8.2	Grammars and context-sensitivity	50
8.3	The halting problem	51
8.4	Back to Chomsky	52

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 online version of this text contains some hyperlinks to webpages which 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?

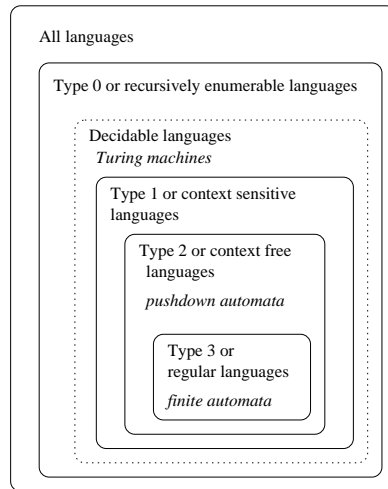
1. Mathematical models of computation, such as:
 - Finite automata,
 - Pushdown automata,
 - Turing machines
2. How to specify formal languages?
 - Regular expressions
 - Context free grammars
 - Context sensitive grammars
3. The relation between 1. and 2.

1.3 Applications

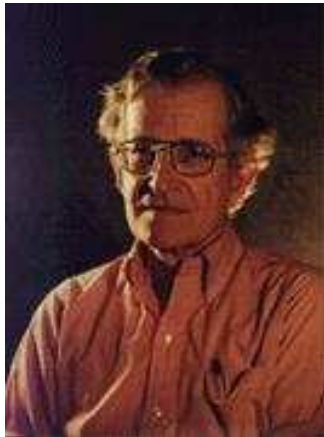
- Regular expressions
Regular expressions are a convenient ways to express patterns (i.e. for search). There are a number of tools which use regular expressions:
 - `grep` pattern matching program (UNIX)
 - `sed` stream editor (UNIX)
 - `lex` A generator for lexical analyzers (UNIX)
- Grammars for programming languages.
The [appendix of \[GJSB00\]](#) contains a *context free grammar* specifying the syntax of JAVA.
`YACC` is a tool which 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* which classifies grammars and languages. This hierarchy can be amended by different types of machines (or automata) which 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 which 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 which 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 0ϵ . It is also important to realize that although there are infinite 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 $L \in \mathcal{P}(\Sigma)$. 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 which contain the same number of 0s and 1s is a language over $\Sigma = \{0, 1\}$.
- The set of words which 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 palindroms using the english alphabet, e.g. words which read the same forwards and backwards like **abba**. This is a language over $\{\mathbf{a}, \mathbf{b}, \dots, \mathbf{z}\}$.
- The set of correct Java programs. This is a language over the set of UNICODE characters (which correspond to numbers between 0 and $2^{16} - 1$).
- The set of programs, which if executed on a Windows machine, will print 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. 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 which are accepted by DFAs and NFAs is the same.

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 $\delta \in Q \times \Sigma \rightarrow Q$,
4. An initial state $q_0 \in Q$,
5. A set of accepting states $F \subseteq Q$.

As an example consider the following automaton

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

where

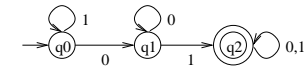
$$\delta_D = \{((q_0, 0), q_1), ((q_0, 1), q_0), ((q_1, 0), q_1), ((q_1, 1), q_2), ((q_2, 0), q_2), ((q_2, 1), q_2)\}$$

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

	0	1
$\rightarrow q_0$	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 \rightarrow 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' which is labelled x .

We write Σ^* for the set of words (i.e. sequences) over the alphabet Σ . This includes the empty word which 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 $0 \notin L(D), 101 \in L(D)$ and $110 \notin L(D)$. Indeed, we have

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

To be more precise we give a formal definition of $L(A)$. First we define the extended transition function $\hat{\delta} \in Q \times \Sigma^* \rightarrow Q$. 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 $\hat{\delta}_D(q_0, 101) = q_1$:

$$\begin{aligned} \hat{\delta}_D(q_0, 101) &= \hat{\delta}_D(\delta_D(q_0, 1), 01) && \text{by (2)} \\ &= \hat{\delta}_D(q_0, 01) && \text{because } \delta_D(q_0, 1) = q_0 \\ &= \hat{\delta}_D(\delta_D(q_0, 0), 1) && \text{by (2)} \\ &= \hat{\delta}_D(q_1, 1) && \text{because } \delta_D(q_0, 0) = q_1 \\ &= \hat{\delta}_D(\delta_D(q_1, 1), \epsilon) && \text{by (2)} \\ &= \hat{\delta}_D(q_2, \epsilon) && \text{because } \delta_D(q_1, 1) = q_2 \\ &= q_2 && \text{by (1)} \end{aligned}$$

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

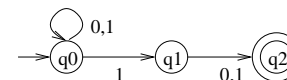
$$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$$

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

2.2 Nondeterministic Finite Automata

2.2.1 What is an NFA?

Nondeterministic finite automata (NFA) have transition functions which 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-deterministic 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 which accepts the same language. Here is an example of an NFA C which accepts all words over $\Sigma = \{0, 1\}$ s.t. the symbol before the last is 1.



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

1. A finite set of states Q ,
2. A finite set of input symbols Σ ,
3. A transition function $\delta \in Q \times \Sigma \rightarrow \mathcal{P}(Q)$,
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

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

where δ_C so given by

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

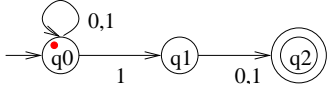
Note that we diverge slightly from the definition in the book, which uses a single initial state instead of a set of initial states. Doing so means that we can avoid introducing ϵ -NFAs (see [HMU01], section 2.5).

2.2.2 The language accepted by an NFA

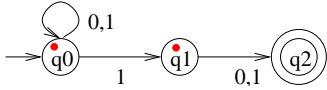
To see whether a word $w \in \Sigma^*$ is accepted by an NFA ($w \in L(A)$) we may have to use several markers. Initially we put one marker on the initial state. Then each time when we read a symbol we look at all the markers: we remove the old marker and put markers at all the states which are reachable via an arrow marked with the current input symbol (this may include the state which 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.

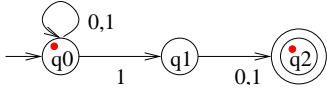
E.g. consider the word 100 (which is not accepted by C). Initially we have



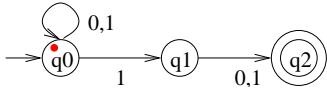
After reading 1 we have to use two markers because there are two arrows from q_0 which are labelled 1:



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:

$$\bigcup\{A_1, A_2, \dots, A_n\} = A_1 \cup A_2 \cup \dots \cup A_n$$

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

$$\bigcup\{\} = \{\} \quad \bigcup\{A\} = A$$

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

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

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

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

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

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

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

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

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

2.2.3 The subset construction

DFAs can be viewed as a special case of NFAs, i.e. those for which 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 which accepts the same language. This shows that NFAs aren't more powerful as 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

$$\begin{aligned} \delta_{D(A)}(S, x) &= \bigcup\{\delta(q, x) \mid q \in S\} \\ F_{D(A)} &= \{S \subseteq Q_N \mid S \cap F \neq \{\}\} \end{aligned}$$

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

which contains at least a final state of the NFA. The transitions just follow the active set of markers, i.e. a state $S \in \mathcal{P}(Q_N)$ corresponds to having markers on all $q \in S$ and when we follow the arrow labelled x we get the set of states which are marked after reading x .

As an example let us consider the NFA C above. We construct a DFA $D(C)$

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

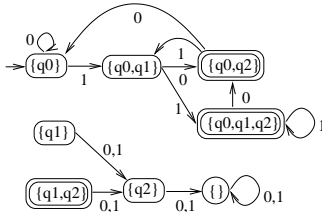
with $\delta_{D(C)}$ given by

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

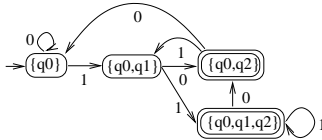
and $F_{D(C)}$ is the set of all the states marked with $*$ above, i.e.

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

Looking at the transition diagram:



we note that some of the states ($\{\}$, $\{q_1\}$, $\{q_2\}$, $\{q_1, q_2\}$) 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

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

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

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

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

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

□

We can now use the lemma to show

Theorem 2.2

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

Proof:

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

□

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 which 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 which match the regular expression, where matching means that the line contains a substring which 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 expression) 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 which 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?

We assume as given an alphabet Σ (e.g. $\Sigma = \{a, b, c, \dots, z\}$) and define the *syntax* of regular expressions (over Σ)

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 expression. We use boldface to emphasize the difference between the symbol a and the regular expression \mathbf{a} .
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
- hallo
- hallo + hallo

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

As in arithmetic they are some conventions how to read regular expressions:

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

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 which 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 $L = \{a, ab\} \subseteq \{a, b\}^*$:

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

You should notice that we use the same symbol as in Σ^* but there is a subtle difference: Σ is a set of symbols but $L \subseteq \Sigma^*$ is a set of words.

Alternatively (and more abstractly) one may describe L^* as the least language (wrt \subseteq) which contains L and the empty word and is closed under concatenation:

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

We now define the *semantics* of regular expressions: 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 may be used as a regular expression (as in $L(\emptyset)$) or the empty set ($\emptyset = \{\}$). Similarly, ϵ in 2. may be a regular expression or a word, in 6. $*$ may be used to construct regular expressions or it is an operation on languages. Which alternative we mean becomes only clear from the context, there is no generally agreed mathematical notation ¹ 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 languages they define:

ϵ

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

By 2.

hallo

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

$$\begin{aligned} L(\text{h}) &= \{\text{h}\} \\ L(\text{a}) &= \{\text{a}\} \end{aligned}$$

Hence by 5:

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

Continuing the same reasoning we obtain:

$$L(\text{hallo}) = \{\text{hallo}\}$$

hallo + hello

From the previous point we know that:

$$\begin{aligned} L(\text{hallo}) &= \{\text{hallo}\} \\ L(\text{hello}) &= \{\text{hello}\} \end{aligned}$$

Hence by using 4 we get:

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

h(a + e)llo

Using 3 and 4 we know

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

Hence using 5 we obtain:

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

¹This is different in programming, e.g. in JAVA we use "... " to signal that we mean things literally.

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

Let us introduce the following notation:

$$w^i = \underbrace{ww \dots w}_i \text{ times}$$

Now using 6 we know that

$$\begin{aligned} L(\mathbf{a^*}) &= \{w_0w_1 \dots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n. w_i \in L(\mathbf{a})\} \\ &= \{w_0w_1 \dots w_{n-1} \mid n \in \mathbb{N} \wedge \forall i < n. w_i \in \{\mathbf{a}\}\} \\ &= \{\mathbf{a}^n \mid n \in \mathbb{N}\} \end{aligned}$$

and hence using 5 we conclude

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

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

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

Let's analyze the parts:

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

Hence, we have

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

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

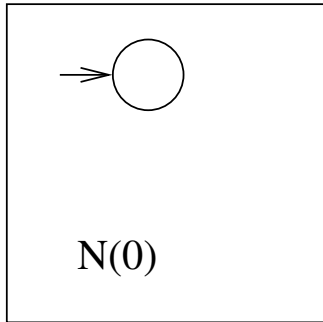
3.3 Translating regular expressions to NFAs

Theorem 3.1 For each regular expression E we can construct an 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:

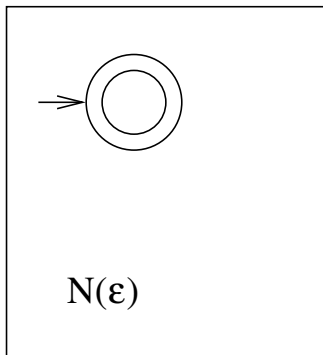
1. $N(\emptyset)$:



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

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

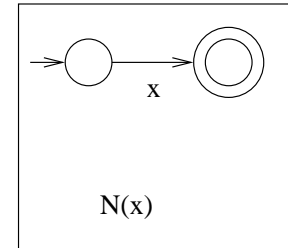
2. $N(\epsilon)$:



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

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

3. $N(x)$:

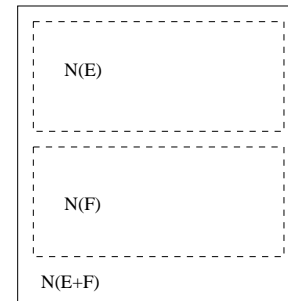


This automaton only accepts the word x , hence:

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

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 accidentally 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

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

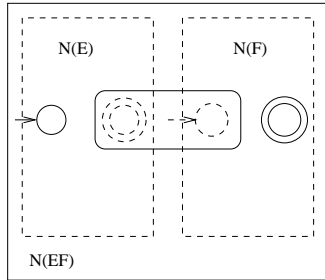
that's what is meant by *induction over the syntax of regular expressions*.

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 $N(F)$:

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

- 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 $N(E)$ which has a transition to a final state of $N(E)$ we add a transition with the same label to all the initial states of $N(F)$.

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

- 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 $N(E)$ which 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

$$N(EF) = (Q_{EF}, \Sigma, \delta_{EF}, S_{EF}, F_{EF})$$

I hope that you are able to convince yourself that

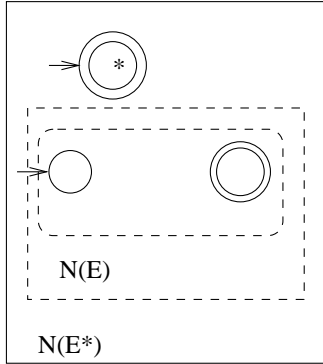
$$L(N(EF)) = \{uv \mid u \in L(N(E)) \wedge v \in L(N(F))\}$$

and hence we can reason

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

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 $*$ which 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 from N_E and for each state which has an arrow to the final state labelled x we also add an arrow to all the initial states labelled x .

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

- 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

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

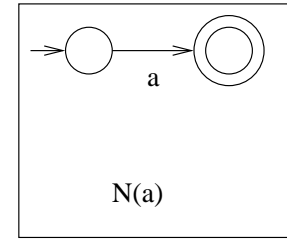
since we can run through the automaton an arbitrary number of times. The new state $*$ allows us also to accept the empty sequence. Hence:

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

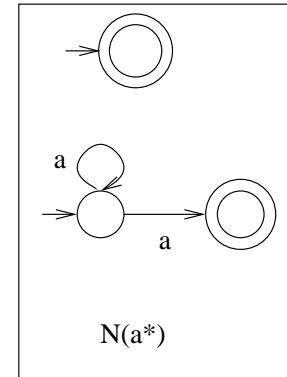
7. $N((E)) = N(E)$

I.e. using brackets does not change anything. □

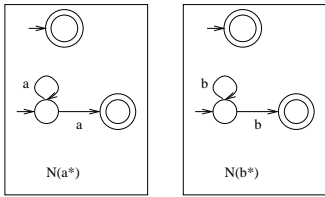
As an example we construct $N(a^*b^*)$. First we construct $N(a)$:



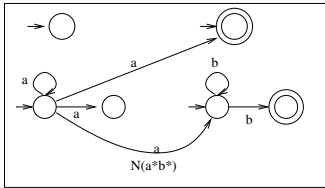
Now we have to apply the $*$ -construction and we obtain:



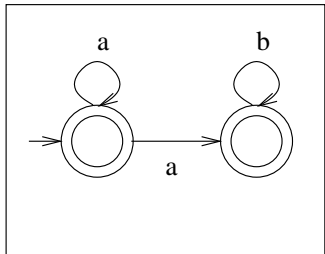
$N(b^*)$ is just the same and we get



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 $R(A)$ which 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 accepted by a DFA.

Proof: We have that 1. \implies 2 by theorem 3.1. We know that 2. \implies 3. by 2.2 and 3. \implies 1. by 3.2. \square

As indicated in the introduction: the languages which 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 which can be recognized by a computer with finite (i.e. fixed) memory. Such a computer corresponds to a DFA. However, there are many languages which 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 which start with a number of 0s followed by the **same** number of 1s. Note that this is different to $L(0^*1^*)$ which is the language of words of sequences of 0s followed by a sequence of 1s but the number has not to be identical (and which we know 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 $32 \cdot 1024 \cdot 1024 \cdot 8 = 268435456$ 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 .

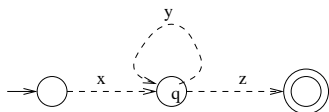
We shall now show a general theorem called *the pumping lemma* which allows us to prove that a certain language is not regular.

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 $w \in L$ which 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 $xy^kz \in L$.

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 $xy^i z$ 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 $xy^i z \in L$

□

4.2 Applying the pumping lemma

Theorem 4.2 *The language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ 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 which is longer than n such that the properties given by the pumping lemma hold. Consider $0^n 1^n \in L$, this is certainly longer than n . We have that $xyz = 0^n 1^n$ 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 $xy^0 z \in L$ but this cannot be the case because it contains at least one 0 less but the same number of 1s as $0^n 1^n$.

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 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 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

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

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. \square

Given a word $w \in \Sigma^*$ we write w^R for the word read backwards. I.e. $\text{abc}^R = \text{bca}$. 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 $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ we have $\text{abba} \in L_{\text{pali}}$. 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 $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ we have that L_{pali} is not regular.*

Proof: We use the pumping lemma: We assume that L_{pali} is regular. Now given a pumping number n we construct $w = \mathbf{a}^n \mathbf{bba}^n \in L_{\text{pali}}$, 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 0s and since $y \neq \epsilon$ at least one. We conclude that $xz \in L_{\text{pali}}$ where $xz = \mathbf{a}^m \mathbf{bba}^n$ where $m < n$. However, this word cannot be a palindrome since only the first half contains any a's.

Hence our assumption L_{pali} is regular must be wrong. \square

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 $A \rightarrow \alpha$.

As an example we define a grammar for the language of arithmetical 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:

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

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

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

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 $V \rightarrow \alpha$ 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 :

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

Note that \Rightarrow_G here stands for the relation *derives in one step* and has nothing to do with implication. In the example we have always replaced the leftmost non-terminal symbol (hence it is called a leftmost derivation) but this is not necessary.

Given any grammar $G = (V, \Sigma, S, P)$ we define the relation *derives in one step*

$$\begin{aligned}
&\Rightarrow_G \subseteq (V \cup T)^* \times (V \cup T)^* \\
\alpha V \gamma \Rightarrow_G \alpha \beta \gamma &: \iff V \rightarrow \beta \in P
\end{aligned}$$

The relation *derives* is defined as²

$$\begin{aligned}
&\Rightarrow_G^* \subseteq (V \cup T)^* \times (V \cup T)^* \\
\alpha_0 \Rightarrow_G^* \alpha_n &: \iff \alpha \Rightarrow_G \alpha_1 \Rightarrow_G \dots \alpha_{n-1} \Rightarrow_G \alpha_n
\end{aligned}$$

this includes the case $\alpha \Rightarrow_G^* \alpha$ 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) which can be derived in any number of steps, i.e.

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

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

5.3 More examples

Some of the languages which 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

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

Also the language of palindromes

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

² \Rightarrow_G^* is the *transitive-reflexive closure* of \Rightarrow_G .

turns out to be context-free;

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow \epsilon \mid aSa \mid bSb\})$$

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. a^*b^* can be translated into

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

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

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

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

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

which allows us to derive interesting sentences like

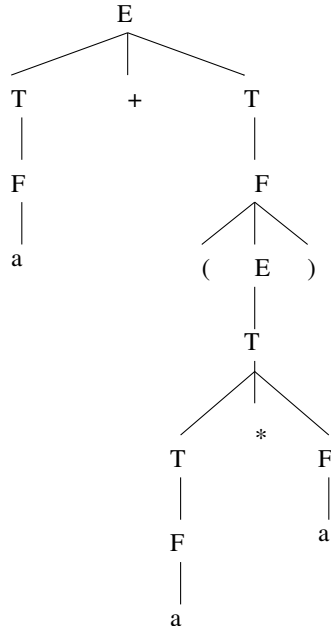
the dog which catches the cat which bites barks

An important example for context-free languages is the syntax of programming languages. We have already mentioned [appendix of \[GJSB00\]](#) which uses a formalism slightly different from the one introduced here. Another example is the toy language used in the compilers course ([Bac02], see [Mini-Triangle Concrete and Abstract Syntax](#)

However, 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.

5.4 Parse trees

With each derivation we also associate a *derivation tree*, which 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 which 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.

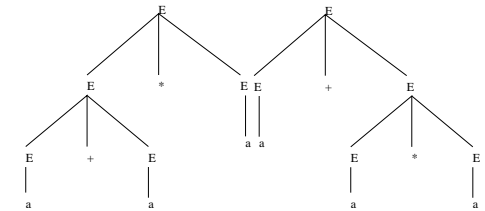
5.5 Ambiguity

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. 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 $G' = (\{E\}, \{(\cdot, a, +, *)\}, E, P')$ where P' is given by:

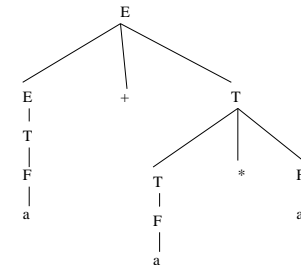
$$P' = \{E \rightarrow E + E \mid E * E \mid a \mid (E)\}$$

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 ambiguous: 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.

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 which 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 which can be recognized by PDA are precisely the context free languages.

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 symbols (the alphabet),
- A finite set Γ of stack symbols,
- A transition function

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

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

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

- 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 which 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:

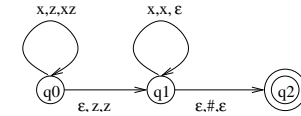
$$\begin{aligned} \delta(q_0, 0, \#) &= \{(q_0, 0\#)\} \\ \delta(q_0, 1, \#) &= \{(q_0, 1\#)\} \\ \delta(q_0, 0, 0) &= \{(q_0, 00)\} \\ \delta(q_0, 1, 0) &= \{(q_0, 10)\} \\ \delta(q_0, 0, 1) &= \{(q_0, 01)\} \\ \delta(q_0, 1, 1) &= \{(q_0, 11)\} \\ \delta(q_0, \epsilon, \#) &= \{(q_1, \#)\} \\ \delta(q_0,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

To save space we may abbreviate this by writing:

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

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 $w \in \Sigma^*$ which 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 $\vdash_P \subseteq \text{ID} \times \text{ID}$ between IDs which 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 \vdash_P :

1. $(q, xw, z\gamma) \vdash_P (q', w, \alpha\gamma)$ if $(q', \alpha) \in \delta(q, x, z)$
2. $(q, w, z\gamma) \vdash_P (q', w, \alpha\gamma)$ 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 which replace the current 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 $(q_0, 0110, \#)$?

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

We write $(q, w, \gamma) \vdash_P^* (q', w', \gamma)$ 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

$$(q_0, 0110, \#) \vdash_{P_0}^* (q_2, \epsilon, \epsilon).$$

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

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

We have shown $(q_0, 0110, \#) \vdash_{P_0}^* (q_1, 110, 0\#)$. Here the PDA gets stuck there is no state after $(q_1, 110, 0\#)$.

If we start with a word which 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

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

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 $(q_0, 0110, \#) \vdash_{P_0}^* (q_2, \epsilon, \epsilon)$ 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

$$L(P) = \{w \mid (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon)\}$$

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 which use one acceptance method into one which uses the other. Hence, both acceptance criteria specify the same class of languages.

6.4 Deterministic PDAs

We have introduced PDAs as nondeterministic machines which may have several alternatives how to continue. We now define Deterministic Pushdown Automata (DPDA) as those which 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 which recognizes the language L ! **Nondeterministic PDAs are more powerful than deterministic PDAs.**

However, we can define a similar language L' over $\Sigma = \{0, 1, \$\}$ which can be recognized by a deterministic PDA:

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

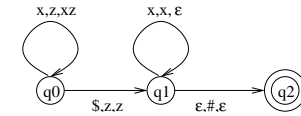
That is L' contains palindroms with a marker $\$$ in the middle, e.g. $01\$10 \in L'$. We define a DPDA P_1 for L' :

$$P_1 = (\{q_0, q_1, q_2\}, \{0, 1, \$\}, \{0, 1, \#\}, \delta', q_0, \#, \{q_2\})$$

where δ is given by:

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

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 languages. 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 is 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:

$$\delta(q_0, \epsilon, A) = \{(q_0, \alpha) \mid A \rightarrow \alpha \in P\}$$

for all $A \in V$.

$$\delta(q_0, a, a) = \{(q_0, \epsilon)\}$$

for all $a \in \Sigma$.

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

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

we define

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

with

$$\begin{aligned} \delta(q_0, \epsilon, E) &= \{(q_0, T), (q_0, E + T)\} \\ \delta(q_0, \epsilon, T) &= \{(q_0, F), (q_0, T * F)\} \\ \delta(q_0, \epsilon, F) &= \{(q_0, a), (q_0, (E))\} \\ \delta(q_0, (\cdot, \cdot)) &= \{(q_0, \epsilon)\} \\ \delta(q_0, a, a) &= \{(q_0, \epsilon)\} \\ \delta(q_0, +, +) &= \{(q_0, \epsilon)\} \\ \delta(q_0, *, *) &= \{(q_0, \epsilon)\} \end{aligned}$$

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

$$\begin{aligned} (q_0, a + (a*a), E) &\vdash (q_0, a + (a*a), E+T) \\ &\vdash (q_0, a + (a*a), T+T) \\ &\vdash (q_0, a + (a*a), F+T) \\ &\vdash (q_0, a + (a*a), a+T) \\ &\vdash (q_0, + (a*a), +T) \\ &\vdash (q_0, (a*a), T) \\ &\vdash (q_0, (a*a), F) \\ &\vdash (q_0, (a*a), (E)) \\ &\vdash (q_0, a*a, E) \\ &\vdash (q_0, a*a, T) \\ &\vdash (q_0, a*a, T * F) \\ &\vdash (q_0, a*a, F * F) \\ &\vdash (q_0, a*a, a * F) \\ &\vdash (q_0, *a, *F) \\ &\vdash (q_0, a), F) \\ &\vdash (q_0, a), a) \\ &\vdash (q_0, \cdot, \cdot) \\ &\vdash (q_0, \epsilon, \epsilon) \end{aligned}$$

Hence $a + (a*a) \in L(P(G))$.

This example hopefully already illustrates the general idea:

$$\begin{aligned} w &\in L(G) \\ &\iff \\ S &\Rightarrow \dots \Rightarrow w \\ &\iff \\ (q_0, w, S) &\vdash \dots \vdash (q_0, \epsilon, \epsilon) \\ &\iff w \in L(P(G)) \end{aligned}$$

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 alternative.

7 How to implement a recursive descent parser

A parser is a program which 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 which 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 which 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. $\$ \notin V \cup \Sigma$,
- $P' = P \cup \{S' \rightarrow S\$\}$

The idea is that

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

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

$$\begin{aligned} \text{First}(A) &= \{a \mid a \in \Sigma \wedge A \Rightarrow^* a\beta\} \\ \text{Follow}(A) &= \{a \mid a \in \Sigma \wedge S' \Rightarrow^* \alpha A a \beta\} \end{aligned}$$

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

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

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

7.2 How to calculate First and Follow

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

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

We calculate First and Follow in a similar fashion:

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

And for Follow:

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

7.3 Constructing an LL(1) grammar

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

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

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

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

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

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

Hence the grammar G is **not LL(1)**.

However, luckily there is an alternative grammar G' which defines the same language: $G' = (\{E, E', T, T', F\}, \{(\cdot, a, +, *)\}, E, P')$ where

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

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

```

First(E) = First(T) = First(F) = {a, (}
      First(E') = {+}
      First(T') = {*}
Follow(E) = Follow(E') = {), $}
Follow(T) = Follow(T') = {+, ), $}
Follow(F) = {+, *, ), $}

```

Now we calculate the Lookahead-sets:

```

Lookahead(E → TE') = {a, (}
Lookahead(E' → +TE') = {+}
  Lookahead(E' → ε) = Follow(E') = {), $}
Lookahead(T → +FT') = {a, (}
Lookahead(T' → *FT') = {*}
  Lookahead(T' → ε) = Follow(T') = {+, ), $}
  Lookahead(F → a) = {a}
  Lookahead(F → (E)) = {(}

```

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* which 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` which 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 LL(1) - use LR(1) 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 from hand. Hence there are automated tools which get the grammar as an input and which 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].

8 Turing machines and the rest

A *Turing machine* (TM) is a generalization of a PDA which 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 which 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 which 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 which will always stop. There are *type-0-languages* which are not recursive — the most famous one is the *halting problem*. This is the language of encodings of Turing machines which will always stop.

There is no type of grammars which captures all recursive languages (and for theoretical reasons there cannot be one). However there is a subset of recursive languages which are called *context-sensitive languages* which are given by *context-sensitive grammars*, these are those 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 which 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

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

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 $\delta \in Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$. 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

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

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 \vdash_M similar as for PDAs:

1. $(\gamma_l, q, x\gamma_r) \vdash_M (\gamma_l y, q', \gamma_r)$ if $\delta(q, x) = (q', y, \text{R})$
2. $(\gamma_l z, q, x\gamma_r) \vdash_M (\gamma_l, q', z y \gamma_r)$ if $\delta(q, x) = (q', y, \text{L})$
3. $(\gamma_l, q, \epsilon) \vdash_M (\gamma_l y, q', \gamma_r)$ if $\delta(q, B) = (q', y, \text{R})$
4. $(\epsilon, q, x\gamma_r) \vdash_M (\gamma_l, q', B y \gamma_r)$ if $\delta(q, x) = (q', y, \text{L})$

The cases 3. and 4. are only needed to deal with the situation if we have 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

$$L(M) = \{w \in \Sigma^* \mid (\epsilon, q_0, w) \vdash_M^* (\gamma_l, q', \gamma_r) \wedge q' \in F\}$$

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 which accepts the language $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ — this is a language which cannot be recognized by a PDA or be defined by a CFG.

We define $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ by

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $\Sigma = \{a, b, c\}$

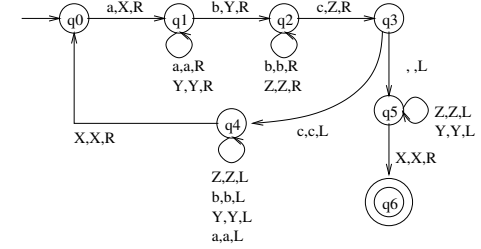
- $\Gamma = \Sigma \cup \{X, Y, Z, \sqcup\}$

- δ is given by

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

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

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 **cs** left it moves left to the next **a** (q_4) and repeats the cycle. Otherwise it checks whether there are no **as** and **bs** 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**:

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

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 \Rightarrow_G (and \Rightarrow_G^*) is defined as before

$$\begin{aligned} \Rightarrow_G &\subseteq (V \cup T)^* \times (V \cup T)^* \\ \alpha\beta\gamma \Rightarrow_G \alpha'\beta'\gamma' &: \iff \beta \rightarrow \beta' \in P \end{aligned}$$

and as before the language of G is defined as

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

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 $\alpha \rightarrow \beta \in P$ 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

- $V = \{S, B, C\}$
- $\Sigma = \{a, b, c\}$
-

$$\begin{aligned} P = \{ & S \rightarrow aSBC \\ & S \rightarrow aBC \\ & aB \rightarrow ab \\ & CB \rightarrow BC \\ & bB \rightarrow bb \\ & bC \rightarrow bc \\ & cC \rightarrow cc \} \end{aligned}$$

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 which are accepted by a TM (i.e. type 0 languages) but which 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 $[M] \in \{0, 1\}^*$ 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

$$L_{\text{halt}} = \{[M]w \mid M \text{ holds on input } w.\}$$

It is easy (although the details are quite daunting) to define a TM which accepts this language: we just simulate M and accept if M stops.

However, Turing showed that there is no TM which decides this language. To see this let us assume that there is a TM H which decides L . Now using H we construct a new TM F which 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.

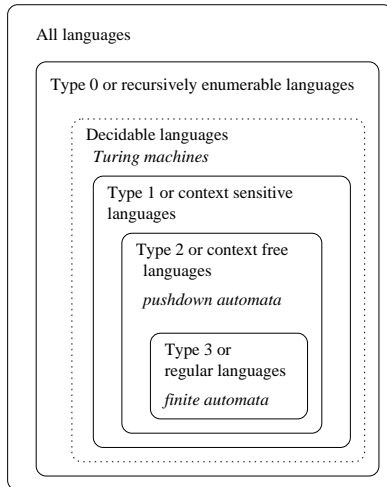
The question is what happens if I run F on $[F]$? Let us assume it terminates, then H applied to $[F][F]$ returns yes and hence we must conclude that F on $[F]$ loops???. On the other hand if F with input $[F]$ loops then H applied to $[F][F]$ will stop and reject and hence we have to conclude that F on $[F]$ will stop?????

This is a contradiction and hence we must conclude that our assumption that there is a TM H which decides L_{halt} is false. We say L_{halt} is undecidable.

We have shown that a Turing machine cannot decide whether a program (for a Turing machine) halts. Maybe we could find a more powerful programming language which overcomes this problem? It turns out that all computational formalisms (i.e. programming languages) which 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 subclasses of grammars or equivalently by different types of automata which 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). Correspondingly we have introduced different grammatical formalisms: regular expressions, context-free grammars and grammars.

Note that at each level there are languages which 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 which only use their tapes as a stack, and Turing machines which never use the tape apart for reading the input. Again correspondingly we can define restrictions on the grammar side: first introduce context-free grammars and then grammars where all productions are of the form $A \rightarrow aB$ or $A \rightarrow a$, 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 hierarchy 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. [www](#), 2001. 21
- [Bac02] Roland Backhouse. Compilers (g52cmp) — lecture notes. [www](#), 2002. 32
- [GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2nd edition edition, 2000. 4, 32
- [HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition edition, 2001. 3, 10, 25, 39, 48
- [Hud99] Scott E. Hudson. Cup parser generator for java. [www](#), 1999. 46
- [Mar01] Simon Marlow. Happy - the parser generator for haskell. [www](#), 2001. 46
- [Sch99] Uwe Schöning. *Theoretische Informatik — kurzgefaßt*. Spektrum - Akademischer Verlag, 3. Auflage edition, 1999. 3