Introduction to Language Theory and Compiling
Solution of exercises

Gilles Geeraerts *et al*

October 21, 2024

Contents

Chapter 2

All Things Regular: Languages, Expressions…

2.1 Definition of regular languages

Ex. 2.1.

1. • $1 \in \Sigma$ and $0 \in \Sigma$, thus $\{1\}$ and $\{0\}$ are both regular languages (RL).
   • The Kleene closure of a RL is also a RL, thus $\{1\}^*$ and $\{0\}^*$ are RL.
   • The concatenation of RL is a RL, thus $\{1\}^* \cdot \{0\} \cdot \{1\} \cdot \{0\}^*$ is a RL.
2. An odd binary number always ends with a 1.
   $\{1\}$ and $\{0\}$ are RL; $\{1\} \cup \{0\}$ is regular; $(\{1\} \cup \{0\})^*$ is regular; $(\{1\} \cup \{0\})^* \cdot \{1\}$ is regular.

Ex. 2.2.

1. **Show that any finite language is regular** (by *induction*:) *Idea of the proof*: L is finite, so there exists $n \in \mathbb{N}$ (the *size* of the language L) and n words $w_1, \dots, w_n \in \Sigma^*$ such that $L = \{w_1, \dots, w_n\}$. Thus, $L = \bigcup_{i=1}^{n} L_i$, where for each $i \in \{1, \dots, n\}$, L_i is the singleton language containing only the word w_i: $L_i = \{w_i\}$. Moreover, for each i, there exists $n_i \in \mathbb{N}$ (the *length* of the word w_i) and n_i letters $c_1, \dots, c_{n_i} \in \Sigma$ such that $w_i = c_1 \dots c_{n_i}$, so for each i, $L_i = \{c_1 \dots c_{n_i}\} = \{c_1\} \cdot \dots \cdot \{c_{n_i}\} = \bullet_{j=1}^{n} c_j$. Since each $\{c_j\}$ is regular, L_i is also regular because it is a concatenation of regular languages. Thus, $L = \bigcup_{i=1}^{n} L_i$ is regular, as a finite union of regular languages.

   Note however that the use of "$\cdots$" is not very formal here, so the really formal way of writing such proof is by induction.

   First, let us show by induction on the length of the word l that for all $l \in \mathbb{N}$, any word $w \in \Sigma^l$ (Σ^l denotes the set of words of length l), $\{w\}$ is a regular language.

   • For $l = 0$, we have that $w = \varepsilon$, and by definition $\{\varepsilon\}$ is regular.
   • Although this is not needed for the induction, we also treat the case $l = 1$ because the case $l = 0$ might seem "pathological" for some of you: for $l = 1$, $w = a$ for some $a \in \Sigma$, so $L = \{a\}$ is regular by definition.
   • Now, assume that the property holds for some $l \in \mathbb{N}$, and let w be some word of length $l+1$. w can be decomposed into $w = w'a$ for some w' of length l and some $a \in \Sigma$. By the induction hypothesis, $\{w'\}$ is regular since w' is of length n, so $\{w\} = \{w'\} \cdot \{a\}$ is regular as a concatenation of two regular languages.

   Remark. Note that we could have shown using the same technique a more general result, namely that the l-th power of a regular language is regular, where we define $L^0 = \{\varepsilon\}$ and for all $l \in \mathbb{N}$, $L^{l+1} = L^l \cdot L$. Then, we could have deduced that Σ^l, the set of all words of length l, is regular, because Σ is regular (as a finite union of regular languages $\{a\}$).

Now, let us show by induction on n the size of L that for all $n \in \mathbb{N}$, for all finite language L of size n, L is regular:

- Again, we can start at $n = 0$: then $L = \emptyset$, which is regular by definition.
- Again, although this is not needed for the mathematical correctness of the proof, we treat the case $n = 1$: then, L contains a single word $w \in \Sigma^*$: $L = \{w\}$, so L is regular as we showed earlier.
- Now, assume the property holds for some $n \in \mathbb{N}$, and let L be a language of size $n+1$. $L = L' \cup \{w\}$ for some L of size n and some $w \in \Sigma^*$. By the induction hypothesis, we known that L' is regular since it is a language of size n. Now, $\{w\}$ is regular, as shown earlier, so L is regular, as a union of two regular languages.

Remark. An entirely different way of showing that any finite language L is regular would have been to build a finite automaton recognising L, and then use Kleene's theorem. This is left as an exercise[1].

2. **Show that the language $L = \{0^n1^n \mid n \in \mathbb{N}\}$ is not regular** (by *contradiction*:)
Assume, towards contradiction that L is a regular language. By Kleene's theorem, we know that there exists a finite (*possibly non-deterministic)* automaton $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ that accepts L. By definition, the state set Q is finite. Let m be its size. Now, observe that the language L is *infinite*, as for each natural number $n \in \mathbb{N}$, there is a corresponding word 0^n1^n in L. For instance, consider the word $0^{2m}1^{2m}$. Clearly ($2m \in \mathbb{N}$ and $2m = 2m$!), this word is in L. Thus, there exists an accepting run of the automaton A on the word $0^{2m}1^{2m}$. This run is of the following form:

$$q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{0} \dots \xrightarrow{0} q_{2m} \xrightarrow{1} q_{2m+1} \xrightarrow{1} \dots \xrightarrow{1} q_{4m}$$

where $q_{4m} \in F$ (and where $q_i \xrightarrow{0} q_{i+1}$ stands for "A moves from state q_i to state q_{i+1} by reading 0"). Let us look at the run prefix $q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{0} \dots \xrightarrow{0} q_{2m}$ that goes through the first half of the word $0^{2m}1^{2m}$. This run prefix is composed of $2m+1$ states. Recall that A has only m different states. Thus, by the pigeonhole principle, there exist $q \in Q$, $i, j \in \mathbb{N}$ such that $0 \leq i < j \leq 2m$ and $q_i = q_j = q$. That is, the run prefix is of the following form:

$$q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_2 \xrightarrow{0} \dots \xrightarrow{0} q_i = q \xrightarrow{0} \dots \xrightarrow{0} q_j = q \xrightarrow{0} \dots \xrightarrow{0} q_{2m} \xrightarrow{1} q_{2m+1} \xrightarrow{1} \dots \xrightarrow{1} q_{2m}$$

This means that the path $q_i = q \xrightarrow{0} \dots \xrightarrow{0} q_j = q$ is actually a *loop*, and, furthermore, that we can repeat it (or delete it) and still obtain an accepting run of A.

Indeed, if this:

$$q_0 \xrightarrow{0} \dots \xrightarrow{0} q_i = q \xrightarrow{0} \dots \xrightarrow{0} q_j = q \xrightarrow{0} \dots \xrightarrow{0} q_{2m} \xrightarrow{1} q_{2m+1} \xrightarrow{1} \dots \xrightarrow{1} q_{2m} \xrightarrow{1} q_{2m+1} \xrightarrow{1} \dots \xrightarrow{1} q_{4m}$$

is an accepting run, then the following one, where the loop is repeated twice,

$$q_0 \xrightarrow{0} \dots \xrightarrow{0} q_i = q \xrightarrow{0} \dots \xrightarrow{0} q_j = q = q_i \xrightarrow{0} \dots \xrightarrow{0} q_j = q \xrightarrow{0} \dots \xrightarrow{0} q_{2m} \xrightarrow{1} q_{2m+1} \xrightarrow{1} \dots \xrightarrow{1} q_{2m} \xrightarrow{1} q_{2m+1} \xrightarrow{1} \dots \xrightarrow{1} q_{4m}$$

is also an accepting run, as it fully respects the transition function of A, and ends in q_{4m} which is an accepting state. Let $k = j - i$ and $\ell = 2m - j$. Observe now which word is accepted by this run: $0^i0^k0^k0^\ell1^{2m}$. Recall that $i + k + \ell = 2m$, thus $i + 2k + \ell > 2m$ as $k > 0$. Thus, the word $0^i0^k0^k0^\ell1^{2m}$ is *not* in the language L. This is a contradiction with the assumption that L was accepted by A, which means, in particular, that any word not in L has to be rejected by A. This shows that there cannot exist a finite automaton accepting L. Hence, L is not regular.

Remark. Note that by repeating the loop j times, or by deleting it, we can also show that for all $j \in \mathbb{N}$ (even $j = 0$), $0^i0^{j\times k}0^l1^{2m}$ is accepted by A, so A *wrongly* accepts an infinite number of words.

[1]Do not hesitate to ask us if you want advice on this.

Ex. 2.3.
To prove that $(L^*M^*)^* = (L \cup M)^*$, we need to prove an equality between two sets, which can be done using two inclusions. We provide two different proofs of these inclusions: one remaining at the set level and another one done at the word level. The latter is more precise, but also more tedious.

At set level Both directions rely on the following fact: Kleene closure of both sets preserves inclusion.

$\boxed{(L^*M^*)^* \subseteq (L \cup M)^*}$ $L \subseteq L \cup M$ therefore $L^* \subseteq (L \cup M)^*$. Similarly, $M \subseteq L \cup M$ therefore $M^* \subseteq (L \cup M)^*$. So $L^*M^* \subseteq (L \cup M)^*(L \cup M)^* = (L \cup M)^*$. Taking the Kleene closure on both sides yields $(L^*M^*)^* \subseteq ((L \cup M)^*)^*$. However, the Kleene closure is an idempotent operator (for every language K, we have $(K^*)^* = K^*$), therefore $((L \cup M)^*)^* = (L \cup M)^*$, and we obtain $(L^*M^*)^* \subseteq (L \cup M)^*$.

$\boxed{(L^*M^*)^* \supseteq (L \cup M)^*}$ We notice that $L = L^1 \cdot M^0 \subseteq L^*M^*$: indeed a word of L can be decomposed a exactly one word of L followed by ε. Similarly, $M = L^0 \cdot M^1 \subseteq L^*M^*$. As a result $L \cup M \subseteq L^*M^*$. Taking the Kleene closures on both sides, we obtain $(L \cup M)^* \subseteq (L^*M^*)^*$.

At word level

$\boxed{(L^*M^*)^* \subseteq (L \cup M)^*}$ Let $w \in (L^*M^*)^*$. Then we can decompose w according to this regular expression:

$$w = u_{1,1} \cdots u_{1,n_1} \cdot v_{1,1} \cdots v_{1,p_1} \cdot u_{2,1} \cdots u_{2,n_2} \cdot v_{2,1} \cdots v_{k,p_k}$$

where for each i, j, $u_{i,j} \in L$ and $v_{i,j} \in M$. Note that n_is and p_is can be zero. Then, by definition of inclusion for each i, j, we have $u_{i,j} \in L \cup M$ and $v_{i,j} \in L \cup M$. That means w can be decomposed into words of $L \cup M$ and so $w \in (L \cup M)^*$.

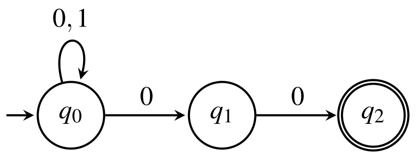
$\boxed{(L^*M^*)^* \supseteq (L \cup M)^*}$ Let $w \in (L \cup M)^*$. We can write $w = w_1 \cdots w_n$ with each $w_i \in (L \cup M)$. That means that for each i,

- if $w_i \in L$, then $w_i = w_i \cdot \varepsilon \in L^*M^*$;
- if $w_i \in M$, then $w_i = \varepsilon \cdot w_i \in L^*M^*$.

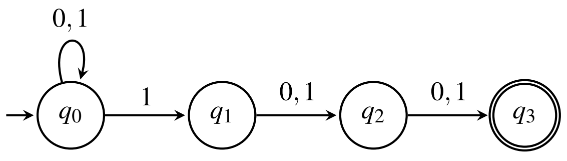
In both cases $w_i \in L^* \cdot M^*$, therefore $w \in (L^*M^*)^*$.

2.2 Finite automata

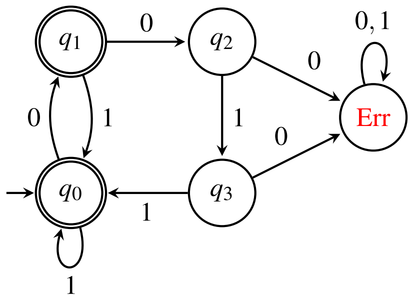
Ex. 2.4.



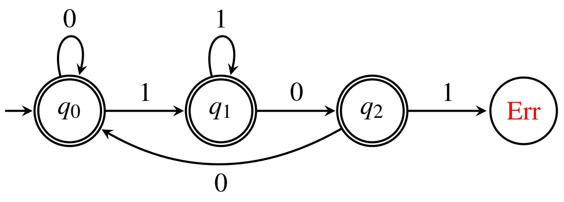
(1) The set of strings ending with 00



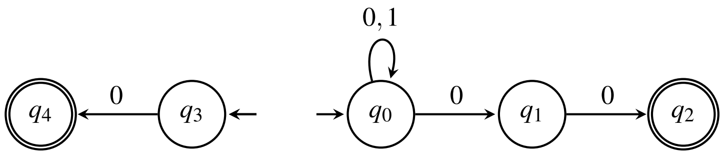
(2) The set of strings whose 3rd symbol, counted from the end of the string, is a 1.



(3) The set of strings where each pair of zeroes is directly followed by a pair of ones.

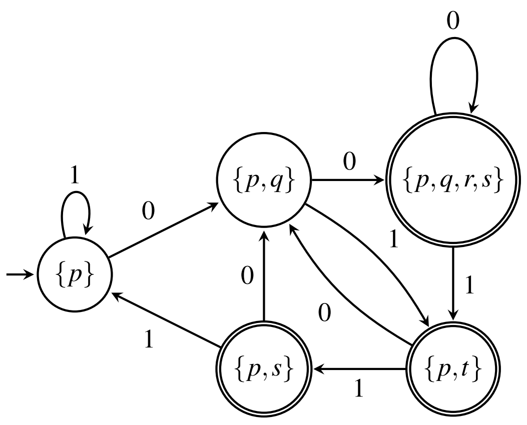


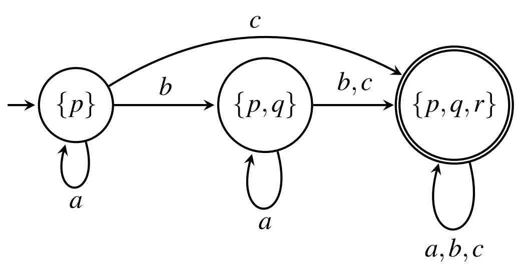
(4) The set of strings that do not contain the sequence 101.



(5) The set of binary numbers divisible by 4 (this is automaton has two initial states).

Ex. 2.5.





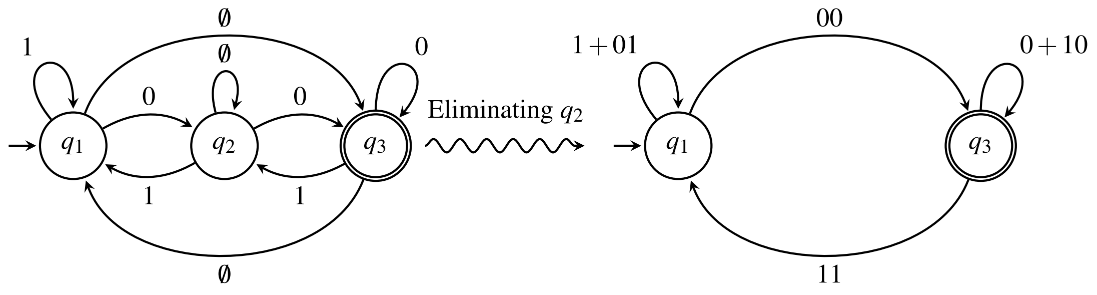
2.3 Regular expressions

Ex. 2.6. RE are:

1. $(0+1)^*00$
2. $(0+1)^*1(0+1)(0+1)$
3. $(1+01+0011)^*(0+\varepsilon)$
4. $0^*(1+00(0^*))^*0^*$
5. $(0+1)^*00+0$

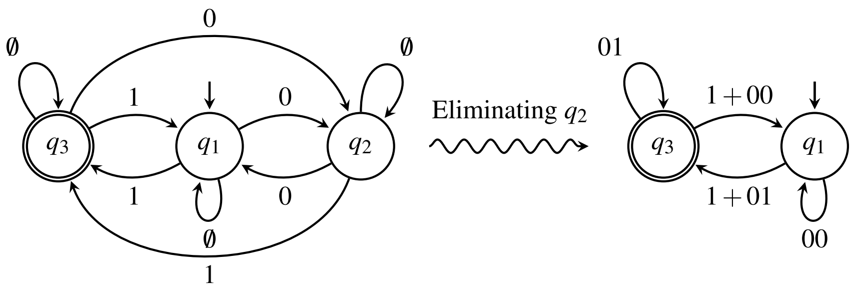
Ex. 2.7. RE are:

a) We calculate the RE corresponding to the paths from the an initial state (here q_1) to a final state (here q_3) by eliminating other states:



As the initial and final state differ, the RE will be $((1+01)+00\cdot(0+10)^*\cdot 11)^*\cdot 00\cdot(0+10)^*$.

b) We calculate the RE corresponding to the paths from the an initial state (here q_1) to a final state (here q_3) by eliminating other states:



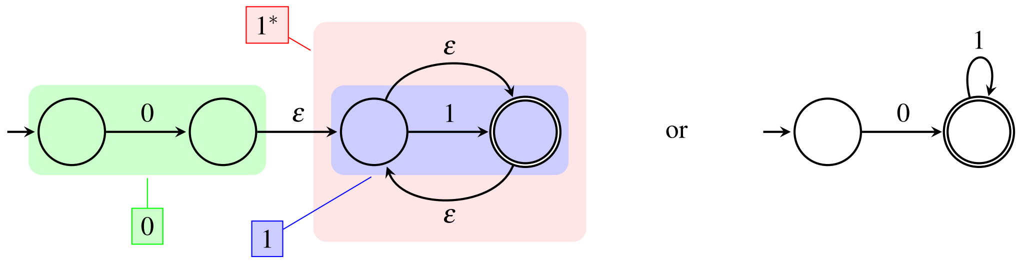
As the initial and final state differ, the RE will be $(00+(1+01)\cdot(01)^*\cdot(1+00))^*(1+01)\cdot(01)^*$.

Ex. 2.8. Remember that

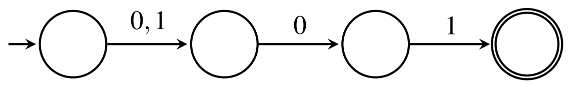
AB gives $(Q_1)\longrightarrow_A (Q_2)\longrightarrow_B (Q_3)$

$A+B$ gives $(Q_4)\longleftarrow_\varepsilon (Q\prime_3)\longleftarrow_A (Q\prime_2)\longleftarrow_\varepsilon (Q_1)\longrightarrow_\varepsilon (Q_2)\longrightarrow_B (Q_3)\longrightarrow_\varepsilon (Q_4)$

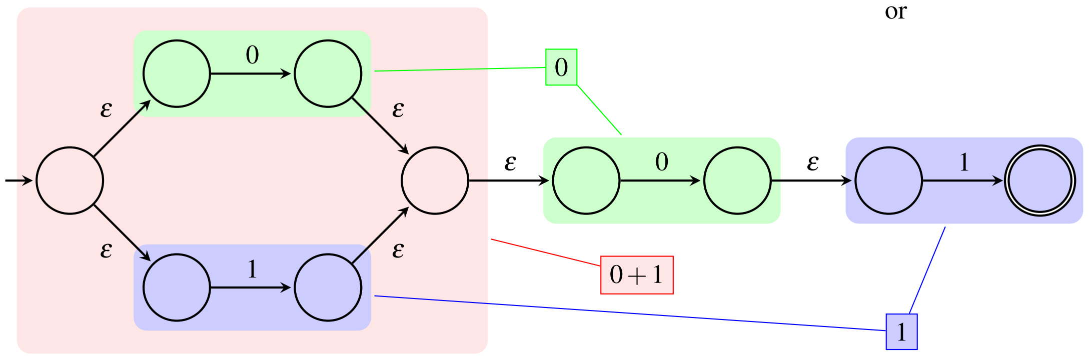
X^* $(Q_2)\longleftarrow_X (Q_1)\longleftrightarrow_\varepsilon (Q_2)$



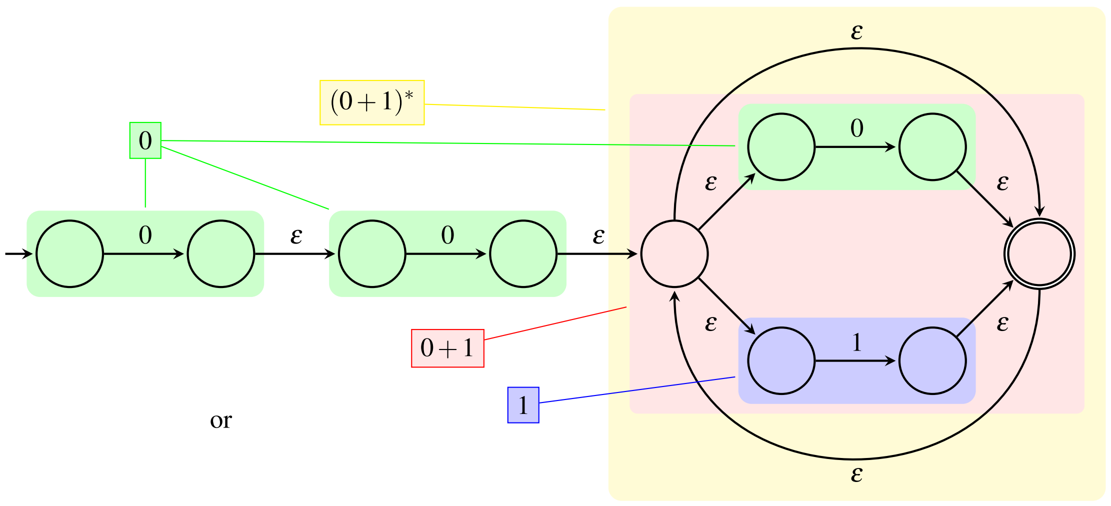
1. 01^*



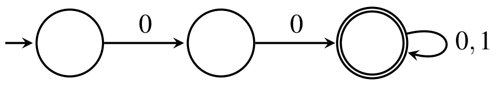
or



2. $(0+1)01$



or



3. $00(0+1)^*$

2.4 Extended regular expressions

Ex. 2.9. `(.|\n){5}`

Ex. 2.10. `\\*\**`

Ex. 2.11. `^#.*$`

Ex. 2.12. `[0-9]+(\.[0-9]+)?(E[+-]?[0-9]+)?`

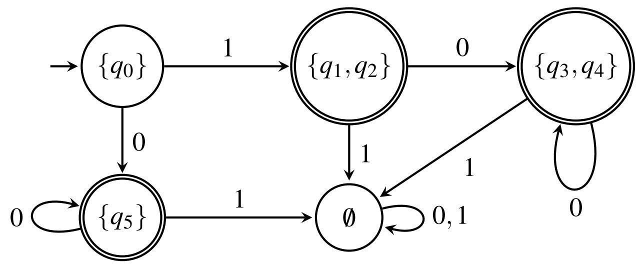
Ex. 2.13. `^[A-Z][A-Za-z]*(\ [A-Za-z]+)*\.$`

Ex. 2.14. `abcde[A-Za-z_]{3}(?=\.ext)`

2.5 Minimisation of automata and other operations

Ex. 2.15.

1.



2. The sink state can be removed; in addition, states $\{q_5\}$ and $\{q_3,q_4\}$ are the same and could be merged (as formally described in the next question).

3. It is easier to start this from the last strongly connected components:

   $\emptyset$: $\emptyset$ (not accepting and cannot reach any other state)

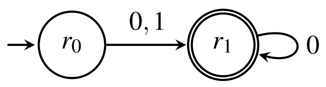
   $\{q_5\}$: 0^*

   $\{q_3,q_4\}$: 0^*

   $\{q_1,q_2\}$: $\varepsilon + 0 \cdot 0^* = \varepsilon + 0^+ = 0^*$

   $\{q_0\}$: $0 \cdot 0^* + 1 \cdot 0^* = (0+1)0^*$

4. We remove sink state $\emptyset$ (as we want a minimal DFA, not a *minimal and complete* one) and merge states $\{q_5\}$, $\{q_3,q_4\}$, $\{q_1,q_2\}$ into a single state r_1 (and we rename $\{q_0\}$ into r_0):



5. We build an upper triangular matrix to build the equivalence relation, and refine it:

$\{q_5\}$	$\{q_1,q_2\}$	$\{q_3,q_4\}$	$\emptyset$	
0	0	0	1	$\{q_0\}$
	1	1	0	$\{q_5\}$
		1	0	$\{q_1,q_2\}$
			0	$\{q_3,q_4\}$

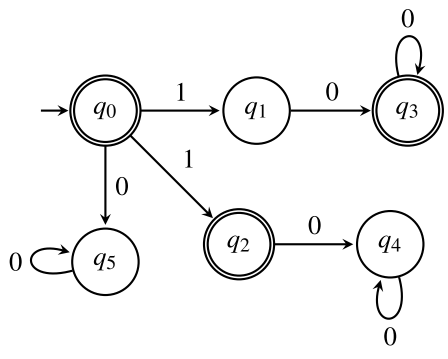
$$\boxed{\begin{array}{c} \{q_0\} \xrightarrow{0} \{q_5\} \\ \emptyset \xrightarrow{0} \emptyset \\ \{q_5\} \nsim \emptyset \end{array}}$$

$\{q_5\}$	$\{q_1,q_2\}$	$\{q_3,q_4\}$	$\emptyset$	
0	0	0	0	$\{q_0\}$
	1	1	0	$\{q_5\}$
		1	0	$\{q_1,q_2\}$
			0	$\{q_3,q_4\}$

   We can check that the refinement process stabilises because for each letter the states $\{q_5\}$, $\{q_3,q_4\}$, $\{q_1,q_2\}$ lead to equivalent states. We can then build the same automaton as above (we need to remove state $\emptyset$ in this case as well). The process was however much more efficient and formal.

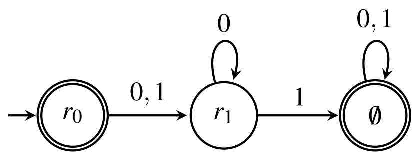
Ex. 2.16.

1. Accepting states become non-accepting and non-accepting states become accepting.



2. $L(A') = \varepsilon + 1 + 10^+$ The word 10 is accepted by both A (run $q_0 \xrightarrow{1} q_2 \xrightarrow{0} q_4$) and A' (run $q_0 \xrightarrow{1} q_1 \xrightarrow{0} q_3$). So the intersection of the languages is not empty (one can check that the intersection is 10^+) and therefore they cannot be the complement of each other.

3. We use the *complete (minimal) DFA* built in exercise 2.15 and then invert the accepting and non-accepting states, yielding $\overline{A}$:

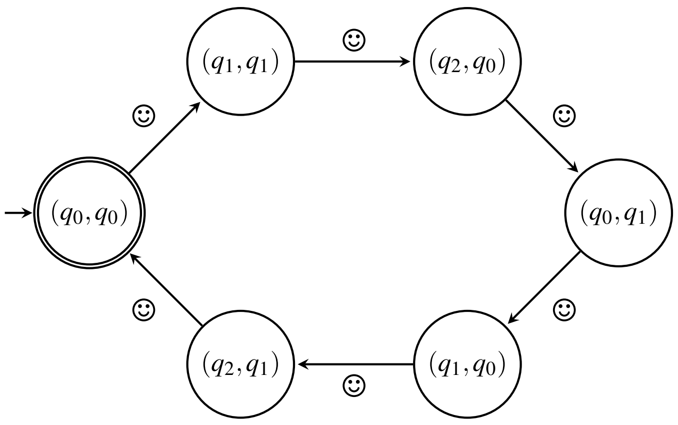


Note that we didn't need to use the minimal version, what is required is that it is both complete and deterministic so that for any word there exactly one corresponding run in the automaton. Flipping accepting states then only turns accepting runs into rejecting runs and vice versa. The language accepted by $\overline{A}$ is $L(\overline{A}) = \varepsilon + (0+1) \cdot 0^* \cdot 1 \cdot (0+1)^*$. That is indeed the complement: A accepts words that are one letter followed by some zeroes. This one accepts either the empty word or a word that has a 1 at a position at least 2.

Ex. 2.17.

1. A accepts words whose length is divisible by 3. B accepts words whose length is divisible by 2 (even length).

2.



This automaton accepts words whose length is divisible by 6. That is also the set of words whose length is divisible both by 2 and 3.

2.6 The scanner generator JFlex

Ex. 2.18. See `exercise2.18.flex` file provided on the UV.

Ex. 2.19. See `exercise2.19.flex` and `exercise2.19-bis.flex` files provided on the UV.

Ex. 2.20. See `exercise2.20.flex` file provided on the UV.

Ex. 2.21. See `exercise2.21.flex` and `exercise2.21-bis.flex` files provided on the UV.

Ex. 2.22. See `exercise2.22.flex` file provided on the UV.

Chapter 3

Grammars

3.1 Context-free languages

Ex. 3.1. $L = \{1^n \mid \exists m \in \mathbb{N} : n = m^2\}$ is not a regular language. The intuition is testing whether a word belongs to this language requires counting an *a priori* unbounded value, which is beyond the capabilities of a finite automaton. We can prove it by using a pumping argument: pumping a subword an appropriate number of times will yield a word not in L. The formal proof is given below.

Assume, by way of contradiction, that L is regular. Therefore there is a finite automaton A with $p > 1$ states[1] that accepts L. Then let $w = 1^{p^2} \in L$ and $q_0 \xrightarrow{1} q_1 \cdots \xrightarrow{1} q_{p^2}$ be an accepting run of w by A. Since $p^2 > p$, there is a state appearing twice in the run: there are $0 \le i < j \le p^2$ such that $q_i = q_j$ and $j - i \le p$. We can decompose $w = 1^i \cdot 1^{j-i} \cdot 1^{p^2-j}$ and let $t = 1^i$, $u = 1^{j-i}$, $v = 1^{p^2-j}$. Since $q_i = q_j$, any word $w_k = t \cdot u^k \cdot v$ is accepted by A using the run $q_0 \xrightarrow{1} q_1 \cdots q_i \left(\xrightarrow{1} \cdots \xrightarrow{1} q_j\right)^k \xrightarrow{1} \cdots \xrightarrow{1} q_{p^2}$. In particular, $w' = t \cdot u^2 \cdot v = 1^{p^2+(j-i)}$ is accepted by A. Because $j - i \le p$, we have $|w'| = p^2 + (j-i) < p^2 + 2p + 1 = (p+1)^2$ and $|w'|$ cannot also be a square. So w' is not in L despite being accepted by A, which is a contradiction.

3.2 Grammars

Ex. 3.2.

a) Right-regular grammar giving all strings made of 1s which may be followed by a single 0. 0 itself is also accepted. This language is the regular language: $1^*(0+1)$.

   1110 can be derived as $S \Rightarrow 1S \Rightarrow 11S \Rightarrow 111S \Rightarrow 1110$.

b) Context-free grammar giving all arithmetical expressions (in polish notation) using addition and multiplication with the mathematical variable called a. Not class 3 because S appears twice in the right hand-side of rules (2) and (3).

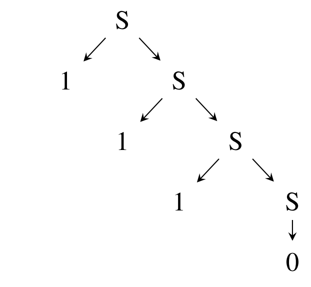
   $*+a+aa*aa$ can be derived as $S \Rightarrow *SS \Rightarrow *+SSS \Rightarrow *+aSS \Rightarrow *+a+SSS \Rightarrow *+a+aSS \Rightarrow *+a+aaS \Rightarrow *+a+aa*SS \Rightarrow *+a+aa*aS \Rightarrow *+a+aa*aa$

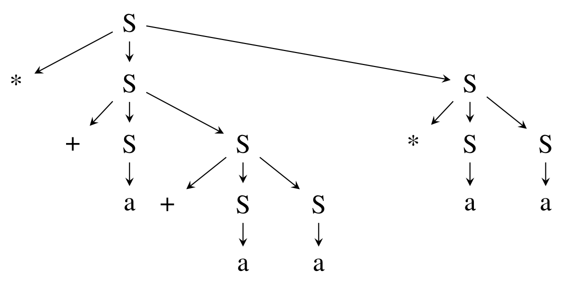
c) Unrestricted grammar giving all strings made of abc taken at least once. Not context-sensitive because $A \to \varepsilon$ is a rule (the only rule which can generate ε in a CS grammar is $S \to \varepsilon$).

   $abcabc$ can be derived as $S \Rightarrow abcA \Rightarrow abcS \Rightarrow abcabcA \Rightarrow abcabc\varepsilon = abcabc$.

In the following figure are pictured parse trees for the first two words. Note however that we cannot write a parse tree for $abcabc$, since such structure does not fit when there are more than one variable on the left-hand-side.

[1] If $p \le 1$ there are not many possible automata. It can be checked that only $\emptyset$ and 1^* can be accepted by an empty or single-state automaton, and neither of these languages is L.

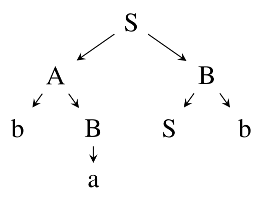

(1) 1110

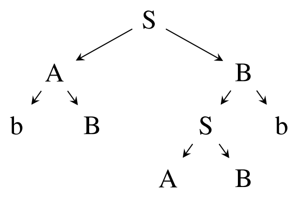

(2) $*+a+aa*aa$

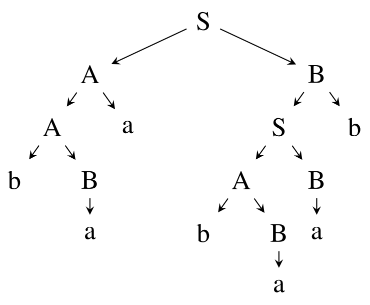
Ex. 3.3.

1. The given grammar is context-free but *not regular*: there cannot be a rule of the form $A \to \alpha B$ combined with a rule of the form $A \to B\alpha$ in a regular grammar.

2. The *parse tree* are as follow:


(3) *baSb*


(4) *bBABb*


(5) *baabaab*

3. The *leftmost* derivation of *baabaab* is: $S \Rightarrow AB \Rightarrow AaB \Rightarrow bBaB \Rightarrow baaB \Rightarrow baaSb \Rightarrow baaABb \Rightarrow baabBBb \Rightarrow baabaBb \Rightarrow baabaab$

   The *rightmost* derivation is: $S \Rightarrow AB \Rightarrow ASb \Rightarrow AABb \Rightarrow AAab \Rightarrow AbBab \Rightarrow Abaab \Rightarrow Aabaab \Rightarrow bBabaab \Rightarrow baabaab$

Ex. 3.4. ***Trick****: when you add a letter b, you also add an a in order to have at least $|a|$ = $|b|$.*

$$S \quad \to \quad bSa \mid aSb \mid abS \mid baS \mid Sab \mid Sba \mid Sa \mid aS \mid a$$

We can derive *baaba* from this grammar:

$$S \Rightarrow baS \Rightarrow baabS \Rightarrow baaba$$

Alternative solution*:* If w contains strictly more as than b, then either it contains only as, or it is of the form $bw'w''$, $w'bw''$, or $w'w''b$, where w' and w'' contain strictly more as than bs.

Indeed, let us write $w = w_1 \dots w_n$, and let us assume that w contains at least one b. Let i be the index of the first b: then, we can already say that either $w_1 \dots w_{i-1}$ is empty, or contains strictly more as than bs — since it contains only as.

Then, either:

- we have $i = 1$, and therefore, we have $\mid w_2 \dots w_n \mid_a \geq \mid w_2 \dots w_n \mid_b +2$. Therefore, there exists an index j such that both $w_2 \dots w_j$ and $w_{j+1} \dots w_n$ contain strictly more as than bs, and w is of the form $bw'w''$, using the notations given above;

- or $i \neq 1$, and both $w_1 \dots w_{i-1}$ and $w_{i+1} \dots w_n$ contain strictly more *a*s than *b*s, and then, we have $w = w'bw''$ with notations above;

- or $i \neq 1$ and either $w_1 \dots w_{i-1}$ or $w_{i+1} \dots w_n$ contains at least as many *b*s than *a*s — it cannot be $w_1 \dots w_{i-1}$, so it is necessarily $w_{i+1} \dots w_n$. Then, we have $\mid w_1 \dots w_{i-1} \mid_a \geq \mid w_1 \dots w_{i-1} \mid_b +2$. If $i = n$, then w is of the form $w'w''b$. Otherwise, i is not the index of the last b: we can therefore define j as the index of the next b. Then, we also have $\mid w_1 \dots w_{j-1} \mid_a > \mid w_1 \dots w_{j-1} \mid_b$, and we can make the same disjunction with j instead of i.

Therefore, an alternative grammar generating this language is:

$$S \rightarrow bSS \mid SbS \mid SSb \mid aS \mid a$$

Ex. 3.5.

n°	Rule	Idea
(1)	$S \rightarrow T$	generate *a*s at the start and *c*s at the end
(2)	$T \rightarrow aTc$	$\lvert a\rvert{+}{+}, \lvert c\rvert{+}{+}$
(3)	$T \rightarrow U$	generate *b*s at the start and *c*s at the end
(4)	$U \rightarrow bUc$	$\lvert b\rvert{+}{+}, \lvert c\rvert{+}{+}$
(5)	$U \rightarrow \varepsilon$	no more letters

aaabbccccc can be derived from this grammar:

$$S \overset{(1)}{\Rightarrow} T \overset{(2)}{\Rightarrow} aTc \overset{(2)}{\Rightarrow} aaTcc \overset{(2)}{\Rightarrow} aaaTccc \overset{(3)}{\Rightarrow} aaaUccc \overset{(4)}{\Rightarrow} aaabUcccc \overset{(4)}{\Rightarrow} aaabbUccccc \overset{(5)}{\Rightarrow} aaabbccccc$$

(*Bonus*) $L = \{a^n b^m c^\ell \in \Sigma^* \mid n = m = \ell\}$ cannot be generated by any context-free grammar. We first give an intuition: CFGs are recognized by pushdown automata, *i.e.* finite automata with a stack. Such stack can be used to count and compare the number of occurrences of two letters (for example a context-free grammar recognizing $\{a^n b^n \in \Sigma^* \mid n \in \mathbb{N}\}$). However, it cannot be used to compare the number of occurrences of three different letters at the same time, since the comparison is made by incrementing and decrementing the stack (which can here be seen as a counter), therefore losing information.

Now, for those who are interested, here is the formal proof. It uses the pumping lemma for CFLs:

Lemma. Let L be a CFL. Then there exists $p \in \mathbb{N}$ such that for all $z \in L$ s.t. $|z| \geq p$, there exists $u, v, w, x, y \in \Sigma^*$ such that $z = uvwxy$ and:

1. $|vwx| \leq p$ (the middle portion is not larger than p)

2. $v \neq \varepsilon$ or $x \neq \varepsilon$ (we will pump v and x, so at least one of the two should not be empty)

3. For all $i \geq 0$, $uv^iwx^iy \in L$ (we pump v and x)

The proof of this lemma is along similar lines as the one for finite automata (p corresponds to the number of states of the pushdown automaton).

Then, by taking the string $a^p b^p c^p \in L$, we have that our vwx can contain at most two distinct symbols (since it is of length at most p). Let us assume for example that $vwx = a^j b^k$ for some $j, k \geq 1$. Then it is easy to see that uv^2wx^2y does not contain as many *a*s as *b*s and *c*s: here, there will be more *a*s than *c*s (and more *b*s than *c*s). Thus, L does not satisfy the pumping lemma: it is not context-free.

Ex. 3.6.

n°	Rule	Idea
(1)	$S \rightarrow TU$	generate $a^m c^m$ at the start and $b^n d^n$ at the end
(2)	$T \rightarrow ATC$	$\lvert a\rvert = \lvert c\rvert$
(3)	$T \rightarrow AC$	$\lvert a\rvert = \lvert c\rvert = 1$
(4)	$U \rightarrow BUD$	$\lvert b\rvert = \lvert d\rvert$
(5)	$U \rightarrow BD$	$\lvert b\rvert = \lvert d\rvert = 1$
(6)	$CB \rightarrow BC$	swap c and b
(7)	$A \rightarrow a$	variable to terminal
(8)	$aB \rightarrow ab$	variable to terminal (first b)
(9)	$bB \rightarrow bb$	variable to terminal (subsequent b)
(10)	$bC \rightarrow bc$	variable to terminal (first c)
(11)	$cC \rightarrow cc$	variable to terminal (subsequent c)
(12)	$D \rightarrow d$	variable to terminal

The idea of this grammar is to generate $a^m c^m b^n d^n$, then by successive swaps of cb to bc, get to $a^m b^n c^m d^n$. The terminal generation prevents the swapping process to stop in the middle, as the actual letter b can only be produced by following an a or another (terminal) b; similarly, c can only be produced following a b or a previous (terminal) c.

The word $aabbbccddd$ can be derived from this grammar:

$$S \overset{(1)}{\Rightarrow} TU \overset{(2)}{\Rightarrow} ATCU \overset{(3)}{\Rightarrow} AACCU \overset{(4)}{\Rightarrow} AACCBUD \overset{(4)}{\Rightarrow} AACCBBUDD \overset{(5)}{\Rightarrow} AACCBBBDDD \overset{(6)}{\Rightarrow} AACBCBBDDD \overset{(6)}{\Rightarrow}$$

$$AABCCBBDDD \overset{(6)}{\Rightarrow} AABCBCBDDD \overset{(6)}{\Rightarrow} AABBCCBDDD \overset{(6)}{\Rightarrow} AABBCBCDDD \overset{(6)}{\Rightarrow} AABBBCCDDD \overset{(7)}{\Rightarrow}$$

$$aABBBCCDDD \overset{(7)}{\Rightarrow} aaBBBCCDDD \overset{(8)}{\Rightarrow} aabBBCCDDD \overset{(9)}{\Rightarrow} aabbBCCDDD \overset{(9)}{\Rightarrow} aabbbCCDDD \overset{(10)}{\Rightarrow}$$

$$aabbbcCDDD \overset{(11)}{\Rightarrow} aabbbccDDD \overset{(12)}{\Rightarrow} aabbbccdDD \overset{(12)}{\Rightarrow} aabbbccddD \overset{(12)}{\Rightarrow} aabbbccddd$$

This language cannot be generated by a context-free grammar. One intuition about this fact is that it would then be accepted by a pushdown automaton that has to count two different values at the same time. A more formal argument could be made using the pumping lemma given in Exercise 3.5: assuming it is CFL, take p as in the lemma and consider $a^p b^p c^p d^p$. Any subword of length at most p contains at most two symbols (a and b, b and c, or c and d), so pumping on any part of this subword would create an imbalance with respect to (at least) another letter not being pumped on: for example if the subword contains only as and bs then pumping will create some more as (creating an imbalance with the number of cs) or some more bs (creating imbalance with the ds).

Ex. 3.7.

n°	Rule	Idea
(1)	$D \rightarrow U+U$	$2=1+1$
(2)	$D \rightarrow D\cdot U$	$2=2\cdot 1$
(3)	$D \rightarrow U\cdot D$	$2=1\cdot 2$
(4)	$D \rightarrow D+Z$	$2=2+0$
(5)	$D \rightarrow Z\cdot D$	$2=0+2$
(6)	$U \rightarrow U+Z$	$1=1+0$
(7)	$U \rightarrow Z+U$	$1=0+1$
(8)	$U \rightarrow U\cdot U$	$1=1\cdot 1$
(9)	$U \rightarrow 1$	$1=1$
(10)	$Z \rightarrow Z+Z$	$0=0+0$
(11)	$Z \rightarrow P\cdot 0$	$0=x\cdot 0$
(12)	$Z \rightarrow 0\cdot P$	$0=0\cdot x$
(13)	$Z \rightarrow 0$	$0=0$
(14)	$P \rightarrow P\cdot P$	product of terms
(15)	$P \rightarrow (X)$	well-parenthesised expression
(16)	$P \rightarrow V$	value
(17)	$X \rightarrow S$	sum
(18)	$X \rightarrow P$	product
(19)	$X \rightarrow V$	value
(20)	$S \rightarrow X+X$	sum of expressions
(21)	$V \rightarrow 0$	value 0
(22)	$V \rightarrow 1$	value 1

This grammar is best explained starting from the question “how to generate 0?” (here variable Z). The idea is that the sum of expressions which are 0 is also going to be 0, the product of *anything* with 0 will be 0, and indeed integer 0 is 0. Variables P, X, S, V help define what this *anything* can be: X generates any expression, sum (S), product (P), or value (V, either 0 or 1). Care is taken to enclose any sum within parentheses before taking a product.

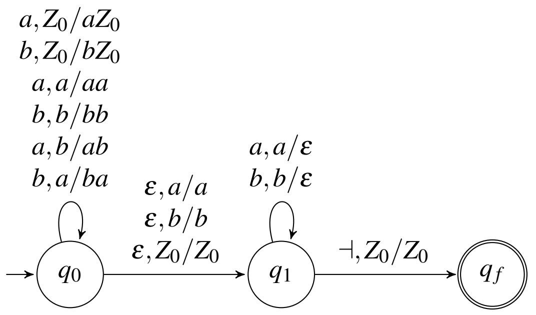
Now that 0 can be generated through Z, we generate 1 through U: $1 = 1+0 = 1\cdot 1$. Finally 2 is generated as the sum of two expressions of value 1: $2 = 1+1$, or with 2 in operation with the corresponding neutral element.

Chapter 4

All things context free…

4.1 Pushdown automata

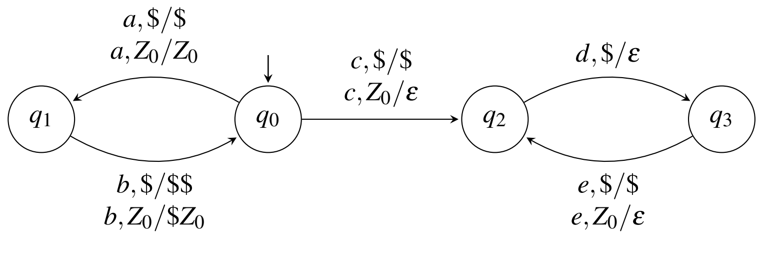
Ex. 4.1. In the case of L_{pal}, we cannot, as we did with $L_{pal\#}$, design a *deterministic* pushdown automaton to accept the language! We need the power of non-determinism to be able to "guess" properly the middle of the input word and start checking if the remaining input is indeed the mirror image of the first half. In the following, we design a PDA such that $L(P) = L_{pal}$, that is P accepts by *final state*. Of course, it is always possible to design a PDA P' such that $N(P') = L_{pal}$, that is P' accepts by *empty stack*, but we leave that as an exercise. Finally, we choose here to have, as a convention, a distinguished "end of the input word symbol" $\dashv$ to avoid any confusion that could appear while using only ε, but this is not necessary.



Let us now give an *accepting* execution on the input word *abaaaaba*:

$\langle q_0, abaaaaba \dashv, Z_0\rangle \vdash \langle q_0, baaaaba \dashv, aZ_0\rangle \vdash \langle q_0, aaaaba \dashv, baZ_0\rangle \vdash \langle q_0, aaaba \dashv, abaZ_0\rangle \vdash \langle q_0, aaba \dashv, aabaZ_0\rangle \vdash \langle q_1, aaba \dashv, aabaZ_0\rangle\langle q_1, aba \dashv, abaZ_0\rangle \vdash \langle q_1, ba \dashv, baZ_0\rangle \vdash \langle q_1, a \dashv, aZ_0\rangle \vdash \langle q_1, \dashv, Z_0\rangle \vdash \langle q_f, \dashv, Z_0\rangle$

Ex. 4.2. We use a single stack symbol \$ to count the occurrences of *ab* (push one \$) and *de* (pop one \$). Since the DPDA needs to be accepting by empty stack, we pop the Z_0 symbol when reading the last *e* or letter *c* when no *ab* was read beforehand.



4.2 Grammar transformations

Ex. 4.3.

1.
   - The unproductive symbol removal algorithm stabilises with $Prod = \{a,b,S,B\}$ so we get:
   $$G_1 = \langle \{S,B\}, \{a,b\}, \{S \to a, B \to b\}, S\rangle$$
   - We notice B can't be accessed from S in this new grammar and can thus be removed. We end up with:
   $$G' = \langle \{S\}, \{a\}, \{S \to a\}, s\rangle$$

2. STEP 1: Computational steps for *Prod*

i	*Prod*
0	$\{a,b\}$
1	$\{a,b,C,A\}$
2	$\{a,b,C,A,S\}$
3	$\{a,b,C,A,S\}$

STEP 2: We get the following P'

$$\begin{array}{lcl} S & \to & A \\ A & \to & bS \\ & \to & b \\ C & \to & AS \\ & \to & b \end{array}$$

STEP 3: We can now remove the inaccessible symbols

i	*Reach*
0	$\{S\}$
1	$\{S,A\}$
2	$\{S,A,b\}$
3	$\{S,A,b\}$

STEP 4: We finally obtain $G' = \langle V', P', T', S'\rangle$ where

- $V' = \{S,A\}$
- $P' = \{S \to A, A \to bS \mid b\}$
- $T' = \{b\}$
- $S' = S$

Ex. 4.4.

1. Word $ID + ID * ID$ is produced both by

$$\begin{tikzcd}
 & E \arrow[dl] \arrow[d] \arrow[drr] & & & \\
E \arrow[d] & op \arrow[d] & & E \arrow[dl] \arrow[d] \arrow[dr] & \\
ID & + & E \arrow[d] & op \arrow[d] & E \arrow[d] \\
 & & ID & * & ID
\end{tikzcd}$$

and

$$\begin{tikzcd}
 & & & E \arrow[dll] \arrow[d] \arrow[dr] & \\
 & E \arrow[dl] \arrow[d] \arrow[dr] & & op \arrow[d] & E \arrow[d] \\
E \arrow[d] & op \arrow[d] & E \arrow[d] & * & ID \\
ID & + & ID & &
\end{tikzcd}$$

2.
$$\begin{array}{lcl} E & \to & E{+}T \\ & \to & E{-}T \\ & \to & T \\ T & \to & T{*}F \\ & \to & T/F \\ & \to & F \\ F & \to & F \Rightarrow G \\ & \to & ID[E] \\ & \to & G \\ G & \to & ID \end{array}$$

Ex. 4.5.

<stmt>	$\to$	**if** <expr> **then** <stmt-list> <if-tail>
<if-tail>	$\to$	**end if**
<if-tail>	$\to$	**else** <stmt-list> **end if**

Ex. 4.6. There is no indirect left-recursion but two direct left-recursion rules: 1 and 3. We can turn them into right-recursion rules:

$$\begin{array}{rcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \\ & \rightarrow & \varepsilon \\ T & \rightarrow & PT' \\ T' & \rightarrow & *PT' \\ & \rightarrow & \varepsilon \\ P & \rightarrow & ID \end{array}$$

Ex. 4.7. STEP 1: Computational steps for *Prod*

i	*Prod*
0	$\{a,b,c,d\}$
1	$\{a,b,c,d,D,E,G\}$
2	$\{a,b,c,d,D,E,G,F\}$
3	$\{a,b,c,d,D,E,G,F,S\}$

STEP 2: We get the following P'

$$\begin{array}{rcl} S & \rightarrow & aE \\ & \rightarrow & bF \\ E & \rightarrow & bE \\ & \rightarrow & \varepsilon \\ F & \rightarrow & aF \\ & \rightarrow & aG \\ G & \rightarrow & Gc \\ & \rightarrow & d \\ D & \rightarrow & ab \end{array}$$

STEP 3: Last rule is the only unreachable rule ($Reach = \{S\} \uplus \{E,F\} \uplus \{G\}$) and can be removed.

STEP 4: There is a single left-recursion rule $G \rightarrow Gc$. It can be replaced by the three rules $G \rightarrow dG'$, $G' \rightarrow cG'$, and $G' \rightarrow \varepsilon$.

We obtain:

$$\begin{array}{rcl} S & \rightarrow & aE \\ & \rightarrow & bF \\ E & \rightarrow & bE \\ & \rightarrow & \varepsilon \\ F & \rightarrow & aF \\ & \rightarrow & aG \\ G & \rightarrow & dG' \\ & \rightarrow & cG' \\ & \rightarrow & \varepsilon \end{array}$$

STEP 5: Factoring the production rules for F:

$$\begin{array}{rcl} S & \rightarrow & aE \\ & \rightarrow & bF \\ E & \rightarrow & bE \\ & \rightarrow & \varepsilon \\ F & \rightarrow & aF' \\ F' & \rightarrow & F \\ & \rightarrow & G \\ G & \rightarrow & dG' \\ & \rightarrow & cG' \\ & \rightarrow & \varepsilon \end{array}$$