Solutions for Selected Exercises from Basics of Compiler Design

Torben Æ. Mogensen

Last update: May 28, 2009

1 Introduction

This document provides solutions for selected exercises from "Basics of Compiler Design".

Note that in some cases there can be several equally valid solutions, of which only one is provided here. If your own solutions differ from those given here, you should use your own judgement to check if your solution is correct.

2 Exercises for chapter 2

Exercise 2.1

a) 0^*42

b) The number must either be a one-digit number, a two-digit number different from 42 or have at least three significant digits:

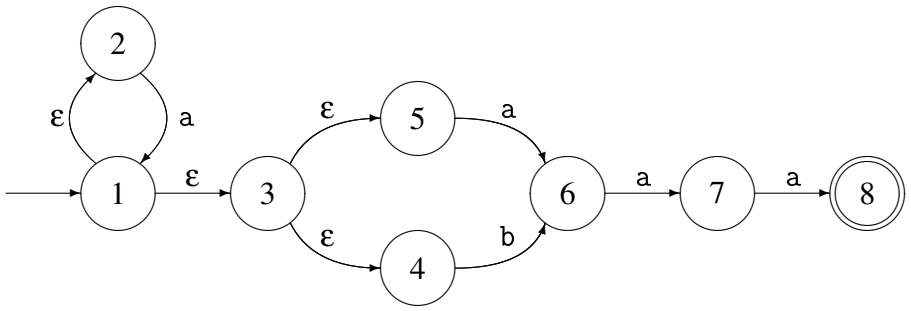
$$0^*([0-9] \mid [1-3][0-9] \mid 4[0-1] \mid 4[3-9] \mid [5-9][0-9] \mid [1-9][0-9][0-9]^+)$$

c) The number must either be a two-digit number greater than 42 or have at least three significant digits:

$$0^*(4[3-9] \mid [5-9][0-9] \mid [1-9][0-9][0-9]^+)$$

Exercise 2.2

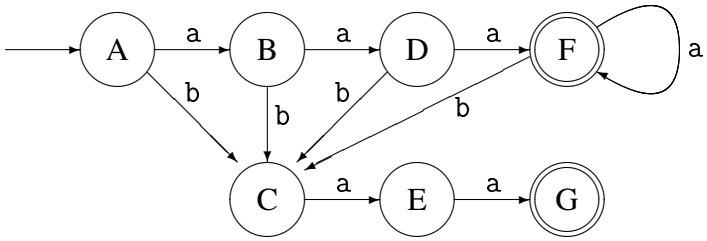
a)



b)

$$
\begin{array}{rcllcl}
A & = & \varepsilon\text{-}closure(\{1\}) & = & \{1,2,3,4,5\} & \\
B & = & move(A,\mathtt{a}) & = & \varepsilon\text{-}closure(\{1,6\}) & = & \{1,6,2,3,4,5\} \\
C & = & move(A,\mathtt{b}) & = & \varepsilon\text{-}closure(\{6\}) & = & \{6\} \\
D & = & move(B,\mathtt{a}) & = & \varepsilon\text{-}closure(\{1,6,7\}) & = & \{1,6,7,2,3,4,5\} \\
 & & move(B,\mathtt{b}) & = & \varepsilon\text{-}closure(\{6\}) & = & C \\
E & = & move(C,\mathtt{a}) & = & \varepsilon\text{-}closure(\{7\}) & = & \{7\} \\
 & & move(C,\mathtt{b}) & = & \varepsilon\text{-}closure(\{\}) & = & \{\} \\
F & = & move(D,\mathtt{a}) & = & \varepsilon\text{-}closure(\{1,6,7,8\}) & = & \{1,6,7,8,2,3,4,5\} \\
 & & move(D,\mathtt{b}) & = & \varepsilon\text{-}closure(\{6\}) & = & C \\
G & = & move(E,\mathtt{a}) & = & \varepsilon\text{-}closure(\{8\}) & = & \{8\} \\
 & & move(E,\mathtt{b}) & = & \varepsilon\text{-}closure(\{\}) & = & \{\} \\
 & & move(F,\mathtt{a}) & = & \varepsilon\text{-}closure(\{1,6,7,8\}) & = & F \\
 & & move(F,\mathtt{b}) & = & \varepsilon\text{-}closure(\{6\}) & = & C \\
 & & move(G,\mathtt{a}) & = & \varepsilon\text{-}closure(\{\}) & = & \{\} \\
 & & move(G,\mathtt{b}) & = & \varepsilon\text{-}closure(\{\}) & = & \{\}
\end{array}
$$

States F and G are accepting since they contain the accepting NFA state 8.
In diagram form, we get:



Exercise 2.5

We start by noting that there are no dead states, then we divide into groups of accepting and non-accepting states:

$$\begin{aligned} 0 &= \{0\} \\ A &= \{1,2,3,4\} \end{aligned}$$

We now check if group A is consistent:

A	a	b
1	A	–
2	A	–
3	A	0
4	A	0

We see that we must split A into two groups:

$$\begin{aligned} B &= \{1,2\} \\ C &= \{3,4\} \end{aligned}$$

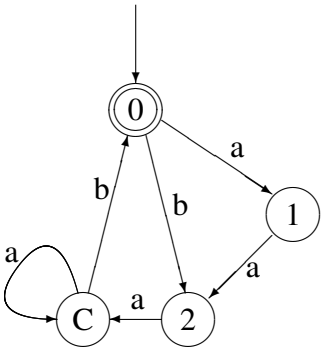
And we now check these, starting with B:

B	a	b
1	B	–
2	C	–

So we need to split B into it individual states. The only non-singleton group left is C, which we now check:

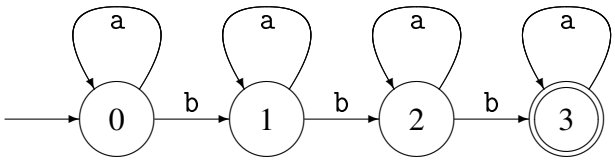
C	a	b
3	C	0
4	C	0

This is consistent, so we can see that we could only combine states 3 and 4 into a group C. The resulting diagram is:

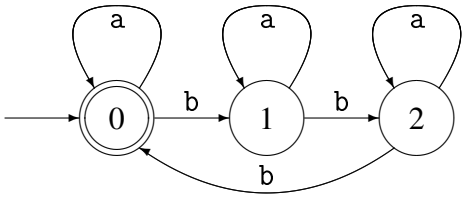


Exercise 2.7

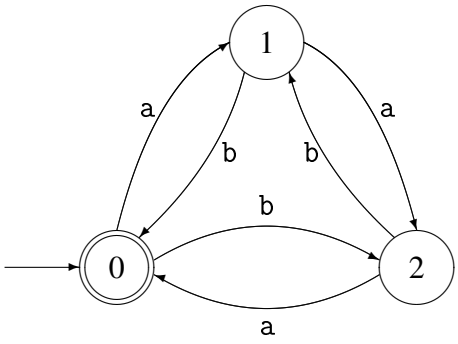
a)



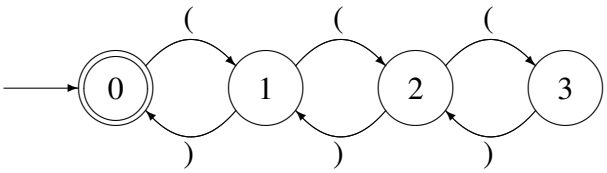
b)



c)

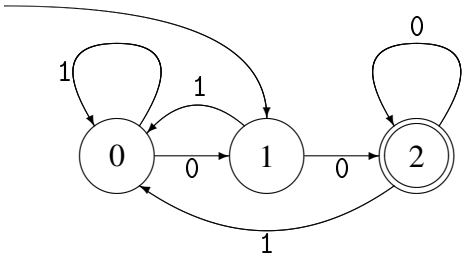


Exercise 2.8



Exercise 2.9

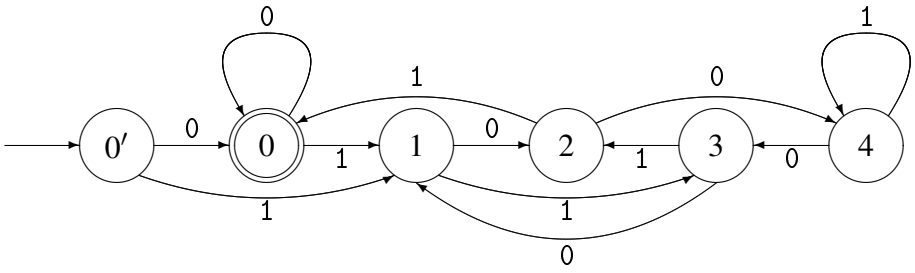
a) The number must be 0 or end in two zeroes:



b) We use that reading a 0 is the same as multiplying by 2 and reading a 1 is the same as multiplying by two and adding 1. So of we have remainder m, reading a 0 gives us remainder $(2m)\,mod\,5$ and reading a 1 gives us remainder $(2m+1)\,mod\,5$. We can make the following transition table:

m	0	1
0	0	1
1	2	3
2	4	0
3	1	2
4	3	4

The state corresponding to $m = 0$ is accepting. We must also start with remainder 0, but since the empty string isn't a valid number, we can't use the accepting state as start state. So we add an extra start state $0'$ that has the same transitions as 0, but isn't accepting:



c) If $n = a * 2^b$, the binary number for n is the number for a followed by b zeroes. We can make a DFA for an odd number a in the same way we did for 5 above by using the rules that reading a 0 in state m gives us a transition to state $(2m) \, mod \, a$ and reading a 1 in state m gives us a transition to state $(2m+1) \, mod \, a$. If we (for now) ignore the extra start state, this DFA has a states. This is minimal because a and 2 (the base number of binary numbers) are relative prime (a complete proof requires some number theory).

If $b = 0$, the DFA for n is the same as the DFA constructed above for a, but with one extra start state as we did for the DFA for 5, so the total number of states is $a+1$.

If $b > 0$, we take the DFA for a and make b extra states: $0_1, 0_2, \ldots, 0_b$. All of these have transitions to state 1 on 1. State 0 is changed so it goes to state 0_1 on 0 (instead of to itself). For $i = 1, \ldots, b-1$, state 0_i has transition on 1 to $0_{(i+1)}$ while 0_b has transition to itself on 0. 0_b is the only accepting state. The start state is state 0 from the DFA for a. This DFA will first recognize a number that is an odd multiple of a (which ends in a 1) and then check that there are at least b zeroes after this. The total number of states is $a+b$.

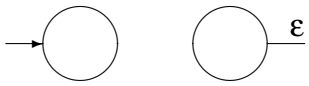
So, if n is odd, the number of states for a DFA that recognises numbers divisible by n is n, but if $n = a * 2^b$, where a is odd and $b > 0$, then the number of states is $a+b$.

Exercise 2.10

a)

$\phi \mid s$	$=$	s	because $L(\phi) \cup L(s) = \emptyset \cup L(s) = L(s)$
ϕs	$=$	ϕ	because there are no strings in ϕ to put in front of strings in s
$s\phi$	$=$	ϕ	because there are no strings in ϕ to put after strings in s
ϕ^*	$=$	ε	because $\phi^* = \varepsilon \mid \phi\phi^* = \varepsilon \mid \phi = \varepsilon$

b)



c) As there can now be dead states, the minimization algorithm will have to take these into consideration as described in section 2.8.2.

Exercise 2.11

In the following, we will assume that for the regular language L, we have an NFA N with no dead states.

Closure under prefix. When N reads a string $w \in L$, it will at each prefix of w be at some state s in N. By making s accepting, we can make N accept this prefix. By making all states accepting, we can accept all prefixes of strings in L.

So an automaton N_p that accepts the prefixes of strings in L is made of the same states and transitions as N, with the modification that all states in N_p accepting.

Closure under suffix. When N reads a string $w \in L$, where $w = uv$, it will after reading u be in some state s. If we made s the start state of N, N would hence accept the suffix v of w. If we made all states of N into start states, we would hence be able to accept all suffixes of strings in L. Since we are only allowed one start state, we instead add a new start state and ε-transitions from this to all the old states (including the original start state, which is no longer the start state).

So an automaton N_s that accepts all suffixes of strings in L is made of the same states and transitions as N plus a new start state s_0' that has ε-transitions to all states in N.

Closure under subsequences. A subsequence of a string w can be obtained by deleting (or jumping over) any number of the letters in w. We can modify N to jump over letters by for each transition $s^c t$ on a letter c add an ε-transition $s^\varepsilon t$ between the same pair of states.

So an automaton N_b that accepts all subsequences of strings in L is made of the same states and transitions as N, with the modification that we add an ε-transitions $s^\varepsilon t$ whenever N has a transition $s^c t$.

Closure under reversal. We assume N has only one accepting state. We can safely make this assumption, since we can make it so by adding an extra accepting state f and make ε-transitions from all the original accepting states to f and then make f the only accepting state.

We can now make N accept the reverses of the strings from L by reversing all transitions and swap start state and accepting state.

So an automaton N_r that accepts all reverses of strings in L is made the following way:

1. Copy all states (but no transitions) from N to N_r.
2. The copy of the start state s_0 from N is the only accepting state in N_r.
3. Add a new start state s_0' to N_r and make ε-transitions from s_0' to all states in N_r that are copies of accepting states from N.
4. When N has a transition $s^c t$, add a transition $t'^c s'$ to N_r, where s' and t' are the copies in N_r of the states s and t from N.

3 Exercises for chapter 3

Exercise 3.3

If we at first ignore ambiguity, the obvious grammar is

$$\begin{array}{lcl} P & \rightarrow & \\ P & \rightarrow & (P) \\ P & \rightarrow & PP \end{array}$$

i.e., the empty string, a parentesis around a balanced sequence and a concatenation of two balanced sequences. But as the last production is both left recursive and right recursive, the grammar is ambiguous. An unambiguous grammar is:

$$\begin{array}{lcl} P & \rightarrow & \\ P & \rightarrow & (P)P \end{array}$$

which combines the two last productions from the first grammar into one.

Exercise 3.4

a)

$$\begin{array}{lcl} S & \rightarrow & \\ S & \rightarrow & \mathtt{a}S\mathtt{b}S \\ S & \rightarrow & \mathtt{b}S\mathtt{a}S \end{array}$$

Explanation: The empty string has the same number of as and bs. If a string starts with an a, we find a b to match it and vice versa.

b)

$$\begin{array}{lcl} A & \rightarrow & AA \\ A & \rightarrow & S\mathtt{a}S \\ S & \rightarrow & \\ S & \rightarrow & \mathtt{a}S\mathtt{b}S \\ S & \rightarrow & \mathtt{b}S\mathtt{a}S \end{array}$$

Explanation: Each excess a has (possibly) empty sequences of equal numbers of as and bs.

c)

$$
\begin{array}{lcl}
D & \to & A \\
D & \to & B \\
A & \to & AA \\
A & \to & S\,\mathtt{a}\,S \\
B & \to & BB \\
B & \to & S\,\mathtt{b}\,S \\
S & \to & \\
S & \to & \mathtt{a}\,S\,\mathtt{b}\,S \\
S & \to & \mathtt{b}\,S\,\mathtt{a}\,S
\end{array}
$$

Explanation: If there are more as than bs, we use A from above and otherwise we use a similarly constructed B.

d)

$$
\begin{array}{lcl}
S & \to & \\
S & \to & \mathtt{a}\,S\,\mathtt{a}\,S\,\mathtt{b}\,S \\
S & \to & \mathtt{a}\,S\,\mathtt{b}\,S\,\mathtt{a}\,S \\
S & \to & \mathtt{b}\,S\,\mathtt{a}\,S\,\mathtt{a}\,S
\end{array}
$$

Explanation: If the string starts with an a, we find later macthing as and bs, if it starts with a b, we find two matching as.

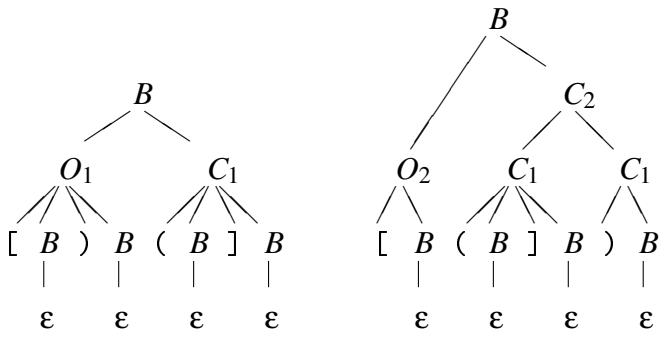
Exercise 3.5

a)

$$
\begin{array}{lcl}
B & \to & \varepsilon \\
B & \to & O_1 C_1 \\
B & \to & O_2 C_2 \\
O_1 & \to & (B \\
O_1 & \to & [B)B \\
O_2 & \to & [B \\
O_2 & \to & O_1 O_1 \\
C_1 & \to &)B \\
C_1 & \to & (B]B \\
C_2 & \to &]B \\
C_2 & \to & C_1 C_1
\end{array}
$$

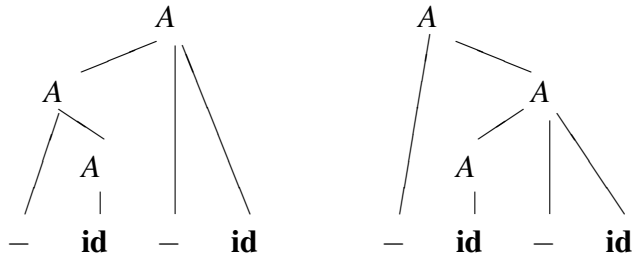
B is "balanced", O_1/C_1 are "open one" and "close one", and O_2/C_2 are "open two" and "close two".

b)



Exercise 3.6

The string $-\mathbf{id}-\mathbf{id}$ has these two syntax trees:

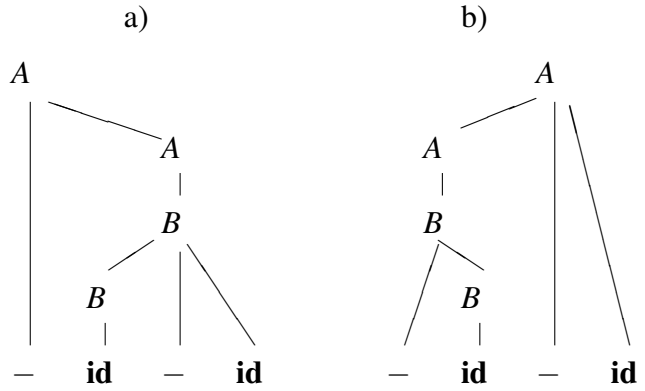


We can make these unambiguous grammars:

a) :
$$\begin{array}{lcl} A & \rightarrow & A-\mathbf{id} \\ A & \rightarrow & B \\ B & \rightarrow & -B \\ B & \rightarrow & \mathbf{id} \end{array}$$

b) :
$$\begin{array}{lcl} A & \rightarrow & -A \\ A & \rightarrow & B \\ B & \rightarrow & B-\mathbf{id} \\ B & \rightarrow & \mathbf{id} \end{array}$$

The trees for the string $-\mathbf{id}-\mathbf{id}$ with these two grammars are:



Exercise 3.9

We first find the equations for *Nullable*:

$$\begin{array}{lcl} \mathit{Nullable}(A) & = & \mathit{Nullable}(BAa) \vee \mathit{Nullable}(\varepsilon) \\ \mathit{Nullable}(B) & = & \mathit{Nullable}(bBc) \vee \mathit{Nullable}(AA) \end{array}$$

This trivially solves to

$$\begin{array}{lcl} \mathit{Nullable}(A) & = & \mathit{true} \\ \mathit{Nullable}(B) & = & \mathit{true} \end{array}$$

Next, we set up the equations for *FIRST*:

$$\begin{array}{lcl} \mathit{FIRST}(A) & = & \mathit{FIRST}(BAa) \cup \mathit{FIRST}(\varepsilon) \\ \mathit{FIRST}(B) & = & \mathit{FIRST}(bBc) \cup \mathit{FIRST}(AA) \end{array}$$

Given that both A and B are `Nullable`, we can reduce this to

$$\begin{array}{lcl} \mathit{FIRST}(A) & = & \mathit{FIRST}(B) \cup \mathit{FIRST}(A) \cup \{a\} \\ \mathit{FIRST}(B) & = & \{b\} \cup \mathit{FIRST}(A) \end{array}$$

which solve to

$$\begin{array}{lcl} \mathit{FIRST}(A) & = & \{a, b\} \\ \mathit{FIRST}(B) & = & \{a, b\} \end{array}$$

Finally, we add the production $A' \rightarrow \$$ and set up the constraints for *FOLLOW*:

$$
\begin{array}{ll}
\{\$\} & \subseteq FOLLOW(A) \\
FIRST(Aa) & \subseteq FOLLOW(B) \\
\{a\} & \subseteq FOLLOW(A) \\
\{c\} & \subseteq FOLLOW(B) \\
FIRST(A) & \subseteq FOLLOW(A) \\
FOLLOW(B) & \subseteq FOLLOW(A)
\end{array}
$$

which we solve to

$$
\begin{array}{lcl}
FOLLOW(A) & = & \{a, b, c, \$\} \\
FOLLOW(B) & = & \{a, b, c\}
\end{array}
$$

Exercise 3.10

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num}\, Exp_1 \\
Exp & \rightarrow & (\ Exp\ )\ Exp_1 \\
Exp_1 & \rightarrow & +\ Exp\ Exp_1 \\
Exp_1 & \rightarrow & -\ Exp\ Exp_1 \\
Exp_1 & \rightarrow & *\ Exp\ Exp_1 \\
Exp_1 & \rightarrow & /\ Exp\ Exp_1 \\
Exp_1 & \rightarrow &
\end{array}
$$

Exercise 3.11

Nullable for each right-hand side is trivially found to be:

$$
\begin{array}{lcl}
Nullable(Exp2\ Exp') & = & false \\
Nullable(+\ Exp2\ Exp') & = & false \\
Nullable(-\ Exp2\ Exp') & = & false \\
Nullable() & = & true \\
Nullable(Exp3\ Exp2') & = & false \\
Nullable(*\ Exp3\ Exp2') & = & false \\
Nullable(/\ Exp3\ Exp2') & = & false \\
Nullable() & = & true \\
Nullable(\textbf{num}) & = & false \\
Nullable((\ Exp\ )) & = & false
\end{array}
$$

The $FIRST$ sets are also easily found:

$$
\begin{array}{lcl}
FIRST(Exp2\ Exp') & = & \{\textbf{num},\ (\} \\
FIRST(+\ Exp2\ Exp') & = & \{+\} \\
FIRST(-\ Exp2\ Exp') & = & \{-\} \\
FIRST() & = & \{\} \\
FIRST(Exp3\ Exp2') & = & \{\textbf{num},\ (\} \\
FIRST(*\ Exp3\ Exp2') & = & \{*\} \\
FIRST(/\ Exp3\ Exp2') & = & \{/\} \\
FIRST() & = & \{\} \\
FIRST(\textbf{num}) & = & \{\textbf{num}\} \\
FIRST((\ Exp\ )) & = & \{(\}
\end{array}
$$

Exercise 3.12

We get the following constraints for each production (abbreviating $FIRST$ and $FOLLOW$ to FI and FO and ignoring trivial constraints like $FO(Exp_{)} \subseteq FO(Exp_1)$):

$$
\begin{array}{lcll}
Exp' & \to & Exp\,\$ & : \ \$ \in FO(Exp) \\
Exp & \to & \textbf{num}\,Exp_1 & : \ FO(Exp) \subseteq FO(Exp_1) \\
Exp & \to & (\ Exp\ )\ Exp_1 & : \ ) \in FO(Exp),\ FO(Exp) \subseteq FO(Exp_1) \\
Exp_1 & \to & +\ Exp\ Exp_1 & : \ FI(Exp_1) \subseteq FO(Exp),\ FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 & \to & -\ Exp\ Exp_1 & : \ FI(Exp_1) \subseteq FO(Exp),\ FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 & \to & *\ Exp\ Exp_1 & : \ FI(Exp_1) \subseteq FO(Exp),\ FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 & \to & /\ Exp\ Exp_1 & : \ FI(Exp_1) \subseteq FO(Exp),\ FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 & \to & & :
\end{array}
$$

As $FI(Exp_1) = \{+, -, *, /\}$, we get

$$FO(Exp) = FO(Exp_1) = \{+, -, *, /,), \$\}$$

Exercise 3.13

The table is too wide for the page, so we split it into two, but for layout only (they are used as a single table).

	num	$+$	$-$	$*$
Exp'	$Exp' \to Exp\,\$$			
Exp	$Exp \to \textbf{num}\,Exp_1$			
Exp_1		$Exp_1 \to +\,Exp\,Exp_1$ $Exp_1 \to$	$Exp_1 \to -\,Exp\,Exp_1$ $Exp_1 \to$	$Exp_1 \to *\,Exp\,Exp_1$ $Exp_1 \to$

	/	(	)	$
Exp'		$Exp' \to Exp\,\$$		
Exp		$Exp \to (\,Exp\,)\,Exp_1$		
Exp_1	$Exp_1 \to / \,Exp\,Exp_1$ $Exp_1 \to$		$Exp_1 \to$	$Exp_1 \to$

Note that there are several conflicts for Exp_1, which isn't surprising, as the grammar is ambiguous.

Exercise 3.14

a)

$$\begin{aligned} E &\to \textbf{num}\,E' \\ E' &\to E + E' \\ E' &\to E * E' \\ E' &\to \end{aligned}$$

b)

$$\begin{aligned} E &\to \textbf{num}\,E' \\ E' &\to E\,Aux \\ E' &\to \\ Aux &\to +E' \\ Aux &\to *E' \end{aligned}$$

c)

	Nullable	*FIRST*
$E \to \textbf{num}\,E'$	*false*	{**num**}
$E' \to E\,Aux$	*false*	{**num**}
$E' \to$	*true*	{}
$Aux \to +E'$	*false*	{+}
$Aux \to *E'$	*false*	{*}

	FOLLOW
E	{+, *, $}
E'	{+, *, $}
Aux	{+, *, $}

d)

	num	+	*	$
E	$E \to \textbf{num}\,E'$			
E'	$E' \to E\,Aux$	$E' \to$	$E' \to$	$E' \to$
Aux		$Aux \to +E'$	$Aux \to *E'$	

Exercise 3.19

a) We add the production $T' \rightarrow T$.

b) We add the production $T'' \rightarrow T'\$$ for calculating $FOLLOW$. We get the constraints (omitting trivially true constraints):

$$\begin{array}{lcll} T'' & \rightarrow & T'\$ & : \ \$ \in FOLLOW(T') \\ T' & \rightarrow & T & : \ FOLLOW(T') \subseteq FOLLOW(T) \\ T & \rightarrow & T->T & : \ -> \in FOLLOW(T) \\ T & \rightarrow & T*T & : \ * \in FOLLOW(T) \\ T & \rightarrow & \textbf{int} & : \end{array}$$

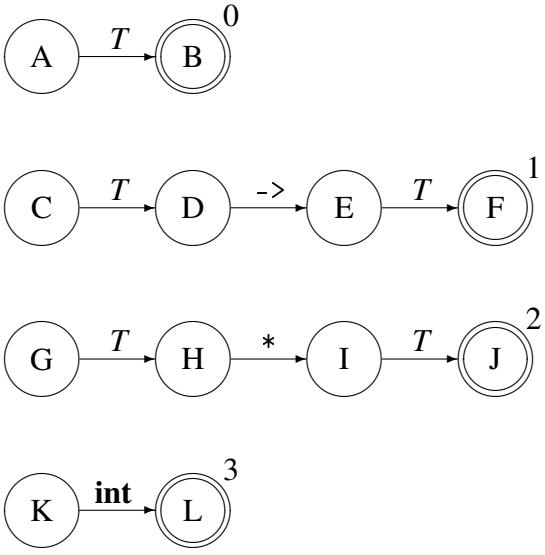
which solves to

$$\begin{array}{lcl} FOLLOW(T') & = & \{\$\} \\ FOLLOW(T) & = & \{\$, ->, *\} \end{array}$$

c) We number the productions:

$$\begin{array}{lcll} 0: & T' & \rightarrow & T \\ 1: & T & \rightarrow & T->T \\ 2: & T & \rightarrow & T*T \\ 3: & T & \rightarrow & \textbf{int} \end{array}$$

and make NFAs for each:



We then add epsilon-transitions:

	ε
A	C, G, K
C	C, G, K
E	C, G, K
G	C, G, K
I	C, G, K

and convert to a DFA (in tabular form):

state	NFA states	**int**	->	*	T
0	A, C, G, K	s1			g2
1	L				
2	B, D, H		s3	s4	
3	E, C, G, K	s1			g5
4	I, C, G, K	s1			g6
5	F, D, H		s3	s4	
6	J, D, H		s3	s4	

and add accept/reduce actions according to the $FOLLOW$ sets:

state	NFA states	**int**	->	*	$	T
0	A, C, G, K	s1				g2
1	L		r3	r3	r3	
2	B, D, H		s3	s4	acc	
3	E, C, G, K	s1				g5
4	I, C, G, K	s1				g6
5	F, D, H		s3/r1	s4/r1	r1	
6	J, D, H		s3/r2	s4/r2	r2	

d) The conflict in state 5 on -> is between shifting on -> or reducing to production 1 (which contains ->). Since -> is right-associative, we shift.

The conflict in state 5 on * is between shifting on * or reducing to production 1 (which contains ->). Since * binds tighter, we shift.

The conflict in state 6 on -> is between shifting on -> or reducing to production 2 (which contains *). Since * binds tighter, we reduce.

The conflict in state 6 on * is between shifting on * or reducing to production 2 (which contains *). Since * is left-associative, we reduce.

The final table is:

state	**int**	->	*	$	T
0	s1				g2
1		r3	r3	r3	
2		s3	s4	acc	
3	s1				g5
4	s1				g6
5		s3	s4	r1	
6		r2	r2	r2	

Exercise 3.20

The method from section 3.16.3 can be used with a standard parser generator and with an unlimited number of precedences, but the restructuring of the syntax tree afterwards is bothersome. The precedence of an operator needs not be known at the time the operator is read, as long as it is known at the end of reading the syntax tree.

Method a) requires a non-standard parser generator or modification of a generated parser, but it also allows an unlimited number of precedences and it doesn't require restructuring afterwards. The precedence of an operator needs to be known when it is read, but this knowledge can be aquired earlier in the same parse.

Method b) can be used with a standard parser generator. The lexer has a rule for all possible operator names and looks up in a table to find which token to use for the operator (similar to how, as described in section 2.9.1, identifiers can looked up in a table to see if they are keywords or variables). This table can be updated as a result of a parser action, so like method a), precedence can be declared earlier in the same parse, but not later. The main disadvantage is that the number of precedence levels and the associativity of each level is fixed in advance, when the parser is constructed.

Exercise 3.21

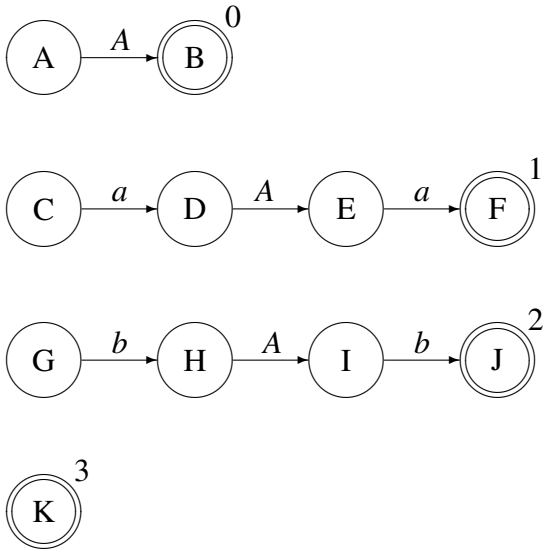
a) The grammar describes the language of all even-length palindromes, i.e., strings that are the same when read forwards or backwards.

b) The grammar is unambiguous, which can be proven by induction on the length of the string: If it is 0, the last production is the only that matches. If greater than 0, the first and last characters in the string uniquely selects the first or second production (or fails, if none match). After the first and last characters are removed, we are back to the original parsing problem, but on a shorter string. By the induction hypothesis, this will have a unique syntax tree.

c) We add a start production $A' \rightarrow A$) and number the productions:

$$\begin{array}{lrcl} 0: & A' & \rightarrow & A \\ 1: & A & \rightarrow & a\ A\ a \\ 2: & A & \rightarrow & b\ A\ b \\ 3: & A & \rightarrow & \end{array}$$

We note that $FOLLOW(A) = \{a, b, \$\}$ and make NFAs for each production:



We then add epsilon-transitions:

	ε
A	C, G, K
D	C, G, K
H	C, G, K

and convert to a DFA (in tabular form) and add accept/reduce actions:

state	NFA states	*a*	*b*	$	A
0	A, C, G, K	s1/r3	s2/r3	r3	g3
1	D, C, G, K	s1/r3	s2/r3	r3	g4
2	H, C, G, K	s1/r3	s2/r3	r3	g5
3	B			acc	
4	E	s6			
5	I	s7			
6	F	r1	r1	r1	
7	J	r2	r2	r2	

d) Consider the string *aa*. In state 0, we shift on the first *a* to state 1. Here we are given a choice between shifting on the second *a* or reducing with the empty reduction. The right action is reduction, so r3 on *a* in state 1 must be preserved.

Consider instead the string *aaaa*. After the first shift, we are left with the same choice as before, but now the right action is to do another shift (and then a reduce). So s1 on *a* in state 1 must also be preserved.

Removing any of these two actions will, hence, make a legal string unparseable. So we can't remove all conflicts.

Some can be removed, though, as we can see that choosing some actions will lead to states from which there are no legal actions. This is true for the r3 actions in *a* and *b* in state 0, as these will lead to state 3 before reaching the end of input. The r3 action on *b* in state 1 can be removed, as this would indicate that we are at the middle of the string with an *a* before the middle and a *b* after the middle. Similarly, the r3 action on *a* in state 2 can be removed. But we are still left with two conflicts, which can not be removed:

state	*a*	*b*	$	A
0	s1	s2	r3	g3
1	s1/r3	s2	r3	g4
2	s1	s2/r3	r3	g5
3			acc	
4	s6			
5	s7			
6	r1	r1	r1	
7	r2	r2	r2	

4 Exercises for chapter 4

Exercise 4.2

In Standard ML, a natural choice for simple symbol tables are lists of pairs, where each pair consists of a name and the object bound to it.

The empty symbol table is, hence the empty list:

```
val emptyTable = []}
```

Binding a symbol to an object is done by prepending the pair of the name and object to the list:

```
fun bind(name,object,table) = (name,object)::table}
```

Looking up a name in a table is searching (from the front of the list) for a pair whose first component is the name we look for. To handle the possibility of a name not being found, we let the lookup function return an option type: If a name is bound, we return `SOME obj`, where `obj` is the object bound to the name, otherwise, we return `NONE`:

```
fun lookup(name,[]) = NONE
  | lookup(name,(name1,obj)::table) =
      if name=name1 then SOME obj
      else lookup(name,table)
```

Entering and exiting scopes don't require actions, as the symbol tables are persistent.

Exercise 4.3

The simplest solution is to convert all letters in names to lower case before doing any symbol-table operations on the names.

Error messages should, however, use the original names, so it is a bad idea to do the conversion already during lexical analysis or parsing.

5 Exercises for chapter 6

Exercise 6.3

We use a synthesized attribute that is the set of exceptions that are thrown but not caught by the expression. If at S_1 **catch** $i \Rightarrow S_2$, i is not among the exceptions that S_1 can throw, we give an error message. If the set of exceptions that the top-level statement can throw is not empty, we also give an error message.

$Check_S(S) =$ `case` S `of`	
throw id	$\{name(\textbf{id})\}$
S_1 **catch id** $\Rightarrow S_2$	$thrown_1 = Check_S(S_1)$ $thrown_2 = Check_S(S_2)$ $if\ name(\textbf{id}) \in thrown_1$ $then\ (thrown_1 \setminus \{name(\textbf{id})\}) \cup thrown_2$ $else$ **error**()
S_1 **or** S_2	$Check_S(S_1) \cup Check_S(S_2)$
other	$\{\}$

$CheckTopLevel_S(S) =$	
	$thrown = Check_S(S)$ $if\ thrown \neq \{\}$ $then$ **error**()

6 Exercises for chapter 7

Exercise 7.1

New temporaries are `t2`, ... in the order they are generated. Indentation shows sub-expression level.

```
 t2 := 2
   t5 := t0
   t6 := t1
  t4 := t5+t6
   t8 := t0
   t9 := t1
  t7 := t8*t9
 t3 := CALL _g(t4,t7)
r := t2+t3
```

Exercise 7.9

a)

$Trans_{Stat}(Stat, vtable, ftable, endlabel) =$ `case` $Stat$ `of`	
$Stat_1$; $Stat_2$	$label_1 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable, label_1)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable, endlabel)$ $code_1$++[`LABEL` $label_1$]++$code_2$
id := Exp	$place = lookup(vtable, name(\textbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$
`if` $Cond$ `then` $Stat_1$	$label_1 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, endlabel, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable, endlabel)$ $code_1$++[`LABEL` $label_1$]++$code_2$
`if` $Cond$ `then` $Stat_1$ `else` $Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable, endlabel)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable, endlabel)$ $code_1$++[`LABEL` $label_1$]++$code_2$ ++[`GOTO` $endlabel$, `LABEL` $label_2$] ++$code_3$
`while` $Cond$ `do` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, endlabel, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable, label_1)$ [`LABEL` $label_1$]++$code_1$ ++[`LABEL` $label_2$]++$code_2$ ++[`GOTO` $label_1$]
`repeat` $Stat_1$ `until` $Cond$	$label_1 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable, label_3)$ $code_2 = Trans_{Cond}(Cond, endlabel, label_1, vtable, ftable)$ [`LABEL` $label_1$]++$code_1$ ++[`LABEL` $label_3$]++$code_2$

b) New temporaries are `t2,...` in the order they are generated. Indentation follows statement structure.

```
LABEL l1
  t2 := t0
```

```
  t3 := 0
 IF t2>t3 THEN l2 ELSE endlab
LABEL l2
  t4 := t0
  t5 := 1
 t0 := t4-t5
  t6 := x
  t7 := 10
  IF t6>t6 THEN l3 ELSE l1
  LABEL l3
    t8 := t0
    t9 := 2
   t0 := t8/t9
GOTO l1
LABEL endlab
```