University of The Gambia

Computer Sciences Department

CPS321: Theory of Computing I

J. Boillat

Keywords: Automata, Regular Expressions, Context-Free Grammars, Complexity, Turing machines, Parsing, Recursion, Numbers

Abstract

An introduction to the modern theory of computing. Topics include automata theory, formal languages, Turing machines, an introduction to lexical analysis and an introduction to parsing.

Preface

Automata's and Languages

A *language*[1] is a set of *words*, i.e. finite *strings* of *letters*, or *symbols*. The inventory from which these letters are taken is called the *alphabet* over which the language is defined. A language is often defined by means of a *grammar*.

Formal language theory rarely concerns itself with particular languages (except as examples), but is mainly concerned with the study of various types of formalisms to describe languages. For instance, a language can be given as

- those strings generated by some *formal grammar*.
- those strings described or matched by a particular *regular expression*.
- those strings accepted by some *automaton*, such as a *Turing machine*, a *deterministic finite automaton*, *non deterministic finite automaton* or a *push down automaton*.

Typical questions asked about such formalisms include:

- What is their *expressive power*? Can formalism X describe every language that formalism Y can describe? Can it describe other languages?
- What is their *recognisability*? How difficult is it to decide whether a given word belongs to a language described by formalism X?
- What is their *comparability*? How difficult is it to decide whether two languages, one described in formalism X and one in formalism Y, or in X again, are actually the same language?

Surprisingly often, the answer to these decision problems is *"it cannot be done at all"*, or *"it is extremely expensive"* (with a precise characterisation of how expensive exactly).

Usage of Automata's and Languages

- Formal language theory is a major application area of *computability theory* and *complexity theory*.
- Deterministic finite Automata's (DFA) are one of the most practical models of computation, since there is a trivial linear time, constant space, on line algorithm to simulate a deterministic DFA on a stream of input.

[1] From Wikipedia `http://en.wikipedia.org/wiki/Formal\char95\relaxlanguage`

- Non deterministic finite Automata's (NFA) have a wide usage in the context of regular expressions. Since they can easily be transformed into DFA's, there is also a linear time, but possibly exponential space simulation algorithm.

  Regular expressions are used for the specification of scanners like FLEX and JAVACC, in a lot of UNIX tools like `grep`, `sed`, etc. They are also part of XML-SHEMA and of some scripting languages like PERL, PYTHON, etc.

- Formal grammars are used for the specification of programming languages. For those grammars there are fast simulation algorithms, thus allowing to develop fast compilers. Usually, compilers are implemented using parser generators like BISON, ANTLR, JAVACC, etc.

Tools

Many exercises are designed to be completed using existing software tools such as programming languages like JAVA or simulation and computation packages like JFLAP or program generation software like JAVACC.

JFLAP

JFLAP[Rm] is software for experimenting with formal languages topics including non deterministic finite Automata's, non deterministic push down Automata's, multi-tape Turing machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar.

JAVACC

JAVACC (JAVA Compiler Compiler) [JCC96] is an open source parser generator for the JAVA programming language. JAVACC is similar to Yacc in that it generates a parser for a formal grammar provided in EBNF notation, except the output is Java source code. Unlike YACC, however, JAVACC generates top-down parsers, which limits it to the LL(k) class of grammars (in particular, left recursion cannot be used). The tree builder that accompanies it, JJTREE, constructs its trees from the bottom up.

Lecture Notes

Sources

These lecture notes are principally based on the book *Introduction to Automata Theory, Languages and Computation*, written by J. E. Hopcroft, R. Motwani and J. D. Ullman [HMU07].

We make an extensive use of the lecture notes *Formal Languages and Automata Theory* of P. Flener [Fle06].

Some introductory texts are adapted from *Wikipedia* [Wik].

Writing

These lecture notes have been edited and published using open source software only:

LaTeX[2] has been used for composition. The original figures are all in SVG[3] format.

Inkscape[4] was a very useful tool for figures post processing and integration into LaTeX (conversion to EPS figures).

The automata's have been created using JFLAP. The generated JFLAP files have been transformed to SVG using an XSL script written by the author. Some other figures have been edited using Xfig[5].

The LaTeX source of the lecture notes has been transformed automatically to XML (thus allowing validation) using XTL[6].

The intermediate XML document has been finally transformed to XHTML + MathML + SVG using an XSL transformation.

[2] LaTeX - A document preparation system [http://www.latex-project.org/]

[3] SVG Scalable Vector Graphics, XML Graphics for the Web [http://www.w3.org/Graphics/SVG/]

[4] Inkscape is an open source vector graphics editor, with capabilities similar to Illustrator, CorelDraw, or Xara X, using the W3C standard Scalable Vector Graphics (SVG) file format [http://inkscape.org]

[5] Xfig is an open source vector graphics editor which runs under the X Window System on most UNIX-compatible platforms [http://www.xfig.org/]

[6] XTL is a language for performing translations from any structured text based document to XML. The language elements are very similar to XSL [https://staff.hti.bfh.ch/blj2/xtl/]

Chapter 1

Automata

1.1 Alphabets and Languages

1.1.1 Alphabets

Definition 1.1 [Alphabet] *An* alphabet Σ *is a finite, nonempty set of symbols*

Example 1.1 [Binary Alphabet] $\Sigma = \{0, 1\}$ the binary alphabet

Example 1.2 [Lower Case Letters] $\Sigma = \{a, b, c, \ldots, z\}$ the set of all lower case letters

1.1.2 Strings

Strings can also be identified with words.

Definition 1.2 [String] *A* string *is a finite sequence of symbols from an alphabet* Σ

Definition 1.3 [Empty String] *The* empty string ϵ *is the string with zero occurrences of symbols from* Σ

Definition 1.4 [Length] *The* length *of a string is the number of positions for symbols in the string.* $|w|$ *denotes the length of the string* w*.*

Definition 1.5 [Powers] *The* powers Σ^k *of an alphabet* Σ *is the set of strings of length* k *with symbols from* Σ*.*

Example 1.3 [Powers] Let $\Sigma = \{0, 1\}$, then

$\Sigma^0 = \{\epsilon\}$

$\Sigma^1 = \{0, 1\}$

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

The set of all strings over an alphabet Σ is usually denoted by Σ^*

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$$

1.1.3 Languages

Following definition of a language is somehow strange. Applied to written languages like English, this would identify a language with a dictionary. In chapter 4 we will see another grammar based definition of languages.

Definition 1.6 [Language] *A* language *L over an alphabet Σ is a set of strings over Σ.*

1.1.4 Operations on Languages

In this section, L, M and $\overline{L}$ are languages over alphabet Σ.

Definition 1.7 [Union]

$$L \cup M = \{w : w \in L \quad or \quad w \in M\}$$

Definition 1.8 [Intersection]

$$L \cap M = \{w : w \in L \quad and \quad w \in M\}$$

Definition 1.9 [Complement]

$$\overline{L} = \Sigma^* - L$$

Definition 1.10 [Concatenation]

$$L \cdot M = \{w : w = xy, x \in L, y \in M\}$$

Definition 1.11 [Powers]

$$L^0 = \{\epsilon\}, L^1 = L, L^{k+1} = L.L^k$$

Definition 1.12 [Kleene[1] Closure]

$$L^* = \bigcup_{i \in \mathbb{N}} L^i$$

Exercise 1.1 [Language] *Let $\Sigma = \{a, b, c, d\}$ be an alphabet and*

$$L_1 = \{\epsilon, ab, bab, bc\}, \qquad L_2 = \{bb, aba, cd\}$$

What are the languages $L_1 \cdot L_2$ and $L_2 \cdot L_1$? What can you conclude?

Exercise 1.2 [Language] *Let $\Sigma = \{a, b, c, \}$ be an alphabet and*

$$L_1 = \{\epsilon, aa, ab, ba, bb\}, \qquad L_2 = \{cb, cb, b, c\}$$

What are the languages L_1^+, L_1^*, L_2^+, L_2^* and $\overline{L_2}$?

[1] Stephen Cole Kleene (1909-1994) was an American mathematician who helped lay the foundations for theoretical computer science. One of many distinguished students of Alonzo Church, Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory. Kleene's work grounds the study of which functions are computable. A number of mathematical concepts are named after him: Kleene hierarchy, Kleene algebra, the Kleene star (Kleene closure), Kleene's recursion theorem and the Kleene fixpoint theorem. He also invented regular expressions, and was a leading American advocate of mathematical intuitionism.

1.2 Deterministic Finite Automata [DFA]

An automaton is a mathematical model for a finite state machine (FSM). A FSM is a machine that, given an input of symbols, transitions, through a series of states according to a transition function (which can be expressed as a table). This transition function tells the automaton which state to go to next given a current state and a current symbol.

The input is read symbol by symbol, until it is consumed completely (similar to a tape with a word written on it, which is read by a reading head of the automaton; the head moves forward over the tape, reading one symbol at a time). Once the input is depleted, the automaton is said to have stopped.

Depending on the state in which the automaton stops, it's said that the automaton either accepts or rejects the input. If it landed in an accepting state, then the automaton accepts the word. If, on the other hand, it lands on a non accepting state, the word is rejected. The set of all the words accepted by an automaton is called the language accepted by the automaton.

Definition 1.13 [DFA] *A* Deterministic Finite Automaton [DFA] *is a quintuple*

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q *is a finite set of states*
- Σ *is a finite alphabet (input symbols)*
- $\delta : Q \times \Sigma \to Q$ *is a transition function*
- $q_0 \in Q$ *is the* start state
- $F \subset Q$ *is a set of* final states

In practise, the function δ may be a partial function, i.e. a function that is not defined for all arguments. This makes the transition diagram more readable.

1.2.1 Language of a DFA

An DFA accepts a string $w = a_1 a_2 \cdots a_n$ an if there is a path in the transition diagram that

1. Begins at a start state
2. Ends at an accepting state
3. Has sequence of labels $a_1 a_2 \cdots a_n$

Let q the state of the DFA after processing $a_1 a_2 \cdots a_{i-1}$. If $\delta(q, a_i)$ is not defined, the DFA stops, i.e. it does not accept.

Definition 1.14 [Extended Transition Function] *The transition function δ can be extended to $\hat{\delta}$ that operates on states and strings (as opposed to states and symbols).*

Basis: $\hat{\delta}(q, \epsilon) = q$

Induction: $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

Definition 1.15 [DFA Language] *The* language *accepted by A is*

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

The languages accepted by DFA's are called *regular languages*.

DFA are usually represented using a *transition table* or a *state diagram*.

A transition table is a table showing what state a DFA will move to, based on the current state and the input symbol.

Transition tables are typically two-dimensional tables. The vertical dimension indicates input symbol, the horizontal dimension indicates current states, and the cells in the table contain the next state for a given input symbol. The start state is marked with an arrow ($\rightarrow$) and the accepting states are marked with stars ($*$).

A state diagram for a DFA is a directed graph with the following elements:

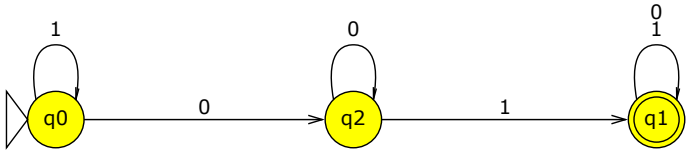
- States Q: a finite set of vertices represented by circles and labelled with unique designator symbols or words written element of Q inside them
- δ: a finite set of directed edges (label-ed with an element $a \in \Sigma$) from the actual state q toward its following state $\delta(q, a)$.
- Start state q_0: The start state is usually represented by an arrow with no origin pointing to the state.
- Accepting state(s) F: It is usually drawn as a double circle.

Example 1.4 [DFA] An automaton A that accepts $L = \{x01y : x, y \in \{0,1\}^*\}$

The automaton $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$ as a transition table:

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

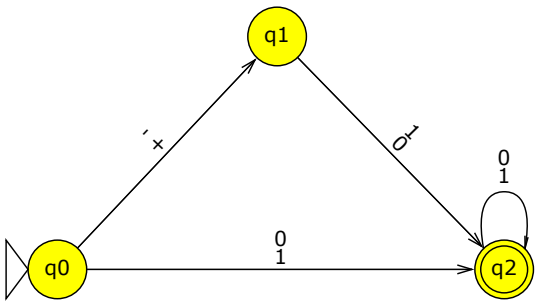
The automaton as a transition diagram:



Example 1.5 [DFA for Integers] An automaton A that accepts $L = \{i : i \in \mathbb{Z}$ with optional sign$\}$. The symbol d represents a digit in binary notation.

Note that δ is partial here.

	+	−	d
$\rightarrow q_0$	q_1	q_1	q_2
q_1			q_2
$*q_2$			q_2



1.2.2 JAVA DFA Implementation

Following Algorithm computes if a string w belongs to the language of some DFA A.

Algorithm 1.1 [DFA-Algorithm]

Let first define two functions.

Function	Description
move(q,a)	$p = \text{move}(q,a) \Leftrightarrow p = \delta(q,a)$
nextChar()	reads the next symbol from the input string w

Let q_0 be the initial state of A

$s = q_0$
c = nextChar()
while ($c \neq$ **eof**)
    s = move(s,c)
    c = nextChar()
endwhile
if ($s \in F$)
    return `true`
else
    return `false`
endif

The transition table of the DFA is implemented with a two dimensional vector. The table must be initialised before starting the simulation. The simulation itself is a loop executing move(q,c) until an accepting state or an error is reached. A special state `-1` (error) has been added. Any non-existing transition is implemented as a transition to state `-1`.

The method `match()` processes the input and returns the final state.

JAVA-Class `DFA`

```
1  public class DFA {
2
3    public int[][] table;
4    public int start;
5    public boolean[] accepting;
6    public final int SE = -1;
7
8    private int nextChar(InputStreamReader reader) {
9      try {
10       return reader.read();
11     } catch (IOException e) {
12       return -1;
13     }
14   }
15
16   private int move(int state, char c) {
17     return table[state][c];
18   }
19
20   public int match(InputStreamReader reader) {
21     int state = start;
22     int i = nextChar(reader);
23     while (i != -1 && state != SE) {
```

```
24          char c = (char) i;
25          state = move(state, c);
26          i = nextChar(reader);
27        }
28        return state;
29      }
30    }
```

Exercise 1.3 [JAVA] *Build a DFA which models* JAVA *identifiers.*

Exercise 1.4 [Binary Strings] *Build a DFA that recognises the set of binary strings having an even number of* 1*'s.*

Exercise 1.5 [Binary Strings] *Build a DFA that recognises the set of binary strings that start with at least three* 0*'s and end with at least two* 1*'s.*

Exercise 1.6 [Binary Strings] *Build a DFA that recognises the set of binary strings that have at least three consecutive* 0*'s.*

Exercise 1.7 [Binary Strings] *Build a DFA that recognises the set of binary strings where the number of* 0*'s is a multiple of* 3*.*

Exercise 1.8 [Exponential] *Build a DFA that recognises the language*

$$L = \{0,1\}^* \cdot \{0\} \cdot \{0,1\} \cdot \{0,1\}$$

Exercise 1.9 [Multiples] *Design a DFA to accept the set of binary strings that, when interpreted as an integer, is divisible by* 5*. Note that the most significant digit is the first to be read. Give the five-tuple notation for your automaton, with the transition function expressed as a table. Then, draw the transition diagram for your DFA.* ***Hint:*** *Think of each state as representing a particular remainder when the number seen so far is divided by* 5*. To simplify, you may make the reasonable assumption that the empty string represents integer* 0*, and thus is one of the accepted strings.*

1.3 Nondeterministic Finite Automata [NFA]

Like a DFA, an NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states. The difference is that the transition function δ of the NFA returns a set of zero or more states.

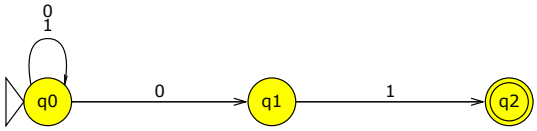
Thus a NFA can be in several states at once, or, viewed another way, it can *guess* which state to go to next.

Definition 1.16 [NFA] *A* NFA *is a quintuple* $A = (Q, \Sigma, \delta, q_0, F)$

- Q *is a finite set of states*
- Σ *is a finite alphabet (input symbols)*
- $\delta : Q \times \Sigma \to P(Q)$ *is a transition function (power set).*
- $q_0 \in Q$ *is the* start state
- $F \subset Q$ *is a set of* final states

Example 1.6 [NFA] The NFA $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$ as a transition table:

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



$L(A)$ is the set of strings over$\Sigma = \{0,1\}$ ending with 01.

Definition 1.17 [NFA Transition Function] *The transition function δ can be extended to $\hat{\delta}$ that operates on states and strings (as opposed to states and symbols).*

Basis: $\hat{\delta}(q,\epsilon) = \{q\}$

Induction: $\hat{\delta}(q,xa) = \bigcup_{p\in\hat{\delta}(q,x)} \delta(p,a)$

1.3.1 Language of an NFA

Definition 1.18 [NFA Language] *The* language *accepted by A is*

$$L(A) = \{w : \hat{\delta}(q_0,w) \cap F \neq \varnothing\}$$

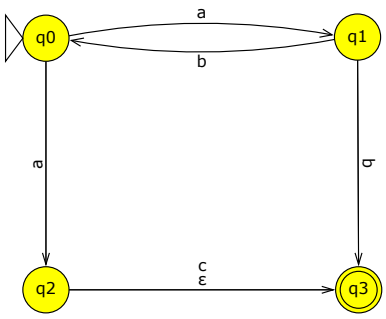
1.3.2 Epsilon-NFA

An extension of the NFA is the ϵ-NFA, which allows a transition to a new state without consuming any input symbols. For example, if it is in state q_1, with the next input symbol an a, it can move to state q_2 without consuming any input symbols provided there is a transition labelled ϵ from q_1 to q_2.

Definition 1.19 [Epsilon-NFA] *An ϵ-NFA is a quintuple (Q,Σ,δ,q_0,F) where δ is a function from $Q\times\Sigma\cup\{\epsilon\}$ to the power set of Q.*

Example 1.7 [Epsilon-NFA] Let $N = (Q,\Sigma,\delta,q_0,F)$ with $\Sigma = \{a,b,c,\epsilon\}$, $Q = \{q_0,q_1,q_2,q_3\}$, $F = \{q_3\}$ and following transition table:

	a	b	c	ϵ
$\rightarrow q_0$	$\{q_1,q_2\}$	$\varnothing$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\{q_0,q_3\}$	$\varnothing$	$\varnothing$
q_2	$\varnothing$	$\varnothing$	$\{q_3\}$	$\{q_3\}$
$*q_3$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$



ECLOSE

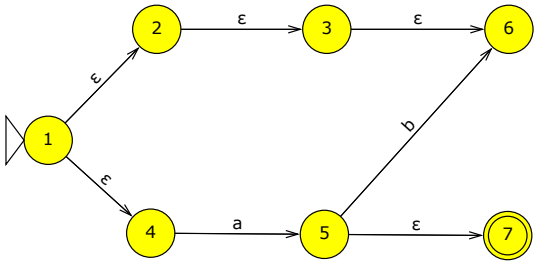
We close a state by adding all states reachable by a sequence $\epsilon\epsilon\epsilon\cdots\epsilon$

Definition 1.20 [ECLOSE] *We define the* ϵ-closure $ECLOSE(q)$ *by induction as follows:*

Basis: $q \in ECLOSE(q)$

Induction: *If* $p \in ECLOSE(q)$ *and* $r \in \delta(p,\epsilon)$ *then* $r \in ECLOSE(q)$

Example 1.8 [ECLOSE]



$\text{ECLOSE}(1) = \{1,2,3,4,6\}$
$\text{ECLOSE}(2) = \{2,3,6\}$
$\text{ECLOSE}(3) = \{3,6\}$
$\text{ECLOSE}(4) = \{4\}$
$\text{ECLOSE}(5) = \{5,7\}$
$\text{ECLOSE}(6) = \{6\}$
$\text{ECLOSE}(7) = \{7\}$

Definition 1.21 [ϵ-NFA Transition Function] *The transition function* δ *can be extended to* $\hat{\delta}$ *that operates on states and strings (as opposed to states and symbols).*

Basis:

$$\hat{\delta}(q,\epsilon) = ECLOSE(q)$$

Induction:

$$\hat{\delta}(q,xa) = \bigcup_{p\in\delta(\hat{\delta}(q,x),a)} ECLOSE(p)$$

JAVA Epsilon-NFA Implementation

Algorithm 1.2 [NFA-Algorithm]

Let first define two functions.

Function	Description
ϵ-closure(T)	The union of the set T with all NFA-states reachable from $s \in T$ through an ϵ-transition.
move (T,c)	Set of NFA states reachable from $s \in T$ through a transition labelled with c.
nextChar()	The next character of the input string

At the begin of the simulation, the ϵ-NFA may be in state of ECLOSE(q_0)

$T = \epsilon$-closure(q_0)
c = nextChar()
while ($c \neq$ **eof**)
  $U = \varnothing$
  forall ($s \in T$)
    $U = U \cup$ move(s, c)
  endforall
  $T = \epsilon$_closure(U)
  c = nextChar()
endwhile
if ($T \cap F \neq \varnothing$)
  return `true`
else
  return `false`
endif

Set of states are implemented with the JAVA-class `Bitset`. As a consequence, the transition table will be a two dimensional `Bitset`-Vector.

JAVA-Class `NFA`

```
1   public class NFA {
2
3     protected BitSet[][] table;
4     protected BitSet[] epsilon;
5     protected BitSet accept;
6     protected int start;
7     protected int states;
8
9     private BitSet closure(BitSet from) {
10      BitSet open = (BitSet) from.clone();
11      BitSet closed = new BitSet(states);
12      while (!open.isEmpty()) {
13        int index = open.nextSetBit(0);
14        open.clear(index);
15        closed.set(index);
16        BitSet e = epsilon[index];
17        if (e != null) {
18          for (int i = e.nextSetBit(0);
19               i >= 0; i = e.nextSetBit(i + 1)) {
20            if (!closed.get(i) && !open.get(i)) {
21              open.set(i);
22            }
23          }
24        }
25      }
26      return closed;
27    }
28
29    private BitSet closure(int state) {
30      BitSet b = new BitSet(states);
31      b.set(state);
32      return closure(b);
33    }
34
35    private int nextChar(InputStreamReader reader) {
36      try {
```

```
37        return reader.read();
38      } catch (IOException e) {
39        return -1;
40      }
41    }
42
43    private BitSet move(BitSet from, char input) {
44      BitSet closed = new BitSet(states);
45      for (int i = from.nextSetBit(0);
46           i >= 0; i = from.nextSetBit(i + 1)) {
47        BitSet e = table[i][input];
48        if (e != null) {
49          closed.or(e);
50        }
51      }
52      return closure(closed);
53    }
54
55    public BitSet match(InputStreamReader reader) {
56      BitSet b = new BitSet(states);
57      b.set(start);
58      BitSet state = closure(b);
59      int i = nextChar(reader);
60      while (i != -1 && state != null) {
61        char c = (char) i;
62        state = closure(move(state, c));
63        i = nextChar(reader);
64      }
65      return state;
66    }
67  }
```

Exercise 1.10 [NFA] *Design an NFA to recognise* $\{abc, abd, aacd\}$. *Assume the alphabet is* $\{a, b, c, d\}$.

Exercise 1.11 [Exponential] *Build a NFA that recognises the language*

$$L = \{0,1\}^* \cdot \{0\} \cdot \{0,1\} \cdot \{0,1\}$$

The NFA should not have more than 4 *states.*

Exercise 1.12 [Nondeterminism] *Design an NFA to accept the set of strings of* 0*'s and* 1*'s that either:*

1. *End in* 010 *and have* 011 *somewhere preceding, or*
2. *End in* 101 *and have* 100 *somewhere preceding.*

In order to make sure that you use nondeterminism, your NFA should have no more than 13 *states and* 15 *edges. You may represent your NFA by a transition diagram.*

1.4 Equivalence NFA/DFA

- NFA's are usually easier to *program* in.
- Surprisingly, for any NFA N there is a DFA D, such that $L(D) = L(N)$, and vice versa.

- This involves the subset construction, an important example how an automaton B can be generically constructed from another automaton A.

Given an NFA

$$N = (Q_N, \Sigma, \delta_N, q_0, F_N)$$

we will construct a DFA

$$D = (Q_D, \Sigma, \delta_D, q_0, F_D)$$

such that

$$L(D) = L(N)$$

Definition 1.22 [Subset Construction] *The details of the* subset construction*:*

- $Q_D = \{S : S \subseteq Q_N\}$. *Note:* $|Q_D| = 2^{|Q_N|}$, *although most states in* Q_D *are likely to be garbage.*
- $F_D = \{S \subseteq Q_N : S \cap F_N \neq \varnothing\}$
- *For every* $S \subseteq Q_N$ *and* $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

Example 1.9 [Subset Construction]

An NFA N that accepts all and only strings ending in 01 can be defined using following transition table:

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

Using the subset construction we can build following DFA D:

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

The states of D correspond to subsets of states of N, but we could have denoted the states of D by, say, $A - F$ just as well.

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Considering only states reachable from B (q_0), we get finally following equivalent DFA:

	0	1
$\rightarrow B$	E	B
E	E	F
$*F$	E	B

Theorem 1.1 [Subset Construction 1] *For every NFA N if D is the DFA obtained by the subset construction,*

$$L(N) = L(D)$$

It is actually easier to prove following stronger claim.

Theorem 1.2 [Subset Construction 2] *For every NFA N if D is the DFA obtained by the subset construction,*

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w) \quad \textit{for all} \quad w \in \Sigma^*$$

Proof: First we show on an induction on w that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$

Basis: $w = \epsilon$ the proof follows directly from the definition of $\hat{\delta}_D(\{q_0\}, w)$ and $\hat{\delta}_N(q_0, w)$

Induction: Suppose that the equality is correct for $|w| = 1$, and let $w = xa$, $|w| = n + 1$

$$\begin{aligned}
\hat{\delta}_D(\{q_0\}, xa) &= \delta_D(\hat{\delta}_D(\{q_0\}, x), a) \\
&= \delta_D(\hat{\delta}_N(q_0, x), a) \\
&= \bigcup_{p \in \hat{\delta}_N(q_0, x)} \hat{\delta}_N(p, a) \\
&= \hat{\delta}_N(q_0, xa)
\end{aligned}$$

The first equality follows from the definition of $\hat{\delta}_D$, the second by induction hypothesis, the third by definition of δ_D and the last by definition of $\hat{\delta}_N$ □

Now it is easy to prove theorem 1.1.

Proof: Let x be a string in Σ^* $L(D) = L(N)$ is equivalent saying that x is accepted by D if and only if x is accepted by N.

$$\begin{array}{lcll}
x \text{ is accepted by } D & \Leftrightarrow & \hat{\delta}_D(\{q_0\}, x) \in F_D & \text{def of accept for DFA} \\
& \Leftrightarrow & \hat{\delta}_D(\{q_0\}, x) \cap F_N \neq \varnothing & \text{def of } F_D \text{ in subset construction} \\
& \Leftrightarrow & \hat{\delta}_N(\{q_0\}, x) \cap F_N \neq \varnothing & \text{previous lemma} \\
& \Leftrightarrow & x \text{ is accepted by } N & \text{def of accept for NFA}
\end{array}$$

□

Theorem 1.3 [Equivalence DFA/NFA] *A language L is $L(N)$ for some NFA N iff and only if it is $L(D)$ for some DFA D.*

Proof: The subset construction was the hard part: Given any NFA, a DFA accepting the same language can be constructed. So, whenever a language is accepted by a NFA, it is also accepted by some DFA.

The other direction of the proof is obvious. Every DFA is essentially an NFA. The only change is that $\delta_N(q, x) = \{\delta_D(q, x)\}$. All other aspects of the FA remain the same, including F. □

Exercise 1.13 [NFA to DFA 1] *Convert to a DFA the following NFA:*

	0	1
$\rightarrow p$	$\{q,s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q,r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\varnothing$	$\{p\}$

Exercise 1.14 [NFA to DFA 2] *Convert to a DFA the following NFA:*

	a	b
$\rightarrow p$	$\{r\}$	$\{q\}$
q	$\{s\}$	$\{q,r\}$
r	$\{p,q\}$	$\{s\}$
$*s$	$\{s\}$	$\{s\}$

1.5 Equivalence Epsilon-NFA/DFA

Given an ϵ-NFA

$$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$$

we will construct a DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

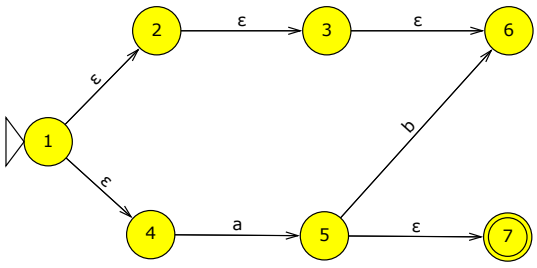
such that

$$L(D) = L(E)$$

Definition 1.23 [ECLOSE Construction]

- $Q_D = \{S : S \subseteq Q_E \quad and \quad S = ECLOSE(S)\}$
- $q_D = ECLOSE(q_0)$
- $F_D = \{S : S \in Q_D \quad and \quad S \cap F_E \neq \varnothing\}$
- $\delta_D(S,a) = \bigcup\{ECLOSE(p) : p \in \delta(t,a) \quad for\ some \quad t \in S\}$

Example 1.10 [ECLOSE Construction]
ECLOSE Construction for following ϵ-NFA:



	a	b
$\rightarrow \{1,2,3,4,6\}$	$\{5,7\}$	$\varnothing$
$*\{5,7\}$	$\varnothing$	$\{6\}$
$\{6\}$	$\varnothing$	$\varnothing$

Theorem 1.4 [Equivalence ϵ-NFA/DFA] *A language L is accepted by some ϵ-NFA E if and only if L is accepted by some DFA D.*

Proof: We use D constructed as above and show by induction that $\hat{\delta}_D(q_0, w) = \hat{\delta}_E(q_D, w)$

Basis: $\hat{\delta}_E(q_0, \epsilon) =$ ECLOSE$(q_0 = d_D = \hat{\delta}_E(q_D, \epsilon)$

Induction:

$$\begin{aligned}\hat{\delta}_E(q_0, xa) &= \bigcup_{p \in \delta_E(\hat{\delta}_E(q_0,x),a)} \text{ECLOSE}(p) \\ &= \bigcup_{p \in \delta_D(\hat{\delta}_D(q_D,x),a)} \text{ECLOSE}(p) \\ &= \bigcup_{p \in \hat{\delta}_D(q_D,xa)} \text{ECLOSE}(p) \\ &= \hat{\delta}_D(q_D, xa)\end{aligned}$$

□

Exercise 1.15 [Convert] *Convert to a DFA the following ϵ-NFA:*

	ϵ	a	b	c
$\rightarrow p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{r\}$	$\{q\}$	$\{r\}$	$\varnothing$
$*r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

Exercise 1.16 [NFA to DFA 3] *Convert to a DFA the following ϵ-NFA:*

	ϵ	a	b	c
$\rightarrow p$	$\{q,r\}$	$\varnothing$	$\{q\}$	$\{r\}$
q	$\varnothing$	$\{p\}$	$\{r\}$	$\{p,q\}$
$*r$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

1.5.1 JAVA NFA to DFA Implementation

The algorithm for the transformation of an NFA into an equivalent DFA is similar to the simulation algorithm 1.2 for NFA's:

Algorithm 1.3 [NFA2DFA-Algorithm] Let first define two functions.

Function	Description
ϵ-closure(T)	The union of the set T with all NFA-states reachable from $s \in T$ through an ϵ-transition.
move (T, c)	Set of NFA states reachable from $s \in T$ through a transition labelled with c.
nextChar()	The next character of the input string

Dstates is the set of states of the DFA Dtrans its set of transitions:

```
DStates = {ε_closure(z0)}
while (there are unmarked statesT in Dstates )
    mark T
    for each symbol a ∈ Σ
        U = ε_closure(move(T, a))
        if (U not in Dstates )
            add U to Dstates (U unmarked)
        Dtran[T, a] = U
    endfor
endwhile
```

JAVA-method `toDfa`

```
1  public DFA toDFA() {
2    DFA dfa = new DFA();
3    Vector<BitSet> open = new Vector<BitSet>();
4    // the vector containing the states of the DFA.
5    // the DFA states are numbered
6    // according to the corresponding index in the vector
7    Vector<BitSet> dStates = new Vector<BitSet>();
8    int from, to;
9    // first compute dStates
10   BitSet b = new BitSet(states);
11   b.set(start);
12   BitSet u = closure(b);
13   open.add(u);
14   dStates.add(u);
15   from = dStates.indexOf(u);
16   dfa.addState(from);
17   dfa.setStart(from);
18   for (int i = 0; i < states; i++) {
19     if (accept.get(i) && u.get(i)) {
20       dfa.addAccepting(from, i);
21       break;
22     }
23   }
24   while (!open.isEmpty()) {
25     BitSet t = open.remove(0);
26     if (!dStates.contains(t)) {
27       dStates.add(t);
28       from = dStates.indexOf(t);
29       dfa.addState(from);
30       for (int i = 0; i < states; i++) {
31         if (accept.get(i) && t.get(i)) {
32           dfa.addAccepting(from, i);
33           break;
34         }
35       }
36     } else {
37       from = dStates.indexOf(t);
38     }
39     for (char a = 0; a < 128; a++) {
40       u = move(t, a);
41       if (!u.isEmpty()) {
42         u = closure(u);
43         if (!dStates.contains(u)) {
44           dStates.add(u);
45           to = dStates.indexOf(u);
46           dfa.addState(to);
47           for (int i = 0; i < states; i++) {
48             if (accept.get(i) && u.get(i)) {
49               dfa.addAccepting(to, i);
50               break;
51             }
52           }
53           if (!open.contains(u)) {
54             open.add(u);
55           }
56         }
57         to = dStates.indexOf(u);
```

```
58            dfa.addTransition(from, a, to);
59          }
60        }
61      }
62      return dfa;
63    }
```

Chapter 2

Regular Expressions

In computing, regular expressions provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a scanner generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- the sequence of characters `car` in any context, such as `car`, `cartoon`, or `bicarbonate`
- the word `car` when it appears as an isolated word
- the word `car` when preceded by the word `blue` or `red`
- a dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits

- A FA (NFA or DFA) is a *blueprint* for constructing a machine recognising a regular language.
- A regular expression is a *user-friendly*, declarative way of describing a regular language.
- Example: $01^* + 10^*$
- Regular expressions are used in e.g.
  1. UNIX grep command
  2. UNIX Lex (Lexical analyser generator) and Flex (Fast Lex) tools.

2.1 Regular Expressions

A regular expression defines a set of strings over an alphabet A, i.e. regular expressions define languages.

Definition 2.1 [Regular Expression]

Regular Expressions *(RE) are defined by induction as follows:*

Basis: *Atomic expressions:* ϵ *is a RE and* $\varnothing$ *is a RE.*
$L(\epsilon) = \{\epsilon\}$, *and* $L(\varnothing) = \varnothing$.

If $a \in \Sigma$, *then* a *is a RE.*
$L(a) = \{a\}$.

Induction: *If E is a RE, then (E) is a RE.*
$L((E)) = L(E)$.

Choice expression: If E and F are RE's then $E + F$ is a RE.
$L(E + F) = L(E) \cup L(F)$.

Sequence expression: If E and F are RE's then $E.F$ is a RE.
$L(E.F) = L(E).L(F)$.

Iteration expression: If E is a RE's then E^* is a RE.
$L(E^*) = L(E)^*$.

Notation 2.1 [Dot] We write EF instead of $E.F$

Example 2.11 [Regular Expression] RE for

$$L = \{w \in \{0,1\} : 0 \quad \text{and} \quad 1 \quad \text{alternate in} \quad w\}$$

$$(01)^* + (10)^* + 0(10)^* + 1(01)^*$$

or, equivalently

$$(\epsilon + 1)(01)^*(\epsilon + 0)$$

Remark 2.1 [Precedence] Order of precedence for operators: Star ($*$), Dot (.), Plus ($+$)

2.1.1 Other Notations

Traditional UNIX regular expression syntax followed common conventions but often differed from tool to tool. Following tables contain the syntax of some regular expressions used in JAVA[1]. Note the extensive usage of escape sequences (\) since regular expressions are defined as JAVA strings.

[1] See `java.util.regex.Pattern` in the JAVA API documentation

Construct	Matches
Characters	
x	The character x
\\	The backslash character
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\t	The tab character
\n	The newline
\r	The carriage-return character
Character classes	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)
Predefined character classes	
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]
Greedy quantifiers	
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
$X\{n\}$	X, exactly n times
$X\{n,\}$	X, at least n times
$X\{n,m\}$	X, at least n but not more than m times
Logical operators	
XY	X followed by Y
$X\|Y$	Either X or Y
(X)	X, as a capturing group

Exercise 2.1 [RE 1] *Give a regular expression that denotes the language*

$$L = \{ab\} \cdot (\{aa, bb\}^* \cup (\{b\} \cdot \{a, b\}^+))$$

Exercise 2.2 [RE 2] *Give a regular expression for the set of binary strings where the number of* 0*'s is a multiple of* 3*.*

Exercise 2.3 [RE 3] *Give a regular expression for the set of strings over the alphabet* $\Sigma = \{a, b\}$ *that start and end with* a*.*

Exercise 2.4 [RE 4] *Give a regular expression for the set of strings over the alphabet* $\Sigma = \{a, b\}$ *that do not contain the substring* aba*.*

Exercise 2.5 [RE 5]

What is the following JACACC *regular expression (see chapter C) used for?*

```
"\""(   (~["\"","\\","\n","\r"])
  | ("\\"
      ( ["n","t","b","r","f","\\","'","\""]
      | ["0"-"7"] ( ["0"-"7"] )?
      | ["0"-"3"] ["0"-"7"] ["0"-"7"]
```

```
        )
    )
)*
"\""
```

2.1.2 Equivalence of FA's and RE's

We have already shown that DFA's, NFA's, and ϵ-NFA's all are equivalent.
To show FA's equivalent to RE's we need to establish that

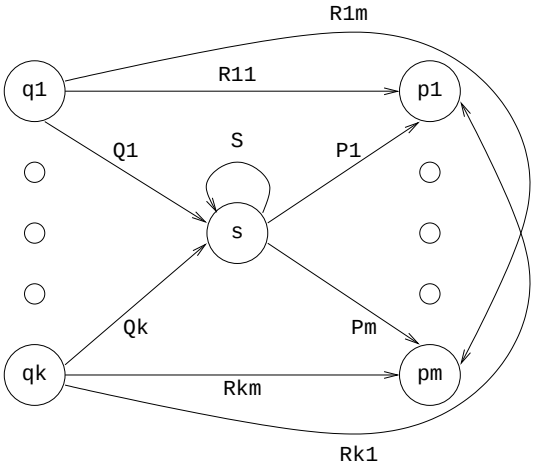
1. For every DFA A we can find (construct, in this case) a RE R, s.t. $L(R) = L(A)$.
2. For every RE R there is a ϵ-NFA A, s.t. $L(A) = L(R)$.

Theorem 2.1 [Equivalence of FA's and RE's] *For every DFA $A = (Q, \Sigma, \delta, q_0, F)$ there is a RE R, s.t.*

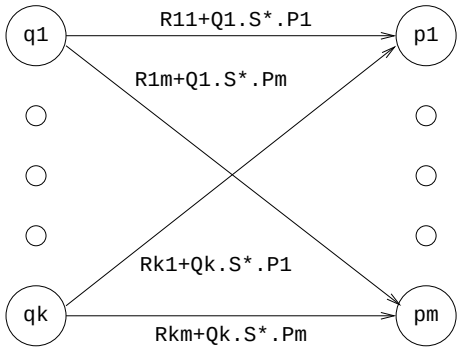
$$L(R) = L(A)$$

State Elimination

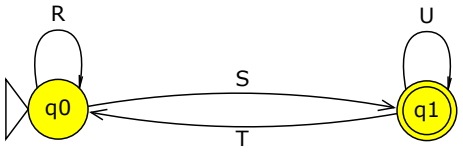
Let's label the edges with RE's instead of symbols:



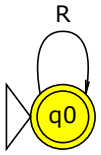
Now, let's eliminate state s.



For each accepting state q eliminate from the original automaton all states except q_0 and q.
For each $q \in F$ we'll be left with an A_q that looks like



that corresponds to the RE $E_q = (R + SU^*T)^*SU^*$ or with A_q looking like

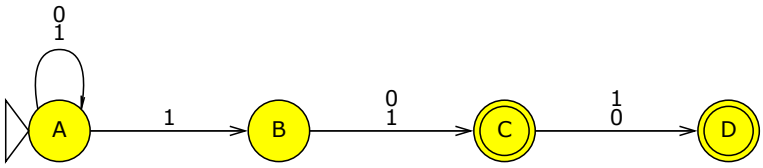


corresponding to the RE $E_q = R^*$.
The final regular expression is

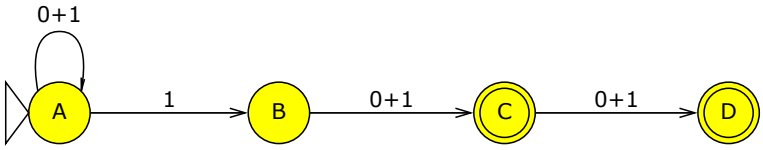
$$\bigoplus_{q \in F} E_q$$

Example 2.12 [State Elimination] A, where

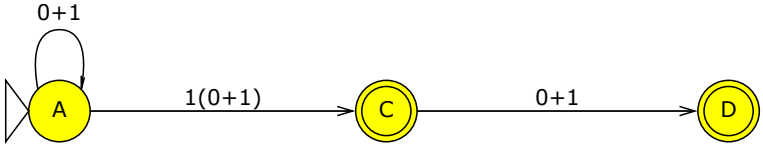
$$L(A) = \{w : w = x1b, \quad \text{or} \quad w = x1bc, x, b, c \in \{0,1\}\}$$



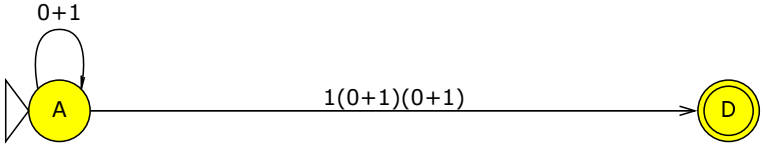
We turn this into an automaton with RE labels



Let's eliminate state B

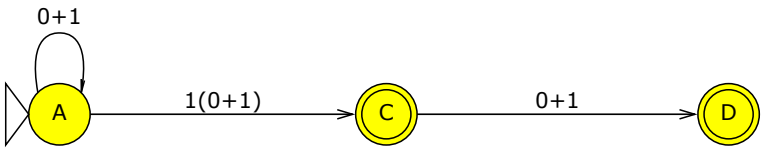


Then we eliminate state C and obtain A_D

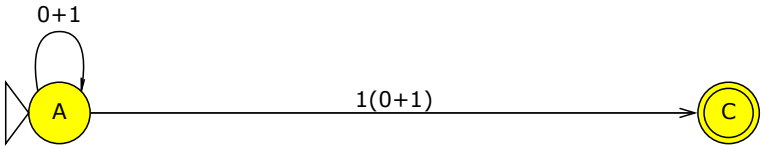


with RE $(0+1)^*1(0+1)(0+1)$

From



we can eliminate D to obtain A_C



with RE $(0+1)^*1(0+1)$

The final expression is the sum of the previous two RE's:

$$(0+1^*)1(0+1)(0+1)+(0+1)^*1(0+1)$$

From RE's to Epsilon-NFA's

Theorem 2.2 [RE's to Epsilon-NFA's] *For every RE R we can construct and ϵ-NFA A, s.t.*

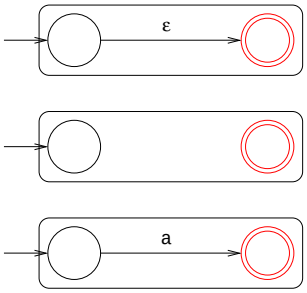
$$L(A) = L(R)$$

The structural induction used in the proof is called Thompson construction

Proof: By structural induction.

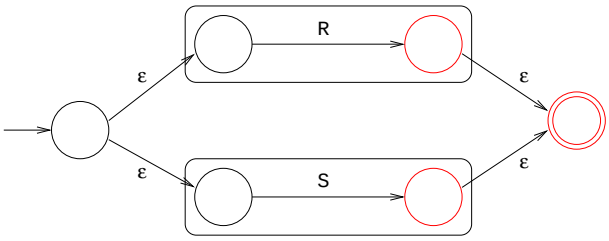
Basis: Automata for ϵ, $\varnothing$ and $a \in \Sigma$:

Atomic expressions:

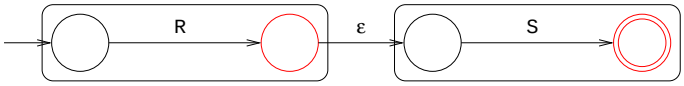


Induction: Automata for $R+S$, RS, and R:

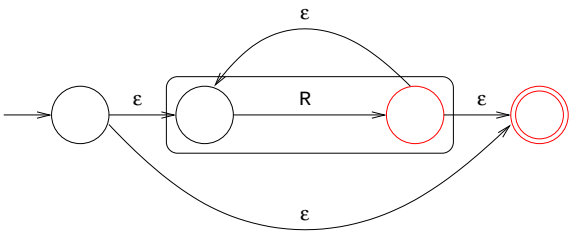
Choice:



Sequence:



Iteration:



□

Exercise 2.6 [DFA to RE] *Convert the following DFA to a regular expression, using the state elimination technique.*

	0	1
$\rightarrow *p$	s	p
$\rightarrow q$	p	s
$\rightarrow r$	r	q
$\rightarrow s$	q	r

Exercise 2.7 [Thompson] *Construct an ϵ-NFA for the regular expression $a(a+b)^*a+a$. Transform the ϵ-NFA to an equivalent DFA.*

2.2 Algebraic Laws for Languages

Theorem 2.3 [Algebraic Laws for Languages]

- $L \cup M = M \cup L$. *Union is commutative.*
- $(L \cup M) \cup N = L \cup (M \cup N)$. *Union is associative.*
- $(LM)N = L(MN)$. *Concatenation is associative. Note: Concatenation is not commutative, i.e., there are L and M such that $LM \neq ML$.*
- $\varnothing \cup L = L \cup \varnothing = L$. *$\varnothing$ is identity for union.*
- $\{\epsilon\}L = L\{\epsilon\} = L$. *$\{\epsilon\}$ is left and right identity for concatenation.*
- $\varnothing L = L\varnothing = \varnothing$. *$\varnothing$ is left and right annihilator for concatenation.*
- $L(M \cup N) = LM \cup LN$. *Concatenation is left distributive over union.*
- $(M \cup N)L = ML \cup NL$. *Concatenation is right distributive over union.*
- $L \cup L = L$. *Union is idempotent.*
- $\varnothing^* = \{\epsilon\}$, $\{\epsilon\}^* = \{\epsilon\}$.
- $L^+ = LL^* = L^*L$, $L^* = L^+ \cup \{\epsilon\}$.
- $(L^*)^* = L^*$. *Closure is idempotent*

□

2.3 Algebraic Laws for Regular Expressions

Definition 2.2 [Regular Language] *A language L is* regular *if it is the language of some RE R, i.e. $L = L(R)$*

Theorem 2.4 [Algebraic Laws for RE's]

- $E + F = F + E$, *that is, $+$ is commutative.*
- $(E + F) + G = E + (F + G)$. *That is, $+$ is associative.*
- $(EF)G = E(FG)$, *that is, concatenation is associative.*
- $E(F + G) = EF + EG$, $(E + F)G = EG + FG$, *that is, the concatenation is distributive over $+$.*
- $\epsilon E = E\epsilon = E$, *that is, ϵ is the neutral element for concatenation.*
- $E^{**} = E^*$, *that is, $*$ is idempotent.*

□

2.4 JAVA Implementation

2.4.1 RE Implementation

Following JAVA-classes may be used to implement the structure of RE's. Each step of definition 2.1 is implemented in a separate class. All classes are subclasses of `RegularExpression`.

JAVA-Class `RegularExpression`

```
1  public abstract class RegularExpression {
2  }
```

Following class corresponds to the RE a.

JAVA-Class `CharacterExpression`

```
1  public class CharacterExpression extends RegularExpression {
2    private final char a;
3
4    public CharacterExpression(char a) {
5      this.a = a;
6    }
7
8    public char getChar() {
9      return a;
10   }
11 }
```

Following class corresponds to the RE ϵ.

JAVA-Class `EmptyExpression`

```
1  public class EmptyExpression extends RegularExpression {
2  }
```

Following class corresponds to the RE $E1 + E2$.

JAVA-Class `ChoiceExpression`

```
1  public class ChoiceExpression extends RegularExpression {
2
3    private final RegularExpression e1;
4    private final RegularExpression e2;
5
6    public ChoiceExpression(RegularExpression e1,
7                            RegularExpression e2) {
8      this.e1 = e1;
9      this.e2 = e2;
10   }
11
12   public RegularExpression getSubAlternative(int nr) {
13     if (nr == 1)
14       return e1;
15     else if (nr == 2)
16       return e2;
17     else
18       return null;
19   }
20 }
```

Following class corresponds to the RE $E1.E2$.

JAVA-Class SequenceExpression

```
1  public class SequenceExpression extends RegularExpression {
2
3    private final RegularExpression e1;
4    private final RegularExpression e2;
5
6    public SequenceExpression(RegularExpression e1,
7                              RegularExpression e2) {
8      this.e1 = e1;
9      this.e2 = e2;
10   }
11
12   public RegularExpression getSubExpression(int nr) {
13     if (nr == 1)
14       return e1;
15     else if (nr == 2)
16       return e2;
17     else
18       return null;
19   }
20 }
```

Following class corresponds to the RE $E*$.

JAVA-Class IterateExpression

```
1  public class IterateExpression extends RegularExpression {
2    private final RegularExpression e;
3
4    public IterateExpression(RegularExpression e) {
5      this.e = e;
6    }
7
8    public RegularExpression getIterateExpression() {
```

```
 9        return e;
10      }
11    }
```

Exercise 2.8 [RE 1] *In this exercise, you will do some programming work. The incomplete source files are available from* `prog/st/afl.jar`. *It may be usefull to generate the* JAVADOC *files.*

1. *The package* `afl.au.dfa` *contains classes for DFA's.*
2. *The package* `afl.au.nfa` *contains classes for NFA's.*
3. *The package* `afl.re` *contains classes for RE's.*
4. *The package* `afl.re.parser` *contains a parser for for RE's.*
5. *The package* `afl.re.tonfa` *contains classes for the transformation of RE's info NFA's.*
6. *The package* `afl.ex` *contains classes for Exception handling.*
7. *The package* `afl.util` *contains some utility classes (messaging).*

Write following programs

1. *Implement all methods of the class* `afl.re.ToString`, *such that the* `toString()` *methods of the RE's return a string using as less parentheses as possible.*

   Hint: You may have to consider the priorities of the RE's.

2. *Implement the Thompson construction for RE's. You will have to implement methods* `thompson` *in all* `RegularExpression` *subclasses.*

   Hint: You will construct a NFA with start state `0` *and accepting state* `1`. *Each* `thompson()` *method may receive a* `FromToPair` *object as a parameter. This objects would tell to the method which are the start and the accepting states of the subexpression.*

3. *Suppose that you would like to check many RE's at the same time. Note that an input string may be matched by many regular expressions at the same time.*

   Which modifications are necessary in the application, such that it is possible to detect the RE's matching the input? You dont need to implement the modifications.

Chapter 3

Properties of Regular Languages

3.1 Proving Languages not to be Regular

The *pumping lemma* states that any string in a regular language of at least a certain length (called the pumping length), contains a section that can be removed, or repeated any number of times, with the resulting string remaining in the language.

Theorem 3.1 [The Pumping Lemma for RE] *Let L be regular. Then* $\exists n \quad \forall w \in L : |w| \geq n \Rightarrow w = xyz$ *such that*

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $\forall k \geq 0, xy^k z \in L$

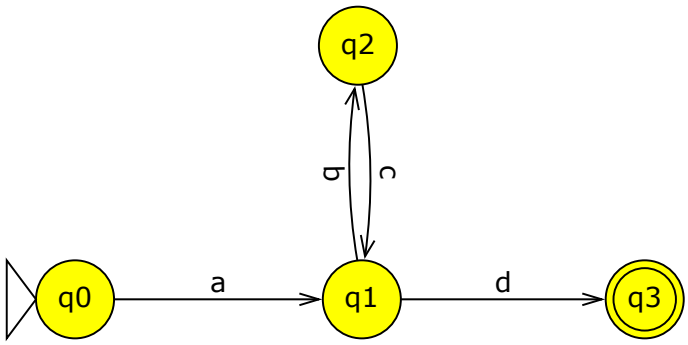
n *is called* pumping length.

`Proof:` Since L is regular, it recognised by some DFA A. Assume that the DFA has n states. Let $w = a_1 a_2 \cdots a_m \in L$, $m > n$. Let $p_i = \hat{\delta}(q_0, a_1 a_2 \cdots a_i)$. Since A has n states, it follows that $\exists i < j : p_i = p_j$

Now $w = xyz$ with $x = a_1 a_2 \cdots a_i$, $y = a_{i+1} a_{i+2} \cdots a_j$ and $z = a_{j+1} a_{j+2} \cdots a_m$.

Evidently $xy^k z \in L, \forall k \geq 0$. □

Example 3.13 [Pumping Lemma] The following image shows an DFA.



The DFA accepts the string: $abcbcd$. Since this is longer than the number of states, there are repeated states by the pigeonhole principle. In this example, the repeated states are states $q1$ and $q2$. Since the substring $bcbc$ of string $abcbcd$ takes the machine through transitions that start at state $q1$ and end at state $q1$, the string portion could be repeated and the DFA would

still accept giving the string $abcbcbcbcd$, and the string portion could be removed and the DFA would still accept giving the string ad.

The pumping lemma is often used to prove that a particular language is non-regular: a proof by contradiction (of the language's regularity) may consist of exhibiting a word (of the required length) in the language which lacks the property outlined in the pumping lemma.

Example 3.14 [L_{eq}] L_{eq} be the language of strings with equal number of zero's and one's is not regular.

Proof: Suppose Leq is regular. Then $= L_{01} = \{w \in \{0,1\}^* : w = 0^n 1^n, n \in \mathbb{N}\}$ is also regular, since $L_{01} \subset L_{eq}$.

By the pumping lemma $w = xyz$, $|xy| \leq n$, $y \neq \epsilon$ and $xy^k z \in L_{eq}$

$$w = \underbrace{00\cdots00}_{x}\underbrace{00\cdots00}_{y}\underbrace{01\cdots11}_{z}$$

In particular, $xz \in L_{eq}$, but xz has fewer 0's than 1's. This is a contradiction, thus L_{eq} cannot be regular. □

The proof that the language of balanced (i.e., properly nested) parentheses is not regular follows the same idea. Given p, there is a string of balanced parentheses that begins with more than p left parentheses, so that y will consist entirely of left parentheses. By repeating y, we can produce a string that does not contain the same number of left and right parentheses, and so they cannot be balanced.

Exercise 3.1 [Palindrome] *A palindrome is a string that equals its own reverse, such as* 0110 *or* 1011101*. Use the pumping lemma to show that the set of palindromes is not a regular language.*

3.2 Closure Properties of Regular Languages

Theorem 3.2 [Properties of Regular Languages] *Let L, L_1 and L_2 be regular languages over an alphabet Σ, then*

1. *$L_1 \cup L_2$ is a regular language.*
2. *The complement $\overline{L} = \Sigma^* - L$ of L is a regular language.*
3. *$L_1 \cap L_2$ is a regular language.*
4. *$L_1 - L_2$ is a regular language.*
5. *L^R the reversal of L is a regular language.*
6. *L^* is a regular language.*
7. *$L_1 \cdot L_2$ is a regular language.*

Proof:

1. If L_1 and L_2 then $L_1 = L(E_1)$ and $L_2 = L(E_2)$ for some RE E_1 and E_2 . Then $L(E_1+E_2) = L_1 \cup L_2$ by definition.
2. Let L be recognised by a DFA $A = (Q, \Sigma, \delta, q_0, F)$. Let $B = (Q, \Sigma, \delta, q_0, Q - F)$. Now $L(B) = \overline{L}$
3. By De Morgan law $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. The rest follows from from 1) and 2).
4. Follows from $L_1 - L_2 = L_1 \cap \overline{L_2}$
5. If L is regular, then there is an NFA A such that $L = L(A)$ (Thompson construction). We turn A into an NFA for L^R:

- Reversing all arcs.
- Make the old start state the new sole accepting state.
- Create a new start state p_0, with $\delta(p_0, \epsilon) = F$ (the old accepting states).

6. Exercise.
7. Exercise.

□

Exercise 3.2 [Kleene Closure] *Prove that if L is regular languages over an alphabet Σ, then L^* is also regular language.*

Exercise 3.3 [Concatenation] *Prove that if L_1 and L_2 are regular languages over an alphabet Σ, then $L_1 \cdot L_2$ is also regular language.*

3.3 Minimisation of Automata

Definition 3.1 [Equivalent States]

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and $\{p, q\} \in Q$. We define

$$p \equiv q \quad \Leftrightarrow \quad \forall w \in \Sigma^* : \hat{\delta}(p, w) \in F \quad \textit{if and only if} \quad \hat{\delta}(q, w) \in F$$

If $p \equiv q$ we say that p and q are equivalent

If $p \not\equiv q$ we say that p and q are distinguishable

In other words p and q are distinguishable if and only if

$$\exists w : \hat{\delta}(p, w) \in F \quad \textit{and} \quad \hat{\delta}(q, w) \notin F \quad \textit{or vice versa}$$

Remark 3.1 [Equivalent States] It is easy to prove that $\equiv$ is an equivalence relation, i.e.

Reflexivity $p \equiv p \quad \forall p \in S$

Symmetry If $p \equiv q$ then $q \equiv p$

Transitivity If $p \equiv q$ and $q \equiv r$ then $p \equiv r$

We can compute distinguishable pairs with the following inductive *table filling algorithm*:

Algorithm 3.1 [Table Filling Algorithm] Basis: If $p \in F$ and $q \notin F$, then $p \not\equiv q$.

Induction: If $\exists a \in \Sigma : \delta(p, a) \not\equiv \delta(q, a)$ then $p \not\equiv q$.

Algorithm 3.2 [Table Filling Implementation]

Let $D = (Q, \Sigma, \delta, q_1, F)$ be a DFA with n states numbered from 1 to n. Build pairs $\{i, j\}$ with $i \neq j$.

```
for i ∈ F
    for j ∉ F
        mark {i, j} as distinguishable
    endfor
endfor
do
    for each unmarked pair {i, j}
        for each input symbol a ∈ Σ
            if {δ(i, a), δ(j, a)} is a distinguishable pair
```

mark $\{i, j\}$ as distinguishable
break
endif
endfor
endfor
while at least one pair could be marked as distinguishable.

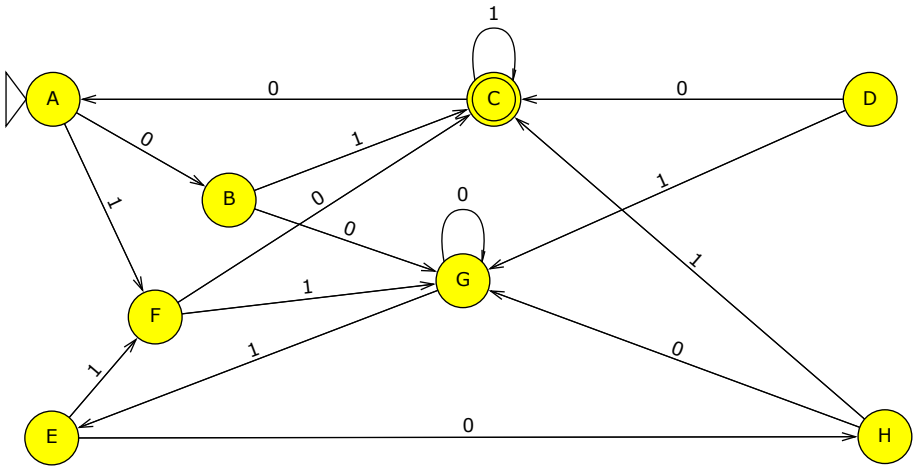
We can minimise a DFA using following algorithm:

Algorithm 3.3 [DFA minimisation]

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The equivalent minimised DFA B can be constructed as follows:

1. Use the table filling algorithm to find all pairs of equivalent states.
2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.
3. Construct the minimal-state equivalent DFA B by using the blocks as its states. Let γ the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states would be distinguishable (contradiction). As a consequence, we can let $\gamma(S, a) = T$.
4. The start state of B is the block containing the start state of A.
5. The set of accepting states of B is the set of blocks containing accepting states of A.

Example 3.15 [Equivalent States]



Round 0:

C is the only accepting state. It follows that $\{A, C\}$, $\{B, C\}$, $\{C, D\}$, $\{C, E\}$, $\{C, F\}$, $\{C, G\}$ and $\{C, H\}$ are pairs of distinguishable states.

Round 1:

$\{A, B\}$ is distinguishable since states A and B go to F and C on input 1.

$\{A, C\}$ is distinguishable since states A and C go to F and C on input 1.

$\{A, D\}$ is distinguishable since states A and D go to B and C on input 0.

$\{A, F\}$ is distinguishable since states A and F go to B and C on input 0.

$\{A, H\}$ is distinguishable since states A and H go to F and C on input 1.

$\{B, D\}$ is distinguishable since states B and D go to G and C on input 0.

$\{B, E\}$ is distinguishable since states B and E go to C and F on input 1.

$\{B, F\}$ is distinguishable since states B and F go to C and G on input 1.

$\{B, G\}$ is distinguishable since states B and G go to C and E on input 1.

$\{D, E\}$ is distinguishable since states D and E go to C and H on input 0.

$\{D, G\}$ is distinguishable since states D and G go to C and G on input 0.

$\{D, H\}$ is distinguishable since states D and H go to C and G on input 0.

$\{E, F\}$ is distinguishable since states E and F go to H and C on input 0.

$\{E, G\}$ is distinguishable since states E and G go to F and E on input 1.

$\{E, H\}$ is distinguishable since states E and H go to F and C on input 0.

$\{F, G\}$ is distinguishable since states F and G go to G and E on input 1.

$\{F, H\}$ is distinguishable since states F and H go to G and C on input 1.

$\{G, H\}$ is distinguishable since states G and H go to E and C on input 1.

Round 2:

$\{A, G\}$ is distinguishable since states A and G go to F and E on input 1.

$\{E, G\}$ is distinguishable since states E and G go to F and E on input 1.

Round 3:

The three remaining pairs $\{A, E\}$, $\{B, H\}$ and $\{D, F\}$ are equivalent:

$\{A, E\}$ are indistinguishable since they both go to B and H on input 0 (and $\{B, H\}$ has not been shown distinguishable) and they both go to F on input 1.

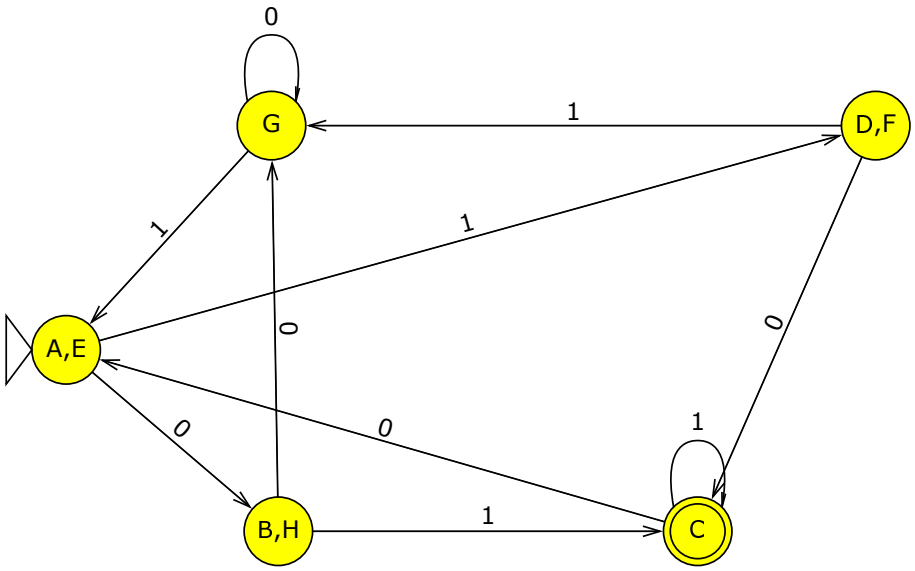
$\{B, H\}$ are indistinguishable since they both go to G on input 0 and to C on input 1.

$\{D, F\}$ are indistinguishable since they both go to C on input 0 and to G on input 1.

Thus the table filling algorithm stops with following table:

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Thus the equivalent minimised DFA has states $Q = \{\{A, E\}, \{B, H\}, \{G\}, \{D, F\}, \{C\}\}$, start state $\{A, E\}$ and accepting states $F = \{\{C\}\}$.



Remark 3.2 [Complexity] If there are n states, there are $\binom{n}{2}$ pairs of states. In one round, we consider all pairs of states, to see if one of their successor pairs has found to be distinguishable. Thus a round has complexity $O(n^2)$. If the algorithm does not stop before, there can not be more than $O(n^2)$ rounds. Thus the complexity of the Table-Filling algorithm is $O(n^4)$.

Exercise 3.4 [Minimisation]

Consider following DFA:

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

1. *Compute the table of distinguishability of the DFA.*
2. *Construct the minimal-state equivalent DFA.*

Exercise 3.5 [Equivalence Relation] *Prove that $\equiv$ is an equivalence relation.*

3.3.1 Implementation of the Table-Filling Algorithm

The JAVA implementation of the Table-Filling algorithm is straightforward. The class `IntPair` implements a state pair.

JAVA-method `dependencies`

```
1  protected LinkedList<IntPair> dependencies() {
2    LinkedList<IntPair> distinguishable = new LinkedList<IntPair>();
3    LinkedList<IntPair> indistinguishable = new LinkedList<IntPair>();
4    int states = dfa.getStates();
5    IntPair.setInstances(states);
6    for (int i = 0; i < states; i++) {
```

```
 7      for (int j = i + 1; j < states; j++) {
 8        if ((dfa.isAccepting(i) && !dfa.isAccepting(j) ||
 9            (dfa.isAccepting(j) && !dfa.isAccepting(i)))) {
10          distinguishable.add(IntPair.get(i, j));
11        } else {
12          indistinguishable.add(IntPair.get(i, j));
13        }
14      }
15    }
16    boolean changed = false;
17    do {
18      changed = false;
19      int k = 0;
20      while (k > -1 && k < indistinguishable.size()) {
21        IntPair pair = indistinguishable.get(k);
22        for (int a = 0; a < INPUTS; a++) {
23          int i = dfa.getTable(pair.i, a);
24          int j = dfa.getTable(pair.j, a);
25          IntPair nextPair;
26          if (i != j) {
27            if (i == SE || j == SE) {
28              if (!(i == SE && j == SE)) {
29                changed = true;
30                k--;
31                indistinguishable.remove(pair);
32                distinguishable.add(pair);
33              }
34            } else {
35              if (i < j) {
36                nextPair = IntPair.get(i, j);
37              } else {
38                nextPair = IntPair.get(j, i);
39              }
40              if (distinguishable.contains(nextPair)) {
41                changed = true;
42                k--;
43                indistinguishable.remove(pair);
44                distinguishable.add(pair);
45                break;
46              }
47            }
48          }
49        }
50        k++;
51      }
52    } while (changed);
53    return indistinguishable;
54  }
```

Remark 3.3 [Complexity] A more careful method can fill the table in $O(n^2)$ time. The idea is to initialise, for each pair of states $\{r,s\}$, a list of pairs $\{p,q\}$ that depend on $\{r,s\}$, i.e. if $\{r,s\}$ is found distinguishable, then $\{p,q\}$ is distinguishable.

For each pair of states $\{p,q\}$ and for each input symbol a, we put $\{p,q\}$ on the list of dependent pairs of $\{\delta(p,a),\delta(q,a)\}$.

If we ever find $\{r,s\}$ to be distinguishable, then we follow the list for $\{r,s\}$. For each pair on that list that is not already distinguishable, we make that pair distinguishable, an we put the pair on

a list of pairs whose lists we must check similarly.

The total work is proportional to the sum of the lengths of the lists. Since the size of the input alphabet is considered a constant, each pair of states is put on $O(1)$ lists. As there are $O(n^2)$ pairs, the total work is $O(n^2)$.

Exercise 3.6 [Table Filling] *Write a* JAVA *method* `tableFilling()` *that returns the linked list of distinguishable pairs of states according to remark 3.3.*

You may use the class IntPair.

3.4 Testing Equivalence of Regular Languages

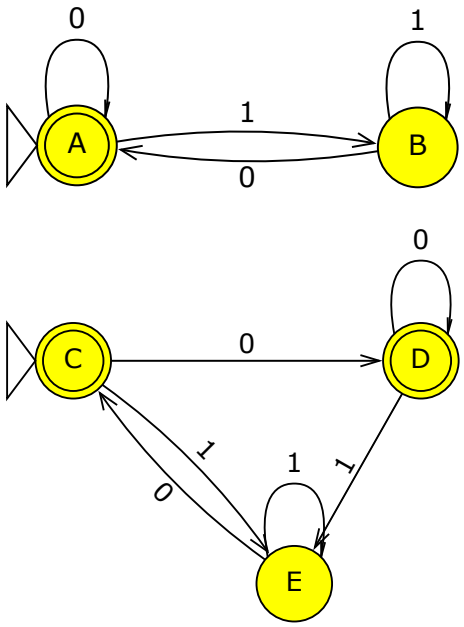
Let L and M be regular languages (each given in some form).

To test if $L = M$

1. Convert both L and M to DFA's.
2. Imagine the DFA that is the union of the two DFA's (never mind there are two start states)
3. If the table filling algorithm says that the two start states are distinguishable, then $L \neq M$, otherwise $L = M$.

Example 3.16 [Equivalent Languages]

Consider following DFA's



The result of the table filling algorithm is

B	x			
C		x		
D		x		
E	x		x	x
	A	B	C	D

Since A and C are equivalent, the automata's are equivalent.

Chapter 4

Context Free Grammars

We have seen that many languages cannot be regular. Thus we need to consider larger classes of languages.

Context-Free Languages (CFL's) played a central role natural languages since the 1950's, and in compilers since the 1960's.

Context-Free Grammars (CFG's) are the basis of BNF-syntax.

Today CFL's are increasingly important for XML and their DTD's or Schema's.

4.1 Context Free Grammars

A grammar is a set of formation rules that describe which strings formed from the alphabet of a formal language are syntactically valid, within the language. A grammar only addresses the location and manipulation of the strings of the language. It does not describe anything else about a language, such as its semantics.

A grammar consists of a set of string rewriting rules with an assigned start symbol; the language described is the set of strings that can be generated by applying these rules arbitrarily, starting with the start symbol.

Definition 4.1 [Context Free Grammars] *A* context-free grammar *is a quadruple* $G = (V, T, P, S)$ *where*

- V *is a finite set of* variables *or* nonterminal symbols.
- T *is a finite set of* terminal symbols.
- P *is a finite set of* productions *of the form* $A \to \alpha$ *, where* A *is a variable and* $\alpha \in (V \cup T)^*$
- S *is a designated variable called the* start symbol.

Remark 4.1 [Grammar] There is a more general definition of a grammar, allowing productions to have the form $\beta \to \alpha$, where $\alpha \in (V \cup T)^*$ and $\beta \in (V \cup T)^*$.

Remark 4.2 [Chomsky Hierarchy] The *Chomsky hierarchy* is a containment hierarchy of classes of formal grammars . This hierarchy of grammars was described by Noam Chomsky[1] in 1956 [Cho56].

[1] Avram Noam Chomsky, born 1928, is an American linguist, philosopher, cognitive scientist, political activist, author, and lecturer. He is an Institute Professor and professor emeritus of linguistics at the Massachusetts Institute of Technology.

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognised by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine (see chapter 5).
- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \to \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \to \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognised by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input) (see chapter 5).
- Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \to \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognised by a non-deterministic pushdown automaton (see section 4.4). Context free languages are the theoretical basis for the syntax of most programming languages.
- Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \to \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a DFA.

Every regular language is context-free, every context-free language, not containing the empty string, is context-sensitive, every context-sensitive language is recursive and every recursive language is recursively enumerable.

Notation 4.1 [BNF] Let A be a nonterminal. If there are more than one A-productions, i.e.:

$$\begin{array}{lcl} A & \to & w_1 \\ A & \to & w_2 \\ & \cdots & \\ A & \to & w_k \end{array}$$

We write

$$A \to w_1 | w_2 | \cdots | w_k$$

This notation is called *BNF*-Notation (Backus-Naur Form).

Notation 4.2 [EBNF] The *EBNF*-Notation is an extension of the BNF-Notations:

- We write $A \to (w)^*$ for the productions $A \to wA|\epsilon$. Zero or more repetitions of the string w
- We write $A \to (w)^+$ for the productions $A \to wA|w$. One or more repetitions of the string w
- We write $A \to (w)^?$ for the productions $A \to w|\epsilon$. Optional w.

Example 4.17 [G_{pal}] Let's define L_{pal} the language of palindromes inductively:

Basis: ϵ, 0, and 1 are palindromes.

Induction: If w is a palindrome, so are $0w0$ and $1w1$.

Circumscription: Nothing else is a palindrome.

The language of palindromes $G_{pal} = (\{P\}, \{0,1\}, A, P)$, where $A = \{P \to \epsilon|1|0|0P0|1P1)$.

Example 4.18 [G_{ari}] The grammar for arithmetic exprexssions

$$G_{ari} = (\{E, N\}, \{0, 1, +, -, *, /, (,)\}, A, E)$$

where A is the following set of productions:

$$\begin{array}{lcl} E & \to & E + E \\ E & \to & E - E \\ E & \to & E * E \\ E & \to & E/E \\ E & \to & (E) \\ E & \to & N \\ N & \to & 1 \\ N & \to & 0 \\ N & \to & N0 \\ N & \to & N1 \end{array}$$

Example 4.19 [G_{re}] Regular expressions over $\{0,1\}$ can be defined by the grammar $G_{re} = (\{E\}, \{0, 1, ., +, ^*, (,)\}, A, E)$ where A is the following set of productions:

$$\begin{array}{lcl} E & \to & 0 \\ E & \to & 1 \\ E & \to & E.E \\ E & \to & E + E \\ E & \to & E^* \\ E & \to & (E) \end{array}$$

4.1.1 The Language of a CFG

There are two ways for testing if a string in Σ^* belong to the language of a given variable:

- *Recursive inference*, using productions from body to head
- *Derivation*, using productions from head to body.

Recursive inference

Example 4.20 [L_{pal}] We show here that the string 001010100 belongs to the language L_{pal} of G_{pal}.

String	Production
001010100	$P \to 1$
0010P0100	$P \to 0P0$
001P100	$P \to 1P1$
00P00	$P \to 0P0$
0P0	$P \to 0P0$
P	

Example 4.21 [L_{ari}] We show here that the string $1 + 1 * 10$ belongs to the language L_{ari} of G_{ari}.

String	Production
$1 + 1 * 10$	$N \to 1$
$N + 1 * 10$	$N \to 1$
$N + N * 10$	$N \to 1$
$N + N * N0$	$N \to N0$
$N + N * N$	$E \to N$
$E + N * N$	$E \to N$
$E + E * N$	$E \to N$
$E + E * E$	$E \to E * E$
$E + E$	$E \to E + E$
E	

Example 4.22 [L_{re}] We show here that the string $0^* + 0.1$ belongs to the language L_{re} of G_{re}.

String	Production
$0^* + 0.1$	$E \to 0$
$E^* + 0.1$	$E \to 0$
$E^* + E.1$	$E \to 1$
$E^* + E.E$	$E \to E^*$
$E + E.E$	$E \to E.E$
$E + E$	$E \to E + E$
E	

Derivation

Definition 4.2 [Derivation] *Let $G = (V, T, P, S)$ be a CFG, and let $\alpha A \beta$ be a string of terminals and nonterminals, with A a nonterminal. Let $A \to \gamma$ be a production of G. The replacement of A by γ in the string $\alpha A \beta$ results in the string $\alpha \gamma \beta$ and is called* derivation.

Notation 4.3 [Derivation] We write

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

or simply

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

if G is well understood

We may extend $\Rightarrow$ to represent zero, one, or many derivation steps, much as the transition function δ of an FA was extended to $\hat{\delta}$.

Definition 4.3 [Reflexive and transitive Closure of Derivation] ***Basis:*** *For any string α of terminals and nonterminals, we say $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. That is any string derives itself.*

Induction: *If $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$ then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. That is if α becomes β by zero or more steps, and one step takes β to γ, then α can become γ.*

Remark 4.3 [Other Interpretation] $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ means that there is a sequence $\gamma_1, \gamma_2, \ldots, \gamma_n$ for some $n \geq 1$, such that

1. $\alpha = \gamma_1$
2. $\beta = \gamma_n$
3. For $i = 1, 2, \ldots, n-1$, we have $\gamma_i \Rightarrow \gamma_{i+1}$

Example 4.23 [L_{pal}] We show here that the string 001010100 belongs to the language of G_{pal}.

$$P \Rightarrow 0P0 \Rightarrow 00P00 \Rightarrow 001P100 \Rightarrow 0010P0100 \Rightarrow 001010100$$

Example 4.24 [G_{ari}] We show here that the string $1 + 1 * 10$ belongs to the language of G_{ari}.

$$E \Rightarrow E+E \Rightarrow N+E \Rightarrow 1+E \Rightarrow 1+E*E \Rightarrow 1+N*E \Rightarrow 1+1*E \Rightarrow 1+1*N \Rightarrow 1+1*1N \Rightarrow 1+1*10$$

Example 4.25 [G_{re}] We show here that the string $0^* + 0.1$ belongs to the language of G_{re}.

$$E \Rightarrow E+E \Rightarrow E^*+E \Rightarrow 0^*+E \Rightarrow 0^*+E.E \Rightarrow 0^*+1.E \Rightarrow 0^*+1.0$$

Remark 4.4 [Derivation Choice] At each step we might have several rules to choose from, e.g. $E + E \Rightarrow E^* + E$ versus $E + E \Rightarrow E + E.E$

Remark 4.5 [Derivation Fails] Not all choices lead to successful derivations of a particular string, for instance $E \Rightarrow E.E$ won't lead to a derivation of $0^* + 0.1$

Definition 4.4 [Leftmost Derivation] *At each step of the* leftmost derivation *the leftmost nonterminal will be replaced by one of its rule-bodies.*

Definition 4.5 [Rightmost Derivation] *At each step of the* rightmost derivation *the rightmost nonterminal will be replaced by one of its rule-bodies.*

Example 4.26 [G_{ari} (Leftmost)] Leftmost derivation of the string $1 + 1 * 10$.

$$\begin{aligned} &E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} N+E \underset{lm}{\Rightarrow} 1+E \underset{lm}{\Rightarrow} 1+E*E \underset{lm}{\Rightarrow} 1+N*E \underset{lm}{\Rightarrow} 1+1*E \underset{lm}{\Rightarrow} 1+1*N \\ &\underset{lm}{\Rightarrow} 1+1*N0 \underset{lm}{\Rightarrow} 1+1*10 \end{aligned}$$

Example 4.27 [G_{ari} (Rightmost)] Rightmost derivation of the string $1 + 1 * 10$.

$$\begin{aligned} &E \underset{rm}{\Rightarrow} E+E \underset{rm}{\Rightarrow} E+E*E \underset{rm}{\Rightarrow} E+E*N \underset{rm}{\Rightarrow} E+E*N0 \underset{rm}{\Rightarrow} E+E*10 \underset{rm}{\Rightarrow} E+N*10 \\ &\underset{rm}{\Rightarrow} E+1*10 \underset{rm}{\Rightarrow} N+1*10 \underset{rm}{\Rightarrow} 1+1*10 \end{aligned}$$

Example 4.28 [G_{re} (Leftmost)] Leftmost derivation for the string $0^* + 0.1$.

$$E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} E^*+E \underset{lm}{\Rightarrow} 0^*+E \underset{lm}{\Rightarrow} 0^*+E.E \underset{lm}{\Rightarrow} 0^*+1.E \underset{lm}{\Rightarrow} 0^*+1.0$$

Example 4.29 [G_{re} (Rightmost)] Rightmost derivation for the string $0^* + 0.1$.

$$E \underset{rm}{\Rightarrow} E+E \underset{rm}{\Rightarrow} E+E.E \underset{rm}{\Rightarrow} E+E.1 \underset{rm}{\Rightarrow} E+0.1 \underset{rm}{\Rightarrow} E^*+1.0 \underset{rm}{\Rightarrow} 0^*+1.0$$

Definition 4.6 [Language of a CFG] *If $G(V, T, P, S)$ is a CFG, then the language of G is*

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

i.e. the set of strings over T^* derivable from the start symbol S.

If G is a CFG, we call $L(G)$ a context-free language.

Example 4.30 [L_{pal}] The language $L(G_{pal})$ is a context-free language.

Theorem 4.1 [L_{pal}]

$$L(G_{pal}) = \{x \in \{0,1\}^* : w = w^R\}$$

Proof: ($\supseteq$)-direction.) Suppose $w = w^R$. We show by induction on $|w|$ that $w \in L(G_{pal})$:

Basis: $|w| = 0$ or If $|w| = 1$.. Then w is ϵ, 0 or 1. Since $P \to \epsilon$, $P \to 0$, and $P \to 1$ are productions, we conclude that $P \underset{G_{pal}}{\overset{*}{\Rightarrow}} w$ in all base cases.

Induction: Suppose $|w| \geq 2$. Since $w = w^R$, we have $w = 0x0$, or $w = 1x1$, and $x = x^R$. We know from induction hypothesis that $P \underset{G_{pal}}{\overset{*}{\Rightarrow}} x$.

Then either

$$P \underset{G_{pal}}{\Rightarrow} 0P0 \underset{G_{pal}}{\overset{*}{\Rightarrow}} 0x0 = w$$

i.e. $w \in L(G_{pal})$ or

$$P \underset{G_{pal}}{\Rightarrow} 1P1 \underset{G_{pal}}{\overset{*}{\Rightarrow}} 1x1 = w$$

i.e. $w \in L(G_{pal})$

($\subseteq$-direction) We assume that $w \in L(G_{pal})$ and must show that $w = w^R$.

Since $w \in L(G_{pal})$, we have $P \underset{G_{pal}}{\overset{*}{\Rightarrow}} w$.

We do an induction on the length of $\underset{G_{pal}}{\overset{*}{\Rightarrow}}$

Basis: The derivation $P \underset{G_{pal}}{\overset{*}{\Rightarrow}} w$ is done in one step. Then w must be ϵ, 0, or 1, all palindromes.

Induction: Let $n \geq 1$, and suppose the derivation takes $n+1$ steps. Then we must have

$$P \underset{G_{pal}}{\Rightarrow} 0P0 \underset{G_{pal}}{\overset{*}{\Rightarrow}} 0x0 = w$$

or

$$P \underset{G_{pal}}{\Rightarrow} 1P1 \underset{G_{pal}}{\overset{*}{\Rightarrow}} 1x1 = w$$

where the second derivation is done in n steps.

By induction hypothesis x is a palindrome, and the inductive proof is complete. □

Definition 4.7 [Sentential Form] *Let $G(V,T,P,S)$ be a CFG, and $\alpha \in (V \cup T)^*$. If*

$$S \overset{*}{\Rightarrow} \alpha$$

we say that α is a sentential form.

If $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$, *we say that* α *is a* left-sentential form *and if* $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$, *we say that* α *is a* right-sentential form

Definition 4.8 [Sentential Form] $L(G)$ *is the set of sentential forms that are in* T^*.

Example 4.31 [G_{re}]

$0^* + E.E$ is a left-sentential form.

$$E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} E^* + E \underset{lm}{\Rightarrow} 0^* + E \underset{lm}{\Rightarrow} 0^* + E.E$$

$E + 0.1$ is a right-sentential form.

$$E \underset{rm}{\Rightarrow} E + E \underset{rm}{\Rightarrow} E + E.E \underset{rm}{\Rightarrow} E + E.1 \underset{rm}{\Rightarrow} E + 0.1$$

Exercise 4.1 [L_{eq}] *Design a context-free grammar for* L_{eq} *be the language of strings with equal number of zero's and one's.*

Exercise 4.2 [Parentheses] *Design a grammar for all and only all strings of round and square parentheses that are balanced.*

Exercise 4.3 [Lisp] *In* LISP *there are two fundamental data types:* atoms *and* lists. *A list is a finite ordered sequence of elements, where each element is in itself either an atom or a list, and an atom is a number or a symbol. A list may be empty. A symbol is essentially a unique named item, written as an alphanumeric string in source code, and used either as a variable name or as a data item in symbolic processing.*

For example, the list `(FOO (BAR 1) 2 ())` *contains four elements: the symbol* `FOO`*, the list* `(BAR 1)`*, the number 2 and the empty list* `()`*.*

Design a grammar for LISP.

Exercise 4.4 [Regular Expression] *The following grammar generates the language of a regular expression:*

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A|\epsilon \\ B & \rightarrow & 0B|1B|\epsilon \end{array}$$

1. *Which regular expression is described by the grammar?*
2. *Give leftmost and rightmost derivation of* 00101

Exercise 4.5 [CFG] *Give a context-free grammar that generates the language*

$$L = \{a^n b^{n+1} : n \in \mathbb{N}\}$$

Exercise 4.6 [Derivation] *Consider the context free grammar:*

$$S \rightarrow SS+ \quad | \quad SS* \quad | \quad a$$

Give the leftmost and the rightmost derivation for the string $aa + aa + *$.

4.2 Parse Trees

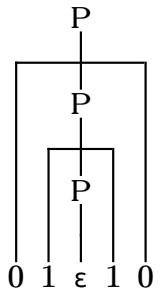
Definition 4.9 [Parse Tree] *Let $G(V, T, P, S)$ be a CFG. A tree is a* parse tree *for G if:*

1. *Each interior node is labelled by a variable in V.*
2. *Each leaf is labelled by a symbol in $V \cup T \cup \{\epsilon\}$. Any ϵ-labelled leaf is the only child of its parent.*
3. *If an interior node is labelled A, and its children (from left to right) labelled*

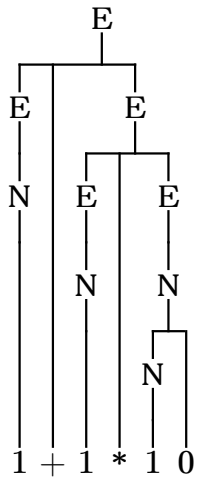
$$X_1, X_2, \ldots, X_k$$

then $A \to X_1 X_2 \ldots X_k \in P$.

Example 4.32 [G_{pal}] Following parse tree shows the derivation of $P \overset{*}{\underset{G_{pal}}{\Rightarrow}} 0110$



Example 4.33 [G_{ari}] Following parse tree shows the derivation of $E \overset{*}{\underset{G_{ari}}{\Rightarrow}} 1 + 1 * 10$



Definition 4.10 [Yield of a Parse Tree]
The yield *of a parse tree is the string of leaves from left to right.*
Important are those parse trees where:

1. The yield is a terminal string.
2. The root is labelled by the start symbol

We shall see the the set of yields of these important parse trees is the language of the grammar.

4.2.1 Inference, Derivation and Parse Trees

Theorem 4.2 [Inference, Derivation and Parse Trees] *Let $G(V, T, P, S)$ be a grammar, and let A be a variable in G. The following are equivalent:*

1. *The recursive inference procedure determines that terminal strings w in the language of variable A.*
2. $A \stackrel{*}{\Rightarrow} w$.
3. $A \underset{lm}{\stackrel{*}{\Rightarrow}} w$.
4. $A \underset{rm}{\stackrel{*}{\Rightarrow}} w$.
5. *There is a parse tree with root A and yield w.*

From Inference to Parse Trees

Theorem 4.3 [From Inference to Parse Trees] *Let $G(V, T, P, S)$ be a CFG, and suppose we can show w to be in the language of a variable A. Then there is a parse tree for G with root A and yield w.*

Proof: We do an induction of the length of the inference.

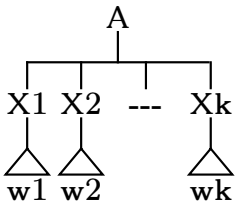
Basis: One step. Then we must have used a production $A \to w$. The desired parse tree is then

Induction: w is inferred in $n + 1$ steps. Suppose the last step was based on a production

$$A \to X_1 X_2 \cdots X_k$$

where $X_i \in V \cup T$. We break w up as $w_1 w_2 \cdots w_k$ where $w_i = X_i$, when $X_i \in T$, and when $X_i \in V$; then w_i was previously inferred being in X_i, in at most n steps.

By the induction hypothesis there are parse trees i with root X_i and yield w_i. Then the following is a parse tree for G with root A and yield w:



□

From Parse Trees to Derivation

Example 4.34 [From Parse Trees to Derivation] Let consider the expression grammar G_{ari}. It is easy to check that there is a derivation

$$E \Rightarrow I \Rightarrow N0 \Rightarrow 10$$

As a result, for any strings α and β, it is also true that

$$\alpha E\beta \Rightarrow \alpha N\beta \Rightarrow \alpha N0\beta \Rightarrow \alpha 10\beta$$

The justification is that we can make the same replacements of production bodies for heads in the context of α and β as we can in isolation.

For instance if we have a derivation that begins $E \Rightarrow E+E \Rightarrow E+(E)$ we could apply the derivation of 10 from the second E by setting α to be $E+"("$ and setting β to be $")"$. The last derivation would then continue

$$E+(E) \Rightarrow E+(N) \Rightarrow E+(N0) \Rightarrow E+(10)$$

Theorem 4.4 [From Parse Trees to Derivation] *Let $G(V,T,P,S)$ be a CFG, and suppose there is a parse tree with root labelled A and yield w. Then $A \overset{*}{\underset{lm}{\Rightarrow}} w$ in G.*

Proof: We do an induction on the height of the parse tree.

Basis: Height is 1. The tree must look like

Consequently $A \rightarrow w$ and $A \underset{lm}{\Rightarrow} w$.

Induction: Height is $n+1$. The tree must look like

Then $w = w_1 w_2 \cdots w_k$, where

1. If $X_i \in T$, then $w_i = X_i$.
2. If $X_i \in V$, then $X_i \overset{*}{\underset{lm}{\Rightarrow}} w_i$ in G by the induction hypothesis.

Now we construct $A \overset{*}{\underset{lm}{\Rightarrow}} w$ by an (inner) induction by showing that

$$\forall i : A \overset{*}{\underset{lm}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

Basis: Let $i = 0$. We already know that

$$A \underset{lm}{\overset{*}{\Rightarrow}} X_1 X_2 \cdots X_k$$

Induction: Make the induction hypothesis that

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k$$

(Case 1:) $X_i \in T$. Do nothing, since $X_i = w_i$ gives us

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_1 w_2 \cdots w_i X_{i+1} X_{i+2} \cdots X_k$$

(Case 2:) $X_i \in V$. By the induction hypothesis there is a derivation

$$X_i \underset{lm}{\Rightarrow} \alpha_1 \underset{lm}{\Rightarrow} \alpha_2 \underset{lm}{\Rightarrow} \cdots \underset{lm}{\Rightarrow} w_i$$

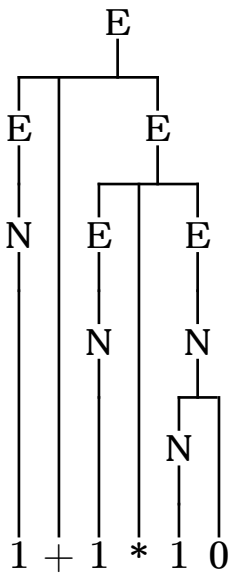
By the context-free property of derivations we can proceed with

$$\begin{array}{ll} A & \underset{lm}{\overset{*}{\Rightarrow}} \quad w_1 w_2 \cdots w_{i-1} X_i X_{i+1} \cdots X_k \\ & \underset{lm}{\Rightarrow} \quad w_1 w_2 \cdots w_{i-1} \alpha 1 X_{i+1} \cdots X_k \\ & \underset{lm}{\Rightarrow} \quad w_1 w_2 \cdots w_{i-1} \alpha 2 X_{i+1} \cdots X_k \\ & \underset{lm}{\Rightarrow} \quad \cdots \\ & \underset{lm}{\Rightarrow} \quad w_1 w_2 \cdots w_{i-1} w_i X_{i+1} \cdots X_k \end{array}$$

□

Remark 4.6 [Rightmost Derivation] We could also prove $A \underset{rm}{\overset{*}{\Rightarrow}} w$ in a similar way.

Example 4.35 [G_{ari}] Let construct the leftmost derivation for the tree



Suppose we have inductively constructed the derivation

$$E \underset{lm}{\Rightarrow} I \underset{lm}{\Rightarrow} 1$$

corresponding to the leftmost subtree, and the leftmost derivation

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} 1 * E \underset{lm}{\Rightarrow} 1 * 1I \underset{lm}{\Rightarrow} 1 * 10$$

corresponding to the rightmost subtree.

For the derivation corresponding to the whole tree we start with $E \underset{lm}{\Rightarrow} E + E$ and expand the first E with the first derivation and the second E with the second derivation:

$$E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} I + E \underset{lm}{\Rightarrow} 1 + E \underset{lm}{\Rightarrow} 1 + E * E \underset{lm}{\Rightarrow} 1 + I * E \underset{lm}{\Rightarrow} 1 + 1 * E \underset{lm}{\Rightarrow} 1 + 1 * I$$
$$\underset{lm}{\Rightarrow} 1 + 1 * 1I \underset{lm}{\Rightarrow} 1 + 1 * 10$$

From Derivation to Recursive Inference

Theorem 4.5 [From Derivation to Recursive Inference] *Let $G(V, T, P, S)$ be a CFG. Suppose $A \overset{*}{\Rightarrow} w$, and that w is a string of terminals. Then we can infer that w is in the language of variable A.*

Proof: We do an induction on the length of the derivation $A \overset{*}{\Rightarrow} w$.

Basis: One step. If $A \Rightarrow w$ there must be a production $A \to w$ in P. Then we can infer that w is in the language of A.

Induction: Suppose $A \overset{*}{\Rightarrow} w$ in $n + 1$ steps. Write the derivation as

$$A \Rightarrow X_1 X_2 \cdots X_k \overset{*}{\Rightarrow} w$$

The we can break w as $w_1 w_2 \cdots w_k$ where $X_i \overset{*}{\Rightarrow} w_i$. Furthermore, $X_i \overset{*}{\Rightarrow} w_i$ can use at most n steps.

Now we have a production $A \to X_1 X_2 \cdots X_k$, and we know by the induction hypothesis that we can infer w_i to be in the language of X_i. Therefore we can infer $w_1 w_2 \cdots w_k$ to be in the language of A. □

4.3 Ambiguity in Grammars and Languages

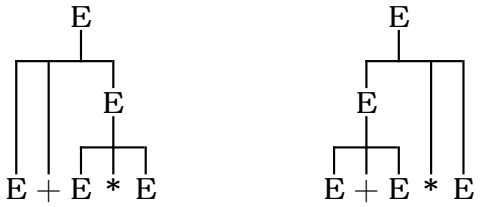
Example 4.36 [G_{ari}] In the grammar G_{ari}, the sentential form $E + E * E$ has two derivations:

$$E \Rightarrow E + E \Rightarrow E + E * E$$

and

$$E \Rightarrow E * E \Rightarrow E + E * E$$

This gives us two different parse trees:



The first parse tree states that the multiplication has a higher priority than the addition. The second parse tree states exactly the contrary.

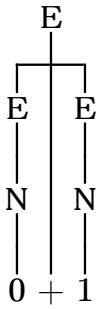
In the same grammar, the string $0 + 1$ has several derivations

$$E \Rightarrow E + E \Rightarrow N + E \Rightarrow 0 + E \Rightarrow 0 + N \Rightarrow 0 + 1$$

and

$$E \Rightarrow E + E \Rightarrow E + I \Rightarrow E + 1 \Rightarrow I + 1 \Rightarrow 0 + 1$$

but only one parse tree



The mere existence of several derivations is not dangerous, it is the existence of several parse trees that ruins a grammar.

Definition 4.11 [Ambiguous Grammar] *Let $G(V, T, P, S)$ be a CFG.We say that G is* ambiguous *if there is a string in T^* that has more than one parse tree.*

If every string in $L(G)$ has at most one parse tree, G is said to be unambiguous.

4.3.1 Removing Ambiguity in Grammars

Good news: Sometimes we can remove ambiguity *by hand*

Bad news: There is no algorithm to do it

More bad news: Some CFL's have only ambiguous CFG's

Example 4.37 [G_{ari}] Consider the grammar G_{ari}. There are two problems:

1. There is no precedence between $*$ and $+$.
2. There is no grouping of sequences of operators, e.g. is $E + E + E$ meant to be $E + (E + E)$ or $(E + E) + E$.

Solution: We introduce more variables, each representing expressions of same *binding strength*.

1. A factor F is an expression that cannot be broken apart by an adjacent $*$, $/$, $-$ or $+$. Our factors are

(a) Numbers N

(b) A parenthesised expression.

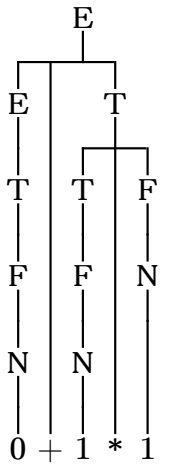
2. A term T is an expression that cannot be broken by $+$ and $-$. For instance $1 * 10$ can be broken by $11*$ i.e. $11 * 1 * 10 = (11 * 1) * 10$. It cannot be broken by $+$, since e.g. $11 + 1 * 10$ is (by precedence rules) same as $11 + (1 * 10)$.

3. The rest are expressions, i.e. they can be broken apart with $*$, $/$, $-$ or $+$.

We modify now the set P of productions of G_{ari} as follows

$$
\begin{array}{lcl}
E & \rightarrow & T \\
E & \rightarrow & E + T \\
E & \rightarrow & E - T \\
T & \rightarrow & F \\
T & \rightarrow & T * F \\
T & \rightarrow & T/F \\
F & \rightarrow & (E) \\
F & \rightarrow & N \\
N & \rightarrow & 1 \\
N & \rightarrow & 0 \\
N & \rightarrow & 0N \\
N & \rightarrow & 1N
\end{array}
$$

Now the only parse tree for $0 + 1 * 1$ will be



Exercise 4.7 [Ambiguity] *Is the following grammar ambiguous (I represents a statement of a programming language, and E is a Boolean expression)?*

$$
\begin{array}{lcl}
I & \rightarrow & \mathit{if}\ \ E\ \ \mathit{then}\ \ I \\
 & | & \mathit{if}\ \ E\ \ I\ \ \mathit{then}\ \ I\ \ \mathit{else}\ \ I \\
 & | & s_1 \ | \ s_2 \ | \ \ldots \ | \ s_n \\
E & \rightarrow & e_1 \ | \ e_2 \ | \ \ldots \ | \ e_m
\end{array}
$$

Exercise 4.8 [Ambiguity]

Consider the grammar

$$S \rightarrow aS \ \ | \ \ aSbS \ \ | \ \ \epsilon$$

This grammar generates all and only all strings of a's and b's such that every prefix has at least as many a's as b's. This grammar is ambiguous.

1. *Show that the string aab has two rightmost derivations.*
2. *Find an unambiguous grammar.*

4.4 Pushdown Automata

A pushdown automaton (PDA) is essentially an ϵ-NFA with a stack.
On a transition the PDA:

1. Consumes an input symbol.
2. Goes to a new state (or stays in the old).
3. Replaces the top of the stack by any string (does nothing, pops the stack, or pushes a string onto the stack)

Example 4.38 [Reverse] Let's consider

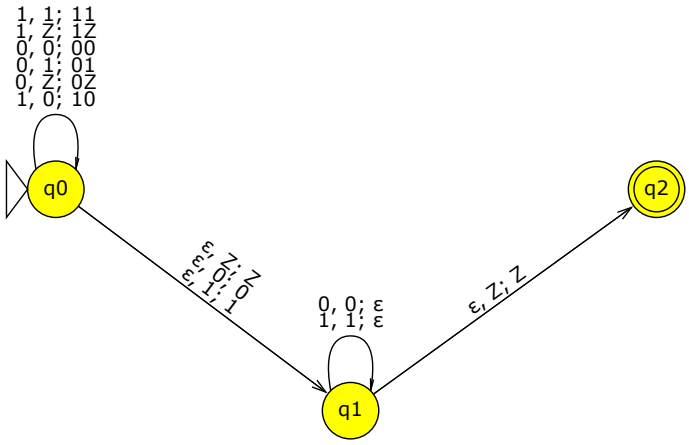
$$L_{ww^R} = \{ww^R : w \in \{01\}^*\}$$

with grammar

$$P \to 0P0, P \to 1P1, P \to \epsilon$$

A PDA for L_{ww^R} has three states, and operates as follows:

1. Guess that you are reading w. Stay in state 0, and push the input symbol onto the stack.
2. Guess that you're in the middle of ww^R. Go spontaneously to state 1.
3. You're now reading the head of w^R. Compare it to the top of the stack. If they match, pop the stack, and remain in state 1. If they don't match, go to sleep.
4. If the stack is empty, go to state 2 and accept.



Definition 4.12 [Pushdown Automaton] *A* pushdown Automaton *(*PDA*) is a seven-tuple:*

$$P = (Q, \Sigma, \delta, \Gamma, q_0, Z_0, F)$$

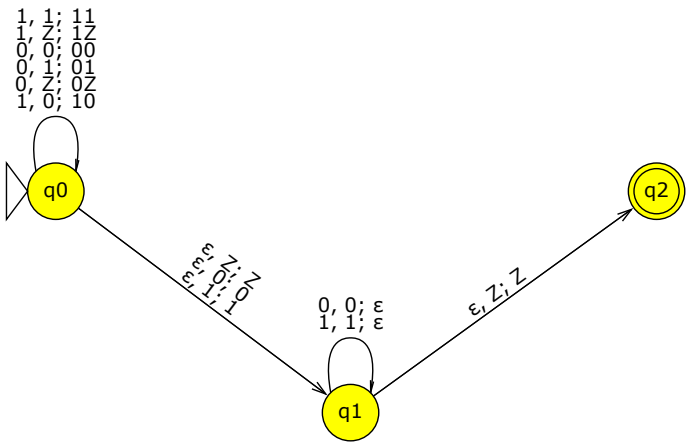
where

- Q *is a finite set of states,*
- Σ *is a finite input alphabet,*
- Γ *is a finite stack alphabet,*
- $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \to 2^{Q \times \Gamma^*}$ *is the transition function.* δ *takes as argument a triple* $\delta(q, a, X)$*, where:*

  1. q *is a state in* Q*.*
  2. a *is either an input symbol in* Σ *or* $a = \epsilon$*.*
  3. X *is a stack symbol in* Γ*.*

  The output of δ *is a finite set of pairs* (p, γ)*, where* p *is the new state and* γ *is the string of stack symbols that replaces* X *at the top of the stack. Note that* γ *may be the empty, i.e.* $\gamma = \epsilon$*.*
- q_0 *is the start state,*
- $Z_0 \in \Gamma$ *is the start symbol for the stack, and*
- $F \in Q$ *is the set of accepting states.*

Example 4.39 [Reverse] The PDA for L_{ww^R}



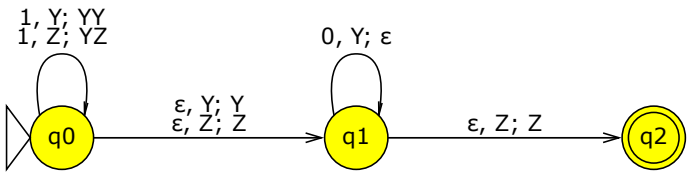
is actually the seven-tuple

$$P = \{\{q_0, q_1, q_2\}, \{0, 1\}, \{01, Z_0\}, q_0, Z_0, \{q_2\}$$

where δ is given by the following table (set brackets missing):

	$0, Z_0$	$1, Z_0$	$0, 0$	$0, 1$	$1, 0$	$1, 1$	ϵ, Z_0	$\epsilon, 0$	$\epsilon, 1$
$\to q_0$	$q_0, 0Z_0$	$q_0, 1Z_0$	$q_0, 00$	$q_0, 01$	$q_0, 10$	$q0, 11$	q_1, Z_0	$q_1, 0$	$q_1, 1$
q_1			q_1, ϵ	q_1, ϵ			q_2, Z_0		
$*q_2$									

Example 4.40 [L10] PDA for $L_{10} = \{w \in \{0, 1\} : w = 1^n 0^n\}$



4.4.1 Instantaneous Description

A PDA goes from configuration to configuration when consuming input.

To reason about PDA computation, we use instantaneous descriptions of the PDA.

Definition 4.13 [ID] *An* ID *or* instantaneous description *is a triple*

$$(q, w, \gamma)$$

where q is the state, w the remaining input, and γ the stack contents.

Definition 4.14 [Move] *Let $P = (Q\Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. Then $\forall w \in \Sigma^*, \beta \in \Gamma^*$, if*

$$(p, \alpha) \in \delta(q, a, X)$$

we write

$$(q, aw, X\beta) \underset{P}{\vdash} (p, w, \alpha\beta)$$

We write also $\vdash$ if P is well understood.

The following properties hold:

1. If an ID sequence is a legal computation for a PDA, then so is the sequence obtained by adding an additional string at the end of component number two.
2. If an ID sequence is a legal computation for a PDA, then so is the sequence obtained by adding an additional string at the bottom of component number three.
3. If an ID sequence is a legal computation for a PDA, and some tail of the input is not consumed, then removing this tail from all ID's result in a legal computation sequence.

Definition 4.15 [Moves] *We use $\overset{*}{\underset{P}{\vdash}}$ or $\overset{*}{\vdash}$ to represent zero or more moves of a PDA P:*

Basis: *$I \overset{*}{\vdash} I$ for any ID I.*

induction: *$I \overset{*}{\vdash} J$ if there exists some ID K such that $I \vdash K$ and $K \overset{*}{\vdash} J$*

4.4.2 Language of a PDA

Definition 4.16 [Language: Acceptance by Final State] *Let $P = (Q\Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The* language accepted by P by final state *is*

$$F(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha), q \in F\}$$

Example 4.41 [Reverse] The PDA of example 4.38 accepts exactly L_{ww^R}

Let P be the PDA, We prove that $L(P) = L_{ww^R}$

($\Subset$-direction) Let $x \in L_{ww^R}$. Then $x = ww^R$, and the following is a legal sequence

$$(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$$

($\subset$-direction) Observe that the only way the PDA can enter q_2 is if it is in state q_1 with an empty stack.

Thus it is sufficient to show that if $(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0)$ then $x = ww^R$, for some word w.

We'll show by induction on $|x|$ that

$$(q_0, x, Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0) \Rightarrow x = ww^R$$

Basis: If $x = \epsilon$ then x is a palindrome.

Induction: Suppose $x = a_1 a_2 \cdots a_n$, where $n > 0$, and the induction hypothesis holds for shorter strings.

There are two moves for the PDA from ID (q_0, x, α):

Move 1: The spontaneous $(q_0 x, \alpha) \vdash (q_1, x, \alpha)$. Now $(q_1 x, \alpha) \overset{*}{\vdash} (q_1 \epsilon, \alpha)$ implies that $|\beta| < |\alpha|$, which implies $\beta \neq \beta|$.

Move 2: Loop and push $(q_0, a_1 a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1 \alpha)$.

In this case there is a sequence

$$(q_0, a_1 a_2 \cdots a_n, \alpha) \vdash (q_0, a_2 \cdots a_n, a_1 \alpha) \vdash \cdots \vdash (q_0, a_n, a_1 \alpha) \vdash (q_1, \epsilon, \alpha)$$

Thus $a_1 = a_n$ and

$$(q_0, a_2 \cdots a_n, a_1 \alpha) \overset{*}{\vdash} (q_1, a_n, a_1 \alpha)$$

Now we remove a_n and we get

$$(q_0, a_2 \cdots a_{n-1}, a_1 \alpha) \overset{*}{\vdash} (q_1, \epsilon, a_1 \alpha)$$

By induction hypothesis $a_2 \cdots a_{n-1} = yy^R$. Then $x = a_1 yy^R a_n$ is a palindrome. □

Definition 4.17 [Language: Acceptance by Empty Stack] *Let $P = (Q\Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. The* language accepted by P by empty stack *is*

$$N(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

Theorem 4.6 [N/F-Equivalence] *If $L = N(P_N)$ for some PDA P_N, then there is a PDA P_F such that $L = F(P_F)$.* □

4.4.3 CFG/PDA-Equivalence

Theorem 4.7 [CFG/PDA-Equivalence] *A language is generated by a CFG if and only if it is accepted by a PDA by empty stack if and only if it is accepted by a PDA by final state.* □

From CFG to PDA

We construct a PDA P such that $L(G) = L(P)$:

We write left-sentential forms as

$$xA\alpha$$

where A is the leftmost variable in the form. For instance,

$$\underbrace{\underbrace{1+}_{x}\underbrace{E}_{A}\underbrace{*E}_{\alpha}}_{\text{tail}}$$

Let $xA\alpha \underset{lm}{\Rightarrow} x\beta\alpha$. This corresponds to the PDA first having consumed x and having $A\alpha$ on the stack, and then on ϵ it pops A and pushes β.

More formally, let y, such that $w = xy$. Then the PDA goes non-deterministically from configuration $(q, y, A\alpha)$ to configuration $(q, y, \beta\alpha)$.

At $(q, y, \beta\alpha)$ the PDA behaves as before, unless there are terminals in the prefix of β. In that case, the PDA pops them, provided it can consume matching input.

If all guesses are right, the PDA ends up with empty stack and input.

Formally, let $G = (V, T, Q, S)$ be a CFG. Define P_G as

$$(\{q\}, T, V \cup T, \delta, q, S)$$

where

$$\delta(q, \epsilon, A) = \{(q, \beta) : A \to \beta \in Q\}$$

for $A \in V$, and

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

for $a \in T$. □

Example 4.42 [Expressions] PDA for $G_{ari} = (\{E, N\}, \{0, 1, +, -, *, /, (,)\}, A, E)$, the grammar of example 4.18

The set of input symbols for the PDA is $\{0, 1, +, -, *, /, (,)\}$. These eight symbols and the symbols N and E form the stack alphabet. The transition function for the PDA is:

1. $\delta(q, \epsilon, N) = \{(q, 1), (q, 0), (q, N1), (q, N0)\}$
2. $\delta(q, \epsilon, E) = \{(q, E + E), (q, E - E), (q, E * E), (q, E/E), (q, (E))\}$
3. $\delta(q, 1, 1) = \{(q, \epsilon)\}$, $\delta(q, 0, 0) = \{(q, \epsilon)\}$, $\delta(q, +, +) = \{(q, \epsilon)\}$, $\delta(q, -, -) = \{(q, \epsilon)\}$, $\delta(q, *, *) = \{(q, \epsilon)\}$, $\delta(q, /, /) = \{(q, \epsilon)\}$, $\delta(q, (, () = \{(q, \epsilon)\}$ and $\delta(q,),)) = \{(q, \epsilon)\}$.

From PDA to CFG

We can construct a CFG G such that $L(G) = L(P)$:

First notice that when a PDA can consume $x = x_1 x_2 \ldots x_k$, the stack will be empty. While emptying the stack, the states may change.

We shall define a grammar with variables of the form $[p_{i-1} Y_i p_i]$ representing going from p_{i-1} to p_i with net effect of popping the string Y_i.

Formally, let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ be a PDA.

Define $G = (V, \Sigma, R, S)$, where

$$V = \{[pXq] : \{p, q\} \in Q, X \in \Gamma\} \cup \{S\}$$

$$
\begin{array}{rcll}
R & = & \{ & S \to [q_0 Z_0 p] : p \in Q\} \cup \\
 & & \{ & [qXr_k] \to a[rY_1r_1]\ldots[r_{k-1}Y_kr_k] : \\
 & & & a \in \Sigma \cup \{\epsilon\}, \\
 & & & \{r_1, \ldots, r_k\} \subseteq Q, \\
 & & & (r, Y_1Y_2 \cdots Y_k) \in \delta(q, a, X) \\
 & & \} &
\end{array}
$$

Example 4.43 [PDA to CFG]

$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$

	e, Z	i, Z
$\to q$	q, ϵ	q, ZZ

We construct the grammar $G = (V, \{i, e\}, RS)$, where $V = \{[qZq], S\}$, and

$$R = \{[qZq] \to i[qZq][qZq], [qZq] \to e\}$$

If we replace $[qZq]$ by A we get the productions

$$S \to A, A \to iAA|e$$

Example 4.44 [PDA to CFG]

Let $P = (\{p, q\}, \{0, 1\}, \{X, Z\}, \delta, q, Z)$, where δ is given by

	$0, Z$	$1, Z$	$0, X$	$1, X$	ϵ, X
$\to q$		q, XZ	p, X	q, XX	q, ϵ
$*p$	q, Q			p, ϵ	

We get $G = (V, \{0, 1\}, RS)$, where $V = \{[pXp], [pXq], [pZp], [pZq], S\}$ and the productions in R are

$S \to [qZq]|[qZ0p]$

From rule $\delta(q, 1, Z) = \{(q, XZ)\}$

$[qZq] \to 1[qXq][qZq]$
$[qZq] \to 1[qXp][pZq]$
$[qZp] \to 1[qXq][qZp]$
$[qZp] \to 1[qXp][pZp]$

From rule $\delta(q, 1, X) = \{(q, XX)\}$

$[qXq] \to 1[qXq][qXq]$
$[qXq] \to 1[qXp][pXq]$
$[qXp] \to 1[qXq][qXp]$
$[qXp] \to 1[qXp][pXp]$

From rule $\delta(q, 0, X) = \{(p, X)\}$

$[qXq] \to 0[pXq]$
$[qXp] \to 0[pXp]$

From rule $\delta(q, \epsilon, X) = \{(q, \epsilon)\}$

$[qXq] \to \epsilon$

From rule $\delta(p, 1, X) = \{(p, \epsilon)\}$

$[pXp] \to 1$

From rule $\delta(p, 0, Z) = \{(q, Z)\}$

$[pZq] \to 0[qZq]$
$[pZp] \to 0[qZp]$

We make the grammar easier to read:

$S \to A$
$S \to B$
$A \to 1CA$
$A \to 1DE$
$B \to 1CB$
$B \to 1DF$
$C \to 1CC$
$C \to 1DG$
$D \to 1CD$
$D \to 1DH$
$C \to 0G$
$D \to 0H$
$C \to \epsilon$
$H \to 1$
$E \to 0A$
$F \to 0B$

The variable G never appears on the lefthand side of a production, thus all productions containing G can be removed.

$S \to A$
$S \to B$
$A \to 1CA$
$A \to 1DE$
$B \to 1CB$
$B \to 1DF$
$C \to 1CC$
$D \to 1CD$
$D \to 1DH$
$D \to 0H$
$C \to \epsilon$
$H \to 1$
$E \to 0A$
$F \to 0B$

Exercise 4.9 [PDA] *Construct a pushdown automaton that recognises the language defined over* $\{a, b, c\}$

$$L = \{wcw^R \quad : \quad w \in \{a, b\}^*\}$$

Exercise 4.10 [PDA] *Show that if P is a PDA, then there is a one-state PDA P_1 such that $L(P) = L(P_1)$.*

Exercise 4.11 [L_{eq}] *Design a PDA to accept the language of the set of stings of 0's and 1's with an equal number of 0's and 1's.*

4.5 Simplification of CFG's

A symbol is useless if it does not appear in any derivation $S \stackrel{*}{\Rightarrow} w$ for start symbol S and terminal w.

Definition 4.18 [Useful Symbol] *A symbol X is* useful *for a grammar $G = (V, T, P, S)$, if there is a derivation*

$$S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$$

for a terminal string w. Symbols that are not useful are called useless.

Definition 4.19 [Generating Symbol] *A symbol X is* generating *if $X \overset{*}{\Rightarrow} w$, for some $w \in T^*$.*

Definition 4.20 [Reachable Symbol] *A symbol X is* reachable *if $S \overset{*}{\Rightarrow} \alpha X \beta$, for some $\{\alpha, \beta\} \subseteq (V \cup T)^*$.*

It turns out that if we eliminate non-generating symbols first, and then non-reachable ones, we will be left with only useful symbols.

Example 4.45 [Useless Symbols]

Let G be

$$S \to AB | a, A \to b$$

S and A are generating, B is not. If we eliminate B we have to eliminate $S \to AB$, leaving the grammar

$$S \to a, A \to b$$

Now only S is reachable. Eliminating A and b leaves us with

$$S \to a$$

with language $\{a\}$.

Note that, if we eliminate non-reachable symbols first, we find that all symbols are reachable. From

$$S \to AB | a, A \to b$$

we then eliminate B as non-generating, and are left with

$$S \to a, A \to b$$

that still contains useless symbols

Exercise 4.12 [CFG]

Construct the associated CFG for following PDA:

	$0, Z$	$1, Z$	$0, 0$	$0, 1$	$1, 0$	$1, 1$	ϵ, Z
$\to p$	$p, 0Z$	$p, 1Z$	$p, 00$	p, ϵ	p, ϵ	$p, 11$	q, Z
$*q$							

Try to simplify the result as much as possible.

4.6 The Pumping Lemma for CFG's

Theorem 4.8 [Pumping Lemma] *Let L be a context free language. Then there exists a constant n such that if z is any string in L such that $|z|$ is at least n, then we can write $z = uvwxy$, subject to the following conditions:*

1. *$|vwx| \leq n$. That is, the middle portion is not too long.*
2. *$vx \neq \epsilon$. Since v and x are pieces to be* pumped*, this condition says that at least one of the stings we pump must not be empty.*
3. *For all $i \geq 0$, uv^iwx^iy is in L. That is, the two strings v and x may be* pumped *any number of times, including 0, and the resulting string will still be a member of L.*

□

Example 4.46 [Pumping lemma] Let $L = \{0^i1^i2^i : i \geq 1\}$, Then L is not a context-free language.

Suppose L were context-free, there is an integer n given to us by the pumping lemma. Let us pick $z = 0^n1^n2^n$. We break z as $z = uvwxy$, where $|vwx| \leq n$ and v and x are not both ϵ. Then we know that w cannot involve both 0's and 2's, since the last 0 and the first 2 are separated by $n+1$ positions. We show that L contains some strings not in L, thus contradicting the assumption that L is context-free.

1. vwx has no 2's. Then vx consists of only 0' and 1's, and has at least one of these symbols. Then uwy, which would have to be in L by the pumping lemma, has n 2's, but has fewer than n 0's or fewer then n 1's, or both. It therefore does not belong to L, and we conclude that L is not a CFL.
2. vwx has no 0's. Similarly, uwy has no 0's, but less 1's or less 2's. It therefore is not in L.

Exercise 4.13 [Non CFG] *Explain why the language $L = \{wcw : w \in \{a,b\}^n\}$, defined over the alphabet $\{a,b,c\}$, is not context-free.*

Exercise 4.14 [Non CFG] *Explain why we cannot apply the pumping lemma to the language $L = \{0^n1^n : n \in \mathbb{N}\}$.*

4.7 Properties of CFL's

Theorem 4.9 [Properties of CFL's] *Let L, L_1 and L_2 be context-free languages and let R be a regular language, then*

- *$L_1 \cup L_2$ is a context-free language.*
- *$L_1.L_2$ is a context-free language.*
- *L^R, the reversal of L is a context-free language.*
- *$L \cap R$ is a context-free language.*
- *$L - R$ is a context-free language.*
- *$\overline{L}$ is not necessarily a context-free language.*
- *$L_1 - L_2$ is not necessarily a context-free language.*

Chapter 5

Turing Machines

5.1 Problems that Computers Cannot Solve

It is important to know whether a program is correct, namely that it does what we expect. It is easy to see that the following JAVA program

```
public class Main {
  void main(String [] args)  {
    System.out.println("hello, world");
  }
}
```

prints `hello, world` and terminates.

But about following program:

```
public class Main {
  int exp(int i, int n) {
    int r;
    for (int j = 1; j <= n; j++) {
      r *= i;
    }
    return r;
  }

  void main(String [] args)  {
    int n = 123456;
    int t = 254;
    while (true) {
      for int x = 1; x <= t - 2; x++) {
        for (int y = 1; y <= t - x -1; y++) {
          for (z = 1; z <= t - x - y; z++) {
            if (exp(x,n) + exp(y,n) == exp(z,n) {
              System.out.println("hello, world");
            }
          }
        }
      }
    }
  }
}
```

Given an input n, it prints hello, world only if the equation

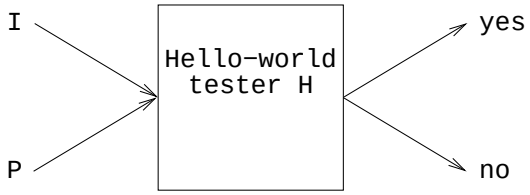
$$x^n + y^n = z^n$$

has a solution where x, y, and z are integers. We know nowadays that it will print hello, world for $n = 2$, and loop forever for $n > 2$.

It took mathematicians 300 years to prove this so-called *Last Theorem of Fermat*[1]. Can we expect to write a program `H` that solves the general problem of telling whether any given program `P`, on any given input `I`, eventually prints `hello, world` or not?

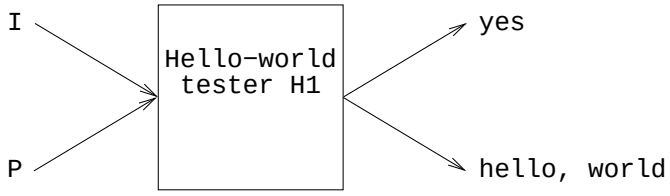
5.1.1 The hypothetical `hello, world` tester

Proof by contradiction that `H` is impossible to write. Suppose that `H` exists:

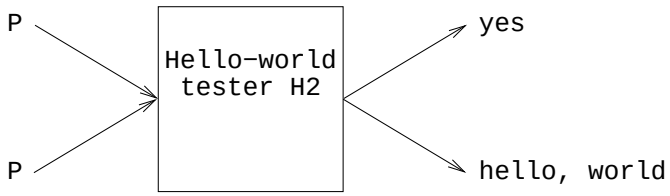


`H` takes as input a program `P` and an input `I`, and tells whether `P` with input `I` prints `hello, world`. In particular, the only output `H` makes is either to print `yes` or to print `no`.

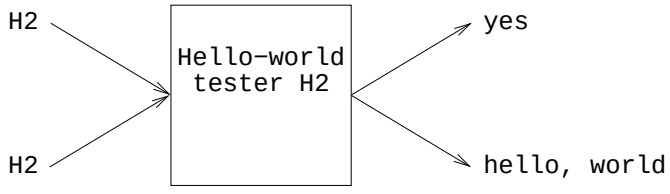
We modify the response no of `H` to `hello, world`, getting a program `H1`:



We modify `H1` to take `P` and `I` as a single input, getting a program `H2`:



We provide `H2` as input to `H2`:



If `H2` prints `yes`, then it should have printed `hello, world`.

If `H2` prints `hello, world`, then it should have printed `yes`.

So `H2` and hence `H` cannot exist. □

Hence we have an *undecidable problem*. It is similar to the language L_d we will see later.

[1] The conjecture has been proved by Andrew Wiles in year 1994

5.1.2 Undecidable Problems

Definition 5.1 [Undecidable] *A problem is* undecidable *if no program can solve it.*

- *Problem* means deciding on the membership of a string in a language.
- Languages over an alphabet are not enumerable.
- Programs (finite strings over an alphabet) are enumerable: order them by length, and then lexicographically.
- Hence there are infinitely more languages than programs.
- Hence there must be undecidable problems (Goedel[2], 1931).

Goedel Incompleteness Theorem

In 1931, Goedel published his famous incompleteness theorems in *Ueber formal unentscheidbare Saetze der Principia Mathematica und verwandter Systeme* [Goe31]. In that article, he proved for any computable axiomatic system that is powerful enough to describe the arithmetic of the natural numbers, that:

1. If the system is consistent, it cannot be complete.
2. The consistency of the axioms cannot be proved within the system.

In hindsight, the basic idea at the heart of the incompleteness theorem is rather simple. Goedel essentially constructed a formula that claims that it is unprovable in a given formal system. If it were provable, it would be false, which contradicts the fact that in a consistent system, provable statements are always true. Thus there will always be at least one true but unprovable statement. That is, for any computably enumerable set of axioms for arithmetic (that is, a set that can in principle be printed out by an idealized computer with unlimited resources), there is a formula that obtains in arithmetic, but which is not provable in that system. To make this precise, however, Goedel needed to solve several technical issues, such as encoding statements, proofs, and the very concept of provability into the natural numbers. He did this using a process known as Goedel numbering.

5.1.3 Problem Reduction

A *reduction* is a transformation of one problem into another problem. Depending on the transformation used this can be used to define complexity classes on a set of problems.

Intuitively, problem A is reducible to problem B if solutions to B exist and give solutions to A whenever A has solutions. Thus, solving A cannot be harder than solving B. We write $A \leq B$, usually with a subscript on the $\leq$ to indicate the type of reduction being used.

Often we find ourselves trying to solve a problem that is similar to a problem we've already solved. In these cases, often a quick way of solving the new problem is to transform each instance of the new problem into instances of the old problem, solve these using our existing solution, and then use these to obtain our final solution. This is perhaps the most obvious use of reductions.

Another, more subtle use is this: suppose we have a problem that we've proved is hard to solve, and we have a similar new problem. We might suspect that it, too, is hard to solve. We argue by contradiction: suppose the new problem is easy to solve. Then, if we can show that every

[2]Kurt Goedel (1906 - 1978) was an Austrian-American logician, mathematician and philosopher. One of the most significant logicians of all time, Goedel made an immense impact upon scientific and philosophical thinking in the 20th century, a time when many, such as Bertrand Russell, A. N. Whitehead and David Hilbert, were pioneering the use of logic and set theory to understand the foundations of mathematics.

instance of the old problem can be solved easily by transforming it into instances of the new problem and solving those, we have a contradiction. This establishes that the new problem is also hard.

Example 5.47 [Multiplication/Squaring]

A very simple example of a reduction is from multiplication to squaring. Suppose all we know how to do is to add, subtract, take squares, and divide by two. We can use this knowledge, combined with the following formula, to obtain the product of any two numbers:

$$a \times b = \frac{\left((a+b)^2 - a^2 - b^2\right)}{2}$$

We also have a reduction in the other direction; obviously, if we can multiply two numbers, we can square a number. This seems to imply that these two problems are equally hard.

Example 5.48 [Function call] Does a program Q, given input q, ever call function `foo()`?

This problem is undecidable.

We construct a program R and an input z such that R with input z, calls `foo()` if and only if Q with input y prints `hello, world`:

1. If Q has a function called `foo()`, rename it and all calls to that function. Clearly the new program Q_1 does exactly what Q does.
2. Add to Q_1 a function `foo()`. This function does nothing, and is not called. The resulting program is Q_2.
3. Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array A. The resulting program is Q_3.
4. Modify Q_3 so that whenever it executes any output statement, it then checks in the array A to see if it has written 12 characters or more, and so, whether `hello, world` are the first 12 characters. In that case call the new function `foo()`. The resulting program is R, and input z is the same as y.

Now suppose that Q with input y prints `hello, world` as its first output. Then R as constructed will call `foo()`. However, if Q with input y does not print `hello, world` as its first output, then R will never call `foo()`. If we can decide whether R with input z calls `foo()`, then we also know whether Q with input y (remember $y = z$) prints `hello, world`. Since we know that no algorithm to decide the hello-world problem exists, so the assumption that there is a calls-foo tester is wrong. Thus the calls-foo problem is undecidable.

Exercise 5.1 [Halt-Tester]

Give a reduction from the hello-world problem to the following problem:

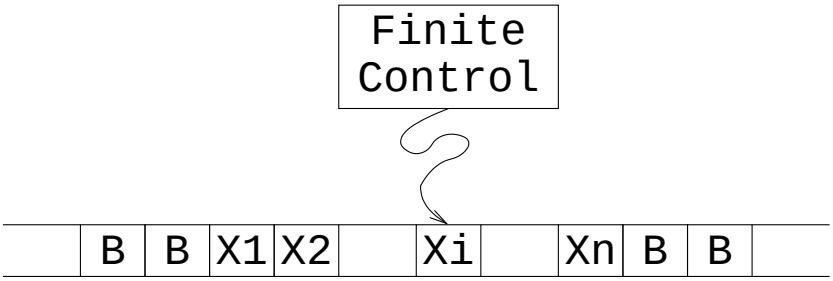
Given a program and an input, does the program eventually halt, i.e., does the program not loop forever on the input?

Use informal style for describing plausible program transformations. Don't forger to consider the exceptions that may be thrown by the original program.

5.2 Turing Machines

A Turing machine is a theoretical device that manipulates symbols contained on a strip of tape. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside of a computer.

The Turing machine was described in 1936 by Alan Turing[3]. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine (thus they have never actually been constructed).



A move of a Turing machine (TM) is a function of the state of the finite control and the tape symbol just scanned.

In one move, the Turing machine will:

1. Change state.
2. Write a tape symbol in the cell scanned.
3. Move the tape head left or right.

5.2.1 Alan Turing and the *Entscheidungsproblem*

In his momentous paper *On Computable Numbers, with an Application to the Entscheidungsproblem* [Tur37], Turing reformulated Kurt Goedel's 1931 results on the limits of proof and computation, replacing Goedel's universal arithmetic-based formal language with what are now called Turing machines, formal and simple devices. He proved that some such machine would be capable of performing any conceivable mathematical computation if it were representable as an algorithm.

Turing machines are to this day the central object of study in theory of computation. He went on to prove that there was no solution to the Entscheidungsproblem by first showing that the halting problem for Turing machines is undecidable: it is not possible to decide, in general, algorithmically whether a given Turing machine will ever halt. While his proof was published subsequent to Alonzo Church's equivalent proof in respect to his lambda calculus, Turing's work is considerably more accessible and intuitive. It was also novel in its notion of a *Universal Machine*, the idea that such a machine could perform the tasks of any other machine. *Universal* in this context means what is now called programmable. The paper also introduces the notion of definable numbers.

5.3 Turing Machines: Formal Definition

Definition 5.2 [Turing Machine] *Formally, a* Turing machine *is a 7-tuple*

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

where:

[3]Alan Turing, 23 June 1912 - 7 June 1954), was an English mathematician, logician, cryptanalyst, and computer scientist. He was influential in the development of computer science and provided an influential formalisation of the concept of the algorithm and computation with the Turing machine. During the Second World War, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's codebreaking centre. For a time he was head of Hut 8, the section responsible for German naval cryptanalysis. He devised a number of techniques for breaking German ciphers, including the method of the bombe, an electromechanical machine that could find settings for the Enigma machine.

1. Q *is the finite set of* states *of the finite control.*
2. Σ *is the finite set of* input symbols.
3. Γ *is the finite set of* tape symbols; $\Sigma \subset \Gamma$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the* transition function, *which is a partial function.*
   The arguments of $\delta(q, X)$ *are a state* q *and a tape symbol* X. *The value of* $\delta(q, X)$, *if it is defined is a triple* (p, Y, D), *where:*
   (a) p *is the next state, in* Q.
   (b) Y *is the symbol, in* Γ, *written in the cell being scanned, replacing whatever symbol was there.*
   (c) D *is a* direction, *either* L *or* R, *standing for* left *or* right, *respectively, and telling us the direction in which the head moves.*
5. $q_0 \in Q$ *is the* start state.
6. $B \in \Gamma$ *is the* blank symbol. $B \notin \Sigma$.
7. $F \subset Q$ *is the set of* final *or* accepting states.

Notation 5.1 [Blank] We denote the blank symbol B by $\square$.

5.3.1 Instantaneous Description and Moves

A Turing machine changes its configuration upon each move.

We use *instantaneous descriptions* (*ID*s) for describing such configurations.

Definition 5.3 [Instantaneous Description]

An instantaneous description *of a TM is a string of the form*

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n$$

where

1. q *is the state of the Turing machine.*
2. *The tape head is scanning the* i*th symbol from the left.*
3. $X_1 X_2 \cdots X_n$ *is the portion of the tape between the leftmost and rightmost nonblanks.*

We use $\underset{M}{\vdash}$ to designate a move of a Turing machine M from one ID to another.

Definition 5.4 [Move] *If* $\delta(q, X_i) = (p, Y, L)$, *then:*

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n$$

If $\delta(q, X_i) = (p, Y, R)$, *then:*

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots X_{i-1} Y p X_{i-+} X_{i+2} \cdots X_n$$

The reflexive-transitive closure of $\underset{M}{\vdash}$ is denoted $\overset{*}{\underset{M}{\vdash}}$.

5.3.2 Language of a TM

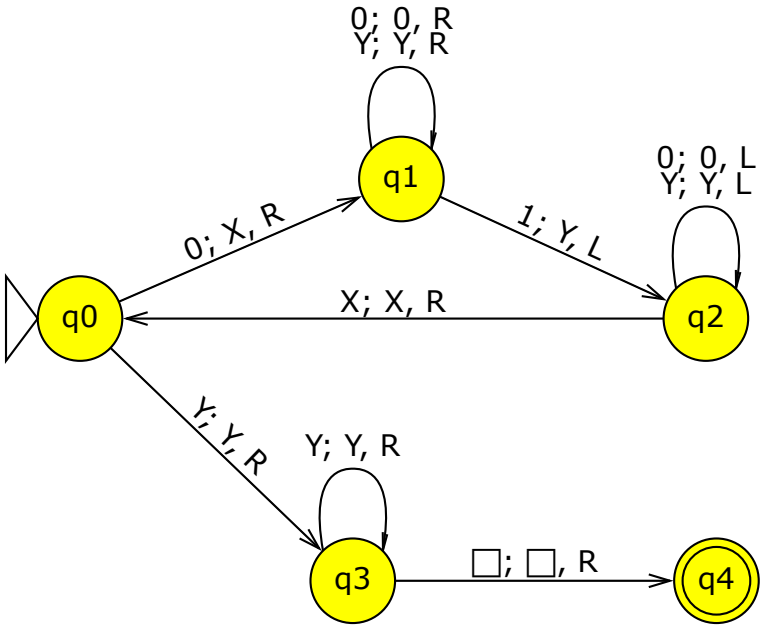
Definition 5.5 [Language] *A Turing machine* $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *accepts the* language

$$L(M) = \{w \in \Sigma^* : q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta, p \in F, \alpha, \beta \in \Gamma^*$$

Example 5.49 [TM for L_{01}] $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0,1\}, \{0,1,X,Y,\Box\}, \delta, q_0, \Box, \{q_4\})$ where δ is given by the following table:

	0	1	X	Y	$\Box$
$\rightarrow q_0$	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	$(q_4, \Box, R)$
$*q_4$					

We can also represent M by the following transition diagram:



As M performs its computation, the portion of the tape, where M's tape head has visited, will always be a sequence of symbols described by the regular expression $X^*0^*Y^*1^*$.

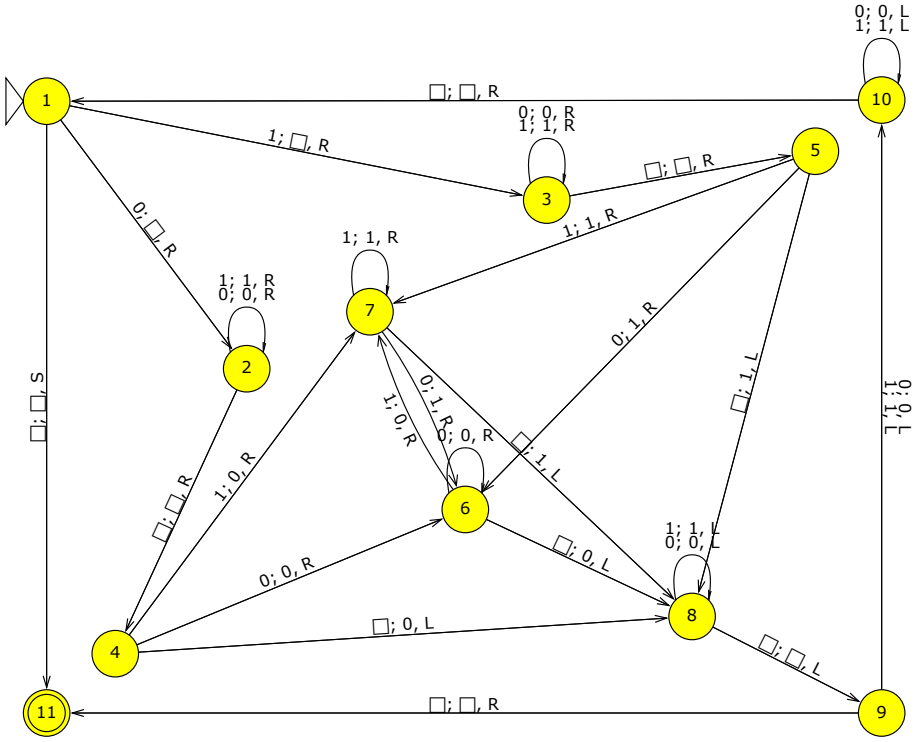
State q_0 is the initial state, and M also enters state q_0 every time it returns to the leftmost remaining 0. If M is in state q_0 and scanning a 0, it will go to state q_1 and change the 0 to an X. Once in state q_1, M keeps moving right over all 0's and Y's remaining in that state. If M sees an X or a $\Box$, it dies. If M sees a 1 when in state q_1 it changes that 1 to a Y, enters state q_2, and starts moving left.

In state q_2, M moves left over 0's and Y's, remaining in state q_2. When M reaches the rightmost X, which marks the right end of the block of 0's that have already been changed to X, M returns to state q_0 and moves right. There are two cases:

1. If M sees a 0, the it repeats the matching cycle we have just described.

2. If M sees a Y, then it has changed all 0's to X's. If all 1's have been changed to Y's, then the input was of the form 0^n1^n, and M should accept. Thus M enters state q_3, and starts moving right, over Y's. If the first symbol other than Y that M sees is a blank, then indeed there were an equal number of 0's and 1's, so M enters state q_4 and accepts. On the other hand, if M encounters another 1, then there are too many 1's ant M dies without accepting. If it encounters a0, then the input was of the wrong form, and M also dies.

Example 5.50 [Reverse] Following TM reserves and input string of 0's and 1's.

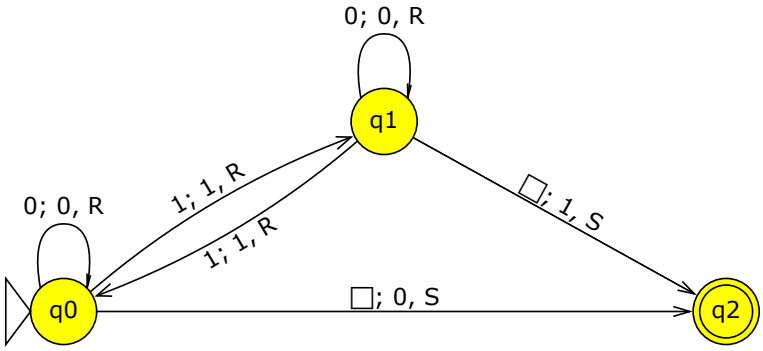


In state 1, M reads the leftmost symbol and replaces it by a blank. If the symbol was a 0, M enters state 2 and moves to the right until it reaches the first blank symbol, otherwise M enters state 3 and moves to the right until it reaches the first blank symbol. States 2 and 3 are used to remember the value of the leftmost symbol.

The leftmost symbol is written in state 4 or 5 after the first blank on the right of the input. This may overwrite an already copied leftmost symbol. If it is the case, that symbol will be copied to the right (states 6 and 7). This process continues until the rightmost blank it reached.

The TM now returns to the next leftmost non blank symbol (states 8, 9 and 10). If there is no more such symbol, the TM enters accepting state 11 and halts.

Example 5.51 [Parity] The input of following TM is a string of 0's and 1's. If the input string has an even number of 1's, the TM will write a 0 at the end of the input and halt, otherwise it will write a 1 at the end of the input and halt.



The TM always moves to the right. If the TM is in state q_0, then it has read an even number of 1's so far, otherwise the machine will be in state q_1. At the end of the input, the Tm will replace the first blank by a 0 if it was in state q_0 and enter q_2, otherwise (the TM was in state q_1) the Tm will replace the first blank by a 1 and enter q_2. The TM halts in accepting state q_2.

5.3.3 Acceptance by Halting

A Turing machine *halts* if it enters a state q, scanning a tape symbol X, and there is no move in this situation, i.e., $\delta(q, X)$ is undefined.

We can always assume that a Turing machine halts if it accepts, as we can make $\delta(q, X)$ undefined whenever q is an accepting state.

Unfortunately, it is not always possible to require that a Turing machine halts even if it does not accept.

Recursive language: there is a TM, corresponding to the concept of algorithm, that halts eventually, whether it accepts or not.

Recursively enumerable language: there is a TM that halts if the string is accepted.

Decidable problem: there is an algorithm for solving it.

Exercise 5.2 [TM for Palindromes] *Construct a Turing Machine for L_{pal}, the language of palindromes.*

Exercise 5.3 [TM for Addition] *Construct a Turing Machine for the addition of binary natural numbers. The initial tape contents is a sum of binary numbers, e.g. 101+11*

Hint: You may replace 0 by a and 1 by b to remember the position of the last bit addition.

Appendix A

Mathematical background

A.1 Axiom

The word *axiom* comes from the Greek word $\alpha\xi\iota\omega\mu\alpha$ (axioma), a verbal noun from the verb $\alpha\xi\iota o\epsilon\iota\nu$ (axioein), meaning *to deem worthy*, but also *to require*, which in turn comes from $\alpha\xi\iota o\zeta$ (axios), meaning *being in balance*, and hence *having (the same) value (as)*, *worthy*, *proper*. Among the ancient Greek philosophers an axiom was a claim which could be seen to be true without any need for proof.

In traditional logic, an *axiom* or *postulate* is a proposition that is not proved or demonstrated but considered to be either self-evident, or subject to necessary decision. Therefore, its truth is taken for granted, and serves as a starting point for deducing and inferring other (theory dependent) truths.

Definition A.1 [Axiom] *In mathematics, the term* axiom *is used in two related but distinguishable senses:* logical axioms *and* non-logical axioms. *In both senses, an axiom is any mathematical statement that serves as a starting point from which other statements are logically derived. Unlike* theorems*, axioms (unless redundant) cannot be derived by principles of deduction, nor are they demonstrable by mathematical proofs, simply because they are starting points; there is nothing else from which they logically follow (otherwise they would be classified as theorems).*

Logical axioms are usually statements that are taken to be universally true (e.g., $A \wedge B \longrightarrow A$), while non-logical axioms (e.g., $a + b = b + a$) are actually defining properties for the domain of a specific mathematical theory (such as arithmetic). When used in that sense, axiom, postulate, and "assumption" may be used interchangeably. In general, a non-logical axiom is not a self-evident truth, but rather a formal logical expression used in deduction to build a mathematical theory. To axiomatise a system of knowledge is to show that its claims can be derived from a small, well-understood set of sentences (the axioms).

A.2 The natural Numbers

A.2.1 The Peano axioms

The Peano[1] axioms define the properties of natural numbers, usually represented as a set $\mathbb{N}$.

Axiom A.1 [Peano Axioms] We begin with a set $\mathbb{N}$ and a mapping S of $\mathbb{N}$ into $\mathbb{N}$. The elements of $\mathbb{N}$ we call *natural numbers*. For $n \in \mathbb{N}$ we call $S(n)$ the *successor* of n.

[1] Giuseppe Peano (27 August 1858 - 20 April 1932) was an Italian mathematician, whose work was of exceptional philosophical value.He was a founder of mathematical logic and set theory, to which he contributed much notation. The standard axiomatisation of the natural numbers is named in his honor. As part of this axiomatisation effort, he made key contributions to the modern rigorous and systematic treatment of the method of mathematical induction. He spent most of his career teaching mathematics at the University of Turin.

1. 0 is a natural number.
2. For every natural number n, $S(n)$ is a natural number.
3. For every natural number n, $S(n) \neq 0$. There is no natural number whose successor is 0.
4. For all natural numbers m and n, if $S(m) = S(n)$, then $m = n$. That is, S is one-to-one (injection).
5. If ϕ is a unary predicate such that:
   - $\phi(0)$ is true, and
   - for every natural number n, if $\phi(n)$ is true, then $\phi(S(n))$ is true,

   then $\phi(n)$ is true for every natural number n.

The first two axioms define the properties of the natural numbers. The constant 0 is assumed to be a natural number, and the naturals are assumed to be closed under a *successor* function S. We denote also $S(n)$ by $n+$.

The two next axioms together imply that the set of natural numbers is infinite, because it contains at least the infinite subset $\{0, S(0), S(S(0)), ...\}$, each element of which differs from the rest.

The final axiom, sometimes called the *axiom of induction*, is a method of reasoning about all natural numbers; it is the only second-order[2] axiom.

A.2.2 Mathematical induction

Mathematical induction is a method of mathematical proof typically used to establish that a given statement is true of all natural numbers. It is done by proving that the first statement in the infinite sequence of statements is true, and then proving that if any one statement in the infinite sequence of statements is true, then so is the next one.

Mathematical induction is a direct consequence of the fifth Peano axiom.

The simplest and most common form of mathematical induction proves that a statement involving a natural number n holds for all values of n. The proof consists of two steps:

Induction base showing that the statement holds when $n = 0$.

Inductive step showing that if the statement holds for some n, then the statement also holds when $n + 1$ is substituted for n

The assumption in the inductive step that the statement holds for some n is called the induction hypothesis (or inductive hypothesis). To perform the inductive step, one assumes the induction hypothesis and then uses this assumption to prove the statement for $n + 1$.

The description above of the basis applies when 0 is considered a natural number, as is common in the fields of combinatorics and mathematical logic. If, on the other hand, 1 is taken to be the first natural number, then the base case is given by $n = 1$.

This method works by first proving that the statement is true for a starting value, and then proving that the process used to go from one value to the next is valid. If these are both proved, then any value can be obtained by performing the process repeatedly. It may be helpful to think of the domino effect; if one is presented with a long row of dominoes standing on end, one can be sure that:

[2] *First-order logic* uses only discrete variables (eg. the variable x represents a person) whereas *second-order logic* uses variables that range over sets of individuals. For example, the second-order sentence $\forall P \forall x (x \in P \vee x \notin P)$ says that for every set P of people and every person x, either x is in P or it is not (this is the principle of bivalence). Second-order logic also includes variables quantifying over functions, and other variables. Both first-order and second-order logic use the idea of a domain of discourse (often called simply the *domain* or the *universe*). The domain is a set of individual elements which can be quantified over.

1. The first domino will fall
2. Whenever a domino falls, its next neighbor will also fall,

so it is concluded that all of the dominoes will fall, and that this fact is inevitable.

A.2.3 Addition

Definition A.2 [Addition] *The* addition *of natural numbers is the mapping A of $\mathbb{N}\times\mathbb{N}$ into $\mathbb{N}$ such that*

1. $A(m,0) = m$ *for all* $m \in \mathbb{N}$
2. $A(m,S(n)) = S(A(m,n)$ *for all* $m,n \in \mathbb{N}$

Notation A.1 [Addition] We write $m+n$ for $A(m,n)$

Remark A.1 [Uniqueness] We can prove directly from the axioms that the addition of natural numbers as defined above is unique.

We show now that some properties of the addition can be prover directly from the axioms (mainly using mathematical induction).

Theorem A.1 [Properties of addition] *Let m, n and p be natural numbers, then*

1. $m+0 = 0+m = m$ *for all* $m \in \mathbb{N}$. *i.e.* 0 *is the neutral element for the addition*
2. $(m+n)+p = m+(n+p)$ *for all* $m,n,p \in \mathbb{N}$, *i.e. the addition is associative.*
3. $m+n = n+m$ *for all* $m,n \in \mathbb{N}$, *i.e. the addition is commutative.*

Proof:

1. Since $m+0 = m$ by definition, we only need to show that $m = 0+m$ for all $m \in \mathbb{N}$. We use hereby induction over m

   Basis: $0+0 = 0$ (by definition)

   Induction: Suppose that $m = 0+m$. By definition $0+S(m) = S(0+m)$. By hypothesis $S(0+m) = S(m)$ □

2. Proof (by induction over p):

   Basis: $(m+n)+0 = m+n = m+(n+0)$

   Induction: Suppose that $(m+n)+p = m+(n+p)$. By definition of the addition, $(m+n)+S(p) = S((m+n)+p)$. By hypothesis $S((m+n)+p) = S(m+(n+p))$. By definition of the addition, $S(m+(n+p)) = m+S(n+p) = m+(n+S(p))$ □

3. We first prove that $S(m)+n = m+S(n)$ for all $m,n \in \mathbb{N}$. We use an induction proof over n.

   Basis: $S(m)+0 = S(m) = S(m+0) = m+S(0)$ (neutral element).

   Induction: Suppose that $S(m)+n = m+S(n)$.By definition, $S(m)+S(n) = S(S(m)+n)$. By hypothesis, $S(S(m)+n) = S(m+S(n))$. By definition of the addition $S(m+S(n)) = m+S(S(n))$

   Now we can show that $m+n = n+m$ using induction over m.

   Basis: $0+n = n+0$ since 0 is neutral element.

   Induction: Suppose that $m+n = n+m$. By definition $m+S(n) = S(m+n)$. By hypothesis $S(m+n) = S(n+m)$. By definition $S(n+m) = n+S(m)$. Now it follows from the result above that $n+S(m) = S(n)+m$. □

A.2.4 Multiplication

Definition A.3 [Multiplication] *The* multiplication *of natural numbers is the mapping M of $\mathbb{N}\times\mathbb{N}$ into $\mathbb{N}$ such that*

1. $M(m,0) = 0$ *for all* $m \in \mathbb{N}$
2. $M(m,S(n)) = M(m,n) + m$ *for all* $m,n \in \mathbb{N}$

Notation A.2 [Multiplication] We write $m * n$ for $M(m,n)$ and 1 for $S(0)$

Remark A.2 [Uniqueness] We can prove directly from the axioms that the multiplication of natural numbers as defined above is unique.

The properties of the multiplication can be proving directly from the axioms (mainly using mathematical induction).

Theorem A.2 [Properties of multiplication] *Let m, n and p be natural numbers, then*

1. $m * 1 = 1 * m = m$ *for all* $m \in \mathbb{N}$. *i.e.* 0 *is the neutral element for the addition*
2. $(m * n) * p = m * (n * p)$ *for all* $m,n,p \in \mathbb{N}$, *i.e. the addition is associative.*
3. $m * n = n * m$ *for all* $m,n \in \mathbb{N}$, *i.e. the addition is commutative.*

Proof: Exercise □

Exercise A.1 [Distributivity] *Prove that*

$$m * (n + p) = m * n + m * p \quad \forall m,n,p \in \mathbb{N}$$

Hint: use induction over p.

Exercise A.2 [Multiplication] *Prove theorem A.2.*

A.2.5 Order

If one number is less than the other, then the second can be obtained by adding a number to the first. For the set $\mathbb{N}$ of natural numbers we have

Theorem A.3 [Order in $\mathbb{N}$] *If T is the subset of $\mathbb{N}\times\mathbb{N}$ consisting of all (m,n) such that $m + q = n$ for some $q \in \mathbb{N}$, then T is an order relation in $\mathbb{N}$, i.e.*

1. $(m,m) \in T$ *for all* $m \in \mathbb{N}$ *(reflexivity)*
2. *If* $(m,n) \in T$ *and* $(n,m) \in T$ *then* $m = n$ *(symmetry)*
3. *If* $(m,n) \in T$ *and* $(n,p) \in T$ *then* $(m,p) \in T$

Notation A.3 [Order in $\mathbb{N}$] We write $m \leq n$ if $(m,n) \in T$.

Proof:

1. $m + 0 = m$ for all $m \in \mathbb{N}$, i.e. $(m,m) \in T$ for all $m \in \mathbb{N}$
2. We first prove that $m + q = m$ if and only if $q = 0$.

   Basis: Let $m = 0$. **(if):** if $q = 0$, then $0 + 0 = 0$. **(only if):** $0 + q = q = 0$.

   Induction: Suppose that $m + q = m$ if and only if $q = 0$. We need to prove that $S(m) + q = S(m)$ if and only if $q = 0$. **(if)**: if $q = 0$ then $S(m) + 0 = S(m)$. **(only if):** if $S(m) + q = S(m)$ then $S(m) + q = S(m + q) = S(m)$, i.e. $m + qm$ the proof follows by induction hypothesis. □

   Now suppose that if $(m,n) \in T$ and $(n,m) \in T$, i.e. $m + p = n$ for some p and $n + q = m$ for some q. It follows that $(m + p) + q = n + q = m$, i.e. $m + (p + q) = m$. If follows that $p + q = 0$, i.e. $p = q = 0$. Finally, since $p = q = 0$, $m = n$ □

3. If $(m,n) \in T$ and $(n,p) \in T$, then $m+q=n$ for some q and $n+r=p$ for some r, i.e. $m+(q+r)=n+r=p$. It follows that $(m,p) \in T$ □

Definition A.4 [Initial segment] *For $n \in \mathbb{N}$, the* initial segment *I_n is the set of all $m \in \mathbb{N}$ such that $m \leq n$.*

A.3 Counting

The familiar process of counting uses $\mathbb{N} - \{0\}$ as a standard set of tags. If the elements of a set can be tagged with the whole numbers form 1 to n, the set is said to have n elements. The usefulness of this process lies in the fact that a set which can be tagged with the whole numbers from 1 to n cannot also be tagged with the whole numbers from 1 to m unless $m=n$.

Theorem A.4 [Counting] *There is no 1-1 mapping of any initial segment I_n onto a proper subset of I_n.*

Proof: Let M be the set of all $n \in \mathbb{N}$ such that there is no 1-1 mapping of the initial segment I_n onto any of its proper subsets. We prove by induction than $M=\mathbb{N}$.

Basis: $1 \in M$, since the only proper subset of $I_1=\{1\}$ is the empty set, and the empty set is not the range of any mapping.

Induction: Suppose that $n \in M$, and let F be a 1-1 mapping of $I_{S(n)}$ onto some proper subset K of $I_{S(n)}$. Then exactly one of the following is true:

(1) $S(n) \in K$ and $F(S(n))=S(n)$
(1) $S(n) \in K$ and $F(S(n)) \neq S(n)$
(2) $S(n) \notin K$

If (1) is true, let

$$F' = \{(m, F(m)) : m \in I_n\}$$

Then F' is a 1-1 mapping of I_n onto $K-\{S(n)\}$, which is a proper subset of I_n, since k is a proper subset of $I_{S(n)}$. This is impossible. since $n \in M$.

If (2) is true, the $F(S(n))=t \neq S(n)$ and $F(k)=S(n)$ for some $k \neq S(n)$. Then set

$$F'' = \{(m, F(m)) : m \neq k, S(n)\} \cup \{(k,t), (S(n), S(n)\}$$

is a 1-1 mapping of $I_{S(n)}$ onto K for which (1) is true. This has been shown to be impossible.

If (3) is true, then $K \subset I_n$ ant the set

$$F''' = F - \{(S(n), F(S(n)))\}$$

is a 1-1 mapping of I_n onto $K-\{F(S(n))\}$. But $K-\{F(S(n))\}$ is a proper subset of I_n, since $F(S(n)) \in K$, by (3), $K \subset I_n$. This is impossible since $n \in M$ □

Theorem A.5 [Counting] *If a set X is not finite, then there is a 1-1 mapping of $\mathbb{N}$ onto some subset of X.*

Proof: This theorem cannot be prove using the axioms stated so far. A proper proof requires usage of the *Axiom of Choice.* □

Definition A.5 [Infinite Set] *A set X is* infinite *if there is a 1-1 mapping of X onto a proper subset of itself.*

Definition A.6 [Denumerable Set] *A set X is called* denumerable *if there is a 1-1 mapping of $\mathbb{N}$ onto X.*

Theorem A.6 [$\mathbb{Z}$ is denumerable] *$\mathbb{Z}$ is denumerable.*

Proof: Let

$$F(n) = \begin{cases} \frac{n}{2} & \text{if} \quad n \quad \text{is even} \\ -\frac{n+1}{2} & \text{if} \quad n \quad \text{is odd} \end{cases}$$

Then F is a 1-1 mapping of $\mathbb{N}$ onto $\mathbb{Z}$ □

Theorem A.7 [$\mathbb{N} \times \mathbb{N}$ is denumerable] *$\mathbb{N} \times \mathbb{N}$ is denumerable.*

Proof: For each $(p, q) \in \mathbb{N} \times \mathbb{N}$ there is exactly one $n \in \mathbb{N}$ such that $p + q = n$ Let

$$R_n = \{(p, q) : p + q = n\}$$

then R_n contains $n + 1$ elements, i.e. $(0, n), (1, n-1), (2, n-2), \ldots (n-1, 1), (n, 0)$. Let $p + q = n$ we define:

$$G(p, q) = p + \sum_{i=0}^{n} |R_i| = p + \sum_{i=0}^{n} (i + 1) = p + \frac{(n+1)(n+2)}{2}$$

Then $F = G^{-1}$ is a 1-1 mapping of $\mathbb{N}$ onto $\mathbb{N} \times \mathbb{N}$ □

Theorem A.8 [$\mathbb{Z} \times \mathbb{Z}$ is denumerable] *$\mathbb{Z} \times \mathbb{Z}$ is denumerable.* □

Theorem A.9 [$\mathbb{Q}$ is denumerable] *$\mathbb{Q}$ is denumerable.* □

Proof: Follows directly from theorem A.7 and theorem A.8 □

Theorem A.10 [$P(\mathbb{N})$ is not denumerable] *$P(\mathbb{N})$ the power set of $\mathbb{N}$ is not denumerable.*

Proof: Suppose there is a mapping F of $\mathbb{N}$ onto $P(\mathbb{N})$. Since the set $A = \{n : n \notin F(n)\} \in P(\mathbb{N})$ and F maps $\mathbb{N}$ onto $P(\mathbb{N})$, there is some $m \in \mathbb{N}$ such that $F(m) = A$. Thus $m \notin F(m)$ if and only if $m \in A$. This is impossible since $F(m) = A$. Hence there is no mapping of $\mathbb{N}$ onto $P(\mathbb{N})$ □

Exercise A.3 [$\mathbb{R}$ is not denumerable] *Prove that $\mathbb{R}$ is not denumerable.*

Hint: *Suppose that the interval $]0, 1] \subset \mathbb{R}$ is denumerable. List the elements of $]0, 1]$ in their binary representation:*

$$\begin{array}{rcl} x_1 & : & 0.a_{11}a_{12}a_{13}a_{14}a_{15}\ldots \\ x_2 & : & 0.a_{21}a_{22}a_{23}a_{24}a_{25}\ldots \\ x_3 & : & 0.a_{31}a_{32}a_{33}a_{34}a_{35}\ldots \\ x_4 & : & 0.a_{41}a_{42}a_{43}a_{44}a_{45}\ldots \end{array}$$

Construct a non-terminating decimal $0.b_1b_2b_3b_4b_5\ldots$ such that $b_n \neq a_{nn}$

Exercise A.4 [$\mathbb{Z} \times \mathbb{Z}$ is denumerable] *Prove theorem A.8.*

Exercise A.5 [$\mathbb{Q}$ is denumerable] *Prove theorem A.9.*

Appendix B

Parsing

A program verifying the language generated by a grammar is called *parser*. It is quite easy to write a JAVA-Program that will check if a string w belongs to the language $L(G)$ of some CFG G. We can just use backtracking, but this method is not efficient [ASU07].

To do this, we will introduce some restrictions to the grammars, such that fast parsing is possible.

To illustrate that process, we will start with the grammar of example B.52.

Example B.52 [Grammar for Expressions]

$$\begin{array}{lcl} E & \to & E+E \\ E & \to & E-E \\ E & \to & E*E \\ E & \to & E/E \\ E & \to & (E) \\ E & \to & \text{num} \end{array}$$

or (BNF notation)

$$E \to \quad E+E \quad | \quad E-E \quad | \quad E*E \quad | \quad E/E \quad | \quad (E) \quad | \quad \text{num}$$

Remark B.1 [Grammar for BNF] An important grammar is G_{BNF}, the grammar for BNF-Grammars:

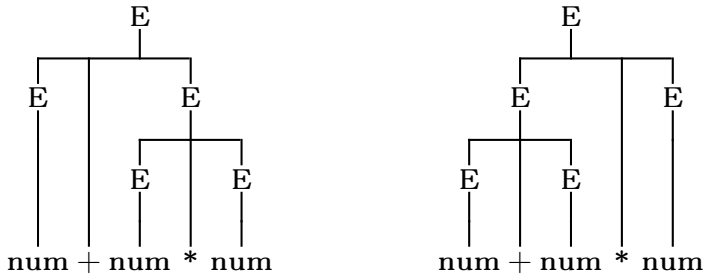
$$\begin{array}{rcl} bnf & \to & production \;\; bnf \;\; | \;\; \epsilon \\ production & \to & \text{<NT>} \;\; \text{"}\to\text{"} \;\; choice \\ choice & \to & sequence \;\; | \;\; sequence \;\; \text{"|"} \;\; choice \\ sequence & \to & factor \;\; | \;\; factor \;\; sequence \\ factor & \to & \text{<NT>} \;\; | \;\; \text{<T>} \end{array}$$

B.1 Writing a Grammar

B.1.1 Ambiguity

We know from example 4.36 that sentential forms may have different parse trees, leading to ambiguity. Grammars that produce more than one leftmost derivation or more than one rightmost derivation for the same sentence are not suitable.

For example, the grammar of example B.52 has two different parse trees for num + num ∗ num:



For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

We know from section 4.3.1 how to remove ambiguity.

Example B.53 [Ambiguity] Removing ambiguity in grammar B.52.

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & E - T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & T/F \\ T & \rightarrow & F \\ F & \rightarrow & (E) \\ F & \rightarrow & \text{num} \end{array}$$

or (BNF notation)

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T/F \mid F \\ F & \rightarrow & (E) \mid \text{num} \end{array}$$

B.1.2 Lexical Versus Syntactic Analysis

Everything that can be described by a regular expression can be also described by a grammar. In compiler construction, it is usual to use both:

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularising the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as a grammar, e.g. `num`: $(0 + 1)^*$.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analysers can be constructed automatically [Les75], [JCC96] from regular expressions than from arbitrary grammars.

B.1.3 Eliminating of Left Recursion

Definition B.1 [Left Recursive Grammar] *A grammar is* left recursive *if it has a nonterminal A such that there is a derivation $A \stackrel{*}{\Rightarrow} A\alpha$ for some string α.*

Top-down parsers cannot handle left recursion.

Example B.54 [Left Recursion] Removing left recursion in grammar B.53.

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \\ E' & \rightarrow & -TE' \\ E' & \rightarrow & \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \\ T' & \rightarrow & /FT' \\ T' & \rightarrow & \epsilon \\ F & \rightarrow & (E) \\ F & \rightarrow & \text{num} \end{array}$$

or (BNF notation)

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid -TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid /FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \text{num} \end{array}$$

The grammar has been obtained by eliminating immediate left recursion from the grammar B.54. The left recursive productions $E \rightarrow E+E|E-E|T$ are replaced by $E \rightarrow TE'$ and $E' \rightarrow +TE'|-TE'|\epsilon$. The new productions for T and T' are obtained similarly.

Immediate left recursion can be eliminated by the following technique. First group the productions as

$$A \rightarrow A\alpha_1 | \cdots | A\alpha_m | \beta_1 | \cdots | \beta_n$$

where no β_i begins with A. Then replace the A-productions by

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' | \cdots | \beta_n A' \\ A' & \rightarrow & \alpha_1 A' | \cdots | \alpha_m A' | \epsilon \end{array}$$

Notice that the left recursion has been transformed into a right recursion.

This procedure does not eliminate left recursion involving derivations of two or more steps.

Algorithm B.1 [Eliminating Left Recursion] Input: Grammar G with no cycles or ϵ-productions

Output: An equivalent grammar with no left recursion.

Method: Apply following algorithm to G. The resulting non-left recursive grammar may have ϵ-productions:

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
for (each i from 1 to n) {
    for (each j from 1 to n) {
        replace each production of the form $A_i \rightarrow A_j\gamma$
            by the production $A_i \rightarrow \delta_1\gamma | \cdots | \delta_k\gamma$, where
            $A_j \rightarrow \delta_1 | \cdots | \delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

B.1.4 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough input has been seen that we can make the right choice.

Example B.55 [Left Factoring] Left factoring of grammar B.53

$$\begin{array}{lcl} E & \rightarrow & EE'' \\ E & \rightarrow & T \\ E'' & \rightarrow & +T \\ E'' & \rightarrow & -T \\ T & \rightarrow & TT'' \\ T'' & \rightarrow & *F \\ T'' & \rightarrow & /F \\ T & \rightarrow & F \\ F & \rightarrow & (E) \\ F & \rightarrow & \text{num} \end{array}$$

or (BNF notation)

$$\begin{array}{lcl} E & \rightarrow & EE'' \mid T \\ E'' & \rightarrow & +T \mid -T \mid \\ T & \rightarrow & TT'' \mid F \\ T'' & \rightarrow & *F \mid /F \mid F \\ F & \rightarrow & (E) \mid \text{num} \end{array}$$

Algorithm B.2 [Left Factoring] Input: Grammar G.

Output: An equivalent left-factored grammar.

Method: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ – i.e., there is a nontrivial common prefix – replace all of the A-productions $A \rightarrow \alpha\beta_1 | \cdots | \alpha\beta_n | \gamma$, where γ represent all alternatives that do not begin with α, by

$$\begin{array}{lcl} A & \rightarrow & \alpha A' | \gamma \\ A' & \rightarrow & \beta_1 | \cdots | \beta_n \end{array}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have common prefix.

B.1.5 Non-Context-Free language Constructs

A few syntactic constructs found in typical programming languages cannot be specified using CFG's alone.

Example B.56 [Declarations] The language in this example abstracts the problem of checking that identifiers are declared before they are used in a program. The language consists of strings of the form wcw, where the first w represent the declaration of an identifier w, c represents an intervening program fragment, and the second w represents the use of the identifier.

The abstract language is $L_1 = \{wcw | w \in \Sigma^*\}$. It is possible to prove that L_1 cannot be the language of some CFG G.

Exercise B.1 [Left Factoring] *Left factor the following grammar:*

$$E \rightarrow int \mid int + E \mid int - E \mid E - (E)$$

Exercise B.2 [Left Recursion] *Eliminate left-recursion from the following grammar:*

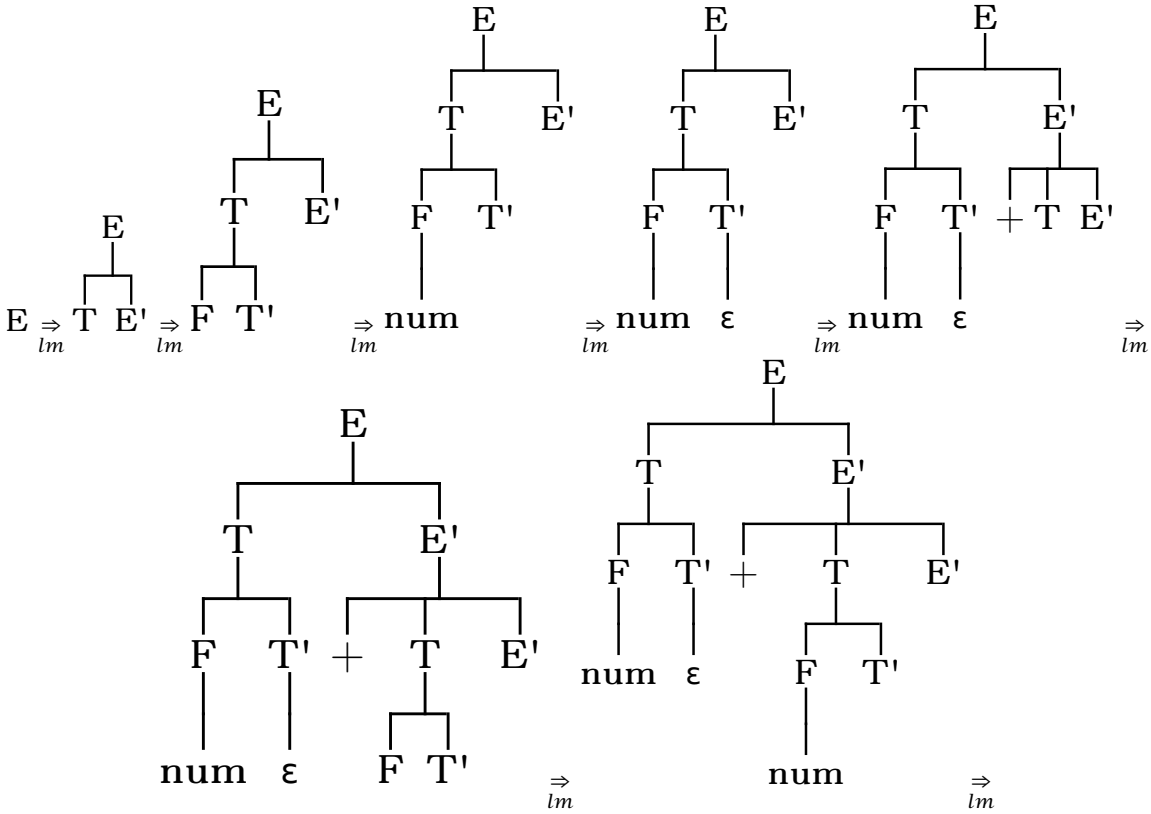
$$\begin{array}{lcl} A & \rightarrow & A + B \mid B \\ B & \rightarrow & int \mid (A) \end{array}$$

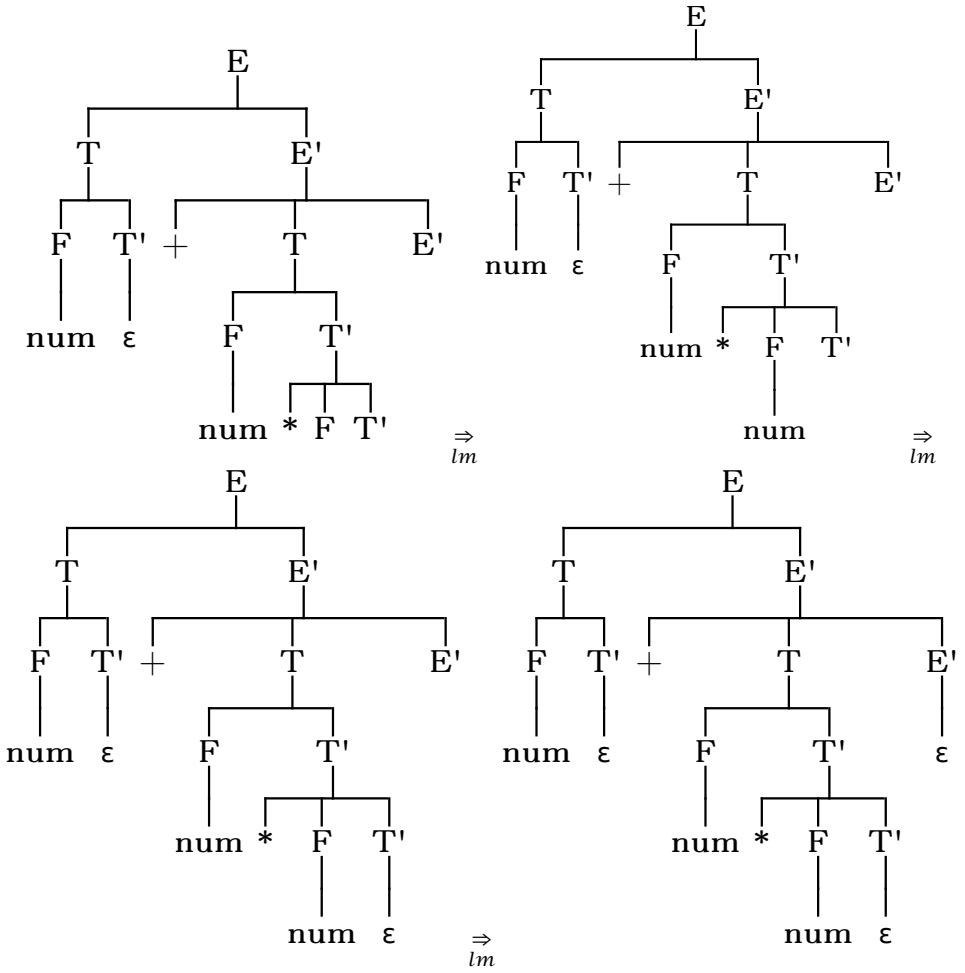
B.2 Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example B.57 [Top-Down Parsing] Sequence of parse trees for the input num + num ∗ num according to the grammar

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid -TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid /FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \text{num} \end{array}$$





B.2.1 Recursive-Descent Parsing

Definition B.2 [Recursive-Descent Parsing] *A* recursive-descent parser *program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Following pseudo code describes the program:*

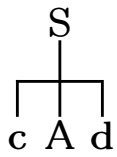
```
void A() {
    Choose an A-production, A → X1 ··· Xk
    for (i = 0 to k) {
        if ( Xi is nonterminal) {
            call procedure Xi();
        } else if ( Xi equals the current input symbol a) {
            advance the input to the next symbol;
        } else {
            /* an error occurred */
        }
    }
}
```

Note that this pseudo code is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

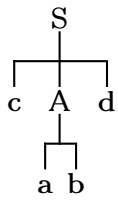
Example B.58 [Recursive-Descent Parsing] Consider the Grammar

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab|a \end{aligned}$$

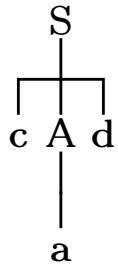
To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labelled S, and the input pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree



The leftmost leave labelled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf la baled A. We expand A using the first alternative $A \rightarrow ab$



We have a match for the second symbol a of w, so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labelled b. Since b does not match d, there is an error and we go back to A to see whether there is another alternative for A. The second alternative produces the tree

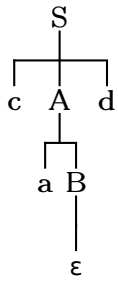


and we are done.

Remark B.2 [Recursive-Descent Parsing] The grammar B.58 has not been left factorised. Following factorised version will allow recursive-descent parsing without backtracking.

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow aB \\ B &\rightarrow b|\epsilon \end{aligned}$$

The parse tree for the input $w = cad$:

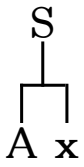


Remark B.3 [Recursive-Descent Parsing] A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go an infinite loop. That is, when we try to expand a nonterminal A, we may eventually find ourselves again trying to expand A without having consumed any input.

Example B.59 [Recursive-Descent Parsing] Consider the Grammar

$$\begin{array}{lcl} S & \rightarrow & Ax \\ A & \rightarrow & x|\epsilon \end{array}$$

To construct a parse tree top-down for the input string $w = x$, begin with a tree consisting of a single node labelled S, and the input pointer pointing to x, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree



Since A may start with x, we expand A and get

We consume x. The remaining of the input string is empty but there still a leaf x in the syntax tree that not been matched. There is an error and we have to go back, replacing A by its second alternative.

and we are done.

Remark B.4 [Recursive-Descent Parsing] The grammar B.59 has not been left factorised. Following factorised version will allow recursive-descent parsing without backtracking.

$$\begin{array}{lcl} S & \rightarrow & xB \\ B & \rightarrow & x|\epsilon \end{array}$$

The parse tree looks as follows:

B.2.2 FIRST and FOLLOW

The construction of top-down parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. FIRST and FOLLOW allow us to choose which production to apply, based on the next input.

Definition B.3 [FIRST] *Let $G = (V, T, P, S)$ be a CFG ant let $w \in (V \cup T)^*$ be a string of grammar symbols.*

$$FIRST(w) = \begin{cases} \{a \in T : w \stackrel{*}{\Rightarrow} av, v \in (V \cup T)^*\} \cup \{\epsilon\} & \text{if } w \stackrel{*}{\Rightarrow} \epsilon \\ \{a \in T : w \stackrel{*}{\Rightarrow} av, v \in (V \cup T)^*\} & \text{else} \end{cases}$$

Definition B.4 [FOLLOW] *Let $G = (V, T, P, S)$ be a CFG ant let $A \in V$ be a nonterminal.*

$$FOLLOW(A) = \{a \in T : S \stackrel{*}{\Rightarrow} \alpha A a \beta, \alpha, \beta \in (V \cup T)^*\}$$

Algorithm B.3 [FIRST] Let $G = (V, T, P, S)$ be a CFG.

1. If X is a terminal, then FIRST$(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), FIRST(Y_2), ... FIRST(Y_{i-1}). If ϵ is in all of FIRST(Y_j), $j = 1, 2, \ldots k$, then add ϵ to FIRST(X)
3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X)

Algorithm B.4 [FOLLOW] Let $G = (V, T, P, S)$ be a CFG.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right end marker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is an element of FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example B.60 [FIRST/FOLLOW] Consider the non-left-recursive grammar B.54. Then:

1. FIRST(F) = FIRST(T) = FIRST(E) = {(, num}. To see why, note that the two productions for F have bodies that start with these two terminal symbols, num and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).
2. FIRST$(E') = \{+, -, \epsilon\}$. Follows directly from the three E'-productions.
3. FIRST$(T') = \{*, /, \epsilon\}$. Follows directly from the three T'-productions.

4. FOLLOW(E) = FOLLOW(E') = $\{), \$\}$. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is an element of FOLLOW(E). For FOLLOW(E'), note this nonterminal appears only at the end of bodies of E-productions. Thus FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW(T') = $\{+, -,), \$\}$. Notice that T appears in bodies only followed by E'. Thus everything, except ϵ that is in FIRST(E') must be in FOLLOW(T), that explains the symbols + and -. However, since FIRST(E') contains ϵ, and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the end of the T-productions, it must be that FOLLOW(T) = FOLLOW(T')

6. FOLLOW(F) = $\{+, -, *, /,), \$\}$. Reasoning analogous to point (5).

Remark B.5 [Computing FIRST and FOLLOW] It is difficult to compute FIRST and FOLLOW without using a computer. Most compiler construction tools like JAVACC [JCC96] can compute FIRST and FOLLOW and also detect left-recursions automatically.

Exercise B.3 [First and Follow] *Consider the context free grammar:*

$$S \to SS+ \quad | \quad SS* \quad | \quad a$$

Compute FIRST and FOLLOW.

B.3 LL(1)-Grammars

We start withe two important rules:

Rule B.1 [FIRST] A CFG G satisfies the FIRST-rule if for each production $A \to w_1|w_2|...|w_n$,

$$\text{FIRST}(w_i) \cap \text{FIRST}(w_j) = \varnothing \quad \text{for} \quad i \neq j$$

Rule B.2 [FOLLOW] A CFG G satisfies the FOLLOW-rule if for each nonterminal A such that $\epsilon \in \text{FIRST}(A)$,

$$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \varnothing$$

Definition B.5 [LL(1)-Grammars] *A* LL(1)*-grammar is a grammar satisfying the rules B.1 and B.2.*

Remark B.6 [LL(1)-Grammars] Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for LL(1)-grammars. The first L in LL(1) stands for scanning the input from left to right, the second L for producing a leftmost derivation, and the 1 for using one input symbol of lookahead at each step to make parsing decisions.

Remark B.7 [LL(k)-Grammars] The first L in LL(k) stands for scanning the input from left to right, the second L for producing a leftmost derivation, and the k for using k input symbols of lookahead at each step to make parsing decisions.

Definition B.6 [PREDICT] *Let $A \to w$ be a production.*

$$PREDICT(A \to w) = \begin{cases} (FIRST(w) - \epsilon) \cup FOLLOW(w) & if \quad \epsilon \in FIRST(w) \\ FIRST(w) & otherwise \end{cases}$$

PREDICT($A \rightarrow w$) contains all terminal expected when processing production $A \rightarrow w$.

Definition B.7 [LL(1) Parsing] *A* recursive LL(1) parser *program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. The program uses a variable called* lookahead *pointing to the actual position of the input string. Following pseudo code describes the program:*

```
void A() {
    if (∃A → X_1 ··· X_k : lookahead ∈ PREDICT(A → X_1 ··· X_k)) {
        for (i = 0 to k) {
            if ( X_i is nonterminal) {
                call procedure X_i()
            } else if ( X_i = lookahead) {
                lookahead = nextSymbol()
            } else {
                /* syntax error (unexpected token) */
            }
        }
    } else {
        /* syntax error (unexpected token) */
    }
}
```

Note that this pseudo code is now deterministic, since the grammar is LL(1).

Example B.61 [Recursive Parser for Expressions]

Following JAVA-classes contain a complete implementation of a recursive parser for the grammar B.54.

The terminal symbols of the grammars are implemented as tokens. The tokens are defined using following regular expressions:

```
token: LPAR "\("
token: RPAR "\)"
token: PLUS "+"
token: MINUS "\-"
token: TIMES "\*"
token: DIV "/"
token: NUM "[0-9][0-9]*"
```

JAVA-Class Token

```
 1  public class Token {
 2    public String lexem;
 3    public int type;
 4    public int startLine;
 5    public int startColumn;
 6    public int endLine;
 7    public int endColumn;
 8
 9    public Token(String lexem, int type, int startLine,
10        int startColumn, int endLine, int endColumn) {
11      this.lexem = lexem;
12      this.type = type;
13      this.startLine = startLine;
```

```
14      this.endLine = endLine;
15      this.startColumn = startColumn;
16      this.endColumn = endColumn;
17    }
18
19    public String toString() {
20      return "<[" + lexem + "]," + type + "," + startLine
21        + "," + startColumn + "," + endLine + ","
22        + endColumn + ">";
23    }
24  }
```

Here follows a simple scanner.

JAVA-Class Scanner

```
 1  public class Scanner implements Constants {
 2
 3    private final int[][] packedTable = {
 4      { -40, 1, 2, 3, 4, -1, 5, -1, 6, 7, 7, 7, 7, 7, 7, 7, 7,
 5        7, -70 },
 6      { -128 }, { -128 }, { -128 }, { -128 }, { -128 }, { -128 },
 7      { -48, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, -70 }
 8    };
 9    public int[][] table = null;
10    public int start = 0;
11    public boolean[] accepting = { false, true, true, true, true,
12       true, true, true };
13    public int[] type = { -1, 1, 2, 5, 3, 4, 6, 7 };
14    private final int states;
15    public final int INPUTS = 128;
16    private int line = 1;
17    private int column = 1;
18    private final PushbackReader reader;
19
20    public Scanner(PushbackReader reader) {
21      this.reader = reader;
22      this.states = packedTable.length;
23      this.table = new int[this.states][INPUTS];
24      unpackTable();
25    }
26
27    public Token getNextToken() throws Exception {
28      return execute();
29    }
30
31    public Token execute() throws Exception {
32      Token token = null;
33      int state = start;
34      int matchedState = SE;
35      String lexem = "";
36      int startLine = line;
37      int startColumn = column;
38      int endLine = line;
39      int endColumn = column;
40      boolean found = false;
41      try {
42        int i = reader.read();
```

```
43        String s = "";
44        if (i == -1) {
45          token = new Token("", 0, startLine, startColumn,
46            endLine, endColumn);
47          found = true;
48        }
49        while (i != -1) {
50          char c = (char) i;
51          s += "" + c;
52          if (c == '\n') {
53            line++;
54            column = 1;
55          } else {
56            column++;
57          }
58          state = table[state][c];
59          if (state != SE && accepting[state]) {
60            lexem += s;
61            s = "";
62            matchedState = state;
63            endLine = line;
64            endColumn = column;
65            token = new Token(lexem, type[matchedState],
66              startLine, startColumn, endLine, endColumn);
67            found = true;
68          } else if (state == SE) {
69            break;
70          }
71          i = reader.read();
72        }
73        if (found) {
74          if (s.length() != 0) {
75            reader.unread(s.toCharArray());
76          }
77          line = endLine;
78          column = endColumn;
79        } else if (i == -1) {
80          throw new RuntimeException("unexpected end of file "
81            + "after reading [" + s + "] at line " + endLine
82            + ", column " + endColumn);
83        } else {
84          throw new RuntimeException("lexical error found "
85            + "after reading [" + s + "] at line " + endLine
86            + ", column " + endColumn);
87        }
88      } catch (IOException e) {}
89      return token;
90    }
91
92    public void unpackTable() {
93      for (int i = 0; i < packedTable.length; i++) {
94        int j = 0;
95        for (int n = 0; n < packedTable[i].length; n++) {
96          int r = packedTable[i][n];
97          if (r >= 0) {
98            table[i][j] = r;
99            j++;
```

```
100           } else {
101             r = -r;
102             while (r > 0) {
103               table[i][j] = -1;
104               r--;
105               j++;
106             }
107           }
108         }
109       }
110     }
111   }
```

Helper class.

JAVA-Class Constants

```
 1  public interface Constants {
 2
 3    public final int SE = -1;
 4    public final int EOF = 0;
 5    public final int LPAR = 1;
 6    public final int RPAR = 2;
 7    public final int PLUS = 3;
 8    public final int MINUS = 4;
 9    public final int TIMES = 5;
10    public final int DIV = 6;
11    public final int NUM = 7;
12    public final String[] names = { "EOF", "LPAR", "RPAR",
13      "PLUS", "MINUS", "TIMES", "DIV", "NUM" };
14  }
```

For the implementation of the scanner, we use algorithm B.7 and the results of example B.60. The method `match()` is used to consume the next input symbol.

- Implementation of the E-productions: There is no choice. In the implementation of `E()` we expect `lookahead` to belong to FIRST(E) and we call `T()` and `Ep()`, otherwise, there is an error.
- Implementation of the E'-productions: There is a choice. If `lookahead` is +, we eat + and call `T()` and `Ep()`, if `lookahead` is −, we eat − and call `T()` and `Ep()`. Since FIRST(E') contains ϵ, we must accept) and \$ (they belong to FOLLOW(E')), but we do nothing (they will be processed by other procedures). Any other value of `lookahead` is an error.

JAVA-Class Parser

```
 1  public class Parser implements Constants {
 2
 3    private static Token lookahead;
 4    private final Scanner scanner;
 5
 6    public Parser(Scanner scanner) throws Exception {
 7      this.scanner = scanner;
 8      lookahead = this.scanner.getNextToken();
 9    }
10
11    private void match(int type) {
```

```
12      if (lookahead.type == type) {
13        try {
14          lookahead = scanner.getNextToken();
15        } catch (Exception e) {
16          e.printStackTrace();
17        }
18      } else {
19        throw new RuntimeException();
20      }
21    }
22
23    public void E() {
24      if (lookahead.type == LPAR || lookahead.type == NUM) {
25        T();
26        Ep();
27      } else {
28        throw new RuntimeException(expected(lookahead,
29          "[LPAR] or [NUM]"));
30      }
31    }
32
33    public void T() {
34      if (lookahead.type == LPAR || lookahead.type == NUM) {
35        F();
36        Tp();
37      } else {
38        throw new RuntimeException(expected(lookahead,
39          "[LPAR] or [NUM]"));
40      }
41    }
42
43    public void F() {
44      switch (lookahead.type) {
45        case LPAR:
46          match(LPAR);
47          E();
48          match(RPAR);
49          break;
50        case NUM:
51          match(NUM);
52          break;
53        default:
54          throw new RuntimeException(expected(lookahead,
55            "[LPAR] or [NUM]"));
56      }
57    }
58
59    public void Ep() {
60      switch (lookahead.type) {
61        case PLUS:
62          match(PLUS);
63          T();
64          Ep();
65          break;
66        case MINUS:
67          match(MINUS);
68          T();
```

```
69          Ep();
70          break;
71        case RPAR:
72        case EOF:
73          break;
74        default:
75          throw new RuntimeException(expected(lookahead,
76              "[PLUS], [MINUS], [RPAR] or [EOF]"));
77      }
78    }
79
80    public void Tp() {
81      switch (lookahead.type) {
82        case TIMES:
83          match(TIMES);
84          F();
85          Tp();
86          break;
87        case DIV:
88          match(DIV);
89          F();
90          Tp();
91          break;
92        case PLUS:
93        case MINUS:
94        case RPAR:
95        case EOF:
96          break;
97        default:
98          throw new RuntimeException(expected(lookahead,
99              "[PLUS], [MINUS], [TIMES], [DIV], [RPAR] or [EOF]"));
100     }
101   }
102
103   private String expected(Token token, String s) {
104     String r = "unexpected token [" + names[token.type]
105       + "] at line " + token.startLine;
106     r += " column " + token.startColumn;
107     r += " was expexting " + s;
108     return r;
109   }
110 }
```

Here a small tester:

JAVA-Class `Main`

```
1  public class Main {
2
3    public static void main(String[] args) {
4      try {
5        PushbackReader reader =
6          new PushbackReader(new FileReader(args[0]));
7        Scanner scanner = new Scanner(reader);
8        Parser parser = new Parser(scanner);
9        parser.E();
10     } catch (Exception e) {
11       e.printStackTrace();
```

```
12          }
13       }
14    }
```

Exercise B.4 [LL(1)] *Consider the context free grammar:*

$$S \rightarrow SS+ \quad | \quad SS* \quad | \quad a$$

1. *Transform the grammar into an LL(1) grammar.*
2. *Compute FIRST and FOLLOW for the transformed grammar.*

B.4 LL(1)-Grammars in EBNF

Let $G = (V, T, P, S)$ be a LL(1) grammar and assume that the productions are written in EBNF. We show here how to implement each EBNF constructs ($w, w_i \in (V \cup T)^*$). $P(w)$ denotes the procedure associated with w. See also [Wir86] [Wir96].

1. We associate the following procedure $A()$ to a production $A \rightarrow w_1|w_2|\ldots|w_n$:

```
void A() {
    if (lookahead ∈ (FIRST(w1) − {ε})) {
        P(w1)
    } else if (lookahead ∈ (FIRST(w2) − {ε})) {
        P(w2)
    } else if ...
    ...
    } else if (lookahead ∈ (FIRST(wn) − {ε})) {
        P(wn)
    } else if (lookahead ∈ FOLLOW(A)) {
        nothing()
    } else {
        error()
    }
}
```

2. We associate the following procedure $A()$ to a production $A \rightarrow w_1 w_2 \ldots w_n$:

```
void A() {
    P(w1)
    P(w2)
    ...
    P(wn)
}
```

3. We associate the following procedure $A()$ to a production $A \rightarrow w^*$:

```
void A() {
    while (lookahead ∈ (FIRST(w) − {ε})) {
        P(w)
    }
}
```

4. We associate the following procedure $A()$ to a production $A \rightarrow w^+$:

```
void A() {
    do {
        P(w)
    } while (lookahead ∈ (FIRST(w) − {ϵ}))
}
```

5. We associate the following procedure $A()$ to a production $A \rightarrow w^?$:

```
void A() {
    if (lookahead ∈ (FIRST(w) − {ϵ})) {
        P(w)
    }
}
```

6. We associate the following procedure $A()$ to a production $A \rightarrow B$ ($B \in V$):

```
void A() {
    B()
}
```

7. We associate the following procedure $A()$ to a production $A \rightarrow \epsilon$:

```
void A() {
    nothing()
}
```

8. We associate the following procedure $A()$ to a production $A \rightarrow a$ ($a \in T$):

```
void A() {
    if (lookahead = a) {
        match(a)
    } else {
        error()
    }
}
```

Appendix C

JAVACC

In practise, the scanning and parsing phases of a compiler are handled by code that is generated by a parser generator. One parser generator for Java is called JAVACC. Here's a tutorial[1].

C.1 Introduction

JAVACC is a lexer and parser generator for LL(k) grammars. You specify a language's lexical and syntactic description in a `.jj` file, then run javacc on the `.jj` file. You will get seven java files as output, including a lexer and a parser.

This page helps you get started using JAVACC. You still need to read the on line documentation to do anything really useful with the tool though.

We'll look at three things you can do with JAVACC

1. Do a simple syntax check only
2. Make an actual interpreter
3. Generate code

C.2 Getting Started

Make sure you have (or get) *at least* version 4.0.

The home page for JAVACC is `https://javacc.dev.java.net/`.

Download it. Unzip it. You may want to make the `javacc` script accessible from your path.

C.3 A First Example: Syntax Checking

Let's start with the language of integer expressions with addition and multiplication, with addition having lower precedence. The typical LL(1) grammar for this is:

```
Microsyntax
  SKIP -> [\x0d\x0a\x20\x09]
  NUM -> [0-9]+
  TOKEN -> [+*()] | NUM
```

[1] From `http://www.cs.lmu.edu/\char126\relaxray/notes/javacc/`

```
Macrosyntax
  E -> T ("+" T)*
  T -> F ("*" F)*
  F -> NUM | "(" E ")"
```

Make a file called `Exp1.jj` like so:

JAVACC-Specification `SyntaxChecker`

```
1   PARSER_BEGIN(SyntaxChecker)
2
3   public class SyntaxChecker {
4     public static void main(String[] args) {
5       try {
6         new SyntaxChecker(new java.io.StringReader(args[0])).S();
7         System.out.println("Syntax is okay");
8       } catch (Throwable e) {
9         System.out.println("Syntax check failed: " +
10          e.getMessage());
11      }
12    }
13  }
14
15  PARSER_END(SyntaxChecker)
16
17  SKIP:  { " " | "\t" | "\n" | "\r"          }
18  TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
19
20  void S(): {} { E() <EOF>       }
21  void E(): {} { T() ("+" T())*    }
22  void T(): {} { F() ("*" F())*    }
23  void F(): {} { <NUM> | "(" E() ")" }
```

This is pretty much a minimal example. You must define a parser class between the markers `PARSER_BEGIN` and `PARSER_END`, and you must specify tokens in a `TOKEN`: clause, and parsing methods must be defined, one for each non-terminal in the grammar. The parsing functions look rather like the EBNF for a grammar: you'll just notice non-terminals look like function calls, tokens look like themselves, and you have the usual EBNF metasymbols like "|" (choice), "*" (Kleene star), and "+" (Kleene plus).

To process the JJ file, invoke

```
javacc Exp1.jj
```

and notice seven files are output. The one called `SyntaxChecker.java` should be of some interest; take a look at it! (Yeah, it's pretty ugly, but you didn't write it yourself, right?)

Some example runs:

```
$ javacc Exp1.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Exp1.jj . . .
Parser generated successfully.
$ javac *.java
Note: SyntaxChecker.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java SyntaxChecker "100"
```

```
Syntax is okay
$ java SyntaxChecker "8 * 3 * (2 + 2345234)"
Syntax is okay
$ java SyntaxChecker "(((3)) * 12)"
Syntax is okay
$ java SyntaxChecker "this doesn't work"
Syntax check failed: Lexical error at line 1, column 1.
                     Encountered: "t" (116),
 after : ""
$ java SyntaxChecker "(x = 3) + 111"
Syntax check failed: Lexical error at line 1, column 2.
                     Encountered: "x" (120),
 after : ""
$ java SyntaxChecker "1 2"
Syntax check failed: Encountered "2" at line 1, column 3.
Was expecting one of:
  <EOF>
  "+" ...
  "*" ...
```

If you want to get fancy with lexing, you can add comments to the language. For example:

```
SKIP: {
  " "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n"|"\r")>
}
```

For lexical analysis, JAVACC allows other sections besides `TOKEN` and `SKIP`. There are also ways to make "private" tokens, and write complex regular expressions. See the JAVACC documentation for details. Also see the mini-tutorial on the JAVACC site for tips on writing lexer specifications from which JAVACC can generate efficient code.

C.4 Left Recursion

JAVACC cannot parse grammars that have left-recursion. It just can't. Suppose you tried to use the non-LL(k) grammar for the language above:

```
Macrosyntax
  E -> T | E "+" T
  T -> F | T "*" F
  F -> NUM | "(" E ")"
```

If you changed your `.jj` file accordingly and ran JAVACC, you would see this:

```
$ javacc BadLeftRecursion.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file BadLeftRecursion.jj . . .
Error: Line 22, Column 1: Left recursion detected: "T... --> T..."
Error: Line 21, Column 1: Left recursion detected: "E... --> E..."
Detected 2 errors and 0 warnings.
```

You must remove left recursion before writing your grammar rules in JAVACC. The general approach is to replace rules of the form

```
A -> a | Ax
```

with

```
A -> a (x)*
```

C.5 Lookahead

Let's add identifiers and assignment expressions to our language:

```
E -> id ":=" E | T ("+" T)*
T -> F ("*" F)*
F -> NUM | ID | "(" E ")"
```

If you naively write JAVACC for this grammar, JAVACC will say something like

```
$ javacc NotEnoughLookahead.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file NotEnoughLookahead.jj . . .
Warning: Choice conflict involving two expansions at
     line 24, column 16 and line 24, column 32 respectively.
     A common prefix is:
     Consider using a lookahead of 2 for earlier expansion.
Parser generated with 0 errors and 1 warnings.
```

What this means is that when expanding an E and looking at an id, we wouldn't know if that id is starting an assignment or is just a variable, unless we examine not just the id, but also the following token. So the parser needs to look at the next *two* symbols.

Understand that *choice points* can appear at various places within a grammar. They obviously appear at "|" separators, but also in ()*, ()+ and ()? constructs too.

JAVACC-Specification SyntaxChecker1

```
1  PARSER_BEGIN(SyntaxChecker1)
2
3  public class SyntaxChecker1 {
4    public static void main(String[] args) {
5      try {
6        new SyntaxChecker1(new java.io.StringReader(args[0])).S();
7        System.out.println("Syntax is okay");
8      } catch (Throwable e) {
9        System.out.println("Syntax check failed: " +
10         e.getMessage());
11     }
12   }
13 }
14
15 PARSER_END(SyntaxChecker1)
16
17 SKIP: { " " | "\t" | "\n" | "\r" }
```

```
18  TOKEN: {
19    "(" | ")" | "+" | "*" | ":="
20    | <NUM: (["0"-"9"])+> | <ID: (["a"-"z"])+>
21  }
22
23  void S(): {} { E() <EOF>                    }
24  void E(): {} { LOOKAHEAD(2) <ID> ":=" E() | T() ("+" T())* }
25  void T(): {} { F() ("*" F())*                }
26  void F(): {} { <NUM> | <ID> | "(" E() ")"          }
```

Sometimes *no* number of lookahead tokens is sufficient for the parser. Then you'd need to use *syntactic* lookahead. For sophisticated, or bizarre, parsing, sometimes *semantic* lookahead is needed. See the JAVACC Lookahead Mini-Tutorial for details.

C.6 Writing An Interpreter

If the syntax checker above looked like it had a lot of extra markup in its parsing functions, that's because parsers are supposed to *do* stuff while parsing. Parsing functions can take in parameters, return results, and invoke blocks of arbitrary Java code. Here's how we can actually evaluate the expressions, or, as they say, *write an interpreter*.

JAVACC-Specification `Evaluator`

```
1   PARSER_BEGIN(Evaluator)
2
3   public class Evaluator {
4     public static void main(String[] args) throws Exception {
5       int result =
6         new Evaluator(new java.io.StringReader(args[0])).S();
7       System.out.println(result);
8     }
9   }
10
11  PARSER_END(Evaluator)
12
13  SKIP:  { " " | "\t" | "\n" | "\r"          }
14  TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
15
16  int S(): {int sum;}
17  {
18    sum=E() <EOF> {return sum;}
19  }
20
21  int E(): {int sum, x;}
22  {
23    sum=T() ("+" x=T() {sum += x;} )* {return sum;}
24  }
25
26  int T(): {int sum, x;}
27  {
28    sum=F() ("*" x=F() {sum *= x;} )* {return sum;}
29  }
30
31  int F(): {int x; Token n;}
32  {
33    n=<NUM> {return Integer.parseInt(n.image);}
```

```
34   |
35     "(" x=E() ")" {return x;}
36   }
```

Some example runs

```
$ javacc Exp2.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Exp2.jj . . .
Parser generated successfully.
$ javac *.java
Note: Evaluator.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java Evaluator "34"
34
$ java Evaluator "34 * 2 + ((3 + 11))"
82
```

C.7 Generating an Abstract Syntax Tree

A quick and dirty way to do what most real parsers do, namely generate an abstract syntax tree, is to embed the tree classes in the `.jj` file, like so:

JAVACC-Specification `Parser`

```
1   PARSER_BEGIN(Parser)
2
3   public class Parser {
4     public static void main(String[] args) throws Exception {
5       Exp result =
6         new Parser(new java.io.StringReader(args[0])).S();
7       System.out.println(result);
8     }
9   }
10
11  // Classes defining the Abstract Syntax Tree
12  abstract class Exp {}
13  class Num extends Exp {
14    int value;
15    Num(int v) {
16      value = v;
17    }
18    public String toString() {
19      return value + "";
20    }
21  }
22  class BinaryExp extends Exp {
23    String op;
24    Exp left, right;
25    BinaryExp(String o, Exp l, Exp r) {
26      op = o;
27      left = l;
28      right = r;
29    }
```

```
30    public String toString() {
31      return "(" + op + " " + left + " " + right + ")";
32    }
33  }
34
35  PARSER_END(Parser)
36
37  SKIP:  { " " | "\t" | "\n" | "\r"          }
38  TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
39
40  Exp S(): {Exp e;}
41  {
42    e=E() <EOF> {return e;}
43  }
44
45  Exp E(): {Exp e1; Exp e2;}
46  {
47    e1=T() ("+" e2=T() {e1 = new BinaryExp("+", e1, e2);} )*
48    {return e1;}
49  }
50
51  Exp T(): {Exp e1; Exp e2;}
52  {
53    e1=F() ("*" e2=F() {e1 = new BinaryExp("*", e1, e2);} )*
54    {return e1;}
55  }
56
57  Exp F(): {Exp e; Token n;}
58  {
59    n=<NUM> {return new Num(Integer.parseInt(n.image));}
60  |
61    "(" e=E() ")" {return e;}
62  }
```

The `toString()` method is convenient, though it might trick you into believing we made a code generator that targets LISP. But the tree is really there.

```
$ java Parser 54
54
$ java Parser 54+3
(+ 54 3)
$ java Parser "54 + 3 * 2 + 7"
(+ (+ 54 (* 3 2)) 7)
$ java Parser "54 + 3 * (2 + 7) * 33"
(+ 54 (* (* 3 (+ 2 7)) 33))
```

C.8 Integrating the Parser into a Compiler

In a real compiler, you don't dump a main method into the parser. You'd make a nice function called `parse()` perhaps, and call that from your own code. And you'd define the tree classes in their own files.

Exercise C.1 [JAVACC] *Write a* JAVACC *specification for following grammars. Run* JAVACC *and look carefully at the warning or error message.*

1. *The grammar of example B.52.*

2. *The grammar of example B.58.*
3. *The grammar of example B.59.*

Exercise C.2 [Beautifier]

Develop using JAVACC *a* JAVA *beautifier, i.e. a program that reads a* JAVA *source file and produces a* XHTML *file that contains the Java source with some parts displayed in colour, for example:*

1. *Keywords in magenta.*
2. *Class/type names in purple. You may assume that class/type names start with a capital letter.*
3. *Method names in red. You may assume that method names start with a lower case letter.*
4. *Other identifiers in blue. You may assume that identifier names start with a lower case letter.*
5. *String constants in green.*
6. *Character constants in light green.*

Appendix D

Final Project

In this project, you will implement a small scanner generator written in JAVA.

D.1 Requirements

D.1.1 Specification Regular Expressions

Your scanner generator should support following regular constructions over the standard 8 bit ASCII character set:

`*`	Iteration
`|`	Choice
`()`	Parentheses
`x`	The character `x`
`\0`	The empty string
`\n`	The newline character
`\r`	The carriage return character
`\t`	The tabulator character
`\x`	The character `x` (escape)
`.`	Any ASCII-Character (0-127)
`[x-yz]`	Range expression and/or single characters
`[^x-yz]`	Any character except `x-y` t and `z`

Syntax

The input format is defined according to the CFG $G = (V, T, P, S)$, where

$$V = \{S, RE, Choice, Sequence, Iteration, Atomic, Range, \}$$

$$T = \{token, skip, semi, name, quote, bar, star, lPar, rPar, lBraket, rBraket, hat, dash, character\}$$

Productions:

$$
\begin{array}{rcl}
S & \rightarrow & (tokensemi\ \ (name)?\ \ RE)+ \\
RE & \rightarrow & quote\ \ Choice\ \ quote \\
Choice & \rightarrow & Sequence\ \ (bar\ \ Sequence)* \\
Sequence & \rightarrow & (Iteration)+ \\
Iteration & \rightarrow & Atomic\ \ (star)? \\
Atomic & \rightarrow & lPar\ \ Choice\ \ rPar\ \ | \\
 & & Range\ \ | \\
 & & character \\
Range & \rightarrow & lBraket\ \ (hat)?\ \ (character\ \ (dash\ \ character)?)*\ \ rBraket
\end{array}
$$

The terminal symbols T are defined by following regular expressions using our notation:

token	`token|skip`
semi	`:`
name	`[A-Z]([A-Z0-9])*`
quote	`"`
bar	`|`
star	`\*`
lPar	`\(`
rPar	`\(`
lBraket	`\[`
rBraket	`\]`
hat	`\^`
dash	`\-`
character	`\.|\\\.`

Each line of the specification consists of three parts:

1. the first part consists of the reserved words `token` or `skip` followed by a semicolon (`:`) an a whitespace.
2. the second optional part is the name of the regular expression followed by a whitespace.
3. the third part is the quoted regular expression.

Remark D.1 [LL(1)] This grammar is a LL(1) grammar and can be used without change to build a recursive-descent parser.

Semantic

- `skip` tokens do not have a name and denote parts of the input to be ignored, e.g. white spaces, etc.
- `token` and `skip` tokens may occur in any position of the specification.
- A regular expression may be matched by more than one token.

An input may be matched by more than one RE! For example, using the following specifications:

```
token: KW "if|while"
token: ID "[a-z][a-z0-9]*"
token: OP "+|\-|\*|/"
```

the input `if` will be matched by the RE's `KW` and `ID`. If this happens, the answer of the scanner will be the RE listed first in the specification file, here `KW`.

Example

Example D.62 [Token Specification]

This is an example of a specification for the tokens used for regular expressions.

Token specification `re`

```
1  token: BAR "\|"
2  token: STAR "\*"
3  token: LPAR "\("
4  token: RPAR "\)"
5  token: LBRAKET "\["
6  token: RBRAKET "\]"
7  token: HAT "\^"
8  token: DASH "\-"
9  token: CHARACTER ".|\\."
```

D.1.2 Generated Code

Using the specifications of example D.62 scanner generator must generate following JAVA classes:

The class `Token` is independent of the specification file. `Token` defines token objects. Such an object knows to which regular expression it corresponds (`type` attribute), knows the lexeme matched by the regular expression (`lexem` attribute) and knows where the lexem has been matched in the input file (`startLine`, `startColumn`, `endLine` and `endColumn` attributes).

JAVA-Class `Token`

```
1  public class Token {
2    public String lexem;
3    public int type = -1;
4    public int startLine;
5    public int startColumn;
6    public int endLine;
7    public int endColumn;
8    public boolean skip = false;
9
10   public Token(String lexem, int type, int startLine,
11       int startColumn, int endLine, int endColumn) {
12     this.lexem = lexem;
13     this.type = type;
14     this.startLine = startLine;
15     this.endLine = endLine;
16     this.startColumn = startColumn;
17     this.endColumn = endColumn;
18   }
19
20   public String toString() {
21     return "<[" + lexem + "],<" + type + ","
22         + startLine + "," + startColumn
23         + "," + endLine + "," + endColumn + ">";
24   }
25
26   public boolean isSkip() {
27     return skip;
28   }
29
```

```
30    public void setSkip(boolean skip) {
31      this.skip = skip;
32    }
33  }
```

The classes RegularExpressionConstants and RegularExpressionScanner are dependent from the specification file. They contain defines token type constants and the scanner.

JAVA-Class RegularExpressionConstants

```
 1  public interface RegularExpressionConstants {
 2
 3    /** the error state */
 4    public final int SE = -1;
 5    /** the EOF token */
 6    public final int EOF = 0;
 7    /** the BAR token */
 8    public final int BAR = 1;
 9    /** the STAR token */
10    public final int STAR = 2;
11    /** the LPAR token */
12    public final int LPAR = 3;
13    /** the RPAR token */
14    public final int RPAR = 4;
15    /** the LBRAKET token */
16    public final int LBRAKET = 5;
17    /** the RBRAKET token */
18    public final int RBRAKET = 6;
19    /** the HAT token */
20    public final int HAT = 7;
21    /** the DASH token */
22    public final int DASH = 8;
23    /** the CHARACTER token */
24    public final int CHARACTER = 9;
25    /** the regular expressions */
26    public final String[] expressions =
27      { "<EOF>", "\\|", "\\*", "\\(", "\\)", "\\[", "\\]",
28        "\\^", "\\-", ".|\\." };
29    /** the token names */
30    public final String[] names =
31      { "EOF", "BAR", "STAR", "LPAR", "RPAR", "LBRAKET",
32        "RBRAKET", "HAT", "DASH", "CHARACTER" };
33    /** the skip array expressions */
34    public final boolean[] skip =
35      { false, false, false, false, false, false, false,
36        false, false, false};
37  }
```

JAVA-Class RegularExpressionScanner

```
1  import java.io.IOException;
2  import java.io.PushbackReader;
3
4  public class RegularExpressionScanner
5    implements RegularExpressionConstants {
6    private int[][] packedTable = {
7      {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
 8        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 9        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3,
10        4, 1, 1, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1,
11        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
12        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
13        1, 1, 1, 1, 1, 1, 1, 6, 1, 7, 8, 1, 1,
14        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
15        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 9, 1,
16        1, 1},
17      {-128},
18      {-128},
19      {-128},
20      {-128},
21      {-128},
22      {-128},
23      {-128},
24      {-128},
25      {-128}
26    };
27    public int[][] table = null;
28    public int start = 0;
29    public boolean[] accepting =
30      {false, true, true, true, true, true,
31       true, true, true, true};
32    public int[] type = {-1, 9, 3, 4, 2, 8, 5, 6, 7, 1};
33    private int states;
34    public final int INPUTS = 128;
35    private int line = 1;
36    private int column = 1;
37    private final PushbackReader reader;
38
39    public RegularExpressionScanner(PushbackReader reader) {
40      this.reader = reader;
41      this.states= packedTable.length;
42      this.table = new int[this.states][INPUTS];
43      unpackTable();
44    }
45
46    public Token getNextToken() throws IOException {
47      Token token;
48      do {
49        token = execute();
50        if (!token.isSkip()) {
51          return token;
52        }
53      } while (token.type != 0);
54      return token;
55    }
56
57    public Token execute() throws IOException {
58      Token token = null;
59      int state = start;
60      int matchedState = SE;
61      String lexem = "";
62      int startLine = line;
63      int startColumn = column;
64      int endLine = line;
```

```
65      int endColumn = column;
66      boolean found = false;
67      try {
68        int i = reader.read();
69       String s = "";
70        if (i == -1) {
71          token = new Token("", 0, startLine, startColumn,
72                            endLine, endColumn);
73          found = true;
74        }
75        while (i != -1) {
76          char c = (char) i;
77          s += "" + c;
78          if (c == '\n') {
79            line++;
80            column = 1;
81          } else {
82            column++;
83          }
84          state = table[state][c];
85         if (state != SE && accepting[state]) {
86            lexem += s;
87            s = "";
88            matchedState = state;
89            endLine = line;
90            endColumn = column;
91            token = new Token(lexem, type[matchedState],
92                              startLine, startColumn,
93                              endLine, endColumn);
94           found = true;
95          } else if (state == SE) {
96            break;
97         }
98          i = reader.read();
99        }
100       if (found) {
101         if (s.length() != 0) {
102           reader.unread(s.toCharArray());
103         }
104         line = endLine;
105         column = endColumn;
106       } else if (i == -1) {
107         throw new
108           RuntimeException("unexpected end of file after reading ["
109             + s + "] at line " + endLine + ", column " + endColumn);
110       } else {
111         throw new
112           RuntimeException("lexical error found after reading ["
113             + s + "] at line " + endLine + ", column " + endColumn);
114       }
115     } catch (IOException e) {
116       e.printStackTrace();
117     }
118     return token;
119   }
120
121   public void unpackTable() {
```

```
122       for (int i = 0; i < packedTable.length; i++) {
123         int j = 0;
124         for (int n = 0; n < packedTable[i].length; n++) {
125           int r = packedTable[i][n];
126           if (r >= 0) {
127             table[i][j] = r;
128             j++;
129           } else {
130             r = -r;
131             while (r > 0) {
132               table[i][j] = -1;
133               r--;
134               j++;
135             }
136           }
137         }
138       }
139     }
140   }
```

D.1.3 Program Call

The scanner generator class must be named `TokenManagerGenerator`. If the name of the specification file is `re.sg`, the programme call:

```
java TokenManagerGenerator RegularExpression re.sg
```

should generate exactly the three `Java` classes of section D.1.2.

D.2 Templates

The JAR file `prog/fp/fp.jar` contains an ECLIPSE project called `final`. This project contains JAVA packager for DFA's, NFA's, RE's, DFA minimisation, NFA to DFA transformation, RE to NFA transformation and a parser for RE's. You may use these packages for your project.

D.2.1 Unsupported Features

Following features are not supported:

- Range expressions (e.g. `[a-ch-z]`), any character expressions (`.`) and empty expressions (`\0`) are not supported.
- The regular expression parser does not supports the RE's mentioned above.
- Only naive DFA minimisation is supported. You will have to implement the Hopcroft minimiser.
- There is no specification file parser.
- The `match` method of the `DFA` class only checks whether a give input can be matched by a DFA. `match` should return a token corresponding to the longest prefix of the input that can be matched.

D.3 Hints

D.3.1 Completing RE Classes

You will have to add the classes `AnyCharacterExpression`, `EmptyExpression` and `RangeExpression`. Furthermore, yo will need to complete the visitors `ToString` and `RegularExpressionToNFA`. This is almost straightforward.

D.3.2 Finding out the correct RE

We need to find out which RE has been matched among all RE's defined in the specification file.

Here, you may need to add a new construct to the Thompson construction. You may build the NFA in the following way:

- For each RE, you will build the NFA in the standard way.
- You add a new start state with an ϵ-transition to each start state of the NFA's constructed above. If the specification file contains n RE's, your NFA will have one start state (usually state 0) and n accepting states (usually numbered 1 to n), one for each RE.

Note that when matching a given input, the NFA may reach a final state containing more than one accepting state. According to the requirements, the matched RE will correspond to the smallest accepting state, i.e. the RE listed first in the specification file.

D.3.3 NFA to DFA Transformation

The DFA must be able to find out, which accepting state of the NFA has been matched, and thus which is the correct RE matching the input. You may need an array called `type` containing this information for each accepting state of the DFA (see also section D.3.2).

D.3.4 DFA Minimisation

Use the instruction of remark 3.3 to implement the Hopcroft minimising algorithm.

When minimising the DFA, accepting states of different type (see section D.3.2) of the original DFA may be merged. If it happens, it is no more possible to find out, which RE has been matched. For that reason, accepting states of different type **must** be initialised as distinguishable!

D.3.5 Parsing

You will need a parser for reading the specification file. You can just use the grammar of section D.1.1 to do the work. This grammar is LL(1) and can be used directly to implement a recursive-descent parser by hand or using JAVACC. The parser may read the quoted regular expression into a JAVA string and use the class `RegularExpressionParsion` to parse the RE (you may need to complete its code for range expressions, etc.).

D.3.6 Code generation

Having a look to the files of sectionD.1.2, you may notice that most parts of these files are independent of the specification file.

For example, in the class `RegularExpressionScanner`, the only variable components are the class name, the name of the implemented class and the values of the attributes `packedTable`, `start`, `accepting` and `type`.

The rest can be generated independently.

There are two possibilities to do the job:

- You put all lines of the file to be generated into print statements.
- You put a marker, e.g. the string `$$$$$` at each place containing a variable value, thus building a skeleton file. This file can be processed with the `Java` class `java.util.Scanner` to patch the variable parts.

D.3.7 Bootstrapping

Once completed, you may rewrite parts of your project using the project itself anywhere you need to scan an input.

D.4 Administrative Remarks

- This project can be completed in groups of maximal 3 students.
- This project, particularly the design and the tests must be well documented (around 20 A4 pages)
- All modification and ad-dons to the ECLIPSE project `scanner` must be fully documented (JAVADOC).
- The project must be submitted in form of an executable JAR file containing all sources and the generated JAVADOC files.

Appendix E

Abbreviations

- **BNF** Backus-Naur Form.
- **CFG** Context-Free Grammar.
- **CFL** Context-Free Language.
- **DFA** Deterministic Finite Automaton.
- **EBNF** Extended Backus-Naur Form.
- **ECLOSE** The ϵ-Closure.
- **ID** Instantaneous Description.
- **LL(k)** A parser parsing the input from left to right, and constructing a leftmost derivation using a lookahead of k tokens.
- **LM** Leftmost.
- $\mathbb{N}$ Natural numbers.
- **NFA** Nondeterministic Finite Automaton.
- **PDA** Pushdown Automaton.
- $\mathbb{Q}$ Rational numbers.
- $\mathbb{R}$ Real numbers.
- **RE** Regular Expression.
- **RM** Rightmost.
- $\mathbb{Z}$ Integer numbers.

Index

Bibliography

[ALSU08] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compiler*. Pearson Studium, München, 2008.

[ASU07] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Boston, 2nd edition, 2007.

[CE63] L.W. Cohen and G. Ehrlich. *The Structure of the Real Number System*. The University Series in Undergraduate Mathematics. D. van Nostrand, Princeton, 1963.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. http://www.chomsky.info/articles/195609--.pdf.

[Fle06] P. Flener. Formal languages and automata theory, 2006. http://user.it.uu.se/~pierref/courses/FLAT/.

[Goe31] K. Goedel. On formally undecidable propositions of principia mathematica and related systems. *Monatshefte fuer Mathematik und Physik*, 38:173–98, 1931. Translated by M. Hirzel http://www.research.ibm.com/people/h/hirzel/papers/canon00-goedel.pdf.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Automata Theory, Languages and Computation*. Pearson Education, Boston, 3rd edition, 2007.

[JCC96] JavaCC, java compiler compiler [tm], 1996. https://javacc.dev.java.net.

[Les75] M.E. Lesk. Lex – a lexical analyser generator. Technical Report Computer Science Technical Report 39, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[Rm] S. Rodger and many. Jflap. http://jflap.org.

[Tur37] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1937. http://www.comlab.ox.ac.uk/activities/ieg/e-library/sources/tp2-ie.pdf.

[vL90a] J. van Leeuwen. *Handbook of Theoretical Computer Science, Algorithms and Complexity*, volume A. Elsevier, Amsterdam, 1990.

[vL90b] J. van Leeuwen. *Handbook of Theoretical Computer Science, Formal Models and Semantics*, volume B. Elsevier, Amsterdam, 1990.

[Wik] Wikipedia. http://en.wikipedia.org.

[Wir86] N. Wirth. *Compilerbau*. Teubner, Stuttgart, 1986.

[Wir96] N. Wirth. *Compiler Construction*. Addison-Wesley, Reading, 1996. See also http://www-old.oberon.ethz.ch/WirthPubl/CBEAll.pdf.

Contents