

# Solutions for Selected Exercises from Introduction to Compiler Design Third Edition

Torben Æ. Mogensen

Last update: December 7, 2023

## Introduction

This document provides solutions for selected exercises from “Introduction to Compiler Design”, third edition. Most of the exercises are numbered the same in the second edition.

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.

## Exercises for Chapter 1

### Exercise 1.2

- 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:

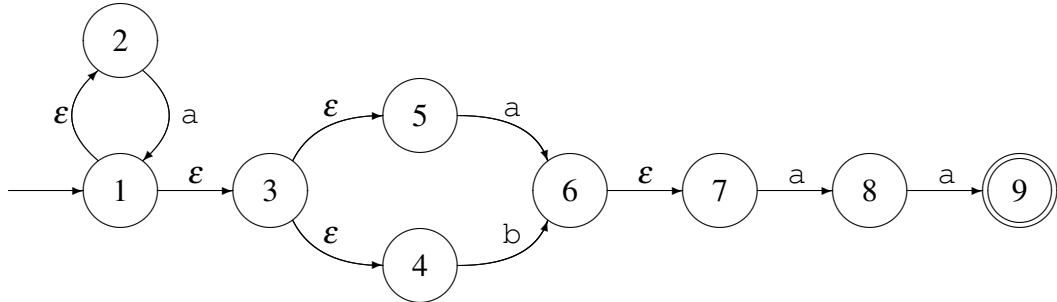
$$0^*([0-9] \mid [1-3][0-9] \mid 4[0-1] \mid 4[3-9] \mid [5-9][0-9] \mid [1-9][0-9][0-9]^+)$$

- 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 1.3

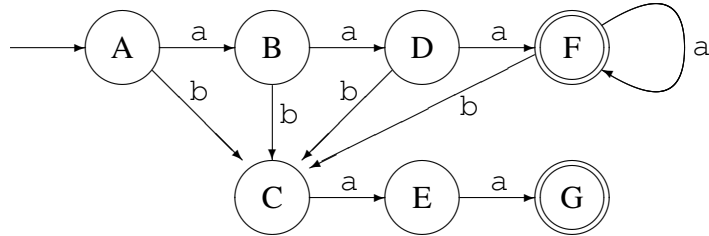
a)



b)

$$\begin{aligned}
 A &= \varepsilon\text{-closure}(\{1\}) = \{1, 2, 3, 4, 5\} \\
 B &= \text{move}(A, a) = \varepsilon\text{-closure}(\{1, 6\}) = \{1, 6, 2, 3, 4, 5, 7\} \\
 C &= \text{move}(A, b) = \varepsilon\text{-closure}(\{6\}) = \{6, 7\} \\
 D &= \text{move}(B, a) = \varepsilon\text{-closure}(\{1, 6, 8\}) = \{1, 6, 8, 2, 3, 4, 5, 7\} \\
 &\quad \text{move}(B, b) = \varepsilon\text{-closure}(\{6\}) = C \\
 E &= \text{move}(C, a) = \varepsilon\text{-closure}(\{8\}) = \{8\} \\
 &\quad \text{move}(C, b) = \varepsilon\text{-closure}(\{\}) = \{\} \\
 F &= \text{move}(D, a) = \varepsilon\text{-closure}(\{1, 6, 8, 9\}) = \{1, 6, 8, 9, 2, 3, 4, 5, 7\} \\
 &\quad \text{move}(D, b) = \varepsilon\text{-closure}(\{6\}) = C \\
 G &= \text{move}(E, a) = \varepsilon\text{-closure}(\{9\}) = \{9\} \\
 &\quad \text{move}(E, b) = \varepsilon\text{-closure}(\{\}) = \{\} \\
 &\quad \text{move}(F, a) = \varepsilon\text{-closure}(\{1, 6, 8, 9\}) = F \\
 &\quad \text{move}(F, b) = \varepsilon\text{-closure}(\{6\}) = C \\
 &\quad \text{move}(G, a) = \varepsilon\text{-closure}(\{\}) = \{\} \\
 &\quad \text{move}(G, b) = \varepsilon\text{-closure}(\{\}) = \{\}
 \end{aligned}$$

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



### Exercise 1.6

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

$$0 = \{0\}$$

$$A = \{1, 2, 3, 4\}$$

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:

$$B = \{1, 2\}$$

$$C = \{3, 4\}$$

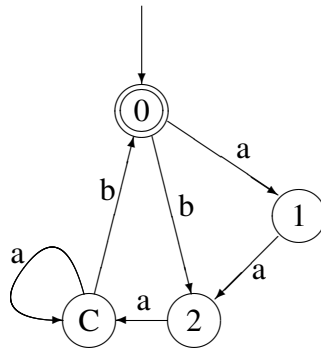
And we now check these, starting with  $B$ :

$B$	$a$	$b$
1	$B$	—
2	$C$	—

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

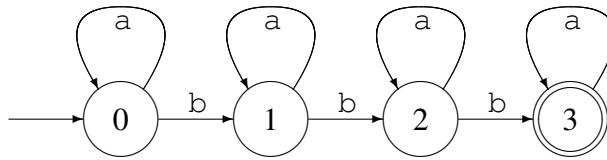
$C$	$a$	$b$
3	$C$	0
4	$C$	0

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

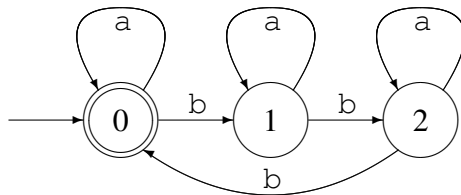


**Exercise 1.8**

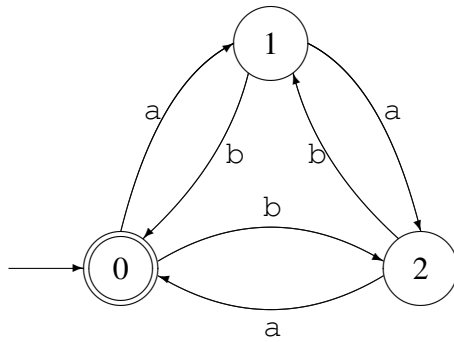
a)



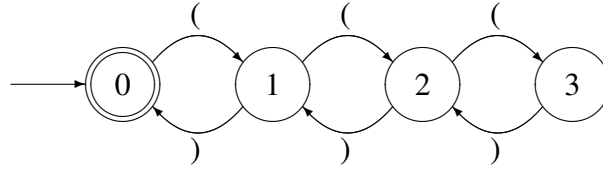
b)



c)

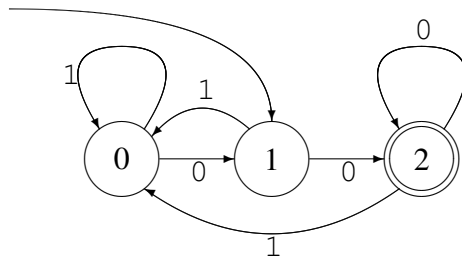


**Exercise 1.9**



**Exercise 1.10**

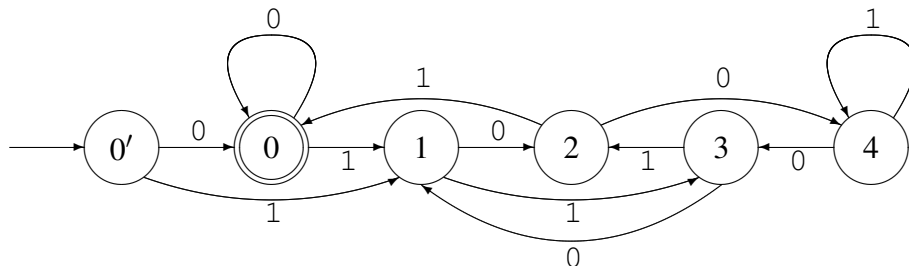
a) The number must be 0 or end in two zeroes:



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 if we have remainder  $m$ , reading a 0 gives us remainder  $(2m) \bmod 5$  and reading a 1 gives us remainder  $(2m + 1) \bmod 5$ . 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  $(2m) \bmod a$  and reading a 1 in state  $m$  gives us a transition to state  $(2m + 1) \bmod a$ . 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:  $0_1, 0_2, \dots, 0_b$ . 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  $i = 1, \dots, b - 1$ , 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 1.11

a)

$\phi | s = s$  because  $L(\phi) \cup L(s) = \emptyset \cup L(s) = L(s)$   
 $\phi s = \phi$  because there are no strings in  $\phi$  to put in front of strings in  $s$   
 $s\phi = \phi$  because there are no strings in  $\phi$  to put after strings in  $s$   
 $\phi^* = \epsilon$  because  $\phi^* = \epsilon | \phi \phi^* = \epsilon | \phi = \epsilon$

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 1.12

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  $\varepsilon$ -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  $\varepsilon$ -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  $\varepsilon$ -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  $\varepsilon$ -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  $\varepsilon$ -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  $\varepsilon$ -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  $N$ .

## Exercises for Chapter 2

### Exercise 2.3

If we at first ignore ambiguity, the obvious grammar is

$$\begin{aligned}P &\rightarrow \epsilon \\P &\rightarrow (P) \\P &\rightarrow PP\end{aligned}$$

i.e., the empty string, a parenthesis 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:

$$\begin{aligned}P &\rightarrow \epsilon \\P &\rightarrow (P)P\end{aligned}$$

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

### Exercise 2.4

a)

$$\begin{aligned}S &\rightarrow \epsilon \\S &\rightarrow aSbS \\S &\rightarrow bSaS\end{aligned}$$

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)

$$\begin{aligned}A &\rightarrow AA \\A &\rightarrow SaS \\S &\rightarrow \epsilon \\S &\rightarrow aSbS \\S &\rightarrow bSaS\end{aligned}$$

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



c)

$$\begin{aligned}D &\rightarrow A \\D &\rightarrow B \\A &\rightarrow AA \\A &\rightarrow SaS \\B &\rightarrow BB \\B &\rightarrow SbS \\S &\rightarrow \\S &\rightarrow aSbS \\S &\rightarrow bSaS\end{aligned}$$

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

d)

$$\begin{aligned}S &\rightarrow \\S &\rightarrow aSaSbS \\S &\rightarrow aSbSaS \\S &\rightarrow bSaSaS\end{aligned}$$

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

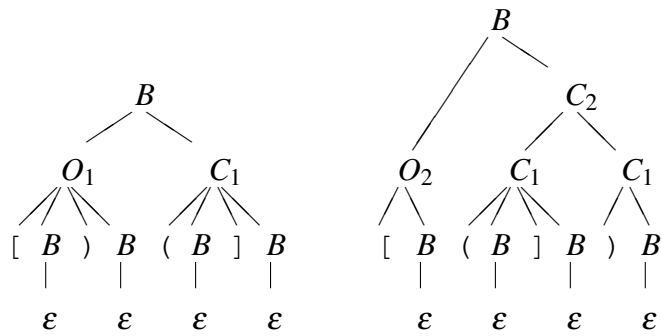
## Exercise 2.5

a)

$$\begin{aligned}B &\rightarrow \varepsilon \\B &\rightarrow O_1C_1 \\B &\rightarrow O_2C_2 \\O_1 &\rightarrow (B \\O_1 &\rightarrow [B)B \\O_2 &\rightarrow [B \\O_2 &\rightarrow O_1O_1 \\C_1 &\rightarrow )B \\C_1 &\rightarrow (B]B \\C_2 &\rightarrow ]B \\C_2 &\rightarrow C_1C_1\end{aligned}$$

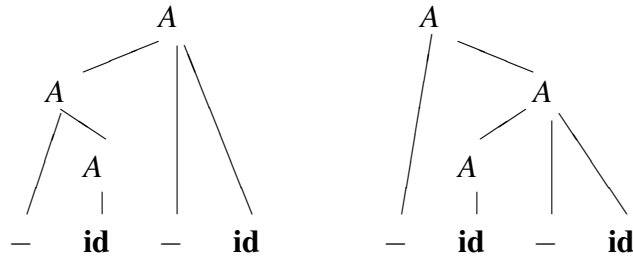
$B$  is “balanced”,  $O_1/C_1$  are “open one” and “close one”, and  $O_2/C_2$  are “open two” and “close two”.

b)



### Exercise 2.6

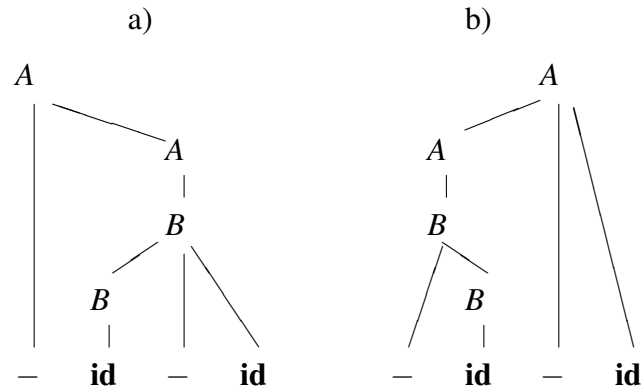
The string  $-id - id$  has these two syntax trees:



We can make these unambiguous grammars:

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

The trees for the string  $-id - id$  with these two grammars are:



### Exercise 2.9

We first find the equations for *Nullable*:

$$\begin{aligned} \text{Nullable}(A) &= \text{Nullable}(BAa) \vee \text{Nullable}(\varepsilon) \\ \text{Nullable}(B) &= \text{Nullable}(bBc) \vee \text{Nullable}(AA) \end{aligned}$$

This trivially solves to

$$\begin{aligned} \text{Nullable}(A) &= \text{true} \\ \text{Nullable}(B) &= \text{true} \end{aligned}$$

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

$$\begin{aligned} \text{FIRST}(A) &= \text{FIRST}(BAa) \cup \text{FIRST}(\varepsilon) \\ \text{FIRST}(B) &= \text{FIRST}(bBc) \cup \text{FIRST}(AA) \end{aligned}$$

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

$$\begin{aligned} \text{FIRST}(A) &= \text{FIRST}(B) \cup \text{FIRST}(A) \cup \{a\} \\ \text{FIRST}(B) &= \{b\} \cup \text{FIRST}(A) \end{aligned}$$

which solve to

$$\begin{aligned} \text{FIRST}(A) &= \{a, b\} \\ \text{FIRST}(B) &= \{a, b\} \end{aligned}$$

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

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

which we solve to

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

### Exercise 2.10

$$\begin{aligned}
Exp &\rightarrow \mathbf{num} Exp_1 \\
Exp &\rightarrow ( Exp ) Exp_1 \\
Exp_1 &\rightarrow + Exp Exp_1 \\
Exp_1 &\rightarrow - Exp Exp_1 \\
Exp_1 &\rightarrow * Exp Exp_1 \\
Exp_1 &\rightarrow / Exp Exp_1 \\
Exp_1 &\rightarrow
\end{aligned}$$

### Exercise 2.11

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

$$\begin{aligned}
Nullable(Exp_2 Exp'_1) &= false \\
Nullable(+ Exp_2 Exp'_1) &= false \\
Nullable(- Exp_2 Exp'_1) &= false \\
Nullable() &= true \\
Nullable(Exp_3 Exp'_2) &= false \\
Nullable(* Exp_3 Exp'_2) &= false \\
Nullable(/ Exp_3 Exp'_2) &= false \\
Nullable() &= true \\
Nullable(\mathbf{num}) &= false \\
Nullable(( Exp )) &= false
\end{aligned}$$

The *FIRST* sets are also easily found:

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

### Exercise 2.12

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

$$\begin{aligned}
Exp' &\rightarrow Exp\$ && : \$ \in FO(Exp) \\
Exp &\rightarrow \mathbf{num}Exp_1 && : FO(Exp) \subseteq FO(Exp_1) \\
Exp &\rightarrow (Exp)Exp_1 && : ) \in FO(Exp), FO(Exp) \subseteq FO(Exp_1) \\
Exp_1 &\rightarrow +ExpExp_1 && : FI(Exp_1) \subseteq FO(Exp), FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 &\rightarrow -ExpExp_1 && : FI(Exp_1) \subseteq FO(Exp), FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 &\rightarrow *ExpExp_1 && : FI(Exp_1) \subseteq FO(Exp), FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 &\rightarrow /ExpExp_1 && : FI(Exp_1) \subseteq FO(Exp), FO(Exp_1) \subseteq FO(Exp) \\
Exp_1 &\rightarrow && :
\end{aligned}$$

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

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

### Exercise 2.13

The table is too wide for the page, so, for layout only, we split the table into two. The two “tables” below are really just parts of a single, very wide table.

Nonterminal	<b>num</b>	+	-	*
$Exp'$	$Exp' \rightarrow Exp\$$			
$Exp$	$Exp \rightarrow \mathbf{num}Exp_1$			
$Exp_1$		$Exp_1 \rightarrow +ExpExp_1$	$Exp_1 \rightarrow -ExpExp_1$	$Exp_1 \rightarrow *ExpExp_1$
		$Exp_1 \rightarrow$	$Exp_1 \rightarrow$	$Exp_1 \rightarrow$

Nonterminal	/	(	)	\$
$Exp'$				$Exp' \rightarrow Exp \$$
$Exp$				$Exp \rightarrow (Exp) Exp_1$
$Exp_1$	$Exp_1 \rightarrow / Exp Exp_1$			$Exp_1 \rightarrow Exp_1 \rightarrow$
	$Exp_1 \rightarrow$			

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

### Exercise 2.14

a)

$$\begin{aligned}
 E &\rightarrow \mathbf{num} E' \\
 E' &\rightarrow E + E' \\
 E' &\rightarrow E * E' \\
 E' &\rightarrow
 \end{aligned}$$

b)

$$\begin{aligned}
 E &\rightarrow \mathbf{num} E' \\
 E' &\rightarrow E Aux \\
 E' &\rightarrow \\
 Aux &\rightarrow + E' \\
 Aux &\rightarrow * E'
 \end{aligned}$$

c)

Production	<i>Nullable</i>	<i>FIRST</i>	Nonterminal	<i>FOLLOW</i>
$E \rightarrow \mathbf{num} E'$	<i>false</i>	{ <b>num</b> }	$E$	{+, *, \$}
$E' \rightarrow E Aux$	<i>false</i>	{ <b>num</b> }	$E'$	{+, *, \$}
$E' \rightarrow$	<i>true</i>	{ }	$Aux$	{+, *, \$}
$Aux \rightarrow + E'$	<i>false</i>	{+}		
$Aux \rightarrow * E'$	<i>false</i>	{*}		

d)

Nonterminal	<b>num</b>	+	*	\$
$E$	$E \rightarrow \mathbf{num} E'$			
$E'$	$E' \rightarrow E Aux$	$E' \rightarrow$	$E' \rightarrow$	$E' \rightarrow$
$Aux$		$Aux \rightarrow + E'$	$Aux \rightarrow * E'$	

### Exercise 2.19

- 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):

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

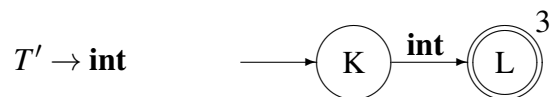
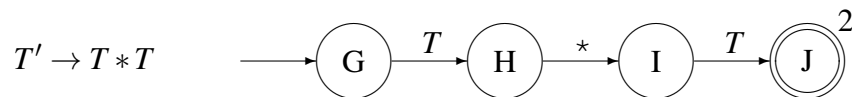
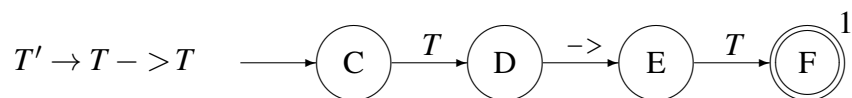
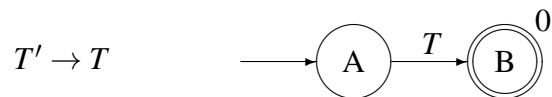
which solves to

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

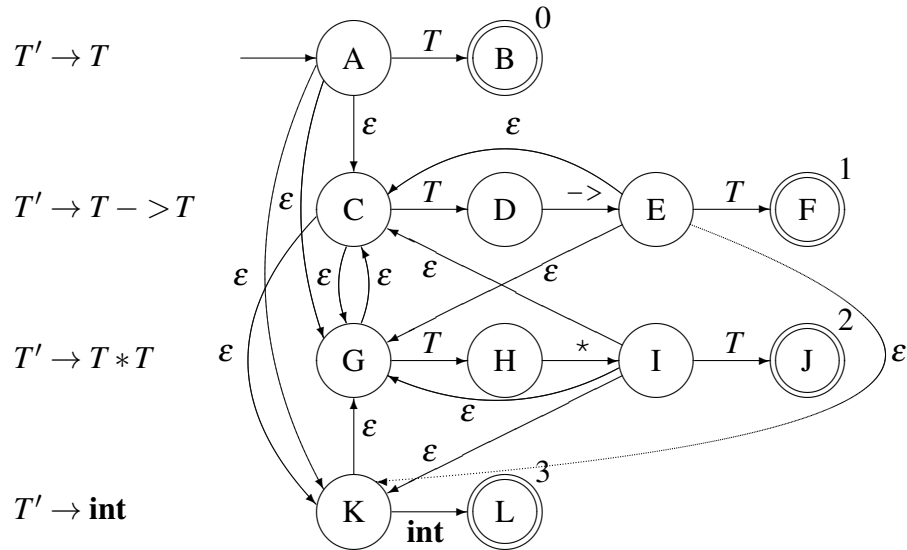
- c) We number the productions:

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

and make NFAs for each:



We then add epsilon-transitions and make a single start state:



Note that we have omitted epsilon-transitions from a state to itself (from C to C and from G to G), as these have no effect.

We now convert the NFA 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  $\rightarrow$  is between shifting on  $\rightarrow$  or reducing to production 1 (which contains  $\rightarrow$ ). Since  $\rightarrow$  is right-associative, we shift.

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

The conflict in state 6 on  $\rightarrow$  is between shifting on  $\rightarrow$  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	$\rightarrow$	$*$	$\$$	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 2.20

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 acquired earlier in the same parse.

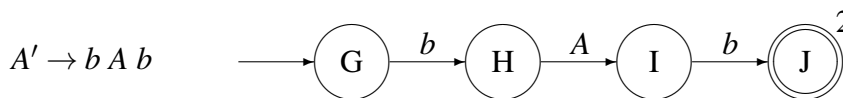
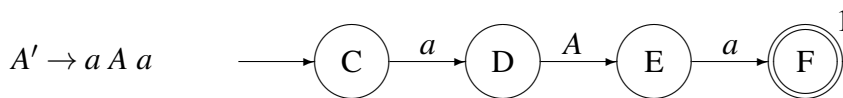
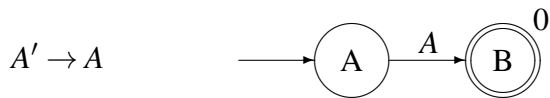
Method b) can be used with a standard parser generator. The lexer has a rule for all possible operator names and looks up in a table to find which token to use for the operator (similar to how, as described in section 2.9.1, identifiers can be looked up in a table to see if they are keywords or variables). 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 2.21

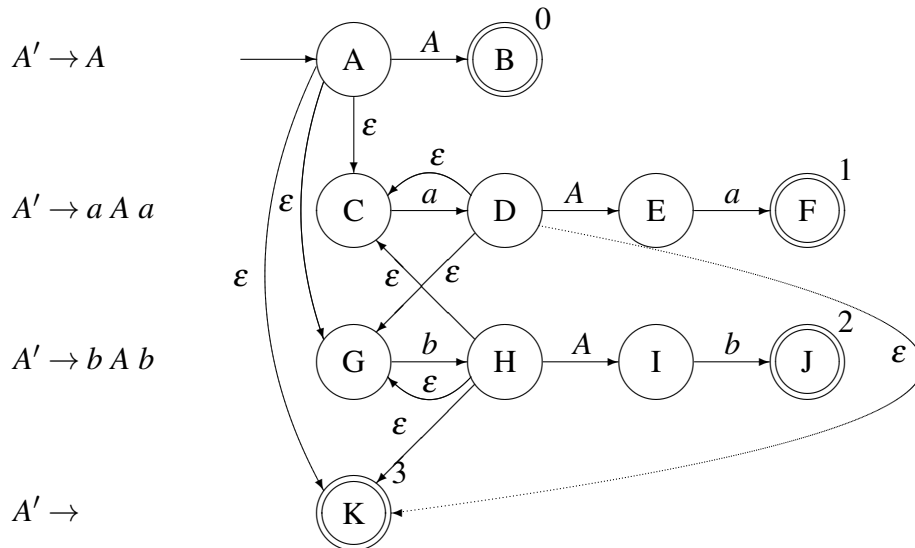
- The grammar describes the language of all even-length palindromes, i.e., strings that are the same when read forwards or backwards.
- The grammar is unambiguous, which can be proven by induction on the length of the string: If it is 0, the last production is the only that matches. If greater than 0, 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.
- We add a start production  $A' \rightarrow A$  and number the productions:

- 0:  $A' \rightarrow A$
- 1:  $A \rightarrow a A a$
- 2:  $A \rightarrow b A b$
- 3:  $A \rightarrow$

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



We then add epsilon-transitions and make a single start state:



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

state	NFA states	<i>a</i>	<i>b</i>	\$	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	<i>a</i>	<i>b</i>	\$	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	

## Exercises for Chapter 3

### Exercise 3.2

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 3.3

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.

## Exercises for Chapter 4

### Exercise 4.1

We add the following cases:

$Eval_{Exp}(Exp, vtable, ftable) = \text{case } Exp \text{ of}$	
$\text{not } Exp_1$	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ <i>if <math>v_1</math> is a Boolean</i> <i>then <math>\neg v_1</math></i> <i>else <b>error()</b></i>
$Exp_1 \text{ and } Exp_2$	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ <i>if <math>v_1</math> and <math>v_2</math> both are Booleans</i> <i>then <math>v_1 \wedge v_2</math></i> <i>else <b>error()</b></i>

### Exercise 4.4

```
int main(int x) =  
  let y = if x<0 then 1 else 0<0 in  
  if y<y then 7 else 11
```

if  $x$  is negative,  $y$  is bound to the integer 1, otherwise to the Boolean `false`. Since we use  $y$  only in a comparison with itself, this causes no type errors. The program will always return 11.

## Exercises for Chapter 5

### Exercise 5.1

We add the following cases:

$$\begin{array}{l|l}
\text{not } Exp_1 & t_1 = \text{Check}_{Exp}(Exp_1, vtable, ftable) \\
& \text{if } t_1 = \text{bool} \\
& \text{then } \text{bool} \\
& \text{else } \mathbf{error}(); \text{bool} \\
\hline
Exp_1 \text{ and } Exp_2 & t_1 = \text{Check}_{Exp}(Exp_1, vtable, ftable) \\
& t_2 = \text{Check}_{Exp}(Exp_2, vtable, ftable) \\
& \text{if } t_1 = \text{bool} \text{ and } t_2 = \text{bool} \\
& \text{then } \text{bool} \\
& \text{else } \mathbf{error}(); \text{bool}
\end{array}$$

### Exercise 5.4

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.

$$\begin{array}{l|l}
\text{throw } \mathbf{id} & \{\text{name}(\mathbf{id})\} \\
\hline
S_1 \text{ catch } \mathbf{id} \Rightarrow S_2 & \text{thrown}_1 = \text{Check}_S(S_1) \\
& \text{thrown}_2 = \text{Check}_S(S_2) \\
& \text{if } \text{name}(\mathbf{id}) \in \text{thrown}_1 \\
& \text{then } (\text{thrown}_1 \setminus \{\text{name}(\mathbf{id})\}) \cup \text{thrown}_2 \\
& \text{else } \mathbf{error}() \\
\hline
S_1 \text{ or } S_2 & \text{Check}_S(S_1) \cup \text{Check}_S(S_2) \\
\hline
\mathbf{other} & \{\} \\
\hline
\text{CheckTopLevel}_S(S) = & \\
& \text{thrown} = \text{Check}_S(S) \\
& \text{if } \text{thrown} \neq \{\} \\
& \text{then } \mathbf{error}()
\end{array}$$

## Exercises for Chapter 6

### Exercise 6.1

New temporaries are  $t_0, \dots$  in the order they are generated. Indentation shows sub-expression level.

```
t0 := 2
  t3 := v0
  t5 := v1
t2 := t3+t5
  t6 := v0
  t7 := v1
  t5 := t6*t7
t1 := CALL _g(t2,t5)
r := t0+t1
```

## Exercise 6.9

a)

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



- b) New temporaries are  $t_2, \dots$  in the order they are generated. Indentation follows statement structure.

```
LABEL 11
    t2 := t0
    t3 := 0
    IF t2>t3 THEN 12 ELSE endlab
LABEL 12
    t4 := t0
    t5 := 1
    t0 := t4-t5
    t6 := x
    t7 := 10
    IF t6>t6 THEN 13 ELSE 11
    LABEL 13
        t8 := t0
        t9 := 2
        t0 := t8/t9
GOTO 11
LABEL endlab
```

## Exercises for Chapter 7

### Exercise 7.2

The generated code is

```
addi d, c, 8
add  a, a, b
sw   a, 0(d)
bge  a, c, label2
label1 :
```

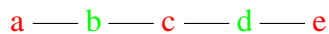
There are two significant differences:

- The addition of 8 to  $d$  is not combined with the load, as there is an instruction between the two, so the ordering of the two first intermediate-code instructions matter, even though they are independent.
- We negated the condition in the jump, so the label for the false case follows immediately as fall-through.

## Exercises for Chapter 8

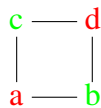
### Exercise 8.6

A graph that is a linear sequence of nodes, e.g.,



can be coloured by alternating colours, as shown.

Additionally, a circle of nodes with an *even* number of nodes can be coloured using alternating colours.



Any collection of such components (that are not connected to each other) can be coloured using two colours.

Note that a triangle or any other odd cycle needs three colours.

## Exercises for Chapter 9

### Exercise 9.4

- If `f` returns directly to the function to which `RETURN y` would return, it will return `x` rather than `y`. Between the return from `f` and the following return, `y` must be placed in the return-value register.
- In this case, we can perform tail-call optimisation: Since the context does not care what is returned, returning `x` instead of `y` does not matter. In `C`, you would get a warning if you return a value from a `void` function, but in the intermediate language `RETURN` requires a parameter, so a variable is required even though it will be ignored at the call site.

## Exercises for Chapter 10

### Exercise 10.6

If `y` and `z` are both pointers or both integers the instruction `x := y - z` assigns an integer (so not a pointer) to `x`. If `y` is an integer and `z` is a pointer, the result is non-sensical. If `y` is a pointer and `z` is an integer, `x` is a pointer. If we don't know

which of the above is true, we will still need to make  $x$  point to anything that  $y$  can point to, but we can ignore  $z$ .

In the case where  $y$  is an integer and  $z$  is a pointer, we might risk that  $x$  is subsequently negated, so we get a valid pointer. So if we want to be completely safe, we must ensure that this does not happen or, alternatively, not do the above optimisation.

Having typed variables in the intermediate language can help resolve such issues, so we get a more precise result.

## Exercises for Chapter 12

### Exercise 12.1

The generated definition and call will be

```
void apply_677(char (*f)(char), char a[], int size) {  
    apply_677(toupper, text, strlen(text))  
}
```

### Exercise 12.3

There are multiple ways to solve this:

- Before adding pointer fields to the work stack, we can test if they point to marked objects, and only add them if they do not.
- If the work stack is full, sort it and remove duplicates before continuing.

The first option adds work even when the work stack is large enough. The second option does not, but the amount of work added when the work stack is full is larger, since sorting is  $O(n \log(n))$ .

### Exercise 12.5

We add the following case:

$$\frac{\text{Trans}_{\text{Pattern}}(\text{Pattern}, \text{vtable}, \text{place}, \text{label}) = \text{case Pattern of}}{- \mid ([], \text{vtable})}$$