THEORY OF FORMAL LANGUAGES

EXERCISE BOOK

A Suite of Exercises with Solutions

DRAFT COPY

Luca Breveglieri

collaborators

Giampaolo Agosta
Alessandro Barenghi
Anna Beletska
Stefano Crespi Reghizzi
Bernardo Dal Seno
Vincenzo Martena
Angelo Morzenti
Licia Sbattella
Martino Sykora

21st January 2008

Contents

Chapter 1

Introduction

Chapter 2

Regular Languages

2.1 Regular Expression

Exercise 1 Consider the two following regular expressions R_1 and R_2 over alphabet $\{a, b\}$:

$$R_1 = a^* \; b^* \; a^* \qquad\qquad R_2 = (\; a \; b \;\mid\; b \; b \;\mid\; a \;)^*$$

These expressions generate regular languages L_1 and L_2, respectively.
Answer the following questions:

1. List all the strings of length ≤ 4 belonging to set differences $L_1 \backslash L_2$ and $L_2 \backslash L_1$ (sometimes also written $L_1 - L_2$ and $L_2 - L_1$ respectively).
2. Write the regular expression R_3 generating language $L_3 = \neg L_1$ (set complement of L_1) and use only union, concatenation, star and cross operators.

Solution of (1)

In order to compute quickly the required set differences one can proceed intuitively and notice what follows:

- in the regular expression R_1 only one sequence of letters b of arbitrary length ≥ 1 may occur, in any position
- in the regular expression R_2 one or more sequences of letters b of arbitrary length ≥ 1 may occur, separated by at least one letter a, internally to or at an end of the string, with the exception that if a sequence is positioned at the beginning of the string then it has to be of even length

Therefore the set differences are characterised as follows:

- the phrases contained in the language L_1 but not in the language L_2 are those of L_1 that begin with an odd number of letters b
- the phrases contained in the language L_2 but not in the language L_1 are those of L_2 that contain at least two letters b separated by at least one letter a

As a consequence, here are the strings belonging to language L_1 but not to language L_2 (on the left), and to L_2 but not to L_1 (on the right), limited to the case of length ≤ 4:

$L_1 \setminus L_2$	$L_2 \setminus L_1$
b	$a\,b\,a\,b$
$b\,a$	$b\,b\,a\,b$
$b\,a\,a$	
$b\,b\,b$	
$b\,a\,a\,a$	
$b\,b\,b\,a$	

The strings are ordered lexicographically, assuming $a < b$. One sees soon that the listed strings satisfy precisely the two intuitive characterisations stated above. □

Observation

As an alternative one can obtain the requested set differences in a purely algorithmic way, first by listing all the strings of length ≤ 4 of languages L_1 and L_2, respectively, and then by determining the two difference sets. The reader is left this task, perhaps a little boring but not long and surely at all trivial.

Solution of (2)

As is has been observed before, the phrases of language L_1 are characterised by not containing any pair of letters b that are not adjacent. More explicitly, a phrase of L_1 is as follows:

- either it does not contain any letter b
- or it contains exactly one letter b
- or it contains two or more letters b, but not separated by any letter a

As a consequence, the phrases of language $L_3 = \neg L_1$ (set complement of L_1) are characterised so as to contain at least two letters b separated by at least one letter a, as follows:

$$L_3 = \{\ x \ \mid \quad x \in \Sigma^* \text{ contains at least two characters } b \text{ that are not adjacent } \}$$

Therefore language L_3 is generated by the following regular expression R_3:

$$R_3 = (a \ \mid \ b)^* \ b \ a^+ \ b \ (a \ \mid \ b)^*$$

If one prefers, the expression can be rewritten as $R_3 = \Sigma^* \ b \ a^+ \ b \ \Sigma^*$, where the role of the universal language Σ^* is put into evidence. □

Observation

Notice that the given regular expression R_3 is highly indeterministic (actually it is even ambiguous), and as such it is very elegant and compact in describing language L_3 without unuseful details. If one prefers a version that is not ambiguous, here it is: $R_3 = a^* \; b^+ \; a^+ \; b \; \Sigma^*$. Such a formulation specifies that the necessary pair of separated letters b generated by component $b^+ a^+ b$ is the leftmost one, while the other such pairs (if any) are generated by component Σ^*.

Of course one could obtain algorithmically, without the need of intuition, a regular expression that generates language $L_3 = \neg L_1$, as follows:

- build the deterministic automaton equivalent to regular expression R_1, by means of the McNaughton-Yamada algorithm or of the Thompson modular construction followed by determinisation with the subset construction
- build the complement automaton, which is deterministic as well
- obtain a regular expression R_3 equivalent to the complement automaton, by means of the Brozozowski node elimination algorithm or of the algorithm of linear equations

In such a strictly algorithmic way no intuition is needed, but the whole process is presumably longer. The reader is left the task of applying the above described approach.

Exercise 2 Each of the two following tables A and B consists of two rows, and each row contains two regular expressions: the former is in column L_1, the latter in L_2.

Answer the following questions:

1. In each row, list by increasing length the three shortest strings belonging to the language defined by set difference $L_1 \setminus L_2$ (or $L_1 - L_2$).

Table A

L_1	L_2	write the three shortest strings of $L_1 \setminus L_2$
$a((b \mid bb)a)^+$	$(ab)^* ba$	
$(a(ab)^* b)^* (\varepsilon \mid a(ab)^* \mid ab)$	$(a(ab)^* b)^*$	

2. In each row, write a regular expression generating the language $L_1 \setminus L_2$ (or $L_1 - L_2$) and use only union, concatenation, star and cross operators.

Table B

L_1	L_2	write a regular expression of $L_1 \setminus L_2$
$a((b \mid bb)a)^+$	$(ab)^* ba$	
$(a(ab)^*b)^*(\varepsilon \mid a(ab)^* \mid ab)$	$(a(ab)^*b)^*$	

Write the answers to the questions in the rightmost column of Tables A and B.

Solution of (1) and (2)

For better convenience, here the answers to questions (1) and (2) are given together for the two rows of the tables (top and bottom).

Top row

In the top row the two languages L_1 and L_2 are characterised in the following way:

- set L_1 contains strings of length ≥ 3, each of which is a list of two or more elements a separated by only one letter b or two consecutive letters b, indifferently
- set L_2 contains strings of two different types:
  - the short string ba, of length 2
  - strings of length ≥ 4, each of which is a list of two or more elements a separated by only one letter b, except that between the second last (penultimate) and last element the separator is always the pair of consecutive letters bb

As a consequence, a list of elements a belongs to language L_1, but not to language L_2, in the following two cases:

case (a) the last separator is letter b (and the others are b or bb indifferently)

case (b) the last separator is the pair of consecutive letters bb and the string contains somewhere at least another separator bb, or even more than one, or as a limit case all the separators may be bb (this case implies however that there are at least three elements a)

L_1	L_2	the three shortest strings and the regexp of $L_1 \setminus L_2$
$a\ ((b \mid bb)\ a)^+$	$(a\ b)^*\ b\ a$	$a\ b\ a \qquad a\ b\ a\ b\ a \qquad a\ bb\ a\ b\ a$ $(a\ (b \mid bb))^*\ a\ b\ a\ \cup (a\ (b \mid bb))^*\ a\ bb\ (a\ (b \mid bb))^*\ a\ bb\ a$

The three shortest strings (all related to case (a) above) and the regular expression are obtained as a consequence. Spacing isolates and puts into evidence the groups of letters. □

Observation

For completeness, notice that there are two shortest strings in the fourth position, namely "$a\,b\,a\,b\,a\,b\,a$" and "$a\,bb\,a\,bb\,a$", and that the latter string is related to the case (b) above.

Bottom row

In the bottom row letters a and b can be interpreted as open and close parenthesis, respectively. Then the following holds:

- set L_1 contains all and only the strings of parentheses that have nesting depth ≤ 2 and satisfy one of the following two constraints:
  - they are well formed Dyck strings (cases (1) and (3) below)
  - they have exactly one exceeding open parenthesis, with respect to well formed Dyck strings (case (2) below)
- set L_2 contains all and only the Dyck strings that have nesting depth ≤ 2

Therefore the strings belonging to set difference $L_1 \setminus L_2$ are those that are of Dyck type with nesting depth ≤ 2 and have exactly one exceeding open parenthesis, that is case (2) of language L_1.

L_1	L_2	three shortest strings and regexp of $L_1 \backslash L_2$
$(a\ (ab)^*\ b)^*\ (\underbrace{\ \varepsilon\ }_{\text{case 1}} \mid \underbrace{a\ (ab)^*}_{\text{case 2}} \mid \underbrace{\ ab\ }_{\text{case 3}})$	$(a\ (ab)^*\ b)^*$	$a \qquad a\ ab \qquad ab\ a$ $(a\ (a\,b)^*\ b)^*\ a\ (a\,b)^*$

The three shortest strings and the regular expression are obtained as a consequence. Spacing isolates and puts into evidence the groups of letters. □

Observation

As an alternative, one can notice that in language L_1 case (3) is contained in the case (1), that case (2) is disjoint from case (1) (and therefore also from case (3)) and that language L_2 coincides with case (1) of L_1. There follows that set difference $L_1 \setminus L_2$ coincides with case (2) of L_2, which is the same conclusion as that obtained by the previous reasoning.

Observation

As an alternative, one can list the three shortest strings first by writing a few short strings of languages L_1 and L_2, and then by computing the set difference of these two sublists of strings.

Observation

The latter regular expression can be further simplified as follows:

$$\left(a\ \left(a\,b\right)^*\ b\right)^*\ a\ \left(a\,b\right)^* = a\ \left(\ b\,a\ \mid\ a\,b\ \right)^*$$

Such a simplification is easily justified by examining a few sample strings, short enough not to spend too long a time for the examination.

However one can obtain the same simplification also by reflecting on the fact that a Dyck string with nesting depth ≤ 2 and exactly one exceeding open parenthesis, is necessarily structured as follows:

- it certainly begins with letter a, as it may not begin with letter b if it has to be of Dyck type and moreover to contain somewhere an exceeding letter a
- after the initial character it continues with letters a and b in equal number, such that at the end it contains exactly one exceeding letter a (it is the initial letter a that unbalances the match though the exceeding open parenthesis may be one of the internal letters a)
- but it does not contain anywhere as a factor either string aaa or bbb (three or more consecutive letters a or b), such that it never has nesting depth > 2 (in fact both forbidden factors would imply to have a nesting depth of at least 3 if not higher)

Such is precisely the behaviour of the simplified regular expression on the right, which begins with letter a and continues with a mix of letters a and b, in any order and in equal number, but never contains forbidden factors because it never juxtaposes more than two identical letters.

By the way, this shows that the language is of local type and that can be expressed also in the following way (where $\Sigma = \{a, b\}$ and overlining indicates set complement):

$$a \ (\ b\,a \ \mid \ a\,b \ )^* = a \ \left(\Sigma^2 \right)^* \cap \overline{\Sigma^* \ \{ \, a\,a\,a, \ b\,b\,b \, \} \ \Sigma^*}$$

Such a simplification (which may look like somewhat unexpected) shows that there may exist notably different lines of reasoning that however reach the same target.

2.2 Finite State Automaton

Exercise 3 Consider the following regular expression over alphabet $\{a, b, c\}$:

$$R = a \ (\ b \ \mid \ b\,c^+ \ )^* \ c$$

Answer the following questions:

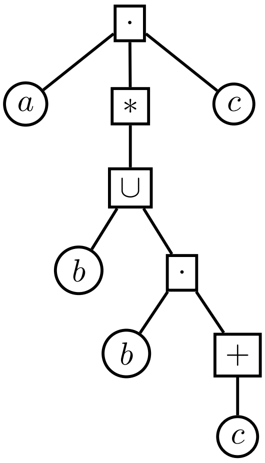
- Construct an indeterministic automaton A that recognises the language generated by regular expression R.
- Construct a deterministic automaton A' equivalent to automaton A.
- If necessary, construct the minimal automaton A'' equivalent to automaton A'.

First solution of (1)

Since an indeterministic automaton A equivalent to regular expression R is requested, a possibility is to use the Thompson or modular construction, which usually produces an indeterministic result. Here is the structure tree of R, which will be useful later for the construction, and all the strings of length ≤ 5 generated by R (to view intuitively the way R works):

structure tree of regular expression

$$R = a \cdot (b \mid b \cdot c^+)^* \cdot c$$



short (≤ 5) strings generated by R

$a\ c$

$a\ b\ c$

$a\ b\ b\ c$

$a\ b\ c\ c$

$a\ b\ b\ b\ c$

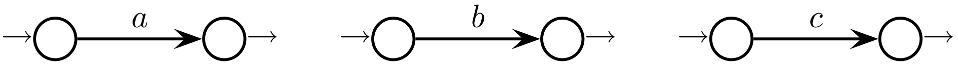
$a\ b\ b\ c\ c$

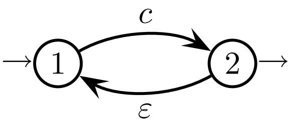
$a\ b\ c\ b\ c$

$a\ b\ c\ c\ c$

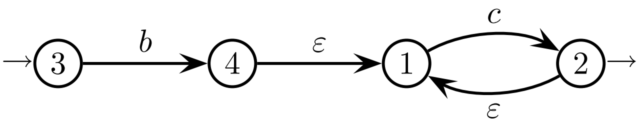
$\dots$

And here is the Thompson or modular construction, arranged step by step scanning bottom-up the structure tree of regular expression R, and in such a way as to have unique initial and final states at each step. Below each intermediate automaton produced by the construction, there is the corresponding fragment of R to which the automaton is equivalent.

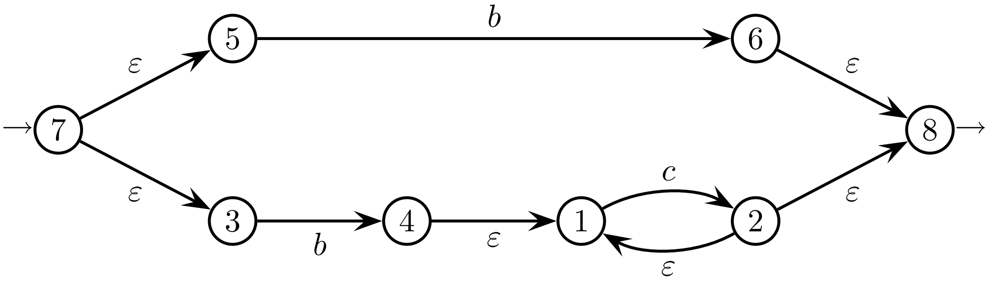




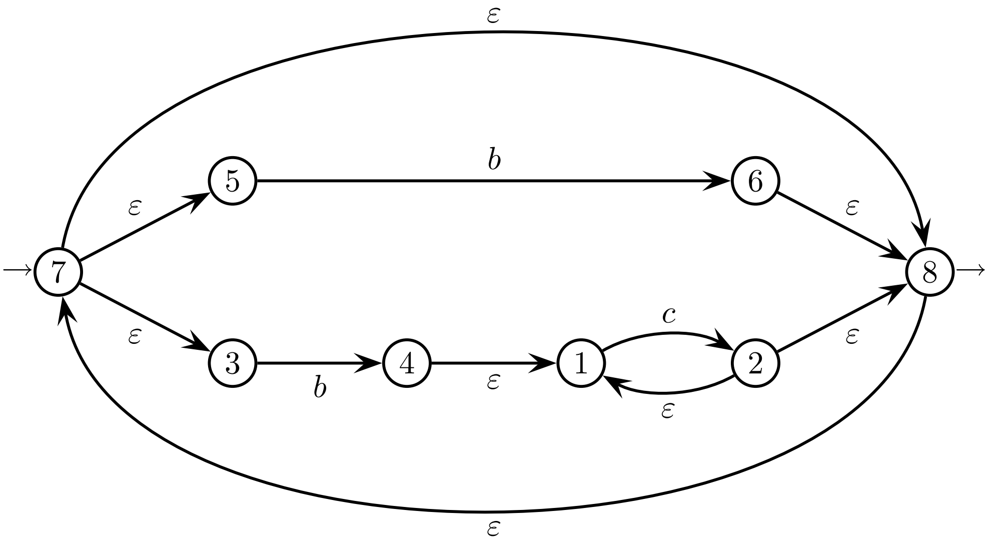
c^+



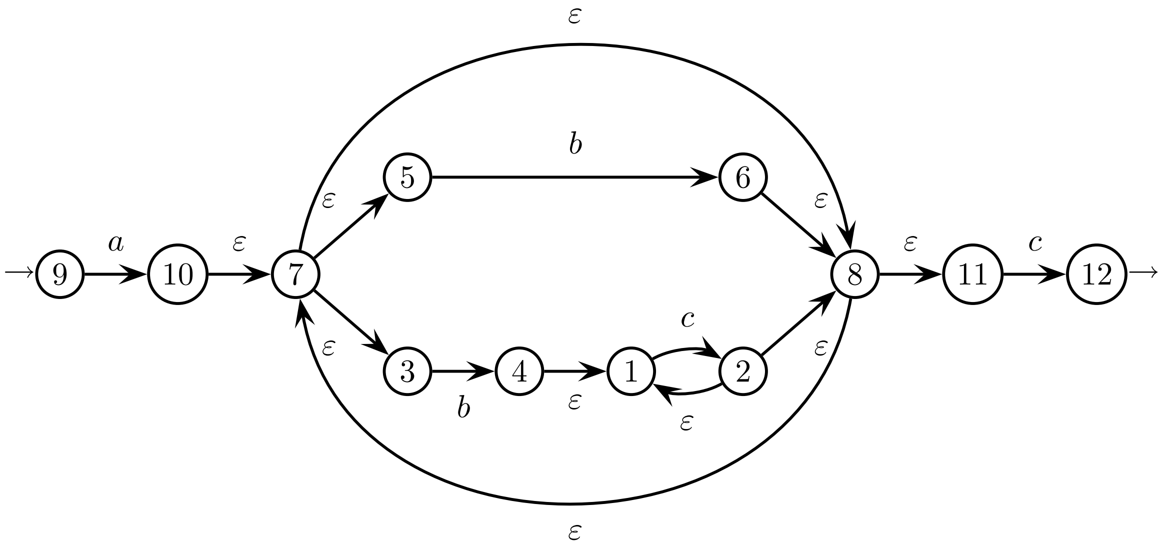
$b \cdot c^+$



$b \;\mid\; (b \cdot c^{+})$



$(b \;\mid\; (b \cdot c^{+}))^{*}$



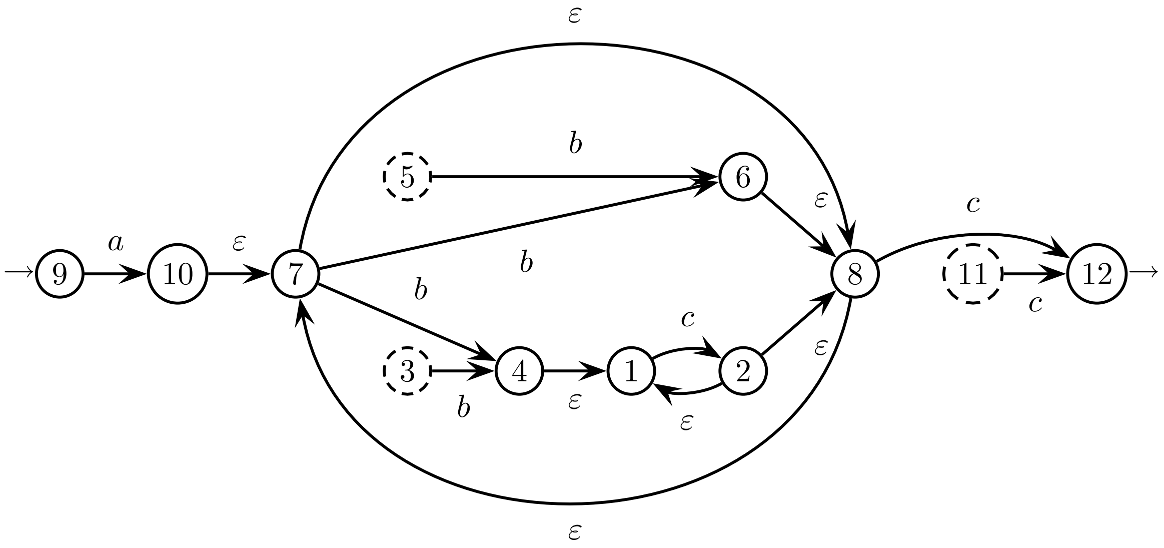
$a \cdot (b \;\mid\; (b \cdot c^{+}))^{*} \cdot c$

indeterministic automaton A

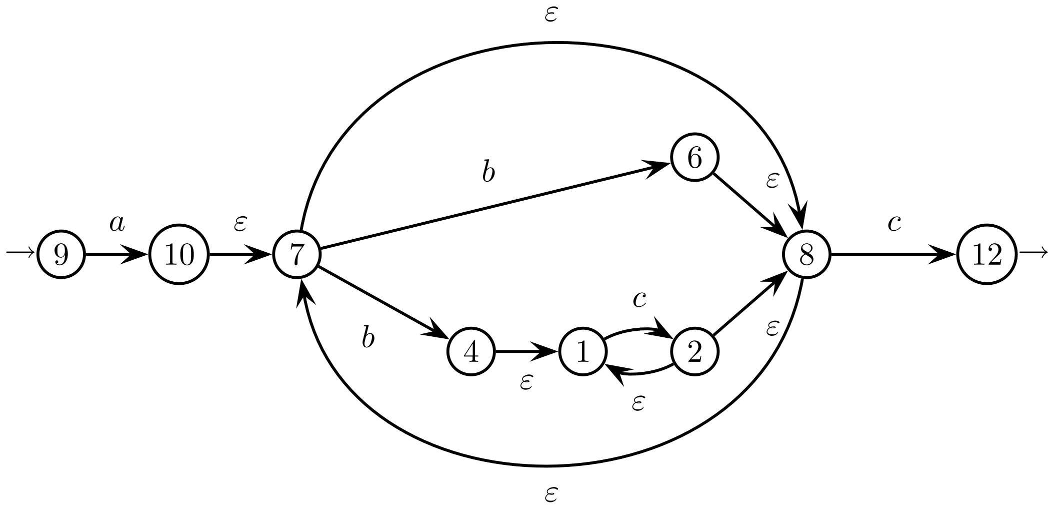
The final automaton A is indeterministic because it has several spontaneous transitions. The reader can check the correctness of A using the sample short strings listed at the beginning. □

First solution of (2)

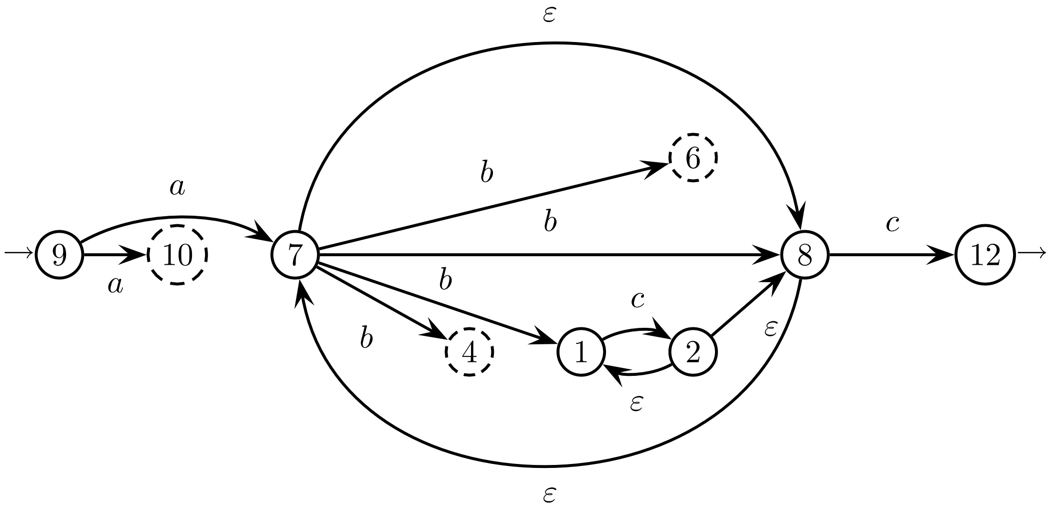
To determinise automaton A (which already has a unique initial state), one has first to cut spontaneous transitions (first phase) and then to use the subset construction (second phase). Here is the first phase. Cut ε-transitions $7 \xrightarrow{\varepsilon} 3$, $7 \xrightarrow{\varepsilon} 5$ and $8 \xrightarrow{\varepsilon} 11$, and duplicate on the source state the arc outgoing from the destination state of the transition (back-propagation rule):



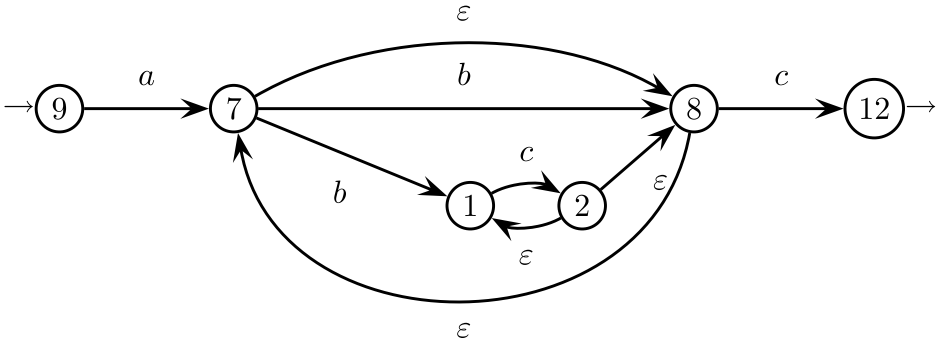
States 3, 5 and 11 become unreachable and can be removed, as follows:



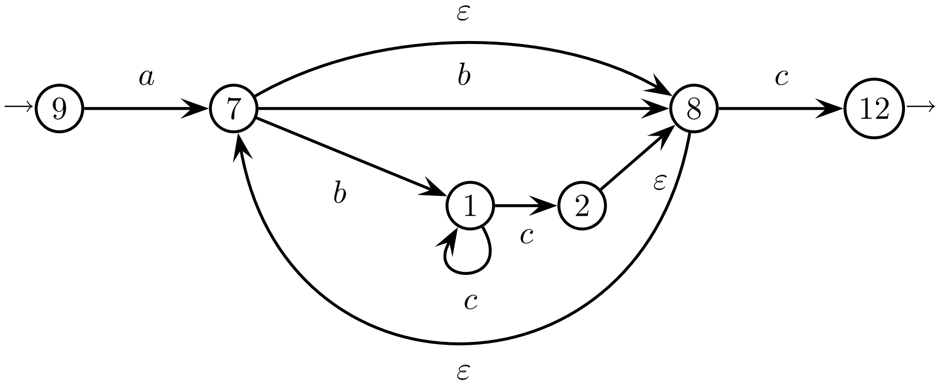
Now cut ε-transitions $10 \xrightarrow{\varepsilon} 7$, $4 \xrightarrow{\varepsilon} 1$ and $6 \xrightarrow{\varepsilon} 8$, and duplicate on the destination state the arc incoming into the source state of the transition (forward-propagation rule):



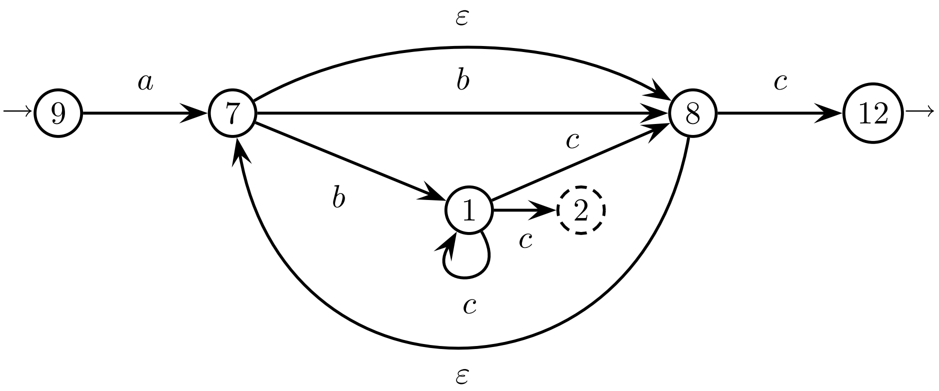
States 10, 4 and 6 become indefinite and can be removed, as follows:



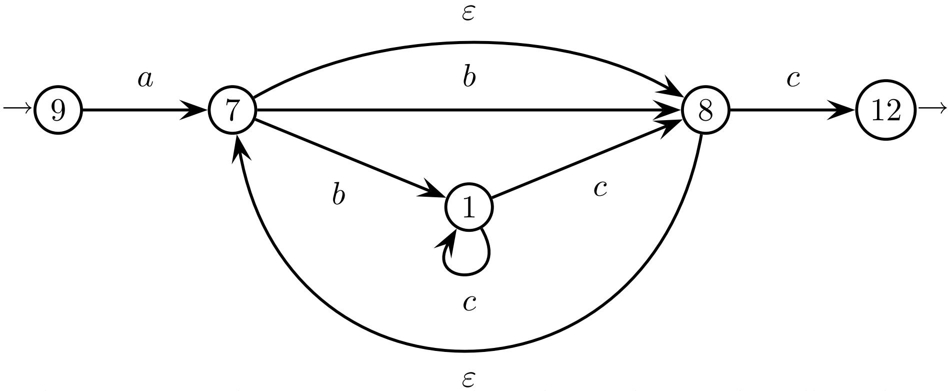
Now cut ε-transition $2 \xrightarrow{\varepsilon} 1$ and duplicate on the destination state the arc incoming into the source state of the transition (forward-propagation rule):



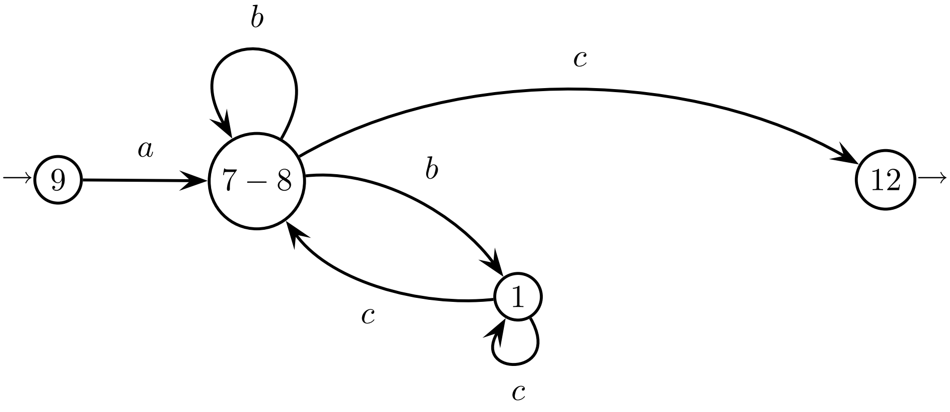
Now cut ε-transition $2 \xrightarrow{\varepsilon} 8$ and duplicate on the destination state the arc incoming into the source state of the transition (forward-propagation rule):



State 2 becomes indefinite and can be removed, as follows:

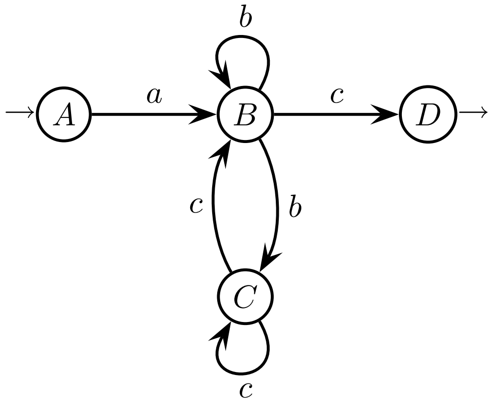


Now, notice that states 7 and 8 constitute an ε-loop, hence they can be collapsed into one state (temporarily named $7 - 8$ for clarity), as follows:



Now the automaton does not have any spontaneous transition left. However it is still indeterministic, as state $7 - 8$ has two outgoing arcs both labeled by letter b (one is a self-loop). The reader may have noticed that the forward and backward propagation rules for cutting spontaneous transitions have been interleaved, for the purpose of minimising at each step the number of arcs to duplicate, and that the ε-loop has been collapsed only at the end not to create a state associated with many transitions.

The second phase is the subset construction. States and transitions of the automaton can be renamed and redisplayed more conveniently as follows (see below on the left), and the successor table is written consequently (see below on the right):



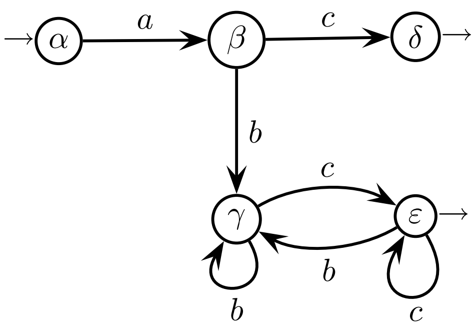
group	a	b	c	final?
A	B	–	–	no
B	–	$B\,C$	D	no
$B\,C$	–	$B\,C$	$B\,C\,D$	no
D	–	–	–	yes
$B\,C\,D$	–	$B\,C$	$B\,C\,D$	yes

successor table

States and transitions of the resulting deterministic automaton A' can be more conveniently renamed and redisplayed as follows:

state	a	b	c	final?
α	β	–	–	no
β	–	γ	δ	no
γ	–	γ	ε	no
δ	–	–	–	yes
ε	–	γ	ε	yes

state table



deterministic automaton A'

Therefore a deterministic automaton A' with five states has been obtained. It may be not in minimal form, anyway (see the next answer). □

First solution of (3)

Normally to minimise the deterministic automaton A' one should resort to the standard minimisation algorithm, based on the triangular implication table. Here however it is possible to conclude more speedily in an intuitive way. Remember that final states are always distinguishable from those that are not final. Then one has the following deductions:

- States δ and ε are distinguishable because they behave differently as for the error state.
- State α is distinguishable from both state β and γ, because α behaves differently from β and γ as for the error state.

- For states β and γ to be indistinguishable, it would be necessary that states δ and ε were indistinguishable as well. However indistinguishability $\delta \sim \varepsilon$ does not hold (see before), and therefore β and γ are distinguishable.

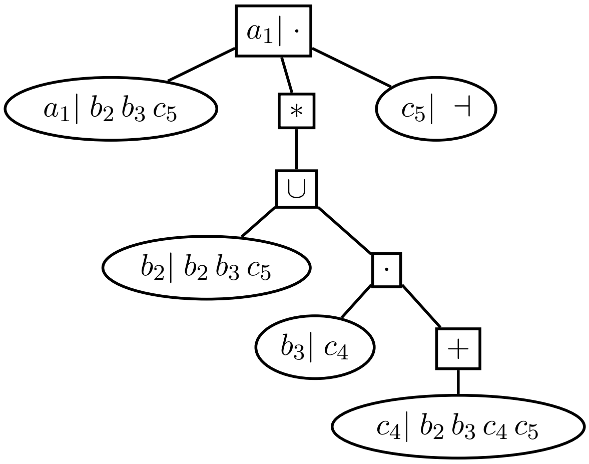
In conclusion all the states of automaton A' are distinguishable from one another. Therefore A' is already in minimal form and the requested minimal automaton A'' coincides with A'. □

Second solution of (1)

As an alternative, one can resort to the Berri-Seti algorithm for obtaining an indeterministic automaton A equivalent to regular expression R. Here is the structure tree of R labeled with start and follow sets at the root and for all generators, respectively (see below on the left), and the table with the start and all the follow sets (see below on the right):

structure tree of regular expression

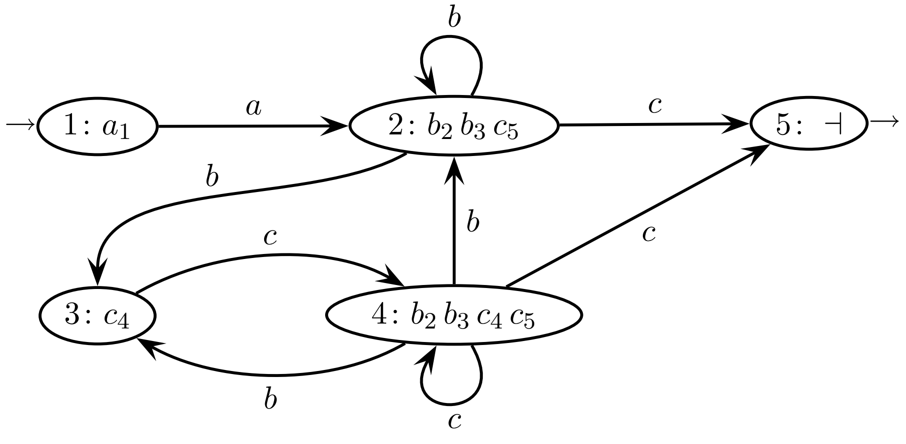
$$R = a_1 \ \cdot \ \left(b_2 \ \mid \ b_3 \ \cdot \ c_4^+ \right)^* \ \cdot \ c_5 \ \dashv$$


$a_1| \cdot$
$a_1| \ b_2 \, b_3 \, c_5$
$*$
$c_5| \ \dashv$
$\cup$
$b_2| \ b_2 \, b_3 \, c_5$
$\cdot$
$b_3| \ c_4$
$+$
$c_4| \ b_2 \, b_3 \, c_4 \, c_5$

start and follow sets

start	a_1
generator	follow
a_1	$b_2 \ b_3 \ c_5$
b_2	$b_2 \ b_3 \ c_5$
b_3	c_4
c_4	$b_2 \ b_3 \ c_4 \ c_5$
c_5	$\dashv$

The indeterministic automaton A is then the following:



The indeterministic automaton A is in reduced form and has five states, somewhat fewer than those obtained with the Thompson or modular construction. The automaton is indeterministic as it has multiple arcs with identical labels outgoing from the same state. □

Second solution of (2)

It is possible to determinise the above indeterministic automaton A directly by means of the subset construction (as there are not any spontaneous transitions), as follows:

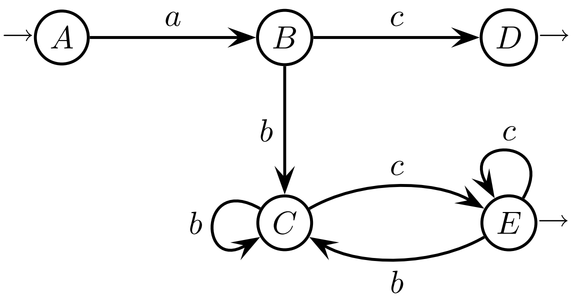
group	a	b	c	final?
1	2	–	–	no
2	–	2 3	5	no
2 3	–	2 3	4 5	no
5	–	–	–	yes
4 5	–	2 3	4 5	yes

successor table

state	a	b	c	final?
A	B	–	–	no
B	–	C	D	no
C	–	C	E	no
D	–	–	–	yes
E	–	C	E	yes

state table

On the left there is the successor table and on the right the state table, with more compact state names. Here is the state-transition graph of the deterministic automaton A':



The deterministic automaton is also in reduced form (since the successor table produces a reduced result if the original automaton is so), but may be not in minimal form, anyway. □

Second solution of (3)

It is relatively straightforward to conclude that the deterministic automaton A' is already in minimal form. Remember that a final state is always distinguishable from a state that is not final. Then one has the following deductions:

- States D and E are distinguishable because they behave differently as for the error state.
- State A is distinguishable from both state B and C, because it behaves differently from them as for the error state.
- For states B and C to be indistinguishable, it would be necessary that states D and E were indistinguishable as well. But equivalence $D \sim E$ does not hold (see before), hence B and C are distinguishable.

Therefore one concludes that automaton A' is already in minimal form and that the requested automaton A'' coincides with A'.

Of course, the automaton A'' here obtained coincides with that designed by means of the Thompson or modular construction before (the reader can check by himself), as the minimal form is unique. □

Observation

If the exercise had not requested explicitly to produce an indeterministic automaton A equivalent to regular expression R, one could have resorted to the McNaughton-Yamada algorithm to obtain directly a deterministic automaton equivalent to R. Again, the resulting automaton may be not in minimal form.

Observation

One may wish to notice that regular expression R can be simplified as follows:

$$R = a \ (\ b \ \mid \ b \, c^{+} \ )^{*} \ c = a \ (\ b \ (\ \varepsilon \ \mid \ c^{+}) \ )^{*} \ c = a \ (\ b \, c^{*} \ )^{*} \ c$$

which may be slightly simpler to deal with for obtaining an equivalent automaton.

Exercise 4 The two following regular expressions R_1 and R_2 are given:

$$R_1 = (\, (\, a \, b \,)^{*} \ c \,)^{*} \qquad\qquad R_2 = (\, c^{*} \ (\, a \, b \,)^{*} \,)^{*}$$

Answer the following questions:

1. Check intuitively whether the two expressions R_1 e R_2 are equivalent (if they are not show a string belonging to either one).
2. Check in an algorithmic way whether the two expressions R_1 and R_2 are equivalent.

Solution of (1)

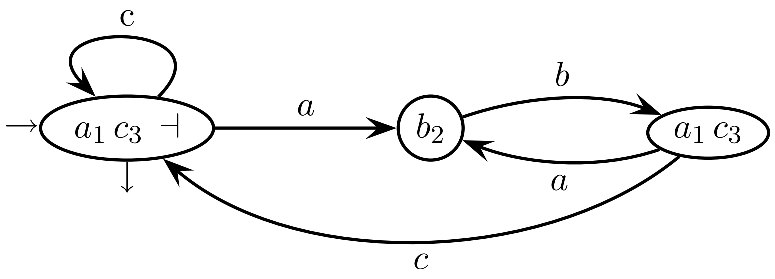
That the two regular expressions R_1 and R_2 are not equivalent, should be almost evident by intuition: the strings generated by R_1 (except ε) necessarily end with at least one letter c, while those generated by R_2 may end with one letter b (besides ending with c). For instance, string $a \, b$ is generated by R_2 but not by R_1. □

Solution of (2)

However here it is requested to provide an algorithmic proof of the inequivalence of expressions R_1 and R_2. One way to do so is to find the deterministic minimal recogniser automata equivalent to R_1 and R_2, and verify whether such automata are identical or not (as the minimal form is unique). Here the two deterministic automata are obtained by means of the McNaughton-Yamada algorithm.

Automaton of $R_1 = (\,(\,a_1\,b_2\,)^*\;c_3\,)^*\;\dashv$:

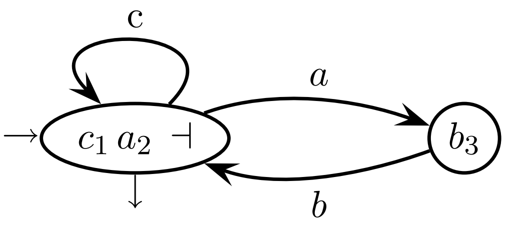
start	$a_1\,c_3\;\dashv$
generator	follow
a_1	b_2
b_2	$a_1\,c_3$
c_3	$a_1\,c_3\;\dashv$



Clearly the automaton is minimal as states b_2 and $a_1\,c_3$ are certainly distinguishable, because the latter has an outgoing arc labeled by c while the former does not, and the final state is obviously distinguishable from the other two states, which are not final.

Automaton of $R_2 = (\,c_1^*\;(\,a_2\,b_3\,)^*\,)^*\;\dashv$:

start	$c_1\,a_2\;\dashv$
generator	follow
c_1	$c_1\,a_2\;\dashv$
a_2	b_3
b_3	$c_1\,a_2\;\dashv$



Again, clearly the automaton is minimal as the two states are one final and the other one not, hence they are necessarily distinguishable.

Since the two deterministic minimal automata are different (as they have three and two states respectively), they are not equivalent and therefore the two regular expressions R_1 and R_2 are not equivalent either. □

Exercise 5 Consider the strings over alphabet $\Sigma = \{a, b, c\}$ and the two following constraints:

1. if the string contains two or more letters a, they are not adjacent

2. if the string contains one or more letters b, at least one of them is followed by at least one letter c, at whatever distance

Answer the following questions:

1. Write the two regular expressions (no matter if ambiguous) that generate the strings corresponding to the former and latter constraint, respectively (that is one expression for each constraint) and use only concatenation, union and star (or cross).

2. By applying an algorithmic method design the deterministic automaton that recognizes the language, the strings of which satisfy both constraints (not one as before).

3. Minimise the number of states of the previously designed deterministic automaton.

Solution of (1)

Here is regular expression R_1, not ambiguous, satisfying the former constraint:

$$R_1 = (b \mid c)^* \ \big(a\ (b \mid c)^+\big)^* \ (a \mid \varepsilon)$$

If there is a letter a in the string, expression R_1 mandatorily inserts soon after such letter a a new letter b or c, unless letter a is a the end of the string. The string can begin optionally by a sequence of letters b and c. In this way all the strings are generated, but those containing two adjacent letters a.

Here is regular expression R_2, strongly ambiguous, satisfying the latter constraint:

$$R_2 = (a \mid c)^* \ \mid \ \Sigma^*\, b\, \Sigma^*\, c\, \Sigma^*$$

For at least one instance of letter b in the string, expression R_2 necessarily inserts into the string a letter c at some distance (unless the string lacks completely letters b). Such a formulation is very simple but clearly is strongly ambiguous as well, as it does not specify which b will be followed by c (see below). For instance, string $b\,c\,b\,c$ is generated ambiguously by R_2. In fact, suppose to number as $\Sigma_1^*\, b_2\, \Sigma_3^*\, c_4\, \Sigma_5^*$, then clearly string $b\,c\,b\,c$ could be generated in three ways: $b_1\, c_1\, b_2\, \varepsilon_3\, c_4\, \varepsilon_5$, $\varepsilon_1\, b_2\, c_3\, b_3\, c_4\, \varepsilon_5$ or $\varepsilon_1\, b_2\, \varepsilon_3\, c_4\, b_5\, c_5$.

However, if one prefers a version that is not ambiguous, here it is:

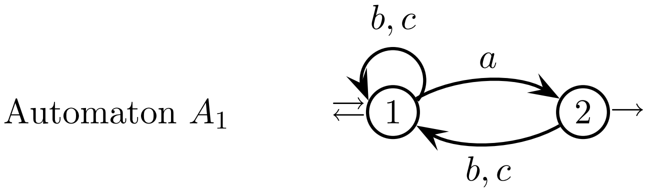
$$R_2' = (a \mid c)^* \ \big(b\ (a \mid b)^*\ c\ (a \mid b \mid c)^* \ \mid \ \varepsilon\big)$$

Expression R_2' specifies that the instance of letter b (if any) necessarily followed (at some distance) by a letter c, is $\cdots \overset{\downarrow}{b}\ (a \mid b)^*\ c \ldots$ (see the pointing dart), that is the leftmost one. This removes ambiguity but does not cause any loss of generality, as if some instance of b must be followed by c sooner or later, then also the leftmost instance of b must be so. □

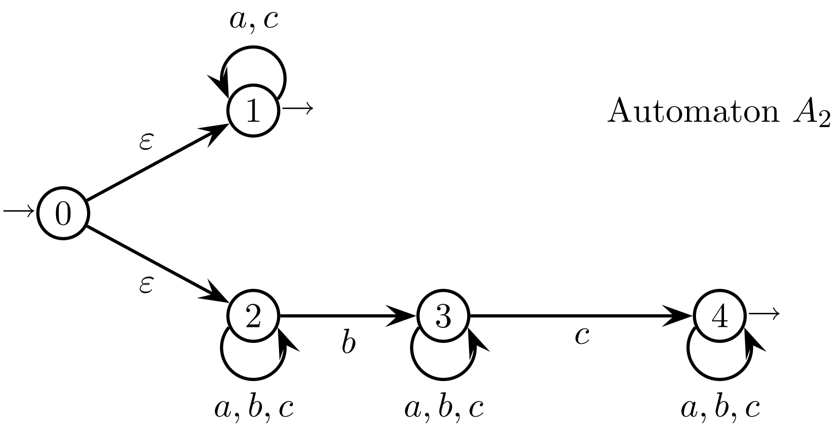
Solution of (2)

One can proceed first by designing the deterministic automata of languages L_1 and L_2, and then by computing the intersection language by means of the construction of the cartesian product of automata.

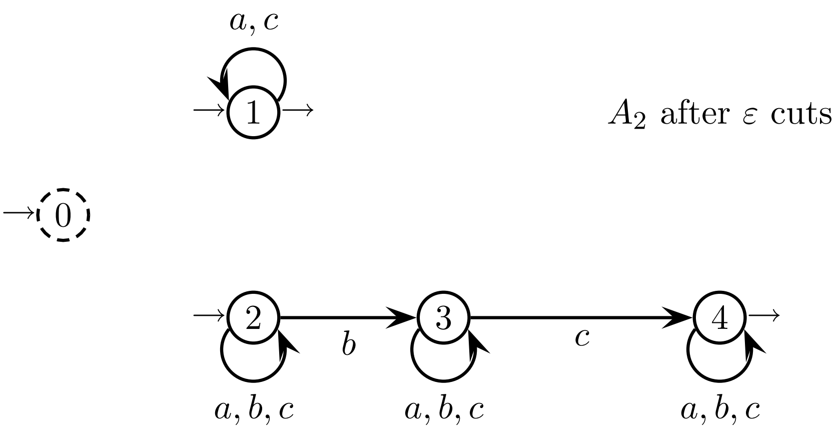
Automaton A_1 is very easy to design intuitively in deterministic form. It can be obtained soon by examining regular expression R_1. Here it is:



If one follows the same approach adopted for obtaining regular expression R_2, the resulting automaton A_2 is indeterministic (remember that R_2 is ambiguous) and for this reason easier and more natural to design. Here it is, derived from R_2 in the natural way, that is by applying the Thompson or modular construction (with some simplifications):



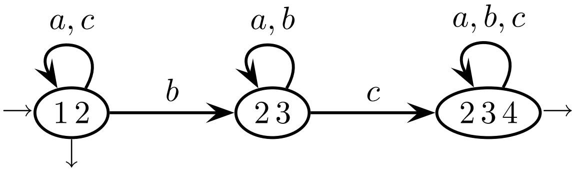
Automaton A_2 can however be easily determinised first by cutting spontaneous transitions (ε-transition) and then by applying the subset construction:



Here spontaneous transitions $0 \xrightarrow{\varepsilon} 1$ and $0 \xrightarrow{\varepsilon} 2$ are cut and the arcs incoming into the source node of the transition are duplicated on the destination node. This causes both states 1 and 2 to become initial, while state 0 remains indefinite (unconnected to any final node) and hence can be removed (in the graph it is dashed). Here is the subset construction (the initial state group is 1 2):

group	a	b	c	final ?
1 2	1 2	2 3	1 2	yes
2 3	2 3	2 3	2 3 4	no
2 3 4	2 3 4	2 3 4	2 3 4	yes

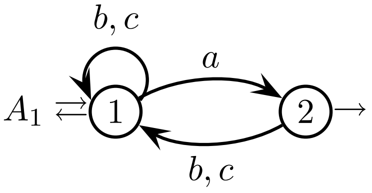
successor table

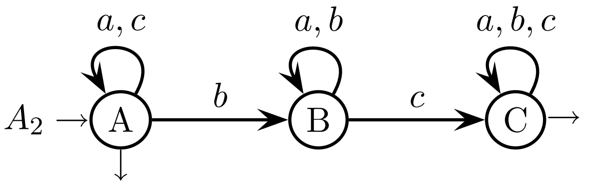


deterministic form of automaton A_3

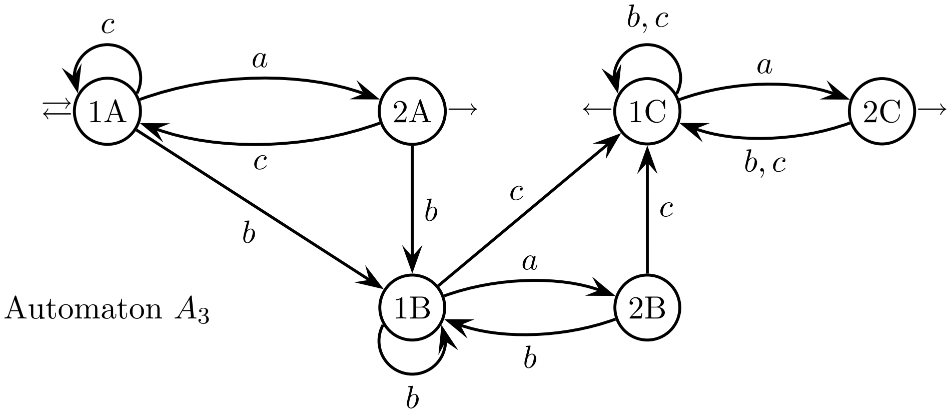
On the right side there is the state-transition graph of the determinised automaton. By the way, it is precisely what is obtained by using regular expression R'_2 as a model to start from.

Now the cartesian product of automata A_1 and A_2 (in determinised form) can be computed. First it is convenient redraw A_1 and rename the states of A_2, as follows:





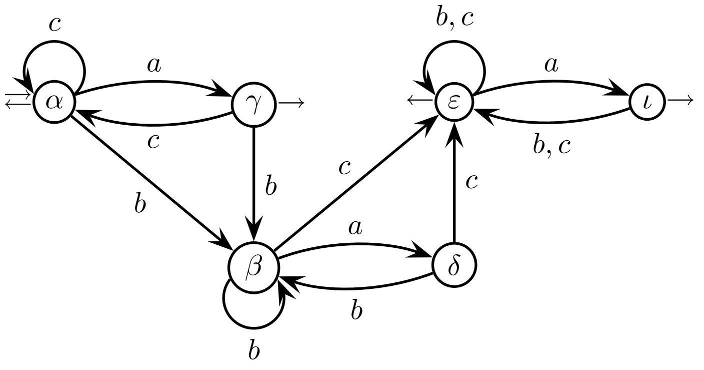
The cartesian product automaton $A_3 = A_1 \times A_2$ is then the following:



Notice that all the six product states of $A_3 = A_1 \times A_2$ happen to be useful, that is both reachable and definite. As both automata A_1 and A_2 are deterministic (one would suffice), A_3 is certainly deterministic as well. It may be not in minimal form, anyway. □

Solution of (3)

In order to minimise the deterministic automaton A_3, first it is convenient to rename more shortly the states and to write the state-transition table of A_3, as follows:



state	a	b	c	final ?
α	γ	–	α	yes
β	δ	β	ε	no
γ	–	β	α	yes
δ	–	β	ε	no
ε	ι	ε	ε	yes
ι	–	ε	ε	yes

automaton A_3

state table

In principle to mimimise one should apply the classical state reduction costruction based on the triangular implication table, but sometimes it is possible to come to a conclusion more quickly, as it happens here. Remember that a final state is always distinguishable from a state that is not final. Thus the following considerations apply:

- of the states that are not final (β and δ), state β is distinguishable form state δ as for input letter a the latter goes into the error state while the former does not
- of the final states (α γ ε and ι), only states γ and ι might be indistinguishable as the other pairs go into the error state for different input letters, but the equivalence $\gamma \sim \iota$ would hold if and only if both equivalences $\beta \sim \varepsilon$ and $\alpha \sim \varepsilon$ held, which by transitivity would imply equivalence $\alpha \sim \beta$, which in turn is impossible for α is final while β is not, and hence γ and ι are distinguishable

Therefore there are not any pairs of indistinguishable states and the conclusion is that automaton A_3 is already in minimal form. □

Exercise 6 Language L over alphabet $\{a, b, c\}$ is defined by means of the following three conditions (every string of L must satisfy all of them):

1. phrases start with letter a

   and

2. phrases end with letter c

   and

3. phrases may not contain any of the following four two-letter factors:

   $b\,a$ $\quad b\,b \quad c\,a \quad c\,c$

For example, the following string:

$$a\ b\ c\ b\ c$$

is a phrase of L, while the string:

$$a \quad \underbrace{b\ a}_{\text{forbidden}} \quad b\ c$$

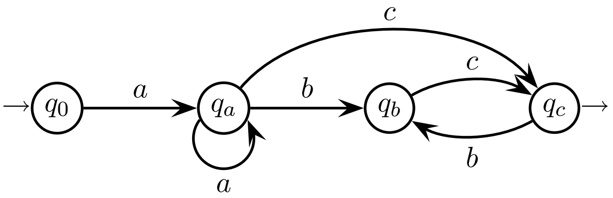
is invalid (and is not a phrase of L), because the middle factor is not admittable.

Answer the following questions:

1. Write a regular expression of language L by using only union, concatenation and star (or cross) operators.
2. Define language $\overline{L}$, the complement of language L, by modifying suitably the three conditions (1, 2, 3) above.

First solution of (1)

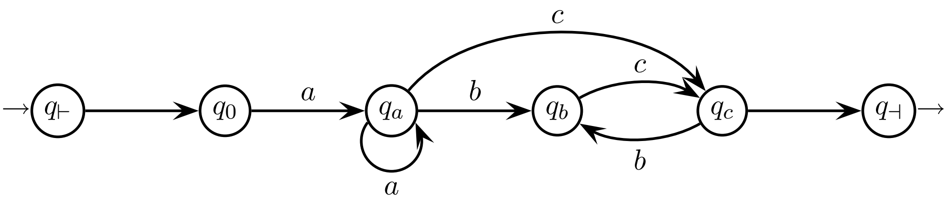
In order to obtain the regular expression of language L, first one can construct the automaton of L and then derive from it the expression. As language L belongs to the family known as local languages, the automaton has states labeled by alphabet letters, plus one initial state. Such an automaton need only record the last read letter and check whether the input string begins and ends with the letters prescribed by conditions (1, 2). Here is the automaton:



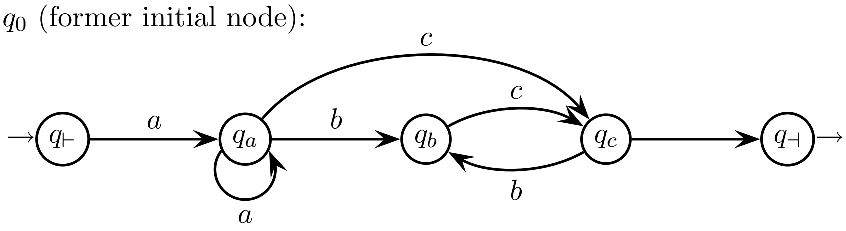
State names (i.e. q_a, q_b and q_c) are in one-to-one correspondence with alphabet letters (but the initial state q_0). Outgoing arcs represent initial and final characters, and the admittable pairs of adjacent characters.

In order to find a regular expression R equivalent to the automaton (and containing only union, concatenation and star or cross operators), one can for instance resort to the Brozozowski algorithm, also called node elimination algorithm. Here it is ($q_{\vdash}$ and $q_{\dashv}$ are the additional initial and final nodes required by the algorithm respectively):

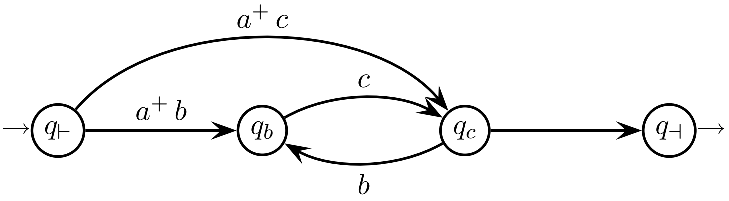
Preparation of the automaton (add unique initial and final nodes):



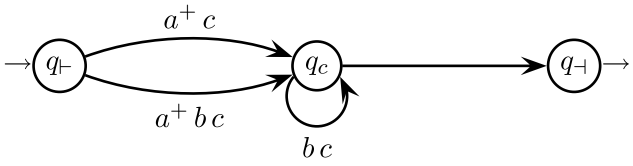
Remove node q_0 (former initial node):



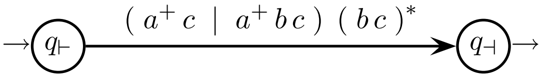
Remove node q_a:



Remove node q_b:



Remove node q_c:



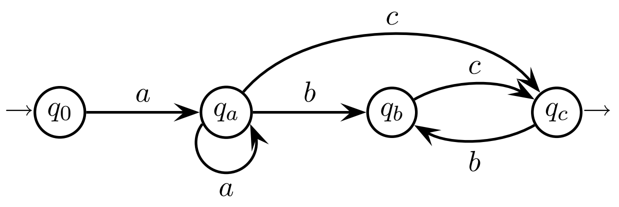
Finally the following regular expression R is obtained:

$$\begin{aligned} R &= (a^+ c \mid a^+ b c) (b c)^* \\ &= a^+ c (b c)^* \mid a^+ b c (b c)^* \\ &= a^+ (c \mid b c) (b c)^* \\ &= a^+ (c b)^* c \mid a^+ (b c)^+ \end{aligned}$$

Here such an expression R is displayed in four different but equivalent forms. The fourth form is obtained from the second one by observing that the identity $c (b c)^* = (c b)^* c$ holds true. □

Second solution of (1)

As an alternative, one can obtain a regular expression R' that generates language L, by means of the method of linguistic linear equations. The first step is the same as before and consists of constructing the finite state automaton of language L. Here it is again:



Then one converts such an automaton into a system of right-linear linguistic equations. Let L_0, L_a, L_b and L_c be the languages recognised by the automaton when assuming state q_0, q_a, q_b and q_c as initial, respectively. The system of equations is then the following (on the left), along with the solution process (follow the chain of implications "⇒" and read the captions):

$$\left\{\begin{array}{lcl} L_0 & = & a\,L_a \\ L_a & = & a\,L_a \mid b\,L_b \mid c\,L_c \\ L_b & = & c\,L_c \\ L_c & = & b\,L_b \mid \varepsilon \end{array}\right. \quad \Rightarrow \quad \left\{\begin{array}{lcl} L_0 & = & a\,L_a \\ L_a & = & a\,L_a \mid b\,L_b \mid c\,L_c \\ L_b & = & c\,(b\,L_b \mid \varepsilon) = \\ & = & c\,b\,L_b \mid c = \\ & = & (c\,b)^*\ c \qquad \text{(Arden rule)} \\ L_c & = & b\,L_b \mid \varepsilon \end{array}\right.$$

equation system

replace eq. L_c into eq. L_b and use Arden rule

$$\left\{\begin{array}{lcl} L_0 & = & a\,L_a \\ L_a & = & a\,L_a \mid b\,L_b \mid c\,L_c \\ L_b & = & (c\,b)^*\ c \\ L_c & = & b\ (c\,b)^*\ c \mid \varepsilon = \\ & = & (b\,c)^+ \mid \varepsilon = \\ & = & (b\,c)^* \end{array}\right. \quad \Rightarrow \quad \left\{\begin{array}{lcl} L_0 & = & a\,L_a \\ L_a & = & a\,L_a \mid b\ (c\,b)^*\ c \mid c\ (b\,c)^* = \\ & = & a\,L_a \mid (b\,c)^+ \mid c\ (b\,c)^* = \\ & = & a^*\ \left((b\,c)^+ \mid c\ (b\,c)^*\right) \qquad \text{(Arden rule)} \\ L_b & = & (c\,b)^*\ c \\ L_c & = & (b\,c)^* \end{array}\right.$$

replace eq. L_b into eq. L_c

replace eq.s L_b and L_c into eq. L_a,
and use Arden rule

$$\left\{\begin{array}{lcl} L_0 & = & a\,a^*\ \left((b\,c)^+ \mid c\ (b\,c)^*\right) = \\ & = & a^+\ \left((b\,c)^+ \mid c\ (b\,c)^*\right) \\ L_a & = & a^*\ \left((b\,c)^+ \mid c\ (b\,c)^*\right) \\ L_b & = & (c\,b)^*\ c \\ L_c & = & (b\,c)^* \end{array}\right. \quad \Rightarrow \quad \begin{array}{lcl} R' & = & L_0 = a^+\ \left((b\,c)^+ \mid c\ (b\,c)^*\right) = \\ & = & a^+\ (b\,c)^+ \mid a^+\ c\ (b\,c)^* \end{array}$$

replace eq. L_a into eq. L_0

regular expression of language L

To solve the system above, it is sufficient to substitute equations and use Arden rule to eliminate recursion. If possible, intermediate expressions are simplified by resorting to known properties of regular operators: for instance the obvious property that $b\,(c\,b)^*\ c = b\,c\,b\,c\,b\ \ldots\ c = (b\ c)^+$, and so on. Thus one obtains eventually regular expression R', which is the solution of equation L_0 corresponding to the original initial state of the automaton. Expression R' is of course equivalent to the expression R obtained before; just notice that $c\ (b\ c)^* = (c\ b)^*\ c$ holds. □

Solution of (2)

Remember that L is a local language, defined as the logical conjunction (and) of the three conditions (1, 2, 3). Therefore the complement language $\overline{L}$ is defined as the logical disjunction (or) of the same three conditions (1, 2, 3) in negated form (according to De Morgan theorem). One then obtains the following:

- a phrase of $\overline{L}$ *does not begin* with letter a (i.e. begins with b or c)

  or

- a phrase of $\overline{L}$ *does not end* with letter c (i.e. ends with a or b)

  or

- a phrase of $\overline{L}$ *does not happen* not to contain some (i.e. has to contain at least one) of the following factors:

$$b\,a \qquad\qquad b\,b \qquad\qquad c\,a \qquad\qquad c\,c$$

Moreover, it is evident that the complement language $\overline{L}$ contains the empty string ε, as language L is ε-free. Pay attention to that now the three conditions above are disjoint, not conjoint, that is it suffices that the string satisfies one of them to belong to $\overline{L}$ (of course the string may satisfy two or even all the three of them). □

Observation

One may wish to explore the question a little more in depth. Since language L is of the local type, a generalised regular expression R' (containing also the operator of set difference or those of set intersection and set complement) that generates L is the following:

$$R' = a\Sigma^* c - (\Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^*) = a\Sigma^* c \cap \overline{\Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^*}$$

Clearly such an expression R' is equivalent to the regular expression R previously found, although it is formulated differently. An ordinary regular expression R'' (with only union, concatenation and star or cross) that generates the complement language $\overline{L}$ is then the following:

$$\begin{aligned}
R'' &= \overline{R} \\
&= \overline{a\Sigma^* c \ \cap \ \overline{\Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^*}} \\
&= \overline{a\Sigma^* c} \ \cup \ \overline{\overline{\Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^*}} \\
&= \overline{(a\Sigma^* \ \cap \ \Sigma^* c)} \ \cup \ \Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^* \\
&= \overline{a\Sigma^*} \ \cup \ \overline{\Sigma^* c} \ \cup \ \Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^* \\
&= \varepsilon \ \cup \ \{b, c\} \Sigma^* \ \cup \ \varepsilon \ \cup \ \Sigma^* \{a, b\} \ \cup \Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^* \\
&= \varepsilon \ \cup \ \{b, c\} \Sigma^* \ \cup \ \Sigma^* \{a, b\} \ \cup \Sigma^* \{b\,a, b\,b, c\,a, c\,c\} \Sigma^*
\end{aligned}$$

Clearly it holds $\overline{L} = L(R'')$. As it seems reasonable, expression R'' has a structure conformant to the three conditions above (plus the empty string ε), where the operator of union represents logical disjunction (or). If one prefers to use the usual symbology, regular expression R'' can be rewritten as follows:

$$\begin{aligned}
R'' &= \varepsilon \mid (\, b \mid c \,) \, (\, a \mid b \mid c \,)^* \mid (\, a \mid b \mid c \,)^* \, (\, a \mid b \,) \\
&\mid \ (\, a \mid b \mid c \,)^* \, (\, b a \mid b b \mid c a \mid c c \,) \, (\, a \mid b \mid c \,)^*
\end{aligned}$$

Observation

One could obtain regular expression R'' also in the following way: first take the deterministic automaton of L that has been found before and construct the complement automaton thereof, which recognises the complemet language $\overline{L}$, then derive from the latter automaton the equivalent regular expression, for instance by means of the Brozozowski or node elimination algorithm again. The reader is left the task of completing this alternative approach.

Exercise 7 Consider the following regular expression R over alphabet $\{a, b, c\}$:

$$R = (a \mid \varepsilon)\ (b \mid c)^*\ (b\,c \mid a\,b)^*$$

Answer the following questions:

1. List all the strings of length ≤ 2 generated by expression R and say which ambiguity degree they have (that is in how many different ways each string is generated by R).
2. Design the indeterministic automaton A of expression R (choose the method to follow).
3. Design the deterministic automaton A' of expression R, obtained from the previous one by means of the subset construction.
4. If necessary, design the minimal automaton A'' of expression R.

Solution of (1)

The strings of length ≤ 2 generated by regular expression R are the following:

$$\varepsilon \qquad a \qquad b \qquad c \qquad a\,b \qquad a\,c \qquad b\,c \qquad b\,b \qquad c\,b \qquad c\,c$$

Of these strings, $a\,b$ and $b\,c$ have ambiguity degree 2, that is they are generated by expression R in two different ways. In fact, number the generators in the expression R as follows:

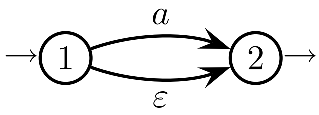
$$R = (a_1 \mid \varepsilon_2)\ (b_3 \mid c_4)^*\ (b_5\,c_6 \mid a_7\,b_8)^*$$

Then expression R can generate both $a_1\,b_3$ and $a_7\,b_8$, and both $b_3\,c_4$ and $b_5\,c_6$. Therefore strings $a\,b$ and $b\,c$ have ambiguity degree 2. □

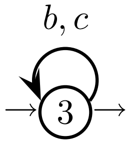
Solution of (2)

Since an indeterministic automaton A equivalent to regular expression R is wanted, one can build it easily by means of the Thompson or modular construction. Here it is, step by step but with some shortcuts to speed it up:

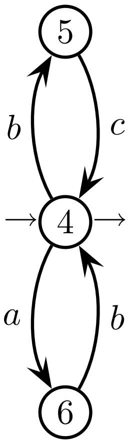
$(a \mid \varepsilon)$



$(b \mid c)^*$

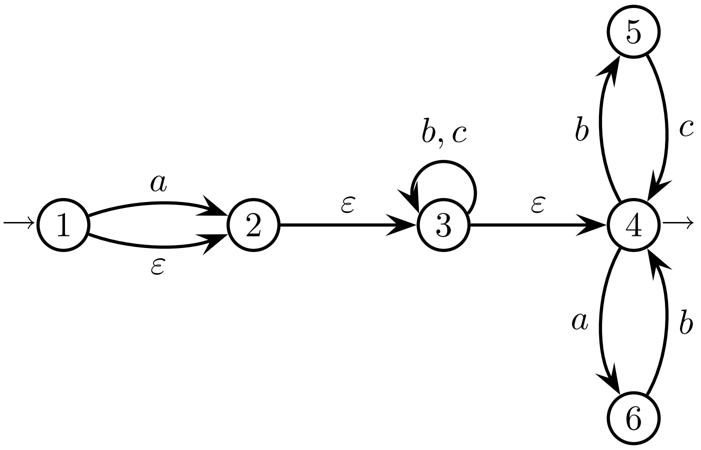


$(b\ c \mid a\ b)^*$



Then the complete (indeterministic) automaton A is obtained by concatenating the three sub-automata above by means of spontaneous transitions (ε-transition), as follows:

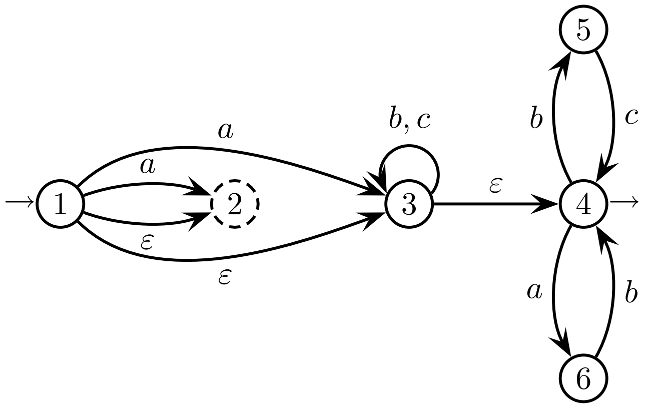
Automaton A



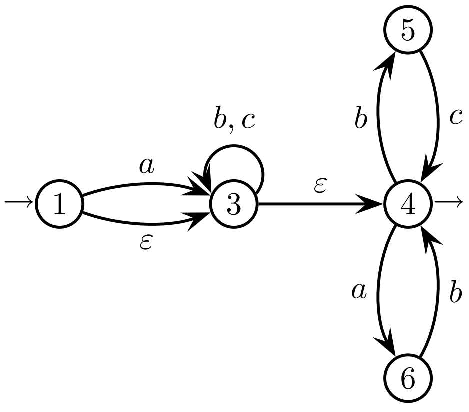
Automaton A is indeterministic mainly because it has a few spontaneous transitions (ε-transitions). □

Solution of (3)

To determinise automaton A, one must cut spontaneous transitions and apply the subset construction. First cut ε-transition $2 \xrightarrow{\varepsilon} 3$ and replicate on destination state 3 the arcs ingoing into source state 2, as follows on the left:



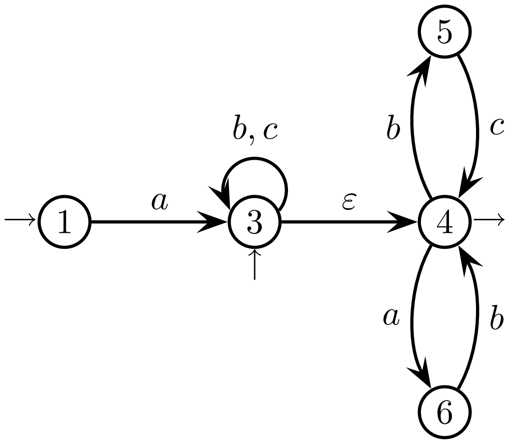
cut transition $2 \xrightarrow{\varepsilon} 3$



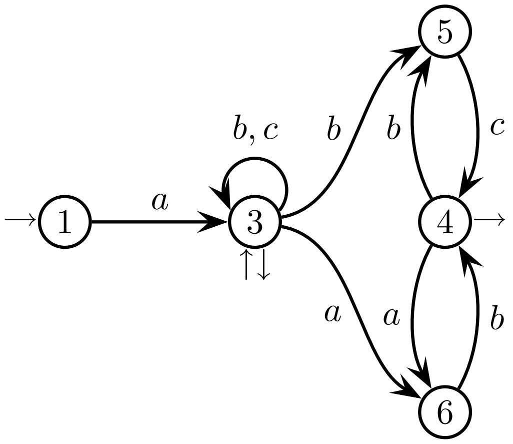
remove undefined state 2

One sees soon that state 2 becomes indefinite (no path from state 2 to final state 4) and that therefore it can be removed; see the figure above on the right.

Next one can cut ε-transition $1 \xrightarrow{\varepsilon} 3$ and replicate on destination state 3 the arcs incoming into source state 1 (namely the initial marker dart), as follows on the left.



cut transition $1 \xrightarrow{\varepsilon} 2$



cut transition $3 \xrightarrow{\varepsilon} 4$

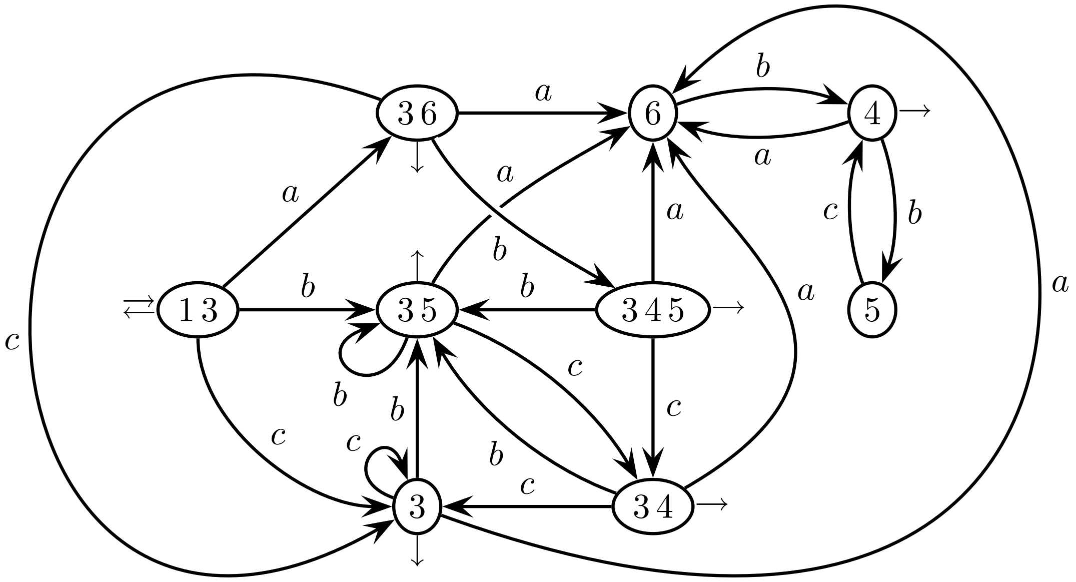
Finally one can cut ε-transition $3 \rightarrow 4$ and replicate on source state 3 the arcs outgoing from destination state 4 (including the final marker dart); see the figure above on the right.

Now there are not any spontaneous transitions left. However the automaton is still indeterministic, as there are two multiple initial states (1 and 3) and duplicate arcs (on state 3). The subset construction has to be applied, as follows (the initial state group is 1 3):

group	a	b	c	final ?
1 3	3 6	3 5	3	yes
3 6	6	3 4 5	3	yes
3 5	6	3 5	3 4	yes
3	6	3 5	3	yes
6	–	4	–	no
3 4 5	6	3 5	3 4	yes
3 4	6	3 5	3	yes
4	6	5	–	yes
5	–	–	4	no

successor table

The subset construction produces a deterministic automaton A' with nine states. Here is the state-transition graph:



deterministic automaton A' - state-transition graph

The error state is not shown explicitly, but of course is present anyway. The obtained deterministic automaton A' is not necessarily in minimal form. □

Solution of (4)

It is pretty evident that the obtained deterministic automaton A' is not in minimal form, as the state table contains identical rows. First it is convenient to rename states in a more compact way, as follows on the left:

state	a	b	c	final ?
A	B	C	D	yes
B	E	F	D	yes
C	E	C	G	yes
D	E	C	D	yes
E	–	H	–	no
F	E	C	G	yes
G	E	C	D	yes
H	E	K	–	yes
K	–	–	H	no

state table

state	a	b	c	final ?
A	B	C	D	yes
B	E	C	D	yes
C	E	C	D	yes
D	E	C	D	yes
E	–	H	–	no
H	E	K	–	yes
K	–	–	H	no

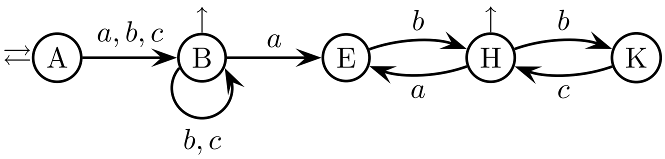
state table

Now notice that rows C, F and D, G are identical, hence states C, F and D, G are indistinguishable, and the equivalences C $\sim$ F and D $\sim$ G hold. One can then remove rows F and G, and replace F and G by C and D, respectively; see the figure above on the right.

Removal and replacement cause other rows to become identical, namely rows B, C and D. Such states are therefore indistinguishable and constitute an equivalence class. One can then remove rows C and D, and replace both C and D by B, as follows on the left:

state	a	b	c	final ?
A	B	B	B	yes
B	E	B	B	yes
E	–	H	–	no
H	E	K	–	yes
K	–	–	H	no

state table



minimal automaton A''

Now, to identify other potential equivalences one should resort to the standard minimisation algorithm based on the triangular implication table. Here however it is possible to conclude more quickly. Remember that final states are always distinguishable from the states that are not final. Notice also that the automaton is in reduced form (as all the states are useful). Then the following reasoning applies:

- States E and K are distinguishable, as they behave differently as for the error state.
- State H is distinguishable from both states A and B, as with letter c state H goes into the error state while states A and B do not.
- State A is distinguishable from state B as their equivalence would imply that of states B and E, which actually does not hold.

Therefore all the states A, B, E, H and K are distinguishable. Minimisation ends here and the state-transition graph of the minimal automaton A'' is shown above on the right. □

Observation

Alternatively, one could minimise the deterministic automaton A' directly by means of the standard algorithm based on the triangular implication table. Here it is, on the left:

B	BE CF							
C	BE DG	CF DG						
D	BE	CF	DG					
E	×	×	×	×				
F	BE DG	CF DG	∼	DG	×			
G	BE	CF	DG	∼	×	DG		
H	×	×	×	×	×	×	×	
K	×	×	×	×	×	×	×	×
	A	B	C	D	E	F	G	H

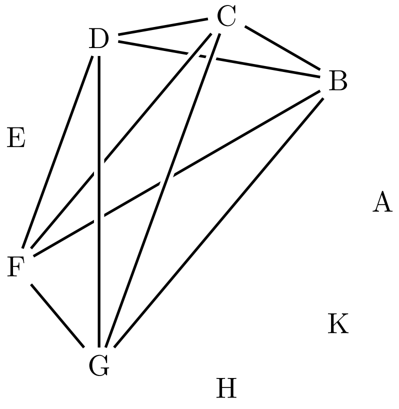
implication table

B	~~BE~~ ∼							
C	~~BE~~ ∼	∼ ∼						
D	~~BE~~	∼	∼					
E	×	×	×	×				
F	~~BE~~ ∼	∼ ∼	∼	∼	×			
G	~~BE~~	∼	∼	∼	×	∼		
H	×	×	×	×	×	×	×	
K	×	×	×	×	×	×	×	×
	A	B	C	D	E	F	G	H

Since there are immediate distinguishable and indistinguishable state pairs, propagation is necessary. In the table on the right above, distinguishable state pairs are barred through and coloured in red, while the indistinguishable ones are replaced by $\sim$. Remember that for a pair of states to be indistinguishable, all the implications it has must be. Then one obtains the following final table, on the left:

B	×							
C	×	∼						
D	×	∼	∼					
E	×	×	×	×				
F	×	∼	∼	∼	×			
G	×	∼	∼	∼	×	∼		
H	×	×	×	×	×	×	×	
K	×	×	×	×	×	×	×	×
	A	B	C	D	E	F	G	H

final implication table

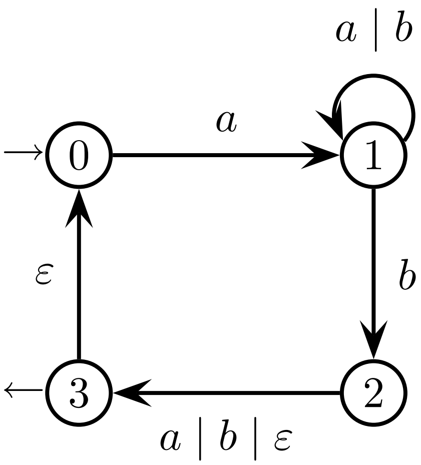


equivalence relation

The table does not contain circular implications and is fully solved. The equivalence relation is shown on the right above. The equivalence classes are $\{A\}$, $\{B, C, D, F, G\}$, $\{E\}$, $\{H\}$ and $\{K\}$, which of course is the same result as before.

Exercise 8 Consider the following finite state automaton A:

Automaton A



Answer the following questions:

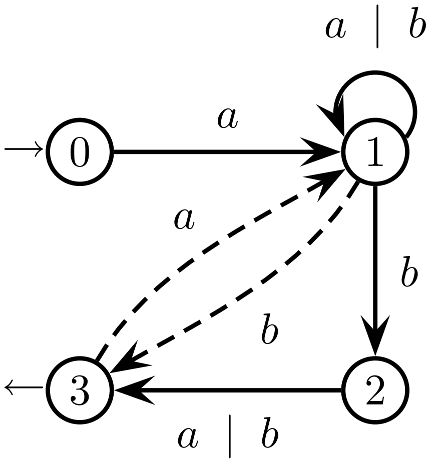
1. Design a deterministic automaton A' equivalent to A (use a method of choice).

2. Minimize the number of states of the deterministic automaton A' obtained before.

Solution of (1)

Notice that automaton A is in reduced form (every state is both reachable and defined). In order to determinise A one had better start cutting spontaneous transitions (ε-transition). It is convenient first to cut arc $3 \xrightarrow{\varepsilon} 0$ and duplicate from states 0 to 3 the arc a outgoing from 0, and then to cut arc $2 \xrightarrow{\varepsilon} 3$ and duplicate from states 2 to 3 the arc b incoming into 2 (of course without removing the original arcs a and b). Such cuts minimise the number of arcs to duplicate and do not introduce any new initial or final states. In this way one obtains the following automaton A', which is still indeterministic (state 1 has three outgoing arcs labeled with letter b):

Automaton A'

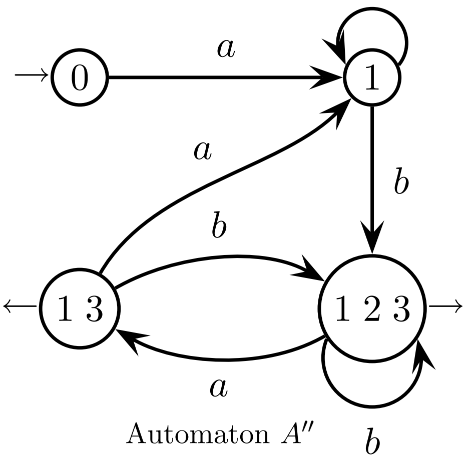


The two dashed arcs labeled with letters a and b are the replicated ones and their function is to reconstruct the labeled paths that are interrupted on cutting the two spontaneous transitions.

Then one can obtain a deterministic automaton A'' equivalent to automaton A' by means of the classical subset construction (carried out with the successor table algorithm):

group	a	b	**final ?**
0	1	–	no
1	1	1 2 3	no
1 2 3	1 3	1 2 3	yes
1 3	1	1 2 3	yes
–	–	–	no

successor table



Automaton A''

The subset construction leads to the deterministic automaton A'' drawn on the right, which still has four states (as usual in drawing deterministic automata the error state is omitted).

The deterministic automaton A'' is in reduced form (all the states are both reachable and definite). In fact if the original indeterministic automaton is in reduced form, the subset construction carried out by means of the successor table algorithm necessarily produces a reduced deterministic automaton. Notice that the automaton may be not in minimal form. □

Solution of (2)

In general to minimise the number of states of a deterministic automaton one should use the classical state minimisation algorithm, based on the triangular implication table. Sometimes however one can reach a conclusion more quickly, as it is here the case.

As a final state is always distinguishable from state that is not final, to check whether the deterministic automaton A'' is in minimal form it suffices to examine pairs of states which are both final or not final. Then one obtains the following results:

- states 0 and 1 are distinguishable, as from 1, but not from 0, there is an outgoing arc labeled with letter b
- states 1 2 3 and 1 3 are distinguishable, as from 1 2 3 the arc labeled with letter a goes into a final state, while from 1 3 the arc labeled in the same way goes into a state that is not final

Therefore the deterministic automaton A'' is already in minimal form and, as verified before, is in reduced form as well. This also implies that A'' is unique. In conclusion it is definitely impossible to further reduce the number of states of A'' under four. □

Observation

Remember that if determinism is not required, there may exist an indeterministic automaton equivalent to automaton A with fewer than four states. The reader may wish to investigate by himself whether such an automaton exists (existence is not granted).

Exercise 9 Given a deterministic finite state automaton M that recognizes language $L(M)$, consider the following constructions (orderly listed):

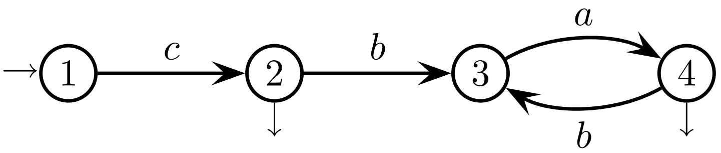
i. transform M into an automaton N_R that recognizes the mirror language $L(M)^R$

ii. transform N_R into a deterministic automaton M_R equivalent to N_R

iii. transform M_R into an automaton M_{RR} that recognizes the mirror language $L(M_R)^R$

Answer the following questions:

1. Start from the automaton M shown below, perform the above mentioned constructions (**i**), (**ii**) and (**iii**) (in the specified order), and draw the state-transition graphs of the requested automata N_R, M_R and M_{RR}.

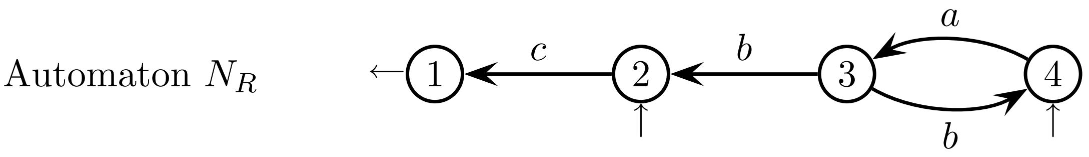


2. Compare the two automata M and M_{RR}, explain what makes them different from each other and why.

Solution of (1)

Automaton M recognises lists (not empty) of type $c\ (b\, a)^*$: such strings always begin with one element c and (possibly) continue only with elements a; the elements (c or a) are always separated by exactly one letter b. The automata constructed after each transformation are listed as follows.

Here is the (indeterministic) automaton N_R, obtained as the mirror image of automaton M (it has the two initial states 2 and 4):



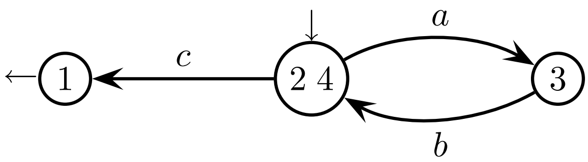
Such an automaton is indeterministic because of state 3, which has two outgoing arcs both labeled with the same letter b, and because of the two initial states 2 and 4.

Here is the deterministic automaton M_R (equivalent to N_R), which is obtained from N_R by applying the subset construction. Since there are multiple initial states the table of state groups begins with the subset of initial states $\{2, 4\}$, which constitutes the unique initial state of the deterministic resulting automaton M_R:

group	a	b	c	**final ?**
2 4	3	–	1	no
3	–	2 4	–	no
1	–	–	–	yes
–	–	–	–	no

successor table

Automaton M_R

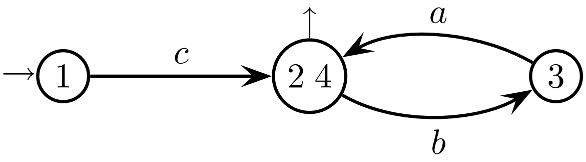


state-transition graph

On the right side there is the state-transition graph of automaton M_R, where as usual the error state is omitted.

Finally here is automaton M_{RR}, obtained as the mirror image of automaton M_R:

Automaton M_{RR}



The deterministic automaton M_{RR} is in reduced form (every state is reachable from state 1 and reaches final state 2 4), and it is easy to verify that it is minimal as well: at the best state 1 and 3 might be indistinguishable, but this is false as they do not have outgoing arcs labeled in the same way. □

Solution of (2)

Clearly the language recognised by automaton M_{RR} is nothing else but the language L itself, as it holds $(L^R)^R = L$. Therefore automaton M_{RR} is equivalent to the original automaton M and differs from M because of the lower number of states. In fact, it will be proved soon that M_{RR} is necessarily in minimal form.

In other words, transformations (i), (ii) and (iii) constitute a state minimisation procedure, as one can easily see from the following reasoning. Two states p and q of automaton M are indistinguishable if and only if the two sets of strings y that label the paths connecting p and q to the final states, respectively, are equal. Consider a label string y of length ≥ 1 such that:

$$p \xrightarrow{y} f \qquad\qquad q \xrightarrow{y} g$$

where f and g are final states (no matter wether different or coincident). In the determinised mirror automaton M_R, the mirrored string y^R labels a path from the initial state (necessaiily unique) to a subset $\{\ldots, p, \ldots, q, \ldots\}$, surely built by the determinisation algorithm, containing both states p and q (besides possibly other states).

When automaton M_{RR} is obtained as the mirror image of automaton M_R (by simply flipping the direction of arcs and swapping initial and final states), states do not change, hence the indistinguishable states p and q happen to be grouped together in the subset $\{\ldots, p, \ldots, q, \ldots\}$, and therefore the result is the same as what would have been obtained with the classical state minimisation algorithm (based on the triangular implication table). □

Observation

There are at least three well known state minimisation algorithms: the standard one, credited to Nerode and based on the triangular implication table; that seen above, credited to Brozozowski; and a third one, credited to Hopcroft.

The standard or Nerode state minimisation algorithm has a worst case quadratic time complexity, i.e. $O\left(n^2\right)$ where n is the number of states of the original automaton to minimise.

The Brozozowski state minimisation algorithm consists of applying to the automaton first mirror, second determinisation, third again mirror and fourth again determinisation (unless the final automaton is already deterministic). Such an algorithm is inefficient anyway, because the intermediate and final automata obtained after determinisation may be not in reduced form and exhibit an exponential explosion of the number of states. One need check whether such automata have unuseful states and, if so, one need put them in reduced form, which in the worst case may take an exponential time with respect to the number of states of the original automaton, i.e. $O(\exp(n))$ where n is the number of states.

The Hopcroft state minimisation algorithm is instead based on the idea of repeatedly splitting the state set of the automaton so as to obtain finally the equivalence classes of indistinguishable states. Such an algorithm is decidedly more sophisticated than the Nerode and Brozozowski ones, and conceptually is more difficult to understand, but in return it is more efficient, as its worst case time complexity is of type $O(n \log n)$, where n is the number of states of the original automaton to minimise.

Exercise 10 The following regular expression R is given (in generalised form as it contains the intersection operator), over alphabet $\{a, b, c\}$:

$$R = a \ \left(b \, c^* \right)^* \quad \cap \quad \left(a \ \mid \ b \right) \ \left(a^* \, b^* \, c^+ \right)^*$$

Answer the following questions:

1. Design in a systematic way a recogniser automaton equivalent to R (that is design it as the intersection of automata).
2. If necessary, put the designed recogniser in deterministic form.

Solution of (1)

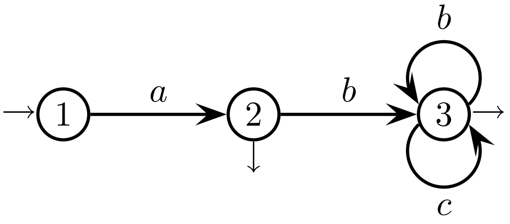
Intuitively the intersection is expressed by the following regular expression R':

$$R' = \left(a \ \left(b^+ \, c^+ \right)^* \right)$$

Expression R' is equivalent to the regular expression R given in the exercise.

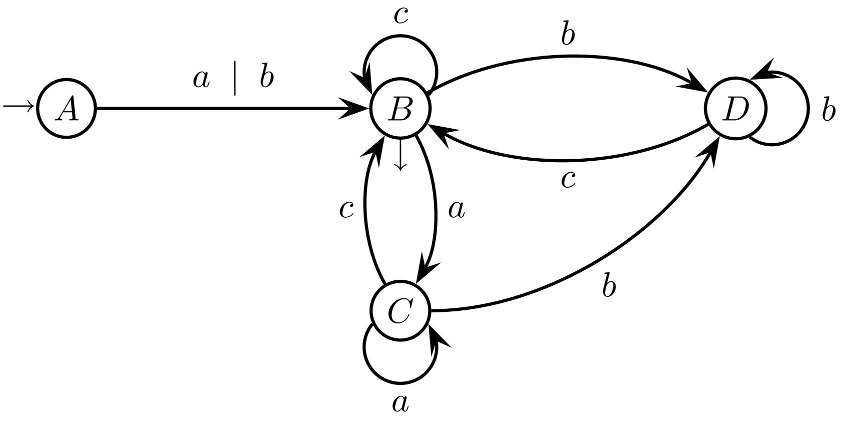
In order to find the intersection in a systematic way, first the two automata of the two subexpressions $a\ (\ b\ c^*\ )^*$ and $(\ a\ \mid\ b\ )\ (\ a^*\ b^*\ c^+\ )^*$ can be obtained by means of the Berri-Sethi or McNaughton-Yamada algorithm, and then they should be intersected by means of the cartesian product construction. Actually the two automata are rather easy to find intuitively.

For subexpression $a\ (\ b\ c^*\ )^*$ one soon has the following automaton:



One designs the automaton quickly by observing that $a\ (\ b\ c^*\ )^* = a\ \mid\ a\, b\ (\ b\ \mid\ c\ )^*$ holds, as $(\ b\ c^*\ )^*$ generates the empty string ε or a generic string beginning with letter b.

For subexpression $(\ a\ \mid\ b\ )\ (\ a^*\ b^*\ c^+\ )^*$ one soon has the following automaton:



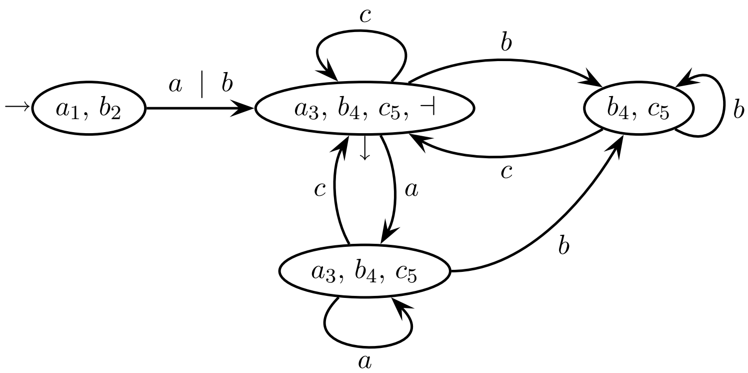
Both automata are already deterministic and in reduced form (as all the states are useful).

Should the latter automaton seem somewhat complex to design in an intuitive way, here is how to obtain it systematically by means of the McNaughton-Yamada algorithm, which is based on a careful examination of the follow characters of the expression generators:

$$(\ a_1\ \mid\ b_2\ )\ \left(\ a_3^*\ b_4^*\ c_5^+\ \right)^*\ \dashv$$

start	a_1, b_2
generator	**follow**
a_1	$a_3\ b_4\ c_5\ \dashv$
b_2	$a_3\ b_4\ c_5\ \dashv$
a_3	$a_3\ b_4\ c_5$
b_4	$b_4\ c_5$
c_5	$a_3\ b_4\ c_5\ \dashv$

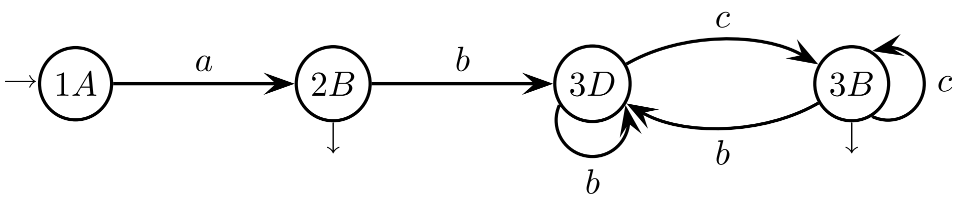
start and follow table



McNaughton-Yamada algorithm

By the way, the constructed automaton coincides with the intuitive version shown before. It is deterministic (by construction) and happens to be minimal (as it can be verified easily).

And here is the intersection automaton (obtained by means of the cartesian product of automata), where unreachable and indefinite states are skipped:



cartesian product automaton

The resulting product automaton is evidently deterministic and minimal: states $1A$ (not final) and $2B$ (final) could be indistinguishable only from states $3B$ (not final) and $3D$ (final), respectively, but both $1A$ and $2B$ do not have outgoing arc labeled with c, while $3B$ and $3D$ do. The automaton generates the regular expression R' of the intersection that is given above; the reader is left the task of verifying so (for instance by means of the Brozozowski or node elimination algorithm). □

Solution of (2)

The product automaton above is already deterministic, therefore nothing else is needed. □

Exercise 11 Consider the two following languages (notice that the alphabets are different):

$$L_1 = \{ x \in \{a, b\}^* \mid \text{the second character of } x \text{ is } a \}$$

$$L_2 = \{ x \in \{a, b, c\}^* \mid \text{the second last (i.e. penultimate) character of } x \text{ is } a \}$$

Answer the following questions:

1. Construct the minimal deterministic automaton that recognises the language $L_1 \cup L_2$ and explain the adopted reasoning.

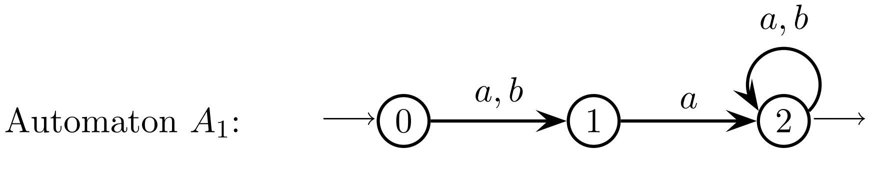
2. Construct the minimal deterministic automaton that recognises the language $L_1 \cap L_2$ and explain the adopted reasoning.

3. Construct the minimal deterministic automaton that recognises the language $L_1 \cup L_2^R$ (L_2^R is the mirror image of L_2) and explain the adopted reasoning.

Solution of (1)

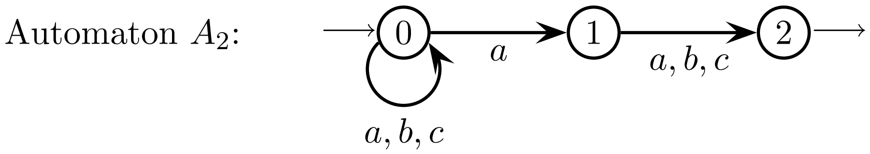
If one wishes to proceed in an algorithmic way, first one ought to design in an intuitive way the two automata A_1 and A_2 that recognise the two languages L_1 and L_2, respectively (such automata may be indeterministic), then one should unite A_1 and A_2 (indeterministically), determinise the resulting automaton and eventually minimise it. Here are the two automata to start with:

Automaton of $L_1 = (\, a \mid b \,)\; a \;(\, a \mid b \,)^*$:



Automaton A_1 is already deterministic.

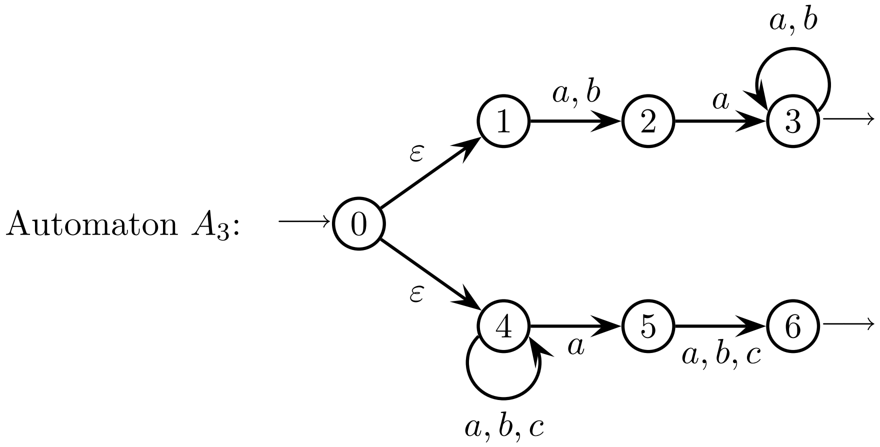
Automaton of $L_2 = (\, a \mid b \mid c \,)^* \; a \;(\, a \mid b \mid c \,)$:



Automaton A_2 is indeterministic (in the state 0).

If one wishes, one can give algoritmically the two automata A_1 and A_2, for instance by means of the Berri-Sethi or McNaughton-Yamada algorithm (in the latter case both automata would be deterministic). Here the two intuitive automata are used.

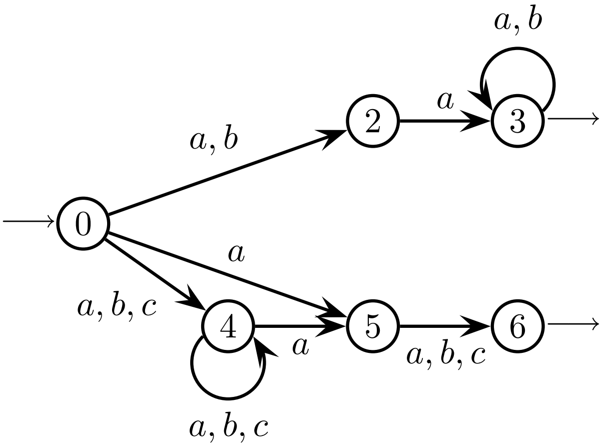
Union automaton (indeterministic) A_3:



Automaton A_3 is indeterministic in the states 0 and 4. State 0 has outgoing spontaneous transitions (ε-transitions).

Elimination of spontaneous transitions (yields automaton A_4):

Automaton A_4:



Automaton A_4 is still indeterministic in the states 0 and 4 (but without spontaneous transitions).

Subset construction (yields automaton A_5):

Automaton A_5

group	a	b	c	final?
0	2 4 5	2 4	4	no
2 4 5	3 4 5 6	4 6	4 6	no
2 4	3 4 5	4	4	no
4	4 5	4	4	no
3 4 5 6	3 4 5 6	3 4 6	4 6	yes
4 6	4 5	4	4	yes
3 4 5	3 4 5 6	3 4 6	4 6	yes
4 5	4 5 6	4 6	4 6	no
3 4 6	3 4 5	3 4	4	yes
4 5 6	4 5 6	4 6	4 6	yes
3 4	3 4 5	3 4	4	yes

successor table

state	a	b	c	final?
A	B	C	D	no
B	E	F	F	no
C	G	D	D	no
D	H	D	D	no
E	E	I	F	yes
F	H	D	D	yes
G	E	I	F	yes
H	L	F	F	no
I	G	M	D	yes
L	L	F	F	yes
M	G	M	D	yes

state table

Minimisation of automaton A_5. As rows I and M coincide, as well as rows E and G, the respective indexing states are indistinguishable and one immediately finds the state equivalences $I \sim M$ and $E \sim G$. Therefore one obtains the automaton A_6:

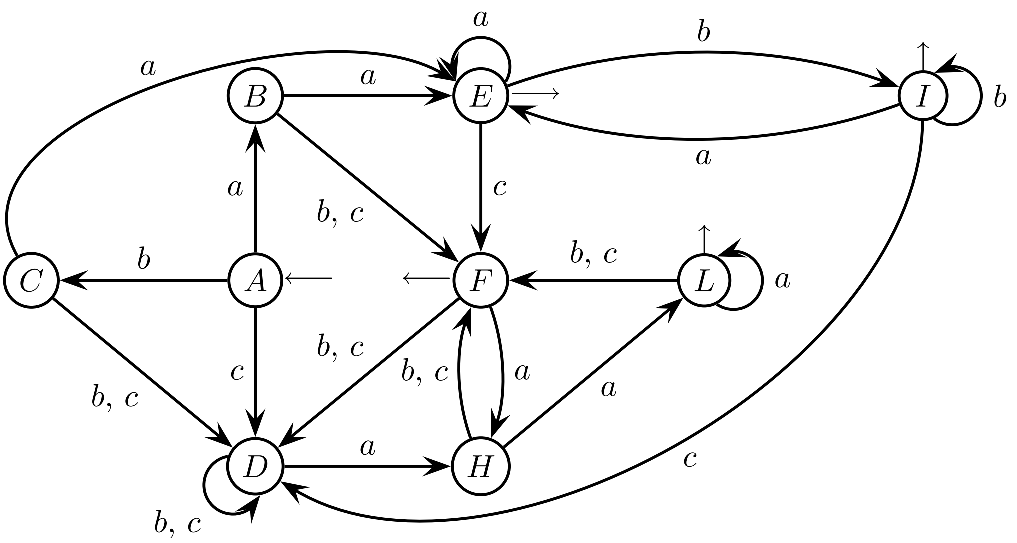
Automaton A_6

state	a	b	c	final?
A	B	C	D	no
B	E	F	F	no
C	E	D	D	no
D	H	D	D	no
E	E	I	F	yes
F	H	D	D	yes
H	L	F	F	no
I	E	I	D	yes
L	L	F	F	yes

state table

It is not difficult to ascertain intuitively that the remaining states are all distinguishable from one another. Therefore automaton A_6 is in the minimal form. Here is its state-transition graph:

Automaton A_6:



state-transition graph

Automaton A_6 has nine states, is deterministic and minimal, and therefore is unique as well. The high number of states, more numerous than the sum of the states of the two original automata A_1 and A_2 (as well as the somewhat twisted and entangled topology), should not surprise: it is necessary for having determinism. In fact, the union of the two original languages L_1 and L_2 is ambiguous as they share some strings (for instance string $b\,a\,b$). Moreover the two respective alphabets are different, hence a string containing a character c necessarily belongs only to L_2 although a prefix of its without c may belong only to L_1 (for instance string $b\,a\,c$ belongs only to L_2 but its prefix $b\,a$ belongs only to L_1). Such two behaviours, more or less indeterministic, are no doubt determinisable (as this can always be done with a finite automaton), but at the expense of using more states. □

Observation

The formal minimality verification of automaton A_6 must be done by means of the triangular implication table. Here it is:

B	BE CF DF							
C	BE CD	DF						
D	BH CD	EH FD	EH					
E	×	×	×	×				
F	×	×	×	×	EH DI DF			
H	BL CF DF	EL	EL DF	HL DF	×	×		
I	×	×	×	×	FD	EH DI	×	
L	×	×	×	×	EL FI	HL DF	×	EL FI DF
	A	B	C	D	E	F	H	I

Now the propagation of distinguishability must be carried out. In the table below, the state pairs that are directly distinguishable are barred through and coloured in **red**.

B	~~BE CF DF~~							
C	~~BE~~ CD	~~DF~~						
D	BH CD	~~EH DF~~	~~EH~~					
E	×	×	×	×				
F	×	×	×	×	~~EH DI DF~~			
H	~~BL CF DF~~	EL	EL ~~DF~~	~~HL DF~~	×	×		
I	×	×	×	×	~~DF~~	~~EH DI~~	×	
L	×	×	×	×	EL FI	~~HL DF~~	×	EL FI ~~DF~~
	A	B	C	D	E	F	H	I

There are three unsolved state pairs left (AC, BH and EL). In the table below the constraints that have become irrelevant are canceled and the state pairs that are made distinguishable in the previous table are barred through and coloured in green.

	A	B	C	D	E	F	H	I
B	×							
C	×	×						
D	BH ~~CD~~	×	×					
E	×	×	×	×				
F	×	×	×	×	×			
H	×	EL	×	×	×	×		
I	×	×	×	×	×	×	×	
L	×	×	×	×	EL ~~FI~~	×	×	×

There is only one unsolved state pair left (BH). In the table below the constraints that have become irrelevant are canceled and the state pairs that are made distinguishable in the previous table are barred through and coloured in **blue**.

	A	B	C	D	E	F	H	I
B	×							
C	×	×						
D	×	×	×					
E	×	×	×	×				
F	×	×	×	×	×			
H	×	~~EL~~	×	×	×	×		
I	×	×	×	×	×	×	×	
L	×	×	×	×	×	×	×	×

final implication table

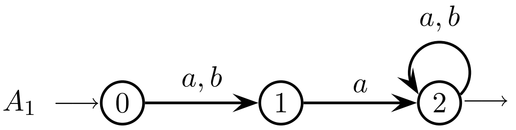
	A	B	C	D	E	F	H	I
B	×							
C	×	×						
D	×	×	×					
E	×	×	×	×				
F	×	×	×	×	×			
H	×	×	×	×	×	×		
I	×	×	×	×	×	×	×	
L	×	×	×	×	×	×	×	×

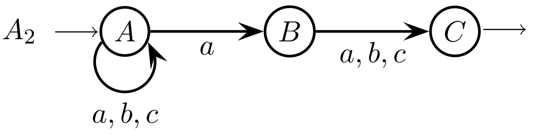
The final implication table, fully solved, is shown on the right. There are not any unsolved state pairs left and all the pairs are distinguishable. Therefore automaton A_6 is already in the minimal form.

Solution of (2)

If one wishes to proceed in an algorithmic way, first one ought to design in an intuitive way the two automata (possibly indeterministic) A_1 and A_2 of the two languages L_1 and L_2, respectively, second to construct the intersection automaton A_3 by means of the cartesian product construction, third to determinise the resulting automaton, and fourth and last to minimise it.

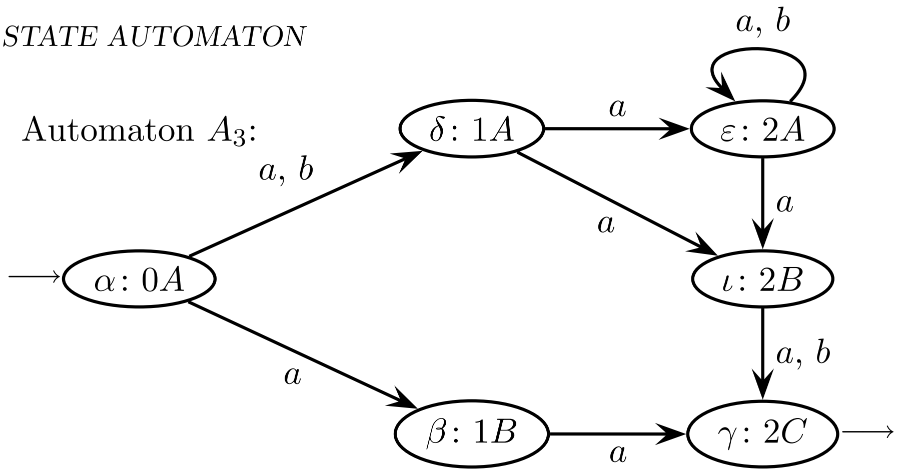
The two automata A_1 and A_2 are the same as for question (1), see before; here they are repeated simply to rename the states of A_2:





The cartesian product automaton A_3 of A_1 and A_2 is the following (unuseful states which are unreachable or do not reach the final state are skipped):

Automaton A_3:



Automaton A_3 turns out to be indeterministic (but without spontaneous transitions) and therefore can be determinised directly by means of the subset construction.

Subset construction (yields automaton A_4):

Automaton A_4

group	a	b	final?
α	$\beta\,\delta$	δ	no
$\beta\,\delta$	$\gamma\,\varepsilon\,\iota$	–	no
δ	$\varepsilon\,\iota$	–	no
$\gamma\,\varepsilon\,\iota$	$\gamma\,\varepsilon\,\iota$	$\gamma\,\varepsilon$	yes
$\varepsilon\,\iota$	$\gamma\,\varepsilon\,\iota$	$\gamma\,\varepsilon$	no
$\gamma\,\varepsilon$	$\varepsilon\,\iota$	ε	yes
ε	$\varepsilon\,\iota$	ε	no

successor table

state	a	b	final?
1	2	3	no
2	4	–	no
3	5	–	no
4	4	6	yes
5	4	6	no
6	5	7	yes
7	5	7	no

state table

Automaton A_4 should be minimised. It would be easy to ascertain that all the states are distinguishable. The formal verification must however be done by means of the triangular implication table. Here it is:

2	×					
3	×	45				
4	×	×	×			
5	24 36	×	×	×		
6	×	×	×	45 67	×	
7	25 37	×	×	×	45 67	×
	1	2	3	4	5	6

Now the propagation of distinguishability must be carried out. In the table below on the left, the state pairs that are directly distinguishable are barred through and coloured in **red**.

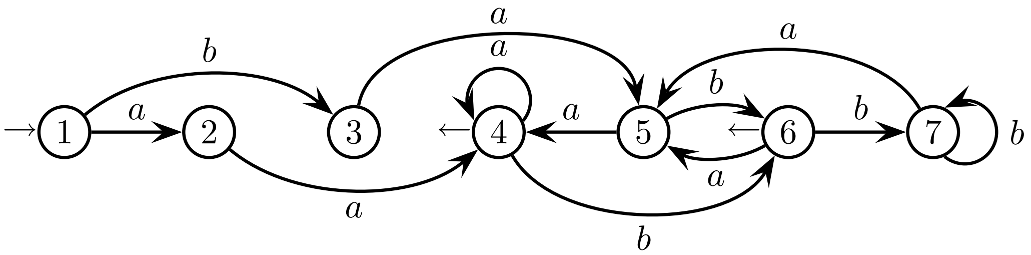
2	×					
3	×	~~45~~				
4	×	×	×			
5	~~24~~ ~~36~~	×	×	×		
6	×	×	×	~~45~~ ~~67~~	×	
7	~~25~~ ~~37~~	×	×	×	~~45~~ ~~67~~	×
	1	2	3	4	5	6

final implication table

2	×					
3	×	×				
4	×	×	×			
5	×	×	×	×		
6	×	×	×	×	×	
7	×	×	×	×	×	×
	1	2	3	4	5	6

The final implication table, fully solved, is shown on the right. There are not any unsolved state pairs left and all the pairs are distinguishable. Therefore automaton A_4 is already in minimal form. Here is the state-transition graph:

Automaton A_4:



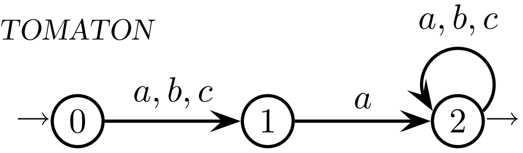
Automaton A_4 is deterministic and minimal, and therefore is unique as well. The somewhat high number of states (though states are not as many as in the union automaton shown before) is the price to be paid for having determinism. □

Solution of (3)

If one mirrors a string (i.e. scans it leftwards instead of rightwards), the second and second last characters are swapped (or keep their position if they coincide). Therefore the effect of mirroring is to switch the two characteristic predicates of the languages L_1 and L_2 (but remember that the two alphabets are different). Hence it follows:

$$L_2^R = \left\{ x \in \{ a, b, c \}^* \mid \underbrace{\text{the second character of } x \text{ is } a}_{\text{predicate of } L_1} \right\}$$

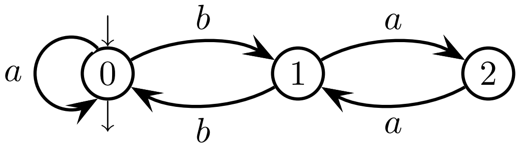
Therefore the difference between the languages L_2^R and L_1 reduces only to the difference of alphabet. As the alphabet of L_2 strictly contains that of L_1, there follows that L_1 is strictly contained in the mirror image of L_2, that is it holds $L_1 \subsetneq L_2^R$. One then concludes that uniting L_1 and L_2^R still yields L_2^R, that is it holds $L_1 \cup L_2^R = L_2^R$. Hence the searched deterministic minimal automaton is the following:



The correctness and minimality of the automaton can be easily verified in an intuitive way. □

Exercise 12 The following (deterministic) automaton A is given:

Automaton A:



Answer the following questions:

1. Transform automaton A into an equivalent "normalised" automaton A_N, which must have the following properties:

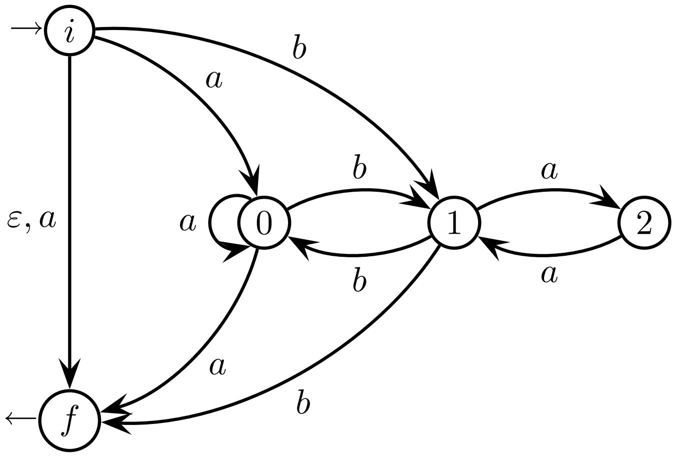
   - has only one initial state i
   - has only one final state f
   - the initial and final states are different, that is $i \neq f$
   - state i does not have any incoming arc
   - state f does not have any outgoing arc

2. Say whether automaton A_N is deterministic and explain why; in the case A_N is not deterministic, say whether it is ambiguous (that is whether it contains two or more different paths that recognise the same string) and explain why.

First solution of (1)

In order to design the "normalised" automaton A_N, the new initial and final states are added to the already given ones, and both have to be connected to the other states so as to preserve the recognised language. Here is automaton A_N:

Automaton A_N:

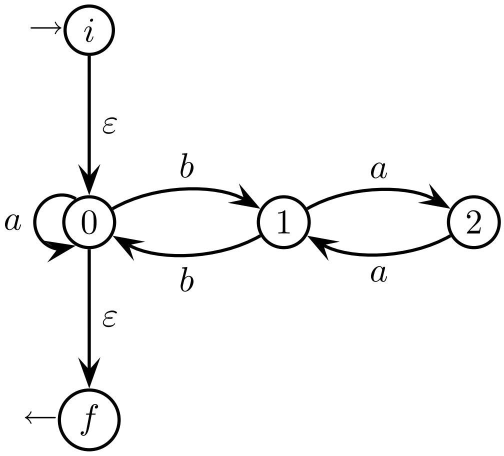


Automaton A_N has two more states with respect to automaton A. It is easy to verify that A_N is equivalent to A. □

Second solution of (1)

An alternative solution makes use of spontaneous transitions (ε-transitions). Here is the alternative "normalised" automaton A'_N:

Automaton A'_N:



Automaton A'_N has two more states with respect to automaton A. It is easy to verify that A'_N is equivalent to A, too. □

Solution of (2)

The original automaton A is certainly not ambiguous. It is evident however that automaton A_N is indeterministic, namely in the states i, 0 and 1, and as a consequence it may be ambiguous. An automaton is said to be ambiguous if a phrase is recognised by two or more different computations (that is paths). One should then examine whether there may exist a phrase accepted by A_N with at least two different computations. It suffices to consider the computations that touch the indeterministic states i, 0 or 1. Notice that A_N is in reduced form, that is every non-final state reaches the final state f by means of some path.

Clearly the empty string ε is recognised only by the path $i \rightarrow f$ (which actually reduces to a single arc), hence it does not cause ambiguity.

If a computation starts from the initial state i and reads one character a, there are only two possibilities, mutually exclusive. Here they are:

- either the computation ends at state f and recognises string a, but nothing else
- or the computation goes into state 0, which is not final, and therefore can reach state f in some way and recognise a string that is certainly longer than a, hence different from a

If a computation labeled by a string y reaches state 0 and here it reads one character a, then it can move in the two following ways: either go into state f or stay in the state 0. Here are the consequences deriving from either move:

- in the former case the computation recognises string $y\,a$, but nothing else
- in the latter case the computation can move one step more, hence it will be able to recognise string $y\,a\,c\,\ldots$ (c is any character), which is certainly longer than $y\,a$ alone

Therefore the two computations cannot recognise the same string. A similar reasoning holds for the computations that touch state 1 (and move with letter b). In conclusion, automaton A_N is not ambiguous. □

Observation

Actually it can be proved in general that the construction shown above, that is to have unique initial and final states, always preserves the property of non-ambiguity of the automaton.

Observation

The alternative automaton shown before is not ambiguous either and a proof is easily obtained by modifying that given for the original solution. The reader is left this task to do by himself.

Exercise 13 Language L over alphabet $\{a, b\}$ is defined by the following conditions:

1. phrases start with string $a\,b$

   and

2. phrases end with character b

   and

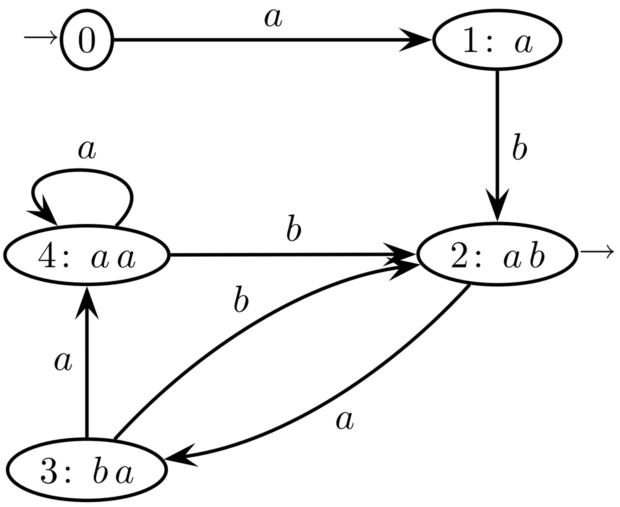
3. phrases do not contain factor $a\,b\,b$

Answer the following questions:

1. Design in a systematic way a deterministic automaton A that recognizes L.
2. If necessary, minimize the previously designed recognizer A.

First solution of (1)

Language L is of local type, though one has to reason with triplets of consecutive characters and not only with pairs. The deterministic automaton A of L only has to remember the two last characters it has read in the input tape. Here it is:



automaton A

The letters recorded in the states are (from left to right) the second last (penultimate) and last character that are read in the input tape; thus, for instance, if one reaches state $2\colon a\,b$ then the second last and last character are a and b, respectively. Of course state 0 (the initial one) does not contain any recorded letter and state 1 (the second state) contains only one.

It is evident that state $b\,b$, actually missing in the graph above, is indefinite (that is if one starts from such a state one cannot reach any final state): if one went into such a state, the string would contain the factor $a\,b\,b$ somewhere (as every string necessarily begins with a), which is a forbidden factor. Therefore state $b\,b$ is not included in the automaton. □

Second solution of (1)

As an alternative, one can construct the automaton by intersecting three simpler automata (that is by means of the cartesian product construction), which are the following (pose $\Sigma = \{\, a,\, b \,\}$): (1) the automaton of $a\,b\,\Sigma^*$ (strings beginning with $a\,b$), (2) that of $\Sigma^*\,b$ (strings ending with b), otherwise directly that of $a\,b\,(\,\varepsilon \mid \Sigma^*\,b\,)$ (strings beginning with $a\,b$ and ending with b), and (3) that of $\neg\,(\,\Sigma^*\,a\,b\,b\,\Sigma^*\,)$ (strings not containing factor $a\,b\,b$); the last one requires to compute a complement, too. All the three (or two) of them are very simple automata and with few states. The reader is left the task of completing this alternative construction. □

Solution of (2)

Intuitively states $3\colon\ b\,a$ and $4\colon\ a\,a$ are indistinguishable and can be unified. However, if one wishes to proceed in an algorithmic way, here is the state table of the automaton (on the left):

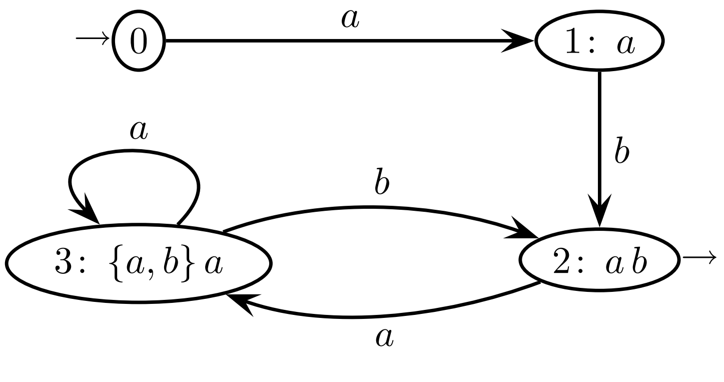
state	a	b	final?
0	1	–	no
1	–	2	no
2	3	–	yes
3	4	2	no
4	4	2	no

state table

state	a	b	final?
0	1	–	no
1	–	2	no
2	3	–	yes
3	3	2	no

minimised state table

Rows 3 and 4 are identical, hence the two indexing states (3 and 4) are indistinguishable and can be unified: row 4 is removed and any occurrence of state 4 is replaced by state 3. The other states are all distinguishable from one another. The minimised state table is shown above on the right. Here is the state-transition graph of the automaton in minimal form:



minimal automaton A

State $3\colon\ \{a, b\}$ records indifferently whether the second last (penultimate) read character is letter a or b. Notice however that states 3 and 1 are distinguishable, as the latter does not have an outgoing arc labeled with a (while the former does). □

Chapter 3

Context-Free Languages

3.1 Grammar Transformation

Exercise 14 Consider the following grammar G (axiom S), over alphabet $\{a, b, c\}$:

$$G \begin{cases} S \rightarrow a\,A\,B\,b \mid A\,E & D \rightarrow A\,A\,B \\ A \rightarrow a\,A\,b\,E \mid \varepsilon & E \rightarrow C\,A \\ B \rightarrow a\,b\,B & F \rightarrow F\,A \mid a \\ C \rightarrow A\,B\,C \mid c & \end{cases}$$

Answer the following questions:

- Identify the indefinite nonterminals of grammar G.
- Identify the unreachable nonterminals of grammar G.
- Transform grammar G into an equivalent grammar G' without unuseful rules.

Solution of (1)

To identify the indefinite (nongenerating) nonterminals of grammar G, first one need identify the definite (generating) ones. Initialise as follows:

$$DEF = \emptyset$$

then start from terminal rules and proceed step by step.

Step 1:

$$\begin{array}{lll} A & \rightarrow & \varepsilon \\ C & \rightarrow & c \\ F & \rightarrow & a \end{array} \qquad DEF = DEF \cup \{A, C, F\} = \{A, C, F\}$$

Step 2:

$$E \rightarrow C\,A \qquad DEF = DEF \cup \{E\} = \{A, C, E, F\}$$

Step 3:

$$S \rightarrow A\,E \qquad DEF = DEF \cup \{S\} = \{S, A, C, E, F\}$$

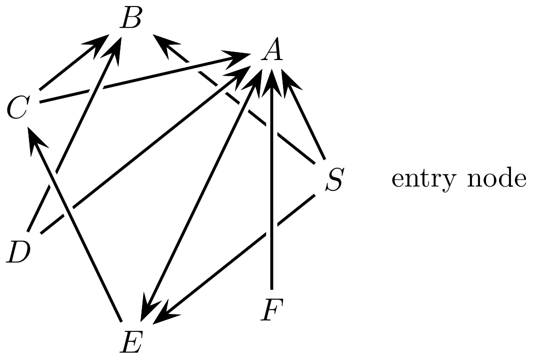
This is the end of the procedure, as at the next step nothing would change any more. Then the sets of definite and indefinite nonterminals are the following:

$$\begin{array}{rcl} DEF & = & \{S, A, C, E, F\} \\ UNDEF & = & V_N - DEF = \{S, A, B, C, D, E, F\} - \{S, A, C, E, F\} = \{B, D\} \end{array}$$

Since axiom S is definite (this means it generates at least one string), language $L(G)$ is certainly not empty (that is $L(G) \neq \emptyset$). □

Solution of (2)

To identify the unreachable nonterminals of grammar G, draw the reachability (directed) graph as determined by the rules of G. Here it is (on the left):



reachability graph

nonterminal	reachable ?
S	yes (by definition)
A	yes
B	yes
C	yes
D	no
E	yes
F	no

reachability table

Axiom S is reachable by definition. Nonterminals A, B and E are directly reachable from S (all of them can also be reached indirectly). Nonterminal C is reachable indirectly through E. Nonterminals D and F are unreachable. See the list on the right of the graph. It holds:

$$\begin{array}{rcl} REACH & = & \{S, A, B, C, E\} \\ UNREACH & = & V_N - REACH = \{S, A, B, C, D, E, F\} - \{S, A, B, C, E\} = \{D, F\} \end{array}$$

Notice that nonterminal D is both indefinite (from before) and unreachable (from here). □

Observation

Notice that the reachability graph contains two reachable loops: $A \to E \to C \to A$ and $A \to E \to A$; hence grammar G is recursive and its derivations have arbitrary length.

Solution of (3)

Eliminating the unuseful rules from grammar G (rules that cannot take part to any successful derivation) means putting G in reduced form. Proceed as follows:

1. first analyse the grammar to identify indefinite nonterminals and remove any rule (or alternative) that contains an indefinite nonterminal
2. then analyse again the grammar to identify unreachable nonterminals and remove any rule that contains a unreachable nonterminal

Step 1 must be executed before step 2, as order is relevant: removing the unuseful rules containing indefinite nonterminals may cause more nonterminals to become unreachable.

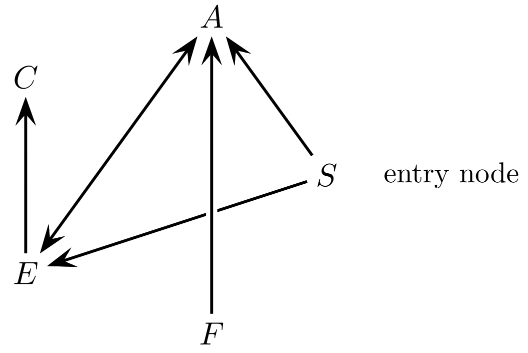
The set of indefinite nonterminals of grammar G has already been identified before and is $UNDEF = \{B, D\}$. First here is the form G_d of grammar G where the rules (or alternatives) containing indefinite nonterminals are barred through for removal:

$$G_d \begin{cases} \begin{array}{lllllll} S & \to & \underline{a\,A\,B\,b} \mid A\,E & \qquad & \underline{D} & \to & \underline{A\,A\,B} \\ A & \to & a\,A\,b\,E \mid \varepsilon & & E & \to & C\,A \\ \underline{B} & \to & \underline{a\,b\,B} & & F & \to & F\,A \mid a \\ C & \to & \underline{A\,B\,C} \mid c & & & & \end{array} \end{cases}$$

By removing such rules it follows:

$$G_d \begin{cases} \begin{array}{lllllll} S & \to & A\,E & \qquad & E & \to & C\,A \\ A & \to & a\,A\,b\,E \mid \varepsilon & & F & \to & F\,A \mid a \\ C & \to & c & & & & \end{array} \end{cases}$$

Then one should repeat the unreachability analysis, as follows:



reachability graph

nonterminal	reachable ?
S	yes (by definition)
A	yes
C	yes
E	yes
F	no

reachability table

Nonterminals A, C and E are reachable (directly or indirectly), while nonterminal F is unreachable. Notice that there is still a reachable loop $A \rightarrow E \rightarrow A$ and hence grammar G_d is recursive. Here is the reduced form G_r of grammar G, where the rules (or alternatives) containing unreachable nonterminals are barred through for removal:

$$G_r \left\{ \begin{array}{lcllcl} S & \rightarrow & A\,E & E & \rightarrow & C\,A \\ A & \rightarrow & a\,A\,b\,E \mid \varepsilon & \underline{F} & \rightarrow & \underline{F\,A \mid a} \\ C & \rightarrow & c & & & \end{array} \right.$$

By removing such rules one finally obtains grammar G':

$$G' \left\{ \begin{array}{lcllcl} S & \rightarrow & A\,E & E & \rightarrow & C\,A \\ A & \rightarrow & a\,A\,b\,E \mid \varepsilon & C & \rightarrow & c \end{array} \right.$$

Grammar G' is equivalent to grammar G but is in reduced form, where all the rules are useful, and uses only nonterminals S, A, C and E. Moreover, G' is still recursive (as G is). □

Observation

Here repeating the unreachability analysis on grammar G_d does not yield more unreachable nonterminals than those that are already identified by analysing directly grammar G; this however happens by chance and in general more nonterminal may turn out to be unreachable after removing the rules containing indefinite nonterminals.

Exercise 15 Consider the following grammar G (axiom S) over alphabet $\{a, b, c\}$, already in reduced form:

$$G \left\{ \begin{array}{lcl} S & \rightarrow & a\,b \mid A\,B\,C\,D \mid A\,D \\ A & \rightarrow & a\,A\,b \mid \varepsilon \\ B & \rightarrow & a\,b\,B \mid a\,b \\ C & \rightarrow & A\,B \mid c \\ D & \rightarrow & A\,A\,D \mid A\,A \end{array} \right.$$

Answer the following questions:

- Identify the nullable nonterminals of grammar G.
- Transform grammar G into an equivalent grammar G' in non-nullable form.
- Choose a representative string $w \in L(G)$ and show the two syntax trees of w in the grammars G and G'.

Solution of (1)

To identify nullable nonterminals, initialise as follows:

$$NULL = \emptyset$$

then start from null rules and proceed step by step.
Step 1:

$$A \rightarrow \varepsilon \qquad NULL = NULL \cup \{A\} = \{A\}$$

Step 2:

$$D \rightarrow A\,A \qquad NULL = NULL \cup \{D\} = \{A, D\}$$

Step 3:

$$S \rightarrow A\,D \qquad NULL = NULL \cup \{S\} = \{S, A, D\}$$

This is the end of the procedure, as nothing would change at the next step any more. Then the sets of nullable and non-nullable nonterminals are the following:

$$\begin{array}{rcl} NULL & = & \{S, A, D\} \\ NON\_NULL & = & V_N - NULL = \{S, A, B, C, D\} - \{S, A, D\} = \{B, C\} \end{array}$$

Since axiom S is nullable, language $L(G)$ is not ε-free (that is $\varepsilon \in L(G)$). □

Solution of (2)

To transform grammar G into non-nullable form, remove all the null rules (but add or keep rule $S \rightarrow \varepsilon$ if the language is not ε-free) and selectively cancel nullable nonterminals in the remaining rules, in all the possible combinations. Here is the non-nullable form G' (axiom S):

$$G' \left\{ \begin{array}{lcl} S & \rightarrow & a\,b \\ S & \rightarrow & A\,B\,C\,D \mid \underbrace{B\,C\,D \mid A\,B\,C \mid B\,C}_{\text{obtained from } A\,B\,C\,D} \\ S & \rightarrow & A\,D \mid \underbrace{A \mid D}_{\text{obtained from } A\,D} \\ S & \rightarrow & \varepsilon \\ A & \rightarrow & a\,A\,b \mid \underbrace{a\,b}_{\text{obtained from } a\,A\,b} \\ B & \rightarrow & a\,b\,B \mid a\,b \\ C & \rightarrow & A\,B \mid c \mid \underbrace{B}_{\text{obtained from } A\,B} \\ D & \rightarrow & A\,A\,D \mid A\,A \mid \underbrace{A\,D \mid A}_{\text{obtained from } A\,A\,D \text{ and } A\,A} \end{array} \right.$$

For instance, since both nonterminals A and D are nullable, from the original rule $S \to A\,B\,C\,D$ new alternative rules $S \to B\,C\,D \mid A\,B\,C \mid B\,C$ are created, which are obtained by canceling from the original one first only A, then only D and finally both A and D. A similar reasoning applies to the other new rules. Moreover, the new null axiomatic rule $S \to \varepsilon$ has to be added as language $L(G)$ is not ε-free.

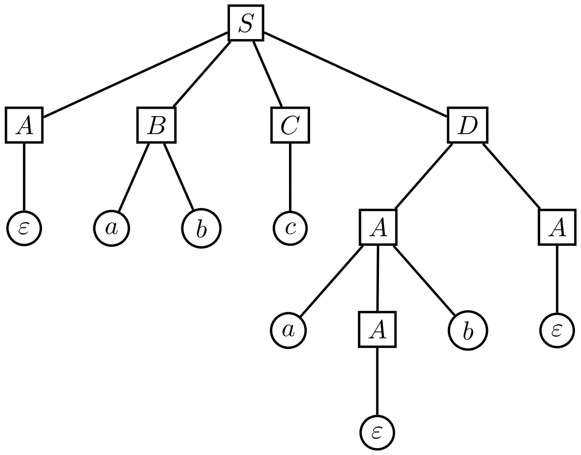
In principle from rule $S \to A\,A\,D$ the identity rule $D \to D$ could be obtained as well, by canceling both occurrences of the nullable nonterminal A. However one can skip such a rule, as clearly it does not change the generating power of the grammar (while it would surely cause it to become ambiguous). Grammar G' can be compacted as follows:

$$G' \left\{ \begin{array}{lcl} S & \to & a\,b \mid A\,B\,C\,D \mid B\,C\,D \mid A\,B\,C \mid B\,C \mid A\,D \mid A \mid D \mid \varepsilon \\ A & \to & a\,A\,b \mid a\,b \\ B & \to & a\,b\,B \mid a\,b \\ C & \to & A\,B \mid c \mid B \\ D & \to & A\,A\,D \mid A\,A \mid A\,D \mid A \end{array} \right.$$

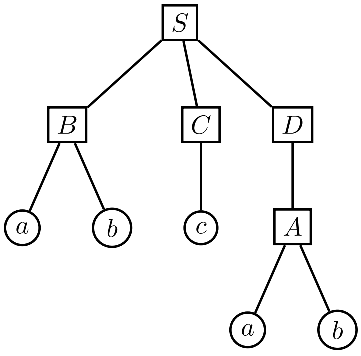
by grouping all the alternative rules into one rule only. Notice that by canceling nullable nonterminals, a few new copy rules have been created. □

Solution of (3)

Here the representative string $w = a\,b\,c\,a\,b$, which belongs to $L(G)$, is chosen and both syntax trees of w in the grammars G and the equivalent non-nullable form G' are shown:



syntax tree of G



syntax tree of G'

Notice that the syntax tree of grammar G' is more compact than that of grammar G, as it does not have any null leaves, but G' itself has more alternative rules than G has. □

Exercise 16 Consider the following grammar G (axiom S) over alphabet $\{a, b, c\}$, already in reduced form:

$$G \begin{cases} S \rightarrow a\,A\,B\,C \mid A\,C \\ A \rightarrow a\,A\,b \mid B \\ B \rightarrow a\,b\,B \mid \varepsilon \\ C \rightarrow B\,D \mid c \\ D \rightarrow A\,A\,D \mid \varepsilon \end{cases}$$

Answer the following questions:

- Identify the nonterminals of grammar G that have renaming derivations.
- Transform grammar G into an equivalent grammar G' without renaming derivations.
- Choose a representative string $w \in L(G)$ and show the two syntax trees of w in the grammars G and G'.

Solution of (1)

A renaming derivation is of type $A \overset{+}{\Rightarrow} B$ (nonterminals A and B may coincide). An immediate group of renaming derivations is given by copy rules, which in grammar G include only rule $A \rightarrow B$. Thus nonterminal A has the elementary (that is of length one) renaming derivation $A \Rightarrow B$. Copy rules might concatenate to one another and yield renaming derivations of length > 1 (and possibly circular derivations as well), but here this does not happen.

However, renaming derivations of any length may also be an indirect outcome of the existence of nullable nonterminals, i.e. $A \Rightarrow B\,C \overset{+}{\Rightarrow} C$ if B is nullable (A and C may coincide). Therefore one should identify nullable nonterminals, as follows:

$$NULL = \emptyset$$

Step 1:

$$\begin{array}{lll} B & \rightarrow & \varepsilon \\ D & \rightarrow & \varepsilon \end{array} \qquad NULL = NULL \cup \{B, D\} = \{B, D\}$$

Step 2:

$$\begin{array}{lll} A & \rightarrow & B \\ C & \rightarrow & B\,D \end{array} \qquad NULL = NULL \cup \{A, C\} = \{A, B, C, D\}$$

Step 3:

$$S \rightarrow A\,C \qquad NULL = NULL \cup \{S\} = \{S, A, B, C, D\}$$

This is the end of the procedure, as nothing changes at the next step any more. All the nonterminals of grammar G are nullable including axiom S, and therefore language $L(G)$ is not ε-free. Now from the rules of grammar G, new renaming derivations (of length two or more) can be obtained by taking care of $S, A, B, C, D \overset{+}{\Rightarrow} \varepsilon$, as follows:

Rule	**Renaming derivation**
$S \rightarrow A\,C$	$S \Rightarrow A\,C \xRightarrow{C \overset{+}{\Rightarrow} \varepsilon} A$
$S \rightarrow A\,C$	$S \Rightarrow A\,C \xRightarrow{A \overset{+}{\Rightarrow} \varepsilon} C$
$C \rightarrow B\,D$	$C \Rightarrow B\,D \xRightarrow{D \overset{+}{\Rightarrow} \varepsilon} B$
$C \rightarrow B\,D$	$C \Rightarrow B\,D \xRightarrow{B \overset{+}{\Rightarrow} \varepsilon} D$
$D \rightarrow A\,A\,D$	$D \Rightarrow A\,A\,D \xRightarrow{A \overset{+}{\Rightarrow} \varepsilon} A\,D \xRightarrow{D \overset{+}{\Rightarrow} \varepsilon} A$
$D \rightarrow A\,A\,D$	$D \Rightarrow A\,A\,D \xRightarrow{A \overset{+}{\Rightarrow} \varepsilon} D$

Notice that the second last renaming derivation is actually ambiguous, as the final nonterminal A can derive from both the first and second occurrence of A in the sentential form $A\,A\,D$.

Summarise all the renaming derivations obtained so far, as follows (on the left):

$S \overset{+}{\Rightarrow} A \mid C$	$S \overset{+}{\Rightarrow} A \mid B \mid C \mid D$
$A \overset{+}{\Rightarrow} B$	$A \overset{+}{\Rightarrow} B$
$C \overset{+}{\Rightarrow} B \mid D$	$C \overset{+}{\Rightarrow} A \mid B \mid D$
$D \overset{+}{\Rightarrow} A \mid D$	$D \overset{+}{\Rightarrow} A \mid B \mid D$
summary	transitive closure

Finally compute the transitive closure of the found renaming derivations, that is concatenate to one another in all the possible ways the renaming derivations obtained so far (see above on the right). These are all the possible renaming derivations of grammar G. □

Observation

One of the above given renaming derivations of grammar G is actually circular (i.e. $D \overset{+}{\Rightarrow} D$), hence G is surely an ambiguous grammar. Such a derivation does not originate directly by concatenating copy rules (the simplest possibility), rather here it is a side effect of the presence of nullable nonterminals. Of course, G might also have multiple derivations of different type and be even more ambiguous thereafter (this is actually the case - see the note above).

Solution of (2)

To eliminate from grammar G all the renaming derivations, and thus indirectly to eliminate the circular ones as well, first one has to transform G into non-nullable form and then to remove copy rules. The set of nullable nonterminals is already known from before (in practice however all of them are nullable), as well as that language $L(G)$ is not ε-free, therefore the non-nullable form G_{nn} of G is the following:

$$G_{nn}\begin{cases} S & \rightarrow & a\,A\,B\,C \mid a\,B\,C \mid a\,A\,C \mid a\,B\,C \mid a\,A \mid a\,B \mid a\,C \\ S & \rightarrow & A\,C \mid A \mid C \\ S & \rightarrow & \varepsilon \\ A & \rightarrow & a\,A\,b \mid a\,b \mid B \\ B & \rightarrow & a\,b\,B \mid a\,b \\ C & \rightarrow & B\,D \mid B \mid D \mid c \\ D & \rightarrow & A\,A\,D \mid A\,D \mid A \mid A\,A \end{cases}$$

Then the copy sets of each nonterminal of grammar G_{nn} are the following:

$$\begin{array}{rclcrcl} \mathit{copy}(S) & = & \{A, C\} & \quad & \mathit{copy}(C) & = & \{B, D\} \\ \mathit{copy}(A) & = & \{B\} & & \mathit{copy}(D) & = & \{A\} \\ \mathit{copy}(B) & = & \emptyset & & & & \end{array}$$

To eliminate copy rules, just apply the removal procedure step by step (that is if there is rule $A \rightarrow B$ then for each rule $B \rightarrow \beta$ add rule $A \rightarrow \beta$ and at the end remove rule $A \rightarrow B$) and remember that at each step the copy sets may be regenerated (this happens when $\beta = C$); identity rules (like $A \rightarrow A$) can be removed directly. Here is the elimination algorithm (order is irrelevant).

Eliminate $\mathit{copy}(A) = \{B\}$:

$$\begin{cases} S & \rightarrow & a\,A\,B\,C \mid a\,B\,C \mid a\,A\,C \mid a\,B\,C \mid a\,A \mid a\,B \mid a\,C \\ S & \rightarrow & A\,C \mid A \mid C \\ S & \rightarrow & \varepsilon \\ A & \rightarrow & a\,A\,b \mid a\,b \mid \underbrace{a\,b\,B}_{\text{from } B} \\ B & \rightarrow & a\,b\,B \mid a\,b \\ C & \rightarrow & B\,D \mid B \mid D \mid c \\ D & \rightarrow & A\,A\,D \mid A\,D \mid A \mid A\,A \end{cases}$$

Eliminate $\mathit{copy}(C) = \{B, D\}$:

$$\begin{cases} S & \rightarrow & a\,A\,B\,C \mid a\,B\,C \mid a\,A\,C \mid a\,B\,C \mid a\,A \mid a\,B \mid a\,C \\ S & \rightarrow & A\,C \mid A \mid C \\ S & \rightarrow & \varepsilon \\ A & \rightarrow & a\,A\,b \mid a\,b \mid a\,b\,B \\ B & \rightarrow & a\,b\,B \mid a\,b \\ C & \rightarrow & B\,D \mid \underbrace{a\,b\,B \mid a\,b}_{\text{from } B} \mid \underbrace{A\,A\,D \mid A\,D \mid A \mid A\,A}_{\text{from } D} \mid c \\ D & \rightarrow & A\,A\,D \mid A\,D \mid A \mid A\,A \end{cases}$$

Eliminate again *copy*$(C) = \{A\}$ (has changed after the previous step):

$$\left\{ \begin{array}{lll}
S & \to & aABC \mid aBC \mid aAC \mid aBC \mid aA \mid aB \mid aC \\
S & \to & AC \mid A \mid C \\
S & \to & \varepsilon \\
A & \to & aAb \mid ab \mid abB \\
B & \to & abB \mid ab \\
C & \to & BD \mid abB \mid ab \mid AAD \mid AD \mid \underbrace{aAb}_{\text{from } A} \mid AA \mid c \\
D & \to & AAD \mid AD \mid A \mid AA
\end{array} \right.$$

Eliminate *copy*$(D) = \{A\}$:

$$\left\{ \begin{array}{lll}
S & \to & aABC \mid aBC \mid aAC \mid aBC \mid aA \mid aB \mid aC \\
S & \to & AC \mid A \mid C \\
S & \to & \varepsilon \\
A & \to & aAb \mid ab \mid abB \\
B & \to & abB \mid ab \\
C & \to & BD \mid abB \mid ab \mid AAD \mid AD \mid aAb \mid AA \mid c \\
D & \to & AAD \mid AD \mid \underbrace{aAb \mid ab \mid abB}_{\text{from } A} \mid AA
\end{array} \right.$$

And finally eliminate *copy*$(S) = \{A, C\}$:

$$\left\{ \begin{array}{lll}
S & \to & aABC \mid aBC \mid aAC \mid aBC \mid aA \mid aB \mid aC \\
S & \to & AC \\
S & \to & \underbrace{aAb \mid ab \mid abB}_{\text{from } A} \\
S & \to & \underbrace{BD \mid AAD \mid AD \mid AA \mid c}_{\text{from } C} \\
S & \to & \varepsilon \\
A & \to & aAb \mid ab \mid abB \\
B & \to & abB \mid ab \\
C & \to & BD \mid abB \mid ab \mid AAD \mid AD \mid aAb \mid AA \mid c \\
D & \to & AAD \mid AD \mid aAb \mid ab \mid abB \mid AA
\end{array} \right.$$

In conclusion, grammar G' without renaming derivations is the following:

$$G' \left\{ \begin{array}{lll} S & \rightarrow & a\,A\,B\,C \mid a\,B\,C \mid a\,A\,C \mid a\,B\,C \mid a\,A \mid a\,B \mid a\,C \mid \\ & & A\,C \mid a\,A\,b \mid a\,b \mid a\,b\,B \mid B\,D \mid A\,A\,D \mid A\,D \mid A\,A \mid \\ & & c \mid \varepsilon \\ A & \rightarrow & a\,A\,b \mid a\,b \mid a\,b\,B \\ B & \rightarrow & a\,b\,B \mid a\,b \\ C & \rightarrow & B\,D \mid a\,b\,B \mid a\,b \mid A\,A\,D \mid A\,D \mid a\,A\,b \mid A\,A \mid c \\ D & \rightarrow & A\,A\,D \mid A\,D \mid a\,A\,b \mid a\,b \mid a\,b\,B \mid A\,A \end{array} \right.$$

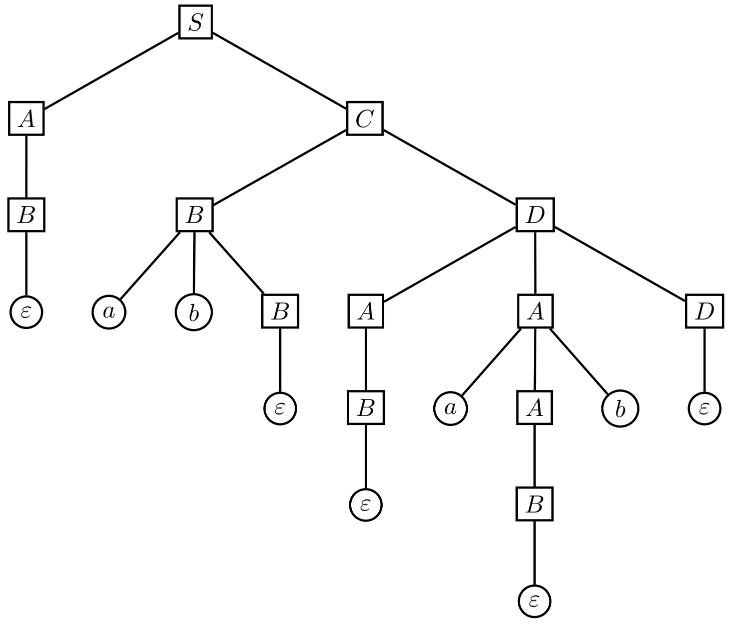
Of course grammar G' is also in non-nullable form (and is in reduced form). □

Observation

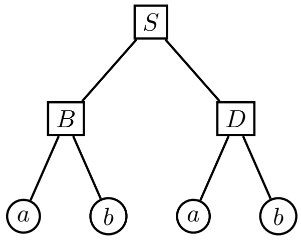
As grammar G' does not have any renaming derivation, it does not have any circular one either. Although such a source of ambiguity has been eliminated, G' may still be ambiguous if grammar G is so. Of course G' is not necessarily the shortest or simplest version of grammar G without renaming derivations. Actually the transformations here applied may have created some redundant rules which could be intuitively simplified.

Solution of (3)

Here the representative string $w = a\,b\,a\,b$, which belongs to $L(G)$, is chosen and both syntax trees of w in the grammars G and the equivalent form G' without renaming derivations (and also without nullable nonterminals) are shown:



syntax tree of G



syntax tree of G'

Notice that the syntax tree of grammar G' is more compact than that of grammar G, as it does not have any linear branches (there are not any unary internal nodes but possibly some of those corresponding to terminal rules of length one) and null leaves, but G' itself has more alternative rules than G has. This demonstrates that copy rules, and more generally renaming derivations, may be helpful for reducing the size of a grammar. □

Exercise 17 Consider the following grammar G (axiom S), over alphabet $\{a, b\}$:

$$G \left\{ \begin{array}{lcl} S & \rightarrow & a\,A\,B \mid B\,C \mid B\,G \\ A & \rightarrow & B\,A \mid a \mid \varepsilon \\ B & \rightarrow & a\,B\,b \mid a\,b\,D\,a \mid A\,A \\ C & \rightarrow & A\,B \mid a \\ D & \rightarrow & F\,D \mid F \\ E & \rightarrow & A\,A \mid \varepsilon \\ F & \rightarrow & A\,b\,A \mid b \\ G & \rightarrow & D\,G \mid H \\ H & \rightarrow & E\,H \end{array} \right.$$

Answer the following questions:

1. Transform grammar G into an equivalent grammar G' in Chomsky normal form.
2. Choose a representative string $w \in L(G)$ and show the two syntax trees of w in the grammars G and G'.

Solution of (1)

To transform grammar G into Chomsky normal form, do as follows: first put G into reduced form, second into non-nullable form, third eliminate copy rules, and fourth and last put all the rules in binary or terminal form (with the possible exception of rule $S \rightarrow \varepsilon$).

Transformation into reduced form:

Identification and elimination of indefinite nonterminals:

$$DEF = \emptyset$$

Step 1:

$$\begin{array}{lcl} A & \rightarrow & a \\ C & \rightarrow & a \\ E & \rightarrow & \varepsilon \\ F & \rightarrow & B \end{array} \qquad DEF = DEF \cup \{A, C, E, F\} = \{A, C, E, F\}$$

Step 2:

$$\begin{array}{lll} B & \rightarrow & A\,A \\ D & \rightarrow & F \end{array} \qquad DEF = DEF \cup \{B, D\} = \{A, B, C, D, E, F\}$$

Step 3:

$$S \rightarrow A\,B \mid B\,C \qquad DEF = DEF \cup \{S\} = \{S, A, B, C, D, E, F\}$$

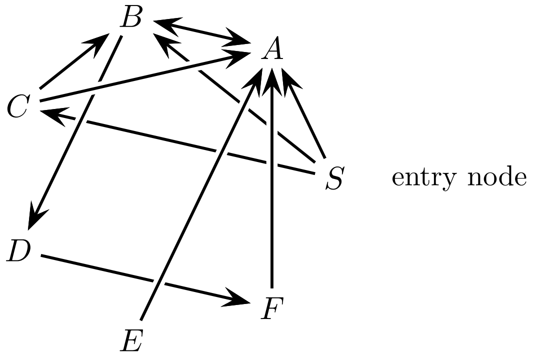
Step 3 is the last one, as nothing changes next. Then it holds:

$$\begin{array}{rcl} DEF & = & \{S, A, B, C, D, E, F\} \\ UNDEF & = & V_N - DEF = \{G, H\} \end{array}$$

The form G_d of grammar G without rules containing indefinite nonterminals is the following:

$$G_d \left\{ \begin{array}{lll} S & \rightarrow & a\,A\,B \mid B\,C \\ A & \rightarrow & B\,A \mid a \mid \varepsilon \\ B & \rightarrow & a\,B\,b \mid a\,b\,D\,a \mid A\,A \\ C & \rightarrow & A\,B \mid a \\ D & \rightarrow & F\,D \mid F \\ E & \rightarrow & A\,A \mid \varepsilon \\ F & \rightarrow & A\,b\,A \mid b \end{array} \right.$$

Identification and elimination of unreachable nonterminals:



nonterminal	reachable ?
S	yes (by definition)
A	yes
B	yes
C	yes
D	yes
E	no
F	yes

reachability graph

reachability table

Nonterminals A, B, C, D and F are reachable (directly or indirectly), while nonterminal E is unreachable. Notice that E has got unreachable as a consequence of nonterminal H being indefinite. Then it holds:

$$\begin{array}{rcl} UNREACH & = & \{E\} \\ REACH & = & V_N - UNREACH = \{S, A, B, C, D, F\} \end{array}$$

The form G_r of grammar G without rules containing unreachable nonterminals is the following:

$$G_r \begin{cases} S \rightarrow a\,A\,B \mid B\,C \\ A \rightarrow B\,A \mid a \mid \varepsilon \\ B \rightarrow a\,B\,b \mid a\,b\,D\,a \mid A\,A \\ C \rightarrow A\,B \mid a \\ D \rightarrow F\,D \mid F \\ F \rightarrow A\,b\,A \mid b \end{cases}$$

Grammars G and G_r are equivalent but G_r is in reduced form.
Transformation into non-nullable form:

$$NULL = \emptyset$$

Step 1:

$A \rightarrow \varepsilon$ $\qquad NULL = NULL \cup \{A\} = \{A\}$

Step 2:

$B \rightarrow A\,A$ $\qquad NULL = NULL \cup \{B\} = \{A, B\}$

Step 3:

$C \rightarrow A\,B$ $\qquad NULL = NULL \cup \{C\} = \{A, B, C\}$

Step 4:

$S \rightarrow B\,C$ $\qquad NULL = NULL \cup \{S\} = \{S, A, B, C\}$

Step 4 is the last one, as nothing changes next. Then it holds:

$$\begin{aligned} NULL &= \{S, A, B, C\} \\ NON\_NULL &= V_N - NULL = \{D, F\} \end{aligned}$$

Since axiom S is nullable, language $L(G)$ is not ε-free. The form G_{nn} of grammar G without nullable nonterminals (and in reduced form) is the following:

$$G_{nn} \begin{cases} S \rightarrow a\,A\,B \mid a\,B \mid a\,A \mid a \mid B\,C \mid C \mid B \mid \varepsilon \\ A \rightarrow B\,A \mid B \mid a \\ B \rightarrow a\,B\,b \mid a\,b \mid a\,b\,D\,a \mid A\,A \mid A \\ C \rightarrow A\,B \mid B \mid A \mid a \\ D \rightarrow F\,D \mid F \\ F \rightarrow A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

The null axiomatic rule $S \rightarrow \varepsilon$ is added as language $L(G)$ is not ε-free.

Elimination of copy rules:

$$\begin{array}{rclcrcl} \mathit{copy}(S) & = & \{B, C\} & \qquad & \mathit{copy}(C) & = & \{A, B\} \\ \mathit{copy}(A) & = & \{B\} & & \mathit{copy}(D) & = & \{F\} \\ \mathit{copy}(B) & = & \{A\} & & \mathit{copy}(F) & = & \emptyset \end{array}$$

Eliminate $\mathit{copy}(A) = \{B\}$:

$$\begin{cases} S & \to & a\,A\,B \mid a\,B \mid a\,A \mid a \mid B\,C \mid C \mid B \mid \varepsilon \\ A & \to & B\,A \mid a\,B\,b \mid a\,b \mid a\,b\,D\,a \mid A\,A \mid a \\ B & \to & a\,B\,b \mid a\,b \mid a\,b\,D\,a \mid a\,b\,a \mid A\,A \mid A \\ C & \to & A\,B \mid B \mid A \mid a \\ D & \to & F\,D \mid F \\ F & \to & A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Eliminate $\mathit{copy}(B) = \{A\}$:

$$\begin{cases} S & \to & a\,A\,B \mid a\,B \mid a\,A \mid a \mid B\,C \mid C \mid B \mid \varepsilon \\ A & \to & B\,A \mid a\,B\,b \mid a\,b \mid a\,b\,D\,a \mid A\,A \mid a \\ B & \to & a\,B\,b \mid a\,b \mid a\,b\,D\,a \mid a\,b\,a \mid A\,A \mid B\,A \mid a \\ C & \to & A\,B \mid B \mid A \mid a \\ D & \to & F\,D \mid F \\ F & \to & A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Notice that nonterminals A and B are expanded by identical rules, hence $L(A) = L(B)$ holds, and therefore A and B are equivalent. This is a consequence of having a copy loop $A \Rightarrow B \Rightarrow A$ (a circular derivation made only of copy rules). One of the two nonterminals can be eliminated, for instance B, and then every occurrence of B can be replaced by A, as follows:

$$\begin{cases} S & \to & a\,A\,A \mid a\,A \mid a \mid A\,C \mid C \mid A \mid \varepsilon \\ A & \to & A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid a \\ C & \to & A\,A \mid A \mid a \\ D & \to & F\,D \mid F \\ F & \to & A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Since the above elimination of nonterminal B may have changed the copy sets, these have to be recomputed, as follows:

$$\begin{array}{rclcrcl} \mathit{copy}(S) & = & \{A, C\} & \qquad & \mathit{copy}(D) & = & \{F\} \\ \mathit{copy}(A) & = & \emptyset & & \mathit{copy}(F) & = & \emptyset \\ \mathit{copy}(C) & = & \{A\} & & & & \end{array}$$

Eliminate $\mathit{copy}(C) = \{A\}$:

$$\begin{cases} S \rightarrow a\,A\,A \mid a\,A \mid a \mid A\,C \mid C \mid A \mid \varepsilon \\ A \rightarrow A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid a \\ C \rightarrow A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid a \\ D \rightarrow F\,D \mid F \\ F \rightarrow A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Again, nonterminals A and C are expanded by identical rules and therefore are equivalent. This happens by chance, as there is not any copy loop that links A and C. One of the two nonterminals can be eliminated, for instance C, and then every occurrence of C is replaced by A, as follows:

$$\begin{cases} S \rightarrow a\,A\,A \mid a\,A \mid a \mid A\,A \mid A \mid \varepsilon \\ A \rightarrow A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid a \\ D \rightarrow F\,D \mid F \\ F \rightarrow A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Now the copy sets are the following:

$$\begin{array}{lll} \mathit{copy}(S) = \{A\} & & \mathit{copy}(D) = \{F\} \\ \mathit{copy}(A) = \emptyset & & \mathit{copy}(F) = \emptyset \\ \mathit{copy}(C) = \emptyset & & \end{array}$$

Eliminate $\mathit{copy}(D) = \{F\}$:

$$\begin{cases} S \rightarrow a\,A\,A \mid a\,A \mid a \mid A\,A \mid A \mid \varepsilon \\ A \rightarrow A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid a \\ D \rightarrow F\,D \mid A\,b\,A \mid b \\ F \rightarrow A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

And finally eliminate $\mathit{copy}(S) = \{A\}$:

$$\begin{cases} S \rightarrow a\,A\,A \mid a\,A \mid a \mid A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid \varepsilon \\ A \rightarrow A\,A \mid a\,A\,b \mid a\,b \mid a\,b\,D\,a \mid a \\ D \rightarrow F\,D \mid A\,b\,A \mid b \\ F \rightarrow A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Thus the form G_c of grammar G without copy rules (and in reduced and non-nullable form) is the following (where for clarity alternatives are rearranged by decreasing length):

$$G_c \begin{cases} S \rightarrow a\,b\,D\,a \mid a\,A\,A \mid a\,A\,b \mid A\,A \mid a\,A \mid a\,b \mid a \mid \varepsilon \\ A \rightarrow a\,b\,D\,a \mid a\,A\,b \mid A\,A \mid a\,b \mid a \\ D \rightarrow A\,b\,A \mid F\,D \mid b \\ F \rightarrow A\,b\,A \mid b \mid b\,A \mid A\,b \end{cases}$$

Notice that clearly grammar G_c contains some redundance as a side effect of its being free of copy rules, which may help considerably in compacting.

Transformation of rules into binary or terminal form:

Terminal rules. Define two new nonterminals X and Y, and use them to replace the occurrences of terminals a and b (only in the alternative rules that are not of terminal type themselves), as follows:

$$\begin{cases} S \rightarrow X\,Y\,D\,X \mid X\,A\,A \mid X\,A\,Y \mid A\,A \mid X\,A \mid X\,Y \mid a \mid \varepsilon \\ A \rightarrow X\,Y\,D\,X \mid X\,A\,Y \mid A\,A \mid X\,Y \mid a \\ D \rightarrow A\,Y\,A \mid F\,D \mid b \\ F \rightarrow A\,Y\,A \mid b \mid Y\,A \mid A\,Y \\ X \rightarrow a \\ Y \rightarrow b \end{cases}$$

Binary rules. Define four new nonterminals U and W_i $(i = 1, 2, 3)$ and related rules, as follows:

$$\begin{cases} S \rightarrow X\,U \mid X\,W_1 \mid X\,W_2 \mid A\,A \mid X\,A \mid X\,Y \mid a \mid \varepsilon \\ A \rightarrow X\,U \mid X\,W_2 \mid A\,A \mid X\,Y \mid a \\ D \rightarrow A\,W_3 \mid F\,D \mid b \\ F \rightarrow A\,W_3 \mid b \mid Y\,A \mid A\,Y \\ X \rightarrow a \\ Y \rightarrow b \\ U \rightarrow Y\,D\,X \\ W_1 \rightarrow A\,A \\ W_2 \rightarrow A\,Y \\ W_3 \rightarrow Y\,A \end{cases}$$

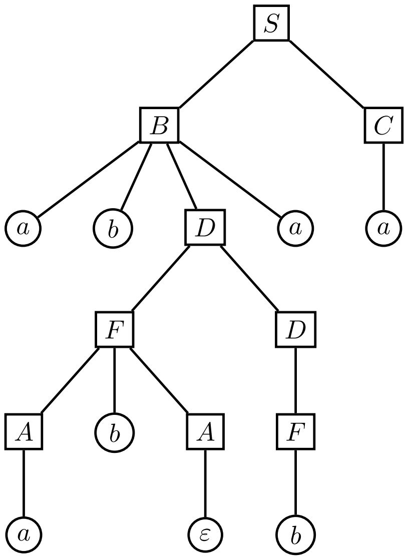
Finally, define one new nonterminal V and the related rule, as follows, and obtain the Chomsky normal form G' of grammar G:

$$G' \left\{ \begin{array}{lcl} S & \to & X\,U \mid X\,W_1 \mid X\,W_2 \mid A\,A \mid X\,A \mid X\,Y \mid a \mid \varepsilon \\ A & \to & X\,U \mid X\,W_2 \mid A\,A \mid X\,Y \mid a \\ D & \to & A\,W_3 \mid F\,D \mid b \\ F & \to & A\,W_3 \mid b \mid Y\,A \mid A\,Y \\ X & \to & a \\ Y & \to & b \\ U & \to & Y\,V \\ V & \to & D\,X \\ W_1 & \to & A\,A \\ W_2 & \to & A\,Y \\ W_3 & \to & Y\,A \end{array} \right.$$

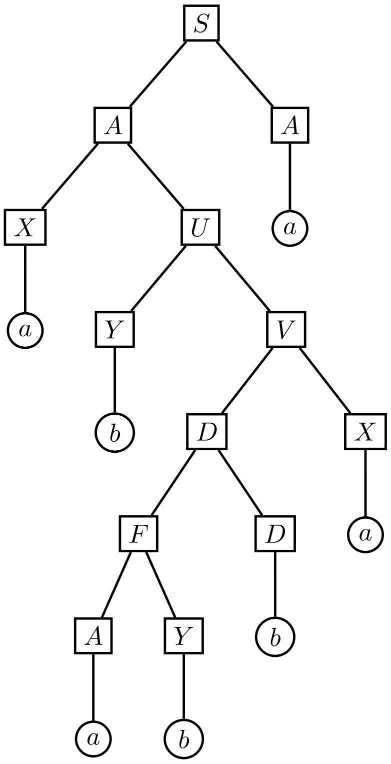
Here all the rules of grammar G' are of binary or terminal type, plus the axiomatic null rule $S \to \varepsilon$ as language $L(G)$ is not ε-free, while the original grammar G contains ternary and quaternary rules as well (and non-axiomatic null rules). Of course, G' may be highly redundant with respect to the original grammar G. □

Solution of (3)

Here the representative string $w = a\,b\,a\,b\,b\,a\,a$, which belongs to $L(G)$, is chosen and both syntax trees of w in the grammars G and the equivalent Chomsky normal form G' are shown:



syntax tree of G



syntax tree of G'

Notice that the syntax tree of grammar G' is deeper than that of grammar G, as a consequence of G' having only binary or terminal rules (and one null axiomatic rule). Of course the syntax tree of G' is strictly binary, as expected, while that of G has nodes of any arity from 1 to 4. □

3.2 Grammar Synthesis

Exercise 18 Consider the Dyck language over alphabet $\{a, b, c, d\}$, where a, c are open parentheses and b, d are the corresponding closed ones, respectively.

Answer the following questions:

1. Write two grammars G_1 and G_2 of languages L_1 and L_2, respectively, both not of extended type (BNF). Languages L_1 and L_2 are defined as follows:

   - L_1 = Dyck language with parentheses structures of depth at most 2

     Examples: ε $\quad a\,b$ $\quad c\,d$ $\quad a\,b\,c\,d$ $\quad a\,c\,d\,b$ $\quad a\,a\,b\,c\,d\,b$

     Counterexamples: $a\,a\,c\,d\,b\,b$ $\quad a\,a\,b\,c\,a\,b\,d\,b$

   - L_2 = Dyck language with parentheses structures of arbitrary depth, but not containing concatenated structures at any level

     Examples: ε $\quad a\,b$ $\quad c\,d$ $\quad a\,c\,d\,b$ $\quad a\,a\,c\,d\,b\,b$

     Counterexamples: $a\,a\,b\,c\,d\,b$ $\quad c\,d\,a\,b$

2. Write a grammar G of language L, not ambiguous and not of extended type (BNF), where L is defined as follows:

$$L = L_1 \cup L_2$$

that is, the union of languages L_1 and L_2.

Solution of (1)

Here are two grammars G_1 and G_2 that generate languages L_1 and L_2, respectively:

$$G_1 \left\{ \begin{array}{lcl} S & \rightarrow & a\,X\,b\,S \mid c\,X\,d\,S \mid \varepsilon \\ X & \rightarrow & a\,b\,X \mid c\,d\,X \mid \varepsilon \end{array} \right.$$

$$G_2 \left\{ \begin{array}{lcl} S & \rightarrow & a\,S\,b \mid c\,S\,d \mid \varepsilon \end{array} \right.$$

The two grammars G_1 and G_2 are obtained from the standard one of the Dyck language:

$$S \rightarrow a\,S\,b\,S \mid c\,S\,d\,S \mid \varepsilon$$

Restrictions are applied to the Dyck grammar to exclude certain types of derivation, as follows:

- grammar G_1 cannot generate parenthesis structures with nesting depth over 2, as nonterminal X does not generate nested structures but only concatenated ones
- grammar G_2 cannot concatenate parenthesis structures but only nest them, as it lacks the instance of nonterminal S concatenated to the right side of its rules

As both grammars G_1 and G_2 are obtained by restriction of the standard Dyck grammar, which is not ambiguous, they are not ambiguous either. The reader is left the task of checking the correctness of the two grammars, which however is immediate (try for instance with the previous examples and counterexamples). □

Solution of (2)

The union of languages L_1 and L_2 is ambiguous, as it generates in two ways the parenthesis structures of simple type, that is those without concatenated structures, of maximum nesting depth 2. The ambiguous phrases are the following:

$$\varepsilon \qquad a\,b \qquad c\,d \qquad a\,a\,b\,b \qquad a\,c\,d\,b \qquad c\,a\,b\,d \qquad c\,c\,d\,d$$

In order to unite languages L_1 and L_2 in a way that is not ambiguous, it is necessary to eliminate the double generation of the above listed ambiguous phrases (of which there are only finitely many). Here is a grammar G, not ambiguous, that generates the union language L (axiom S):

$$G = \left(V_N = \{S, S_1, X, S_2, Y_1, Y_2, Y_{\geq 3}\}, V_T = \{a, b, c, d\}, S, P \right)$$

$$G \left\{ \begin{array}{c} S \;\to\; S_1 \mid S_2 \\ \hline G_1 \left\{ \begin{array}{rcl} S_1 & \to & a\,X\,b\,S_1 \mid c\,X\,d\,S_1 \mid \varepsilon \\ X & \to & a\,b\,X \mid c\,d\,X \mid \varepsilon \end{array} \right. \\ \hline \begin{array}{c} G_2 \\ \text{modified} \end{array} \left\{ \begin{array}{rcl} S_2 & \to & a\,Y_1\,b \mid c\,Y_1\,d \\ Y_1 & \to & a\,Y_2\,b \mid c\,Y_2\,d \\ Y_2 & \to & a\,Y_{\geq 3}\,b \mid c\,Y_{\geq 3}\,d \\ Y_{\geq 3} & \to & a\,Y_{\geq 3}\,b \mid c\,Y_{\geq 3}\,d \mid \varepsilon \end{array} \right. \end{array} \right.$$

Essentially grammar G is the union of grammars G_1 and G_2, obtained by means of the standard modular union construction, which is the following:

- disjoin the nonterminal sets of G_1 and G_2 (unless they are already disjoint)
- unite axioms S_1 and S_2, and define a new axiom S of G

Here however grammar component G_2 is slightly modified so as to be forced to generate parenthesis structures at all similar to those generated by the original grammar G_1, but with nesting depth at least 3, not lower. In such a way the modified grammar component G_2 does not generate any more the phrases that are already generated by grammar component G_1 and therefore the union construction is not ambiguous.

In order to justify such a behaviour, it suffices to observe that in the modified grammar component G_2 a derivation cannot terminate before reaching nonterminal $Y_{\geq 3}$, which has an intentionally expressive name, and in this way it generates unavoidably at least three nested parenthesis structures (not of concatenated type). The reader is left the task of verifying such a behaviour (try for instance with the above listed ambiguous phrases). □

Observation

Of course one could choose to modify component grammar G_1 instead of G_2, perhaps with a few more complications. The reader can make some more exercise and design this second possible solution by himself.

Exercise 19 Consider a subset of the Dyck language over alphabet { '(', ')', '[', ']', '{', '}' }. Such a subset matches both the following constraints:

- the pairs of square brackets contain an even number of pairs of round brackets, at any nesting depth
- the pairs of graph brackets contain an odd number of pairs of round brackets, at any nesting depth

Here are a few sample correct strings:

$$(\,(\,(\,)\,(\,)\,)\,)\,(\,(\,(\,)\,)\,) \qquad (\,(\,[\,(\,)\,(\,)\,]\,)\,)\,\boldsymbol{\{}\,(\,(\,(\,)\,)\,)\,\boldsymbol{\}}$$

$$[\,(\,)\,(\,)\,]\,[\,(\,(\,)\,)\,] \qquad \boldsymbol{\{}\,[\,(\,\boldsymbol{\{}\,(\,(\,)\,(\,)\,)\,\boldsymbol{\}}\,)\,]\,(\,[\,(\,\boldsymbol{\{}\,(\,)\,\boldsymbol{\}}\,)\,]\,)\,\boldsymbol{\}}$$

And here are a few sample wrong strings:

$$[\,(\,)\,(\,)\,(\,)\,]\,\boldsymbol{\{}\,(\,(\,)\,)\,\boldsymbol{\}} \qquad \boldsymbol{\{}\,[\,(\,(\,(\,)\,(\,)\,)\,)\,]\,\boldsymbol{\}} \qquad [\,(\,\boldsymbol{\{}\,(\,)\,(\,)\,\boldsymbol{\}}\,)\,(\,)\,]$$

Answer the following questions:

1. Design a grammar G, not in extended form (BNF), that is not ambiguous and generates the above described language.
2. Explain informally (that is by words), concisely but effectively, how the previously designed grammar G works.

Solution of (1)

Here is a possible solution, not in extended form and not ambiguous, which is modeled as a restriction of the canonical not ambiguous grammar of the Dyck language (axiom S):

$$G \begin{cases} S \;\to\; \text{‘[’}\, E\, \text{‘]’}\, S \;\mid\; \text{‘\{’}\, O\, \text{‘\}’}\, S \;\mid\; \text{‘(’}\, S\, \text{‘)’}\, S \;\mid\; \varepsilon \\ E \;\to\; \text{‘[’}\, E\, \text{‘]’}\, E \;\mid\; \text{‘\{’}\, O\, \text{‘\}’}\, O \;\mid\; \text{‘(’}\, E\, \text{‘)’}\, O \;\mid\; \text{‘(’}\, O\, \text{‘)’}\, E \;\mid\; \varepsilon \\ O \;\to\; \text{‘[’}\, E\, \text{‘]’}\, O \;\mid\; \text{‘\{’}\, O\, \text{‘\}’}\, E \;\mid\; \text{‘(’}\, E\, \text{‘)’}\, E \;\mid\; \text{‘(’}\, O\, \text{‘)’}\, O \end{cases}$$

Grammar G is presented in an ordered form, to put into evidence the similarities and differences among rules. Notice in particular the following:

- the use of syntax classes E (even) and O (odd) in the pairs of square and graph brackets, respectively
- the use of axiom S in the initial rule, to generate pairs of round brackets not nested inside of square or graph brackets
- the swap of classes E and O on switching between the second and third rule

In answering to question (2) (see below), a more extensive and detailed comment to the structure of grammar G is given. □

Solution of (2)

The basic idea is that syntax classes (nonterminals) E and O model the generation of an even and odd number of pairs of round brackets, respectively. All the rules are specialised versions of the foundamental (not ambiguous) Dyck rule, that is:

$$S \to \text{‘('}\ S\ \text{‘)'}\ S \mid \varepsilon$$

where axiom S is replaced by the combinations of nonterminals E and O that preserve the even or odd parity of the number of generated round bracket pairs, respectively. For instance this happens in the rule $O \to$ ‘(’ E ‘)’ E, as here it holds “odd = even + 1 + even”, and so on (obviously the empty string ε indicates the absence of brackets and is supposed to be 0 that is even). Instead square and graph brackets, which do not contribute to the counting of brackets (as only round bracket pairs are counted), contain only the acceptable nonterminal (E or O respectively) in agreement with the parity constraints given by the specification.

At the beginning the outer round bracket nests are unconstrained and therefore are generated directly by axiom S. As the basic Dyck rule is not ambiguous and grammar G is simply obtained as a restriction thereof (restriction excludes some derivations), grammar G is not ambiguous either as restriction cannot cause ambiguity by itself. □

Exercise 20 A list x contains any number of elements e, separated by character g; this means that x is modeled as $(e^+\ g)^*\ e^+$. For example:

$$x = \underbrace{e\ e\ e\ e}_{\text{group } G_1}\ g\ \underbrace{e}_{\text{group } G_2}\ g\ \underbrace{e\ e}_{\text{group } G_3}\ g\ \underbrace{e\ e\ e}_{\text{group } G_4}\ g\ \underbrace{e\ e\ e}_{\text{group } G_5}$$

Name “group G_i” ($i \geq 1$) the i^{th} sublist of elements e included between two consecutive g characters. A group may not be empty (see above). As the sample string x shows, prefix G_1 and suffix G_5 are considered groups as well.

Such a list is a phrase of language L if and only of it contains two groups G_i and G_j ($1 \leq i < j$), not necessarily consecutive ($j - i \geq 1$), such that G_i is shorter than or equal to G_j, that is the inequality $|G_i| \leq |G_j|$ holds.

The sample string x is valid and is a phrase of language L, because if one poses for instance $i = 2$ and $j = 4$ then group G_2 is shorter than group G_4, which suffices to validate x. On the other side, the following string y:

$$y = \underbrace{e\ e\ e}_{\text{group } G_1}\ g\ \underbrace{e\ e}_{\text{group } G_2}\ g\ \underbrace{e}_{\text{group } G_3}$$

is invalid and is not a phrase of L.

Answer the following questions:

1. Write a grammar G, not in extended form (BNF), that generates language L, and draw the syntax tree of the sample phrase x shown above.

2. Examine whether the designed grammar G is ambiguous or not.

Solution of (1)

In order to satisfy the above stated condition, it is evident that the phrases of language L have to contain at least two (not empty) groups. First the complete solution is given, then it is justified in a progressive way. Here is the complete grammar G (axiom S):

$$\begin{aligned} S &\rightarrow U\, e\, M\, e\, V \\ M &\rightarrow e\, M\, e \mid M\, e \mid g\, U \quad (\text{or } V\, g) \\ U &\rightarrow U\, E\, g \mid \varepsilon \\ V &\rightarrow g\, E\, V \mid \varepsilon \\ E &\rightarrow e\, E \mid e \end{aligned}$$

Notice soon the existence of a recursive self-embedding rule ($M \rightarrow e\ M\ e$), clearly dedicated to generating the two groups the presence of which validates the correctness of the string.

And here it follows the justification of the proposed grammar G. The following two rules, with axiom S:

$$\begin{aligned} S &\rightarrow e\, M\, e \\ M &\rightarrow e\, M\, e \mid M\, e \mid g \end{aligned}$$

generate two consecutive (not empty) groups that satisfy condition $|G_i| \leq |G_j|$. The shortest valid string generated by S is $e\, g\, e$. Such a string satisfies the condition in a minimal way, by assuming $i = 1$ and $j = 2$, and observing that $|G_1| = |G_2| = 1$. Therefore in general axiom S generates the valid strings of type $e^h\, g\, e^k$, with $1 \leq h \leq k$.

Then consider the two nonterminals U and V, which are expanded by the following three rules (notice that such rules are essentially linear and hence regular):

$$\begin{aligned} U &\rightarrow U\, E\, g \mid \varepsilon \\ V &\rightarrow g\, E\, V \mid \varepsilon \\ E &\rightarrow e\, E \mid e \end{aligned}$$

Nonterminals U and V generate the strings of type $(\, e^+\, g\,)^*$ and $(\, g\, e^+\,)^*$, respectively. In general such strings are invalid (though they may be valid in some special cases).

Now, if nonterminals U and V are placed on the left and right side of the axiomatic rule, respectively, the effect is that of concatenating to the two crucial groups as many other groups (not empty) as one wishes:

$$S \rightarrow U\, e\, M\, e\, V$$

Hence S generates strings of type $(\, e^+\, g\,)^*\ e^h\, g\, e^k\ (\, g\, e^+\,)^*$, with $1 \leq h \leq k$, which are all valid.

Finally, by adding the following rule (which generates $g\ (e^+ g)^*$):

$$M \rightarrow g\, U$$

other (not empty) groups are inserted between the two crucial groups. Thus S generates valid strings of the following type:

$$\left(e^+ g \right)^* e^h g \left(e^+ g \right)^* e^k \left(g\, e^+ \right)^*$$

with $1 \leq h \leq k$.

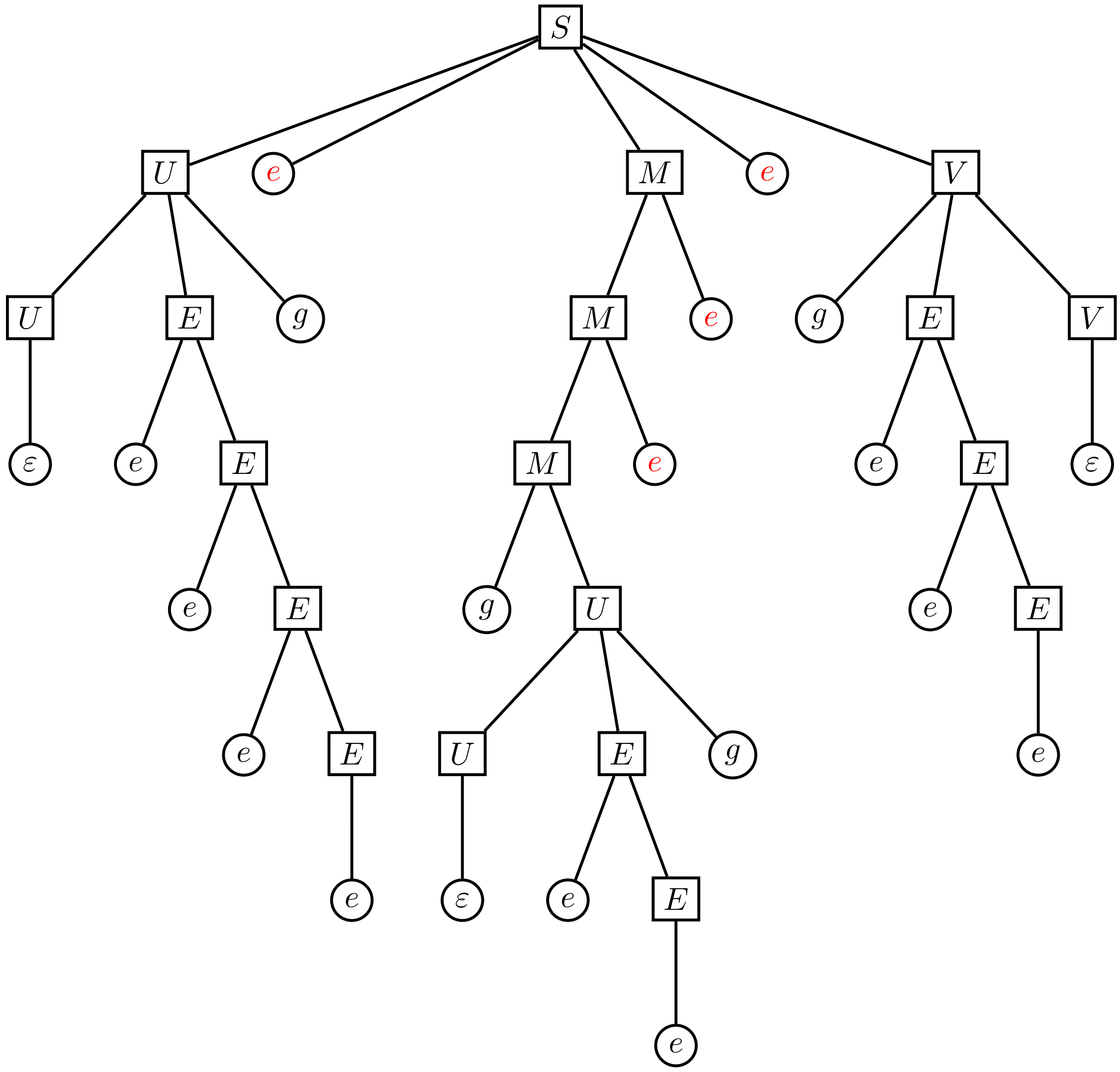
Rule $M \rightarrow V\ g$ would work as correctly as rule $M \rightarrow g\ U$ does, thus using either one is indifferent (but not both as redundancy would cause the grammar to be ambiguous). It is immediate to convince oneself that this is really the most general form of language L. Here it is, presented in a fully formal style:

$$L = \left\{ \left(e^+ g \right)^* e^h g \left(e^+ g \right)^* e^k \left(g\, e^+ \right)^* \mid 1 \leq h \leq k \right\}$$

Therefore language L is eventually complete. Exponent pair h, k and factor (or substring) $e^h\ g\ e^k$ represent the properly free (context-free) aspect of language L, while the rest is a sort of purely regular "fill".

The reader may wish to reconsider the given justification and read it backwards to have a methodology for progressively constructing the solution.

Here is the syntax tree of the sample valid string x shown above:



The crucial groups for the validity of string x are highlighted and coloured in red. One sees soon that the groups $e^h\, g \ldots e^k$ with exponents $h \leq k$ are generated by the two rules with right parts $e\ M\ e$ and $M\ e$, the former of which is necessarily self-embedding. □

Solution of (2)

The grammar G designed before is clearly ambiguous, as two groups G_i and G_j that satisfy the condition are certainly generated by nonterminal M, but other groups, also satisfying the condition and totally indistinguishable from those generated by M, might be generated by nonterminal U or V in a purely regular way. For instance string $e\,g\,e\,g\,e$ is ambiguous, because in the language L it can be modeled as $e^h\, g\, e^k\, g\, e$ or $e\, g\, e^h\, g\, e^k$, with $h = k = 1$.

Reasonably the language L itself is inherently ambiguous, as one may not prevent a grammar, able to mandatorily generate the two indispensable groups G_i and G_j, of generating at some time other groups that satisfy the condition as well, and that therefore are indistinguishable from the two mandatory ones. □

Exercise 21 Consider a language L_1 over alphabet $\Sigma = \{a, b, c\}$:

$$L_1 = (\, x\, c\,)^* \quad \text{where } x \in \Sigma^* \text{ and } |x|_a = |x|_b \geq 0 \text{ and } |x|_c = 0$$

which is exemplified by the following strings:

$$a\,b\,b\,a\,c\,b\,a\,c\,a\,a\,a\,b\,b\,b\,c \qquad\qquad a\,b\,a\,b\,b\,a\,c$$

and consider a regular language R_2 (again over alphabet Σ):

$$R_2 = \big(\, a^+\, b^+\, c\,\big)^*$$

Now, consider the language L_3 defined by the following intersection:

$$L_3 = L_1 \cap R_2$$

Answer the following questions:

1. Write all the phrases belonging to language L_3 with length less than or equal to 6.
2. Design a grammar G, not ambiguous and not in extended form (BNF), that generates language L_3, and draw the syntax tree of a phrase of length exactly 6.

Solution of (1)

Here are the phrases of language L of length ≤ 6:

$$\varepsilon \qquad a\,b\,c \qquad a^2\,b^2\,c \qquad a\,b\,c\,a\,b\,c$$

There are not any other phrases that satisfy the given criterion. □

Solution of (2)

One sees easily that the language L_3 defined by means of the above intersection can be defined (almost) formally as follows:

$$L_1 \cap R_2 = \{\, a^{n_1}\, b^{n_1}\, c\, a^{n_2}\, b^{n_2}\, c \ldots a^{n_k}\, b^{n_k}\, c \mid \qquad \forall\, k \geq 1 \quad \forall\, 1 \leq i \leq k \quad n_i \geq 1\,\}$$

or, if one wishes to be fully formal, as follows:

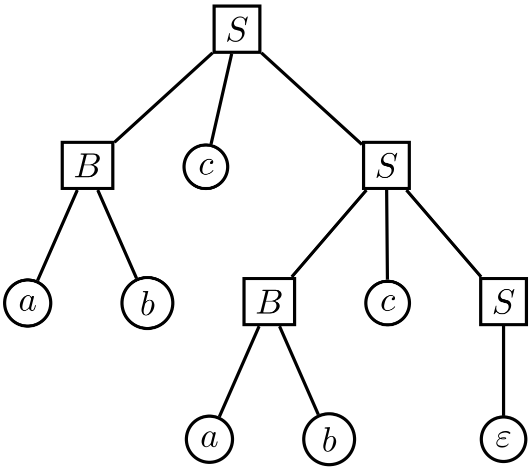
$$L_1 \cap R_2 = \left\{ x \mid \quad \forall\, k \geq 1 \quad \forall\, i\; 1 \leq i \leq k \quad \Rightarrow \quad n_i \geq 1 \;\wedge\; x = \prod_{i=1}^{k} a^{n_i}\, b^{n_i}\, c \right\}$$

where symbol $\prod$ expresses iterated concatenation.

Here is the solution grammar G that generates language L_3 (axiom S):

$$G \left\{ \begin{array}{lcl} S & \rightarrow & B\, c\, S \mid \varepsilon \\ B & \rightarrow & a\, B\, b \mid a\, b \end{array} \right.$$

And here is the syntax tree of the sample string $a\, b\, c\, a\, b\, c$:



Rules $S \rightarrow B\, c\, S \mid \varepsilon$ generate a sequence $B\, c\, B\, c\, \ldots\, B\, c$, while rules $B \rightarrow a\, B\, b \mid a\, b$ expand nonterminal B into the parenthesis structure $a^n\, b^n$ ($n \geq 1$). □

Exercise 22 Consider the following grammar G (axiom S):

$$G \left\{ \begin{array}{lcl} S & \rightarrow & a\, X \mid X \\ X & \rightarrow & a\, X\, b \mid b \mid \varepsilon \end{array} \right.$$

Answer the following questions:

1. Describe in a precise way, either informally (that is in english) or formally (that is by means of a logical formula or by defining a set), at choice, the strings of language $L(G)$.
2. Identify the ambiguous strings of language $L(G)$ (first give some examples and then try to characterize all of them) and show some syntax trees of such strings (a few expressive tree examples suffice).
3. Design a new grammar G', equivalent to grammar G but not ambiguous.

Solution of (1)

The alphabet is $\Sigma = \{a, b\}$. Here is the formal presentation of language $L(G)$:

$$L(G) = \left\{ x \mid \quad \exists\, h, k \geq 0 \quad x = a^h\, b^k \;\wedge\; 0 \leq |h - k| \leq 1 \right\}$$

One sees soon that essentially grammar G generates parenthesis structures of type $a^h\, b^k$, where the numbers of letters a and b do not differ of more than one from each other. In fact rules $X \to a\, X\, b \mid \varepsilon$ generate structures of type $a^n\, b^n$ $(n \geq 0)$, while rules $S \to a\, X$ and $X \to b$ add one more letter a and b at the beginning and end of the derivation, respectively. □

Solution of (2)

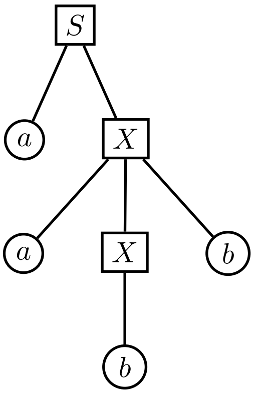
Grammar G is ambiguous, as the balanced parenthesis structures of type $a^n\, b^n$ $(n \geq 1)$ can be generated in two different ways. Here is an example with string $a\,a\,b\,b$ (that is $n = 2$):

$$S \overset{X \to a\, X}{\Longrightarrow} a\; X \overset{X \to a\, X\, b}{\Longrightarrow} a\; a\; X\; b \overset{X \to b}{\Longrightarrow} a\; a\; b\; b$$

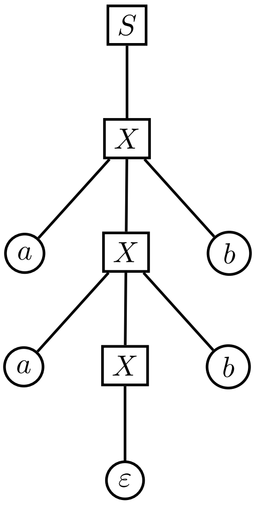
or

$$S \overset{S \to X}{\Longrightarrow} X \overset{X \to a\, X\, b}{\Longrightarrow} a\; X\; b \overset{X \to a\, X\, b}{\Longrightarrow} a\; a\; X\; b\; b \overset{X \to \varepsilon}{\Longrightarrow} a\; a\; b\; b$$

Here are the related syntax trees, which are at all obvious.



1^{st} derivation



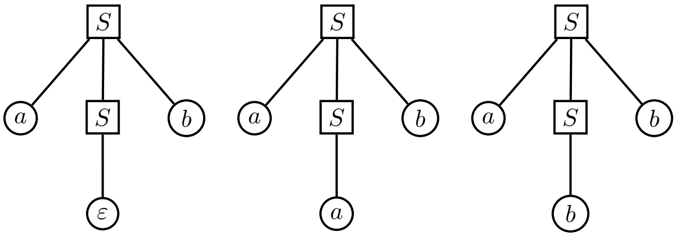
2^{nd} derivation

The difference between the two syntax trees is apparent: the former is asymmetric and generates the initial letter a by appending it directly to the root node, while the latter is symmetric but has a null leaf (namely the middle one). □

Solution of (3)

Since the ambiguity of grammar G depends on how G generates strings $a^h\, b^k$ when $h = k$ rather than $h \neq k$, to obtain a grammar G' equivalent to G, but not ambiguous, it suffices to separate the three cases: $h = k$, $h = k + 1$ and $h + 1 = k$ $(h, k \geq 0)$. A very simple version of G', which uses only one nonterminal, is the following:

$$G' \begin{cases} S \rightarrow a\,S\,b \\ S \rightarrow a \\ S \rightarrow b \\ S \rightarrow \varepsilon \end{cases}$$



string $a\,b$ string $a\,a\,b$ string $a\,b\,b$

In the grammar G' balanced parenthesis structures of type $a^n\,b^n$ $(n \geq 1)$ are generated in a unique way by using repeatedly rule $S \rightarrow a\,S\,b$ (and rule $S \rightarrow \varepsilon$ once at the end), while the possible additional letter a or b is inserted at the end by alternative rules $S \rightarrow a \mid b$, respectively (see the three sample syntax trees above). There may exist other versions of grammar G', more or less simple and compact. □

3.3 Technical Grammar

Exercise 23 Consider a grammar G, not ambiguous and in extended form (EBNF), that generates the if-then-else construct of the C programming language, with some simplifications. The following concepts should be modeled:

- there are variables and integer numbers, denoted as usual in the C language
- there are expressions, containing variables, integer numbers (positive or null), the addition "+" and multiplication "*" operators in infix form, and possibly subexpressions enclosed in round brackets "(" and ")"
- multiplication precedes addition
- there is the assignment statement:

  `var. = expr.;`

  always terminated by";"
- there is the block (that is the list) of assignment statements, not empty and enclosed in graph brackets "{" and "}"
- the graphs brackets are optional if the block contains only one assignment
- there are the relational operators "==", "!=", "<" and ">"
- the if condition is a relational expression:

  `(expr. rel.op. expr.)`

  enclosed in round brackets

- the then branch of the if construct is mandatory and introduces one block of assignments, as described above
- the else branch of the if construct is optional and introduces either one block of assignments, as described above, or another if-then-else construct

Example:

```
if (a + 2 < 10 * b) {
   a = c + 1;
   b = a - 2;
} else if (b == 5)
   a = b;
else {
   c = 0;
}
```

Design the above described grammar G (not ambiguous and in extended form).

Solution

The problem is somewhat standard and the reader could take inspiration from the official grammar of the C (or Pascal) language. However, here is an acceptable simple solution (axiom $\langle \mathsf{IF} \rangle$). Square brackets "[" and "]" are metasymbols representing an interval set:

$$
\begin{array}{rcl}
\langle \mathsf{VAR} \rangle & \rightarrow & [\text{`}\mathsf{A}\text{'} - \text{`}\mathsf{Z}\text{'}]\,([\text{`}\mathsf{A}\text{'} - \text{`}\mathsf{Z}\text{'}] \mid [\text{`}\mathsf{0}\text{'} - \text{`}\mathsf{9}\text{'}])^{*} \\
\langle \mathsf{INT} \rangle & \rightarrow & [\text{`}\mathsf{1}\text{'} - \text{`}\mathsf{9}\text{'}]\,[\text{`}\mathsf{0}\text{'} - \text{`}\mathsf{9}\text{'}]^{*} \\
\langle \mathsf{EXPR} \rangle & \rightarrow & \langle \mathsf{TERM} \rangle\ (\text{`}+\text{'}\ \langle \mathsf{TERM} \rangle)^{*} \\
\langle \mathsf{TERM} \rangle & \rightarrow & \langle \mathsf{FACT} \rangle\ (\text{`}*\text{'}\ \langle \mathsf{FACT} \rangle)^{*} \\
\langle \mathsf{FACT} \rangle & \rightarrow & \langle \mathsf{VAR} \rangle \mid \langle \mathsf{INT} \rangle \mid \text{`(' } \langle \mathsf{EXPR} \rangle \text{ `)'} \\
\langle \mathsf{ASS} \rangle & \rightarrow & \langle \mathsf{VAR} \rangle\ \text{`='}\ \langle \mathsf{EXPR} \rangle\ \text{`;'} \\
\langle \mathsf{BLOCK} \rangle & \rightarrow & \langle \mathsf{ASS} \rangle \mid \text{`\{'}\ \langle \mathsf{ASS} \rangle^{+}\ \text{`\}'} \\
\langle \mathsf{REL\_OP} \rangle & \rightarrow & \text{`=='} \mid \text{`!='} \mid \text{`<'} \mid \text{`>'} \\
\langle \mathsf{REL\_EXP} \rangle & \rightarrow & \text{`('}\ \langle \mathsf{EXPR} \rangle\ \langle \mathsf{REL\_OP} \rangle\ \langle \mathsf{EXPR} \rangle\ \text{`)'} \\
\langle \mathsf{IF} \rangle & \rightarrow & \text{`}\mathsf{if}\text{'}\ \langle \mathsf{REL\_EXP} \rangle\ \langle \mathsf{BLOCK} \rangle\ (\varepsilon \mid \text{`}\mathsf{else}\text{'}\,(\langle \mathsf{BLOCK} \rangle \mid \langle \mathsf{IF} \rangle))
\end{array}
$$

Grammar G is modeled in a modular way by using well known components: the grammar of arithmetic expressions (not ambiguous version), the list construct, and so on. All such components are not ambiguous, therefore reasonably grammar G is not ambiguous either. □

Exercise 24 Consider a simplified version of the query language SQL, defined as follows:

- There are attribute names and relation names (neither one has to be defined here). An attribute may be denoted by a simple name (an identifier) or a compound name (a pathname) with components separated by a dot".":

```
relation-name.attribute-name
```

- There are logical expressions with parentheses "(" and ")", consisting of the logical operators and and not, and of relational predicates.
- There are relational predicates, consisting of the comparison operator "equal-to" or "less-than", i.e. "==" or "<", and of two operands, namely:

```
term1  rel-op  term2
```

  Operator rel-op is comparison, operand term1 is an attribute, while operand term2 may be an attribute or a number (not to be defined here).

- There are the logical constants "T" and "F" (true and false respectively).
- The precedence order is: relational predicate precedes not, which precedes and.
- A single SQL query block must contain all the following clauses (orderly listed):
  - select, followed by:
    - either a list of attributes with separator ","
    - or the terminal "*", which indicates all the attributes
  - from, followed by a list of relations with separator ","
  - where, followed by a logical expression

  The query block may optionally end with the order-by clause, followed by one of the attributes listed in the select clause and a flag ("asc" or "desc") to sort the result of the query in ascending or descending order.

- A SQL query expression consists either of one query block or of two blocks connected by the set operator union or intersect (union or intersection of blocks).

Examples:

```
SELECT   PRODUCTS.NAME, TRANSFERS.PRICE, TRANSFERS.MOTIVATION
FROM     PRODUCTS, TRANSFERS
WHERE    PRODUCTS.PARTNUM == TRANSFERS.PARTNUM AND TRANSFERS.PRICE < 10
    UNION
SELECT   PRODUCTS.NAME, TRANSFERS.PRICE, TRANSFERS.MOTIVATION
FROM     PRODUCTS, TRANSFERS
WHERE    PRODUCTS.PARTNUM == TRANSFERS.PARTNUM AND PRODUCTS.PARTNUM < 20
```

and also

```
SELECT   *
FROM     PRODUCTS
WHERE    PARTNUM < 200
ORDER-BY NAME ASC
```

Answer the following questions:

1. Design a grammar, not ambiguous and of extended type (EBNF), to model the language of SQL query expressions defined before.
2. List the aspects of the proposed problem that cannot be modeled by means of a purely syntactic tool as a grammar in extended form.

Solution of (1)

Here is a grammar in extended form (EBNF) that generates the above described SQL expressions, where the axiom is ⟨QUERY_EXP⟩ and square brackets “[” and “]” indicate optionality:

```
      ⟨QUERY_EXP⟩ → ⟨QUERY_BLOCK⟩ [ ⟨SET_OPERATOR⟩ ⟨QUERY_BLOCK⟩ ]
   ⟨SET_OPERATOR⟩ → ‘UNION’ | ‘INTERSECT’
--------------------------------------------------------------------------
    ⟨QUERY_BLOCK⟩ → ⟨SELECT_CLAUSE⟩ ⟨FROM_CLAUSE⟩ ⟨WHERE_CLAUSE⟩
                    [ ⟨ORDER-BY_CLAUSE⟩ ]
  ⟨SELECT_CLAUSE⟩ → ‘SELECT’ ( ⟨ATTRIBUTE_LIST⟩ | ‘*’ )
    ⟨FROM_CLAUSE⟩ → ‘FROM’ ⟨RELATION_LIST⟩
   ⟨WHERE_CLAUSE⟩ → ‘WHERE’ ⟨LOGICAL_EXP⟩
⟨ORDER-BY_CLAUSE⟩ → ‘ORDER-BY’ ⟨ATTRIBUTE_PATH⟩ [ ‘ASC’ | ‘DESC’ ]
--------------------------------------------------------------------------
 ⟨ATTRIBUTE_LIST⟩ → ⟨ATTRIBUTE_PATH⟩ ( ‘,’ ⟨ATTRIBUTE_PATH⟩ )*
 ⟨ATTRIBUTE_PATH⟩ → [ ⟨RELATION_NAME⟩ ‘.’ ] ⟨ATTRIBUTE_NAME⟩
 ⟨ATTRIBUTE_NAME⟩ → . . .   (generic identifier)
--------------------------------------------------------------------------
  ⟨RELATION_LIST⟩ → ⟨RELATION_NAME⟩ ( ‘,’ ⟨RELATION_NAME⟩ )*
  ⟨RELATION_NAME⟩ → . . .   (generic identifier)
--------------------------------------------------------------------------
    ⟨LOGICAL_EXP⟩ → ⟨LOGICAL_EXP_1⟩ ( ‘AND’ ⟨LOGICAL_EXP_1⟩ )*
  ⟨LOGICAL_EXP_1⟩ → [ ‘NOT’ ] ⟨LOGICAL_EXP_2⟩
  ⟨LOGICAL_EXP_2⟩ → ‘T’ | ‘F’ | ⟨REL_PREDICATE⟩ | ‘(’ ⟨LOGICAL_EXP⟩ ‘)’
  ⟨REL_PREDICATE⟩ → ⟨TERM_1⟩ ⟨REL_OPERATOR⟩ ⟨TERM_2⟩
         ⟨TERM_1⟩ → ⟨ATTRIBUTE_PATH⟩
         ⟨TERM_2⟩ → ⟨ATTRIBUTE_PATH⟩ | ⟨CONST_VALUE⟩
   ⟨REL_OPERATOR⟩ → ‘==’ | ‘¡’
    ⟨CONST_VALUE⟩ → . . .   (generic number)
```

The logical expression is modeled in three levels: operator AND (minimum precedence), operator NOT (middle precedence) and relational predicate (maximum precedence). The whole grammar has a modular structure, as the organisation of the rules in stratified groups puts into evidence. Such an ordered arrangement, the careful choice of expressive names for nonterminal and the visible correspondence between rules and the specifications of the problem, justify sufficiently the correctness of the proposed grammar. □

Solution of (2)

Clearly it is impossible to model the following semantic aspects of the described simplified query language:

- matching between declaration and use of an attribute or relation name
- type compatibility of the attributes occurring as operands of a comparison relational operator (equal to or less than), as such attributes should be of numerical type
- type compatibility of the relations occurring in the query blocks that are united or intersected, as such blocks should have the same relations in the from clause
- that the attribute of the order-by clause is listed along with those of the select clasue

and possibly also other aspects, more or less specific, which the reader may wish to search and describe by himself (if any). □

Exercise 25 Consider a simplified subset of the C programming language, which exhibits the following features:

- a program is a purely sequential list (not empty) of assignment statements in C style; there are not any conditional or loop statements
- the assignment instruction has the following syntax:

  ```
  <variable> = <expression> ;
  ```

- on the left side of the assignment operator "=" there may be an identifier of:
  - a nominal variable, or
  - a pointer variable, preceded by the back-reference operator "*"

  and declaring such variables is not required here
- the expression may contain the following term types:
  - nominal variable
  - back-referenced pointer variable
  - function call
- the function call has the following syntax:

  ```
  <function-name> ( <parameter-list> )
  ```

  - the function name is an identifier (no need of declaring it)
  - the parameter is either a nominal or a pointer variable, possibly (but not necessarily) back-referenced
  - functions return an integer or a pointer value; the latter must be back-referenced
  - parameters are separated by “,” (comma)
  - the parameter list may be empty

- the operators admitted in the expression are “+” (addition) and “/” (division)

- the expression may contain round parentheses, to change the usual precedence rule of operators “+” and “/’ (division precedes addition)

- the back-reference operator “*” has the highest precedence

Here is a sample program (of three statements):

```
z = x / (y + f (z, w));

w = x / (y + *p (l, *m));

*n = y + x / g (z, *m, *i) + z;
```

Answer the following questions:

1. Design a grammar in extended form (EBNF) that is not ambiguous and models the above described simplified subset of the C language.

2. Specify what aspects of the proposed problem can not be modeled by a purely syntactic tool such as an extended (EBNF) grammar.

Solution of (1)

Here is a seemingly reasonable solution, in extended form (EBNF), where square brackets indicate optionality (axiom ⟨PROGRAM⟩):

$$\begin{array}{rcl}
\langle \text{PROGRAM} \rangle & \rightarrow & (\ \langle \text{ASSIGNMENT} \rangle\ \text{`;'}\ )^{+} \\
\hline
\langle \text{ASSIGNMENT} \rangle & \rightarrow & \langle \text{LEFT\_MEMBER} \rangle\ \text{`='}\ \langle \text{EXPRESSION} \rangle \\
\hline
\langle \text{LEFT\_MEMBER} \rangle & \rightarrow & \langle \text{NOMINAL\_VAR} \rangle \mid \text{`*'}\ \langle \text{POINTER\_VAR} \rangle \\
\langle \text{EXPRESSION} \rangle & \rightarrow & \langle \text{TERM} \rangle\ (\text{`+'}\ \langle \text{TERM} \rangle)^{*} \\
\langle \text{TERM} \rangle & \rightarrow & \langle \text{FACTOR} \rangle\ (\text{`/'}\ \langle \text{FACTOR} \rangle)^{*} \\
\langle \text{FACTOR} \rangle & \rightarrow & \langle \text{NOMINAL\_VAR} \rangle \mid \text{`*'}\ \langle \text{POINTER\_VAR} \rangle \mid \\
 & & \mid [\,\text{`*'}\,]\ \langle \text{FUNCTION\_CALL} \rangle \\
\hline
\langle \text{FUNCTION\_CALL} \rangle & \rightarrow & \text{id}\ \text{`('}\ \langle \text{PARAMETER\_LIST} \rangle\ \text{`)'} \\
\langle \text{PARAMETER\_LIST} \rangle & \rightarrow & \langle \text{PARAMETER} \rangle\ (\ \text{`,'}\ \langle \text{PARAMETER} \rangle\ )^{*} \\
\langle \text{PARAMETER} \rangle & \rightarrow & \langle \text{NOMINAL\_VAR} \rangle \mid [\,\text{`*'}\,]\ \langle \text{POINTER\_VAR} \rangle \\
\hline
\langle \text{NOMINAL\_VAR} \rangle & \rightarrow & \text{id} \qquad \text{- generic identifier} \\
\langle \text{POINTER\_VAR} \rangle & \rightarrow & \text{id} \qquad \text{- generic identifier}
\end{array}$$

Applying a back-reference operator to a function return value is optional, as the function can return directly an integer or a pointer and in the latter case the returned value has to be back-referenced. A function parameter may be an integer or a pointer, possibly back-referenced (but not necessarily). In accordance with the specification it is not permitted to pass a whole expression as an actual parameter in a function call.

The grammar has a modular and stratified structure, and the syntax classes have expressive names. Classes NOMINAL_VAR and POINTER_VAR could be unified, if one wishes, as both are expanded into a generic identifier. Of course there may exist different solutions, more or less equivalent to that given above. □

Solution of (2)

The following semantic aspects of the language cannot be modeled in a purely syntactic way:

- declaring a variable (of nominal or pointer type) before using it
- declaring a function before calling it and passing its parameters
- matching (in length and type) the list of formal and actual parameters in a function declaration and call, respectively (notice that here formal parameters are not declared either)
- distinguishing between a nominal and a pointer variable; in fact the above modeled separation of the two syntax classes NOMINAL_VAR and POINTER_VAR is clear and helps in making the grammar more readable, but is unuseful for the type checking in the expression (as well as for the type checking of formal and actual parameters in the

function), because both classes at last are expanded as the same class, that of a generic identifier "id", and therefore are syntactically indistinguishable

There may exist other semantic aspects that are not modeled syntactically, more or less detailed. The reader is left the task of finding them by himself (if any). □

Exercise 26 Consider a text document format in a style similar to LaTeX, slightly simplified. Such a format has the following structure:

- The document is a sequence (possibly empty) of structures, each of which is a dotted or numbered list of text items (text_item).
- The two types of structure are denoted as follows:

Dotted list:	Numbered list:
`%begin_itemize`	`%begin_enumerate`
`%item <text_item>`	`%item <text_item>`
`...`	`...`
`%end_itemize`	`%end_enumerate`

- Dotted and numbered lists may be nested into one another (even if they are of different type), at an arbitrary nesting depth.
- Each dotted or numbered list contains at least one text item (whether a simple text string or a nested substructure), or even two or more.
- The symbol ⟨text_item⟩ corresponds to a simple text string, denoted by the pure terminal c (even if it consists of some characters), or a substructure, that is a dotted or numbered list.

Example:

```
%begin_itemize

%item c

%item %begin_enumerate

%item c

%item c

%end_enumerate
```

```
%item c

%end_itemize

%begin_enumerate

...
```

Write a grammar in extended form (EBNF), not ambiguous, that generates the above described document format, and say whether it is deterministic.

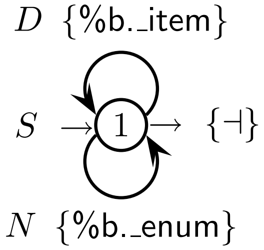
Solution

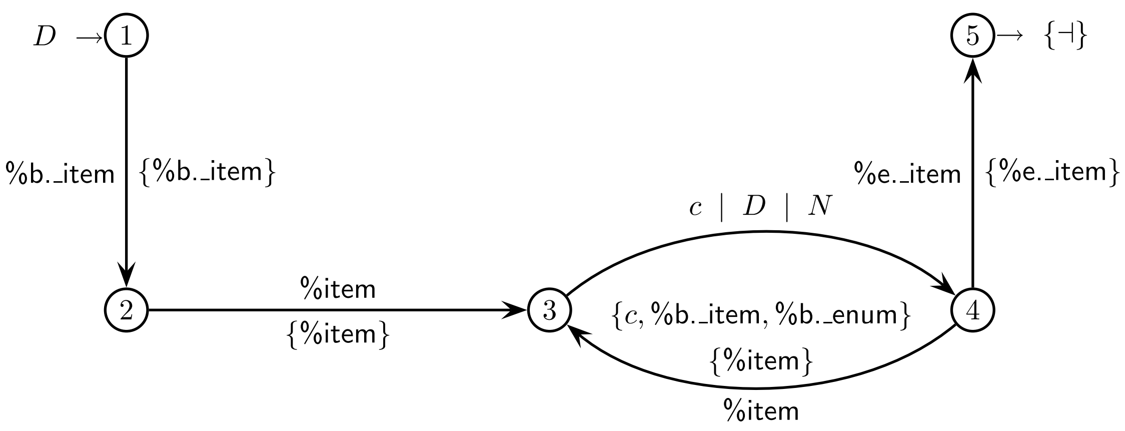
The nonterminals D and N are used here to denote the environments "dotted list" and "numbered list", respectively (in order to save space the keywords `\%begin_item`, etc, are somewhat shortened but are still clearly recognisable).

$$
\begin{array}{rcl}
S & \to & (\ D \mid N\ )^* \\
D & \to & \text{`\%b.\_item.'}\ \big(\ \text{`\%item'}\ (\ c \mid D \mid N\ )^+\ \big)^+\ \text{`\%e.\_item.'} \\
N & \to & \text{`\%b.\_enum.'}\ \big(\ \text{`\%item'}\ (\ c \mid D \mid N\ )^+\ \big)^+\ \text{`\%e.\_enum.'}
\end{array}
$$

The grammar above is in extended form (EBNF). It is easy to verify that such a grammar is of type $LL(1)$ and that for this reason it is certainly not ambiguous.

To verify the $LL(1)$ property of the grammar, it suffices for instance to express the grammar rules in the form of automata and to compute the lookahead sets of order one at every bifurcation point in the graph. Here are the automata that expand nonterminal S and D:





One proceeds in a similar way for the automaton (here omitted) that expands nonterminal N, as such an automaton is structurally analogous to that expanding nonterminal D. It is evident that the condition $LL(1)$ holds true.

Exercise 27

Given the two following languages L_1 and L_2, both over alphabet $\{a, b\}$:

$$\begin{aligned} L_1 &= \{ a^m b^n \mid 1 \leq m \leq n \} \\ L_2 &= \left\{ b^h a^k \mid 1 \leq h \leq k \right\} \end{aligned}$$

consider the following language L_3 (concatenation of L_1 and L_2):

$$L_3 = L_1 L_2$$

Answer the following questions:

1. List all the phrases of language L_3 in order of increasing length, up to and including those of length 5.
2. Write a grammar, not ambiguous and not in extended form (BNF), for language L_3, and draw the syntax tree of one at choice of the longest strings of those previously listed.

Solution of (1)

Here are the requested phrases of language L_3, of length ≤ 5 (how they decompose into factors of languages L_1 and L_2 is shown as well):

$$\underbrace{a\,b}_{L_1}\,\underbrace{b\,a}_{L_2} \qquad \underbrace{a\,b\,b}_{L_1}\,\underbrace{b\,a}_{L_2} \qquad \underbrace{a\,b}_{L_1}\,\underbrace{b\,a\,a}_{L_2}$$

There are not any other phrases of length ≤ 5. Notice that the phrases above decompose into factors of languages L_1 and L_2 in a unique way, hence not in an ambiguous way. □

Solution of (2)

First it is shown that the grammar of language L_3 obtained in the most straightforward way, that is by concatenating the axioms S_1 and S_2 of the grammars that generate the component languages L_1 and L_2, has an ambiguous behaviour.

<table>
<tr><th>Grammar of L_1</th><th>Grammar of L_2</th></tr>
<tr><td>$S_1 \to a\,S_1\,b$</td><td>$S_2 \to b\,S_2\,a$</td></tr>
<tr><td>$S_1 \to a\,b\,B$</td><td>$S_2 \to b\,a\,A$</td></tr>
<tr><td>$B \to b\,B$</td><td>$A \to a\,A$</td></tr>
<tr><td>$B \to \varepsilon$</td><td>$A \to \varepsilon$</td></tr>
<tr><td colspan="2">Grammar of L_3 (axiom (S_3)</td></tr>
<tr><td colspan="2">$S_3 \to S_1\,S_2$</td></tr>
</table>

An ambiguous phrase, of length 7, is the following one (here it is analysed and decomposed into syntax classes in two different ways):

$$\overbrace{a\,b}^{S_1}\overbrace{b\,b\,a\,a\,a}^{S_2} \qquad\qquad \overbrace{a\,b\,b}^{S_1}\overbrace{b\,a\,a\,a}^{S_2}$$

If one observes better the two given languages L_1 and L_2, ambiguity can be easily predicted: it is a classical case of ambiguity induced by the concatenation of two factor languages such that a prefix of the latter (the rightmost one) can be interpreted also as a suffix of the former (the leftmost one). In the example above the second letter b (from the left) can be viewed both as a prefix of L_2 and as a suffix of L_1.

In order to suppress the ambiguous behaviour, it suffices to make one of the two following choices: either that the strings that work both as a prefix and as a suffix, which are all of the type b^*, have to be intended as suffix of the former language, rather than as prefix of the latter one; or as prefix of the latter, rather than as suffix of the former. Here the former choice is made. Notice the following equality:

$$L_1 = \{\ a^m\,b^n \ \mid \quad 1 \leq m \leq n\ \} = \{\ a^m\,b^m \ \mid \quad m \geq 1\ \} \cdot b^*$$

The two languages to be concatenated are then the following:

$$K_1 \quad = \quad L_1 \cdot b^* = \{\ a^m\,b^m \ \mid \quad m \geq 1\ \} \cdot b^* \cdot b^* = L_1$$

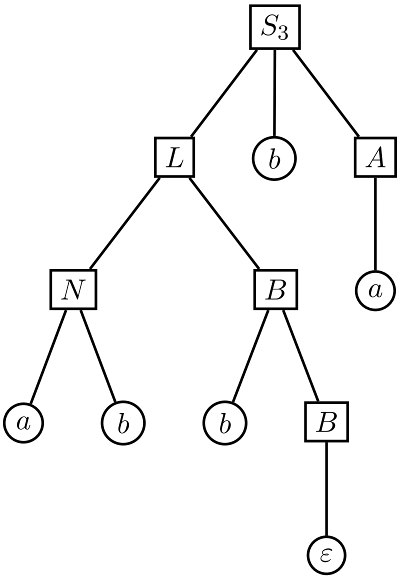
$$K_2 \quad = \quad \{\ b\,a^n \ \mid \quad n \geq 1\ \} = b\,a^+$$

and as an easy consequence the following equality holds:

$$L_3 = L_1\,b\,a^+$$

Thus the grammar G_3 of language L_3, which is not ambiguous, can be written almost immediately. Here it is (axiom S_3):

$$G_3 \left\{ \begin{array}{rcl} S_3 & \rightarrow & L\,b\,A \\ L & \rightarrow & N\,B \\ N & \rightarrow & a\,N\,b \\ N & \rightarrow & a\,b \\ B & \rightarrow & b\,B \\ B & \rightarrow & \varepsilon \\ A & \rightarrow & a\,A \\ A & \rightarrow & a \end{array} \right.$$



The nonterminals N, B and A generate languages $a^m\, b^m$ $(m \geq 1)$, b^* and a^+, respectively. Nonterminal L generates language L_1 by expanding into $N\,B$ and hence into $a^m\, b^m\, b^* = L_1$ $(m \geq 1)$. In conclusion axiom S_3 generates language L_3 in a way that is not ambiguous. Of course, such a solution is not the only possible one.

The syntax tree of the string (of length 5) $a\,b\,b\,b\,a$ is shown on the right side above. It demonstrates clearly that in the grammar G_3 the generation of the sequence of inner letters b is not affected by ambiguity, as all the letters b that equal in number the initial a letters are generated only by nonterminal N by means of a self-embedding rule of the type $N \rightarrow a\,N\,b \mid a\,b$, and as all the potentially exceeding letters b are generated by nonterminal B (but the last one which is fixed and is generated directly by axiom S_3) by means of a right-linear rule of type $B \rightarrow b\,B \mid \varepsilon$, hence a purely regular rule. □

Exercise 28 One wishes one designed a grammar that generates logical expressions with variables, with the binary operators of logical sum "or", logical product "and" and implication "⇒", with the unary operator "not", and with parentheses "(" and ")". The grammar must fulfill the following requirements:

- Variables are schematised by terminal v.
- The precedences of operators are as follows:
  - "not" precedes "and"
  - "and" precedes "or"
  - "or" precedes "⇒"
- Operator "⇒" is assumed to be of not associative type, that is if there are two or more concatenated operators "⇒", the evaluation order must be specified by means of parentheses.

- Operator “or” is assumed to be left-associative.
- Operator “and” is assumed to be right-associative.

Here is an example:

$$(\; v \; \Rightarrow \; v \;) \; \Rightarrow \; v \text{ ‘and’ } v$$

Answer the following questions:

1. Write the requested grammar, not ambiguous and not in extended form (BNF).
2. Draw the syntax tree of a string at choice that contains all the operators (the above example does not contain all the operators).

Solution of (1)

Here is the requested grammar, presented in a modular way (axiom ⟨EXPRESSION⟩):

⟨EXPRESSION⟩	→	⟨SUM_OF_PRODUCTS⟩ ‘⇒’ ⟨SUM_OF_PRODUCTS⟩
⟨EXPRESSION⟩	→	⟨SUM_OF_PRODUCTS⟩
⟨SUM_OF_PRODUCTS⟩	→	⟨SUM_OF_PRODUCTS⟩ ‘or’ ⟨TERM⟩
⟨SUM_OF_PRODUCTS⟩	→	⟨TERM⟩
⟨TERM⟩	→	⟨FACTOR⟩ ‘and’ ⟨TERM⟩
⟨TERM⟩	→	⟨FACTOR⟩
⟨FACTOR⟩	→	‘not’ ‘(’ ⟨EXPRESSION⟩ ‘)’
⟨FACTOR⟩	→	‘(’ ⟨EXPRESSION⟩ ‘)’
⟨FACTOR⟩	→	‘not’ v
⟨FACTOR⟩	→	v

The rules that expand syntax class EXPRESSION generate chains of two or more implication operators “⇒” and mandatorily specify the computation order by means of parentheses. In fact a writing like for instance $v \Rightarrow v \Rightarrow v$ is forbidden, as it would not specify clearly what the computation order should be; one must instead write either $(v \Rightarrow v) \Rightarrow v$ or $v \Rightarrow (v \Rightarrow v)$. Of course writing $v \Rightarrow v$ is permitted, as if there is only one operator associativity does not play any role.

The rules that expand syntax class SUM_OF_PRODUCTS generate the logical sum of logical product terms. Recursion is on the left side to model the left-associativity of the logical sum operator “or”.

The rules that expand syntax class TERM generate the logical product of factors. Recursion is on the right side to model the right-associativity of the logical product operator “and”.

The rules that expand syntax class FACTOR generate a variable or a parenthesised subexpression, possibly negated with the logical complement operator "not".

The rules are stratified: EXPRESSION expands into SUM_OF_PRODUCTS, which expands into TERM, which expands into FACTOR; this ensures the correct precedence order of operators as it is requested in the specification.

The whole grammar is modeled with reference to the standard non-extended grammar of arithmetic expressions, which is not ambiguous. Therefore also the grammar presented here reasonably is not ambiguous. □

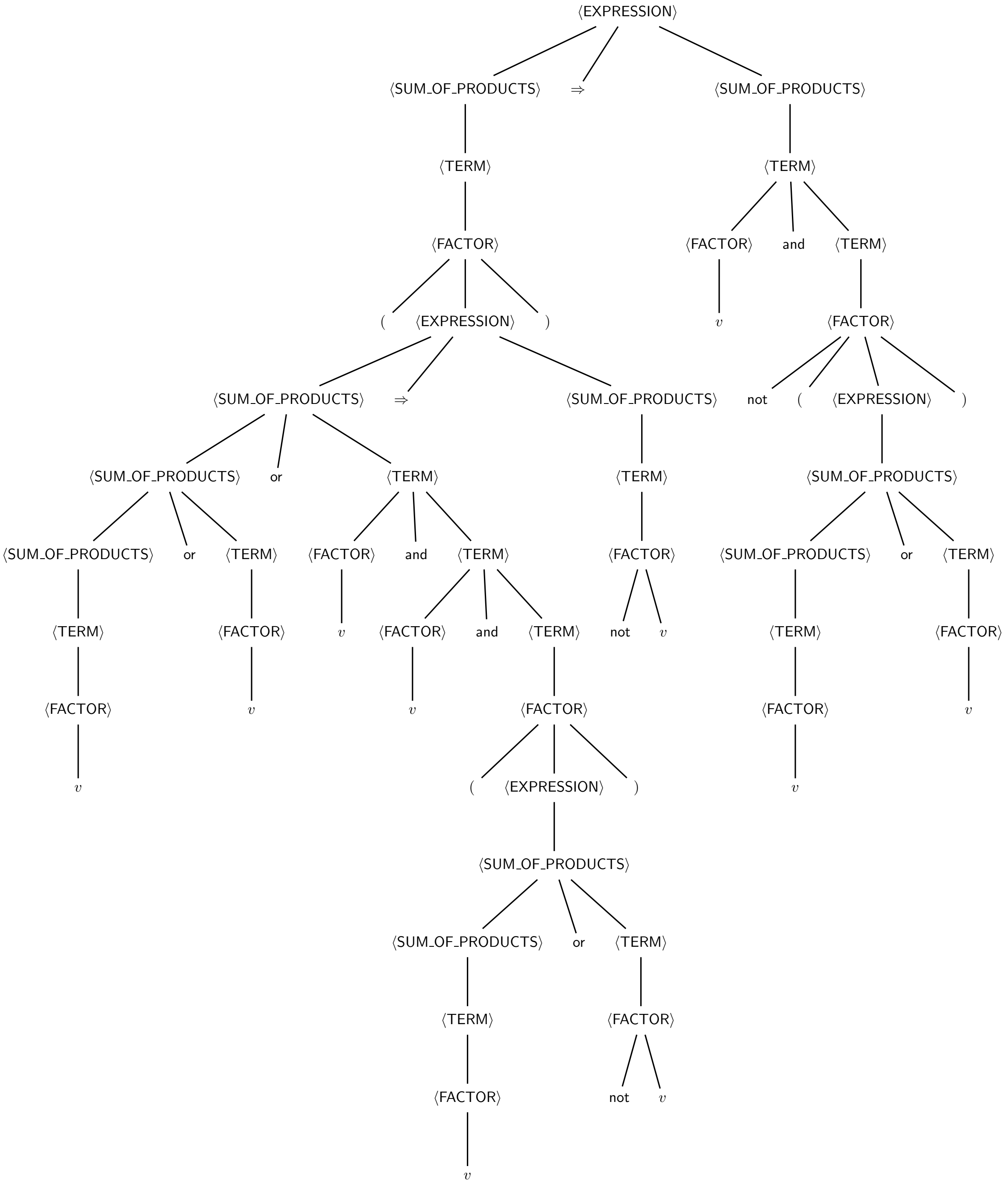
Solution of (2)

A sample string that involves all the operators is the following:

$$(v \ \mathsf{or} \ v \ \mathsf{or} \ v \ \mathsf{and} \ v \ \mathsf{and} \ (\ v \ \mathsf{or} \ \mathsf{not} \ v \ ) \ \Rightarrow \ \mathsf{not} \ v \ ) \ \Rightarrow \ v \ \mathsf{and} \ \mathsf{not} \ (v \ \mathsf{or} \ v \ )$$

Such a string calls in the associativity of both "or" and "and", and contains a chain of two implication operators that requires the usage of parentheses to specify the computation order.

Here is the syntax tree of the sample string shown above:


⟨EXPRESSION⟩
⟨SUM_OF_PRODUCTS⟩ ⇒ ⟨SUM_OF_PRODUCTS⟩
⟨TERM⟩ ⟨TERM⟩
⟨FACTOR⟩ ⟨FACTOR⟩ and ⟨TERM⟩
(⟨EXPRESSION⟩) v ⟨FACTOR⟩
⟨SUM_OF_PRODUCTS⟩ ⇒ ⟨SUM_OF_PRODUCTS⟩ not (⟨EXPRESSION⟩)
⟨SUM_OF_PRODUCTS⟩ or ⟨TERM⟩ ⟨TERM⟩ ⟨SUM_OF_PRODUCTS⟩
⟨SUM_OF_PRODUCTS⟩ or ⟨TERM⟩ ⟨FACTOR⟩ and ⟨TERM⟩ ⟨FACTOR⟩ ⟨SUM_OF_PRODUCTS⟩ or ⟨TERM⟩
⟨TERM⟩ ⟨FACTOR⟩ v ⟨FACTOR⟩ and ⟨TERM⟩ not v ⟨TERM⟩ ⟨FACTOR⟩
⟨FACTOR⟩ v v ⟨FACTOR⟩ ⟨FACTOR⟩ v
v (⟨EXPRESSION⟩) v
⟨SUM_OF_PRODUCTS⟩
⟨SUM_OF_PRODUCTS⟩ or ⟨TERM⟩
⟨TERM⟩ ⟨FACTOR⟩
⟨FACTOR⟩ not v
v

Notice the usage of parentheses for specifying the computation order of implication. □

Observation

By observing the subtrees of the syntax classes SUM_OF_PRODUCTS and TERM one sees easily the left and right associativity of the operators “or” and “and”, respectively.

The sample expression contains a chain of two implications, of type $\dots \Rightarrow \dots \Rightarrow \dots$. At the tree root it is possible to observe that the right implication has a premise, which constitutes the left implication, enclosed in parentheses, hence to be evaluated after the right implication. Therefore the tree specifies a computation order of type $(\dots \Rightarrow \dots) \Rightarrow \dots$. The left implication is elementary (both its premise and consequence do not contain any other implication), hence it does not need parentheses mandatorily (although these may be specified optionally).

Exercise 29 Consider the language L of the programs consisting of (at least) one or more assignment statements, separated and terminated by semicolon ';', with expressions containing only addition and without parentheses. Here follows a sample program:

$$
\begin{array}{lcl}
\hline
i & = & i + i + i + i; \\
i & = & i; \\
i & = & i + i + i + i + i; \\
\hline
\end{array}
$$

With reference to this language L, add the possibility for the program to contain one (and no more than one) syntax error. There are two error types to be considered:

omission of the assignment symbol $=$

omission of the addition symbol $+$

Each of the two following sample programs orderly contains exactly one such syntax error (of either type):

$$
\begin{array}{lcl}
\hline
i & = & i + i + i + i; \\
i & = & i; \\
i & & i + i + i + i + i + i; \\
\hline
i & = & i \; i + i + i; \\
\hline
\end{array}
$$

Now, define language L_F to be the set of all the correct programs and of those that contain exactly one error of either type (but not of both types):

$$
L_F = L \cup \text{ programs that contain exactly one error}
$$

The following sample program does not belong to language L_F, as it contains more than one error:

$$
\begin{array}{lcl}
\hline
i & = & i + i + i \; i; \\
i & = & i; \\
i & & i + i + i + i + i + i; \\
\hline
\end{array}
$$

Answer the following questions:

1. Write an EBNF (extended) grammar G, not ambiguous, that generates language L.
2. Write an EBNF (extended) grammar G_F, not ambiguous, that generates language L_F.
3. Explain why the written grammar G_F of language L_F ensures that the generated string contains at most one syntax error.

Solution of (1)

The solution is presented in a modular way. The extended (EBNF) grammar G is given, which generates only correct programs. Here it is (axiom PROG - pedex 'corr.' stays for "correct"):

$$G \begin{cases} \langle \mathsf{PROG} \rangle_{\text{corr.}} & \rightarrow \ (\langle \mathsf{ASS} \rangle_{\text{corr.}} \ \text{';'} \ )^+ \\ \langle \mathsf{ASS} \rangle_{\text{corr.}} & \rightarrow \ i \ \text{'='} \ \langle \mathsf{EXPR} \rangle_{\text{corr.}} \\ \langle \mathsf{EXPR} \rangle_{\text{corr.}} & \rightarrow \ i \ (\text{'+'} \ i)^* \end{cases}$$

As this grammar is very similar to the usual extended (EBNF) grammar of arithmetic expressions (notice that it is not recursive either), it is not ambiguous. □

Solution of (2)

First the extended (EBNF) grammar that omits only one assignment operator is given (it includes some of the rules of the previous grammar - such rules here are not repeated). Here it is (axiom $\mathsf{PROG}_{\text{without} =}$):

$$\begin{cases} \langle \mathsf{PROG} \rangle_{\text{without} =} & \rightarrow \ (\langle \mathsf{ASS} \rangle_{\text{corr.}} \ \text{';'} \ )^* \ \langle \mathsf{ASS} \rangle_{\text{without} =} \ \text{';'} \ (\langle \mathsf{ASS} \rangle_{\text{corr.}} \ \text{';'} \ )^* \\ \langle \mathsf{ASS} \rangle_{\text{without} =} & \rightarrow \ i \ \langle \mathsf{EXPR} \rangle_{\text{corr.}} \end{cases}$$

The rule that expands the assignment syntax class $\mathsf{ASS}_{\text{without} =}$ does not contain symbol "=". As this grammar is simply a regular extension of the previous one, reasonably it is not ambiguous.

Then the extended (EBNF) grammar that omits only one addition operator "+" is given (it includes some of the rules of the previous grammar - such rules here are not repeated). Here it is (axiom $\mathsf{PROG}_{\text{without} +}$):

$$\begin{cases} \langle \mathsf{PROG} \rangle_{\text{without} +} & \rightarrow \ (\langle \mathsf{ASS} \rangle_{\text{corr.}} \ \text{';'} \ )^* \ \langle \mathsf{ASS} \rangle_{\text{without} +} \ \text{';'} \ (\langle \mathsf{ASS} \rangle_{\text{corr.}} \ \text{';'} \ )^* \\ \langle \mathsf{ASS} \rangle_{\text{without} +} & \rightarrow \ i \ \text{'='} \ \langle \mathsf{EXPR} \rangle_{\text{without} +} \\ \langle \mathsf{EXPR} \rangle_{\text{without} +} & \rightarrow \ i \ (\text{'+'} \ i)^* \ i \ (\text{'+'} \ i)^* \end{cases}$$

The rule that expands the expression syntax class $\mathsf{EXPR}_{\text{without} +}$ forces the presence of exactly two variables i without the infix operator "+". This grammar is not ambiguous as well (for the same reason as before).

Finally, it suffices to unite the three grammars (axiom $\langle \mathsf{PROG} \rangle$). As the three languages are disjoint, union is not ambiguous.

$$\langle \mathsf{PROG} \rangle \rightarrow \langle \mathsf{PROG} \rangle_{\text{corr.}} \mid \langle \mathsf{PROG} \rangle_{\text{without} =} \mid \langle \mathsf{PROG} \rangle_{\text{without} +}$$

Here is the complete formulation of grammar G_F:

$$G_F \left\{ \begin{array}{rcl}
\langle \mathsf{PROG} \rangle & \rightarrow & \langle \mathsf{PROG} \rangle_{\text{corr.}} \mid \langle \mathsf{PROG} \rangle_{\text{without} =} \mid \langle \mathsf{PROG} \rangle_{\text{without} +} \\
\hline
\langle \mathsf{PROG} \rangle_{\text{corr.}} & \rightarrow & (\, \langle \mathsf{ASS} \rangle_{\text{corr.}} \text{ ‘;’ })^{+} \\
\langle \mathsf{ASS} \rangle_{\text{corr.}} & \rightarrow & i \text{ ‘=’ } \langle \mathsf{EXPR} \rangle_{\text{corr.}} \\
\langle \mathsf{EXPR} \rangle_{\text{corr.}} & \rightarrow & i \ (\text{ ‘+’ } i\,)^{*} \\
\hline
\langle \mathsf{PROG} \rangle_{\text{without} =} & \rightarrow & (\, \langle \mathsf{ASS} \rangle_{\text{corr.}} \text{ ‘;’ })^{*} \ \langle \mathsf{ASS} \rangle_{\text{without} =} \text{ ‘;’ } (\, \langle \mathsf{ASS} \rangle_{\text{corr.}} \text{ ‘;’ })^{*} \\
\langle \mathsf{ASS} \rangle_{\text{without} =} & \rightarrow & i \ \langle \mathsf{EXPR} \rangle_{\text{corr.}} \\
\hline
\langle \mathsf{PROG} \rangle_{\text{without} +} & \rightarrow & (\, \langle \mathsf{ASS} \rangle_{\text{corr.}} \text{ ‘;’ })^{*} \ \langle \mathsf{ASS} \rangle_{\text{without} +} \text{ ‘;’ } (\, \langle \mathsf{ASS} \rangle_{\text{corr.}} \text{ ‘;’ })^{*} \\
\langle \mathsf{ASS} \rangle_{\text{without} +} & \rightarrow & i \text{ ‘=’ } \langle \mathsf{EXPR} \rangle_{\text{without} +} \\
\langle \mathsf{EXPR} \rangle_{\text{without} +} & \rightarrow & i \ (\text{ ‘+’ } i\,)^{*} \ i \ (\text{ ‘+’ } i\,)^{*}
\end{array} \right.$$

The modular structure of grammar G_F is put into evidence by dividing the rules into groups. By the way, notice that language L_F is purely regular. There may exist more compact solutions, obtainable by unifying or factoring some rules. □

Solution of (3)

The whole grammar G_F of language L_F is not ambiguous by construction. The explanation is essentially implicit in the reasoning of the previous point: one starts from a base grammar that is not ambiguous and applies only transformations and adjunctions of rules that do not cause ambiguity. □

3.4 Pushdown Automaton

Exercise 30 Suppose the following language L over alphabet $\{a, b, c\}$ is given:

$$L = \{a^n\, b^n \mid \quad n \geq 1\} \cup \left\{a^n\, c^{2n} \mid \quad n \geq 1\right\}$$

Language L is formulated as the union of two components.
Answer the following questions:

1. Design a pushdown automaton A that recognises language L by final state, no matter whether indeterministic or deterministic.
2. If necessary, design a deterministic pushdown automaton A' that recognises language L by final state.
3. Repeat points 1 and 2, but suppose that the two automata recognise by empty stack.

In the deterministic cases, one is free of using a terminator for the input string.

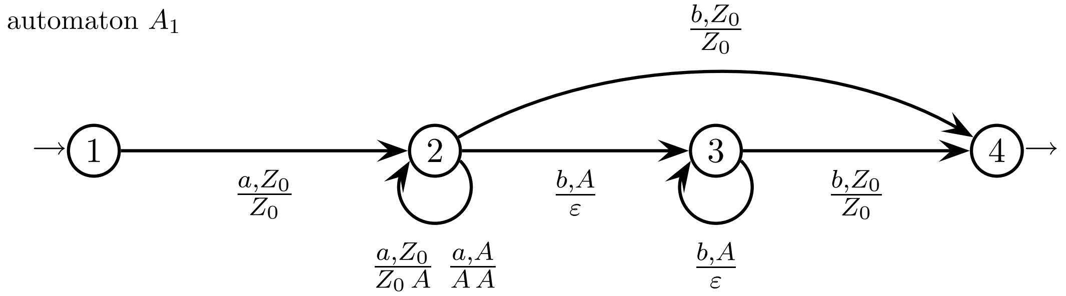
Solution of (1)

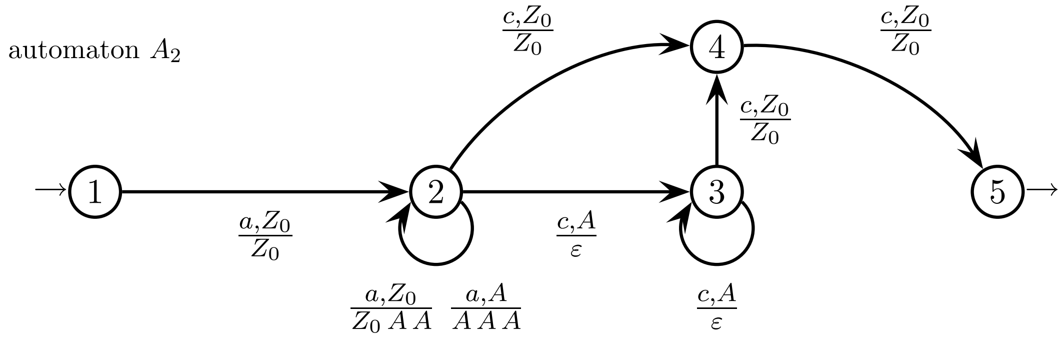
Language L is formulated as the union of two component languages L_1 and L_2, as follows:

$$L_1 = \{a^n\, b^n \mid \quad n \geq 1\} \qquad\qquad L_2 = \left\{a^n\, c^{2n} \mid \quad n \geq 1\right\}$$

Therefore it is natural first to design two pushdown automata A_1 and A_2 that recognise separately L_1 and L_2, respectively, and then to unite A_1 and A_2. In general uniting two pushdown automata yields an indeterministic result (even if both are deterministic), but here determinism is not required.

The two component automata A_1 and A_2 are simple variations of the well known pushdown automaton that recognises (by final state) the structure $a^n\, b^n$ $(n \geq 1)$. Here they are:

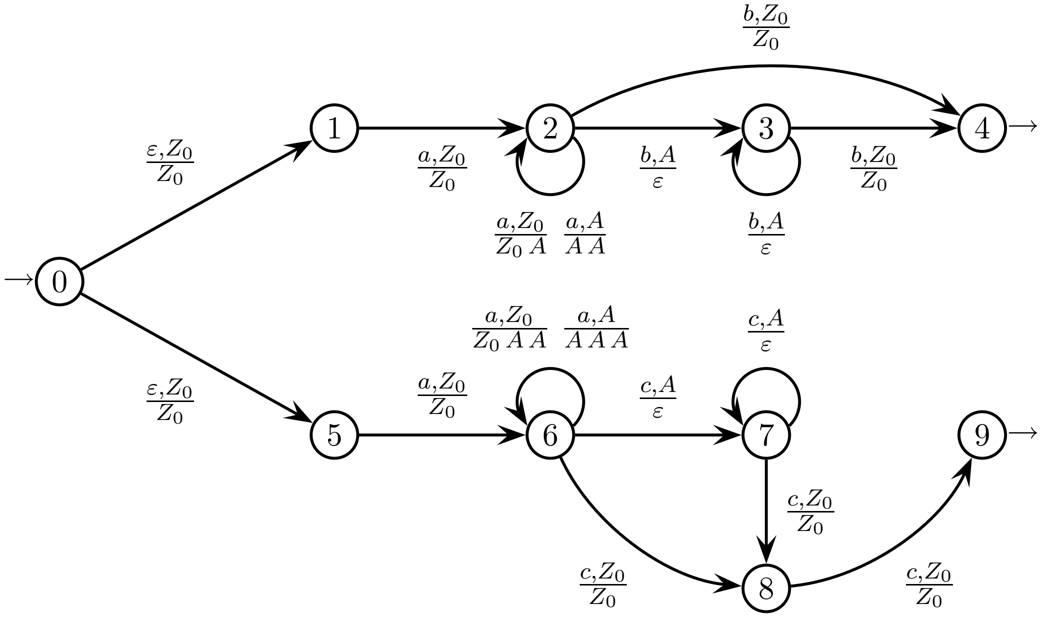

automaton A_1
1
2
3
4
$\frac{a,Z_0}{Z_0}$
$\frac{a,Z_0}{Z_0\,A}$ $\frac{a,A}{A\,A}$
$\frac{b,A}{\varepsilon}$
$\frac{b,A}{\varepsilon}$
$\frac{b,Z_0}{Z_0}$
$\frac{b,Z_0}{Z_0}$


automaton A_2
1
2
3
4
5
$\frac{a,Z_0}{Z_0}$
$\frac{a,Z_0}{Z_0\,A\,A}$ $\frac{a,A}{A\,A\,A}$
$\frac{c,A}{\varepsilon}$
$\frac{c,A}{\varepsilon}$
$\frac{c,Z_0}{Z_0}$
$\frac{c,Z_0}{Z_0}$
$\frac{c,Z_0}{Z_0}$

Both automata A_1 and A_2 deal with the short strings $a\,b$ and $a\,b\,b$, respectively, as a special case (they do not use the stack). For longer strings, A_1 and A_2 push one stack symbol A and two stack symbols $A\,A$ for every input letter a (but for the initial one), respectively. Then they pop one stack symbol A for every input letter b or c (but for the last letter b or the last two letters c), respectively, and recognise by final state. Notice that both A_1 and A_2 are deterministic. The stack is empty when A_1 and A_2 reach their final states 4 and 5, respectively, but may be empty also in other states (e.g. 2, 3 and 4), therefore recognition is necessarily by final state.

To have the union automaton $A = A_1 \cup A_2$ that recognises language L, it suffices to disjoin the state names, include a new initial state and indeterministically unite automata A_1 and A_2 by means for instance of spontaneous transitions (ε-transition). Here is automaton A:

automaton A

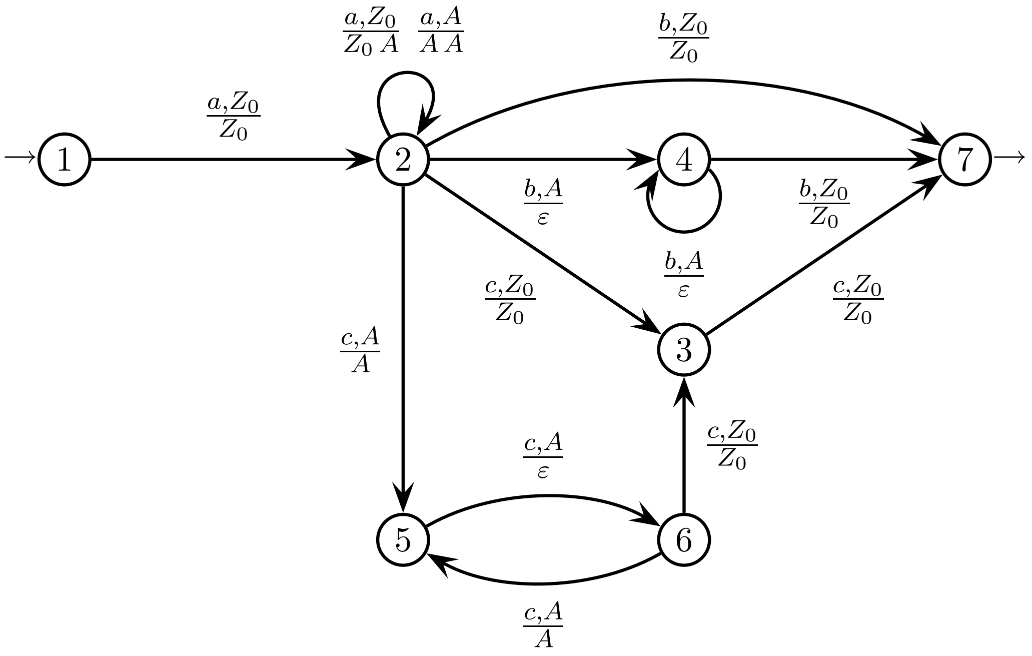

0
1
2
3
4
5
6
7
8
9
$\frac{\varepsilon, Z_0}{Z_0}$
$\frac{\varepsilon, Z_0}{Z_0}$
$\frac{a, Z_0}{Z_0}$
$\frac{a, Z_0}{Z_0 A}$ $\frac{a, A}{A A}$
$\frac{b, A}{\varepsilon}$
$\frac{b, A}{\varepsilon}$
$\frac{b, Z_0}{Z_0}$
$\frac{b, Z_0}{Z_0}$
$\frac{a, Z_0}{Z_0 A A}$ $\frac{a, A}{A A A}$
$\frac{a, Z_0}{Z_0}$
$\frac{c, A}{\varepsilon}$
$\frac{c, A}{\varepsilon}$
$\frac{c, Z_0}{Z_0}$
$\frac{c, Z_0}{Z_0}$
$\frac{c, Z_0}{Z_0}$

Automaton A makes initially an indeterministic choice about the type of input string it will have to recognise and then proceeds on the line it has selected. Recognition is by final state, for the same reasons as before. Of course other (indeterministic) solutions are possible. □

Solution of (2)

In order to have a deterministic automaton A' that recognises the whole language L, clearly one has to unify in some way the two initial series of moves of the previous automaton A that read the prefix a^n ($n \geq 1$) of the input string. One possibility consists of going on pushing anyway one stack symbol A for each input letter a (again but for the initial one), but of popping one stack symbol A for every letter b or every two letters c (again but for the last letter b or the last two letters c), respectively; the finite states of the automaton help to keep separate these two cases. Here is a possible solution, without resorting to the input string terminator:

automaton A'

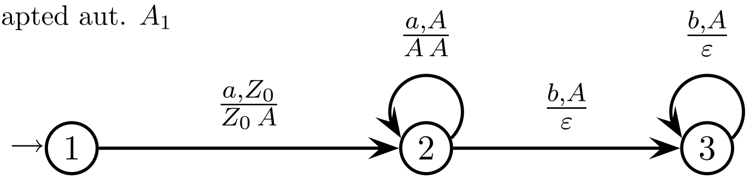


Automaton A' is clearly deterministic. The self-loop on the finite state 4 and the loop of length two on finite states 5 and 6, play the role of counting one letter b or two letters c for each stack symbol A, respectively. Also in this case there may be other deterministic solutions. □

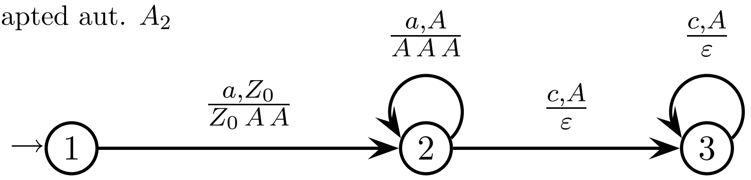
Solution of (3)

In order to have a version of automaton A that recognises by empty stack, one can follow the same approach as in point (1) but adapt component automata A_1 and A_2 to make them recognise by empty stack. Here is a possible adaptation:

adapted aut. A_1

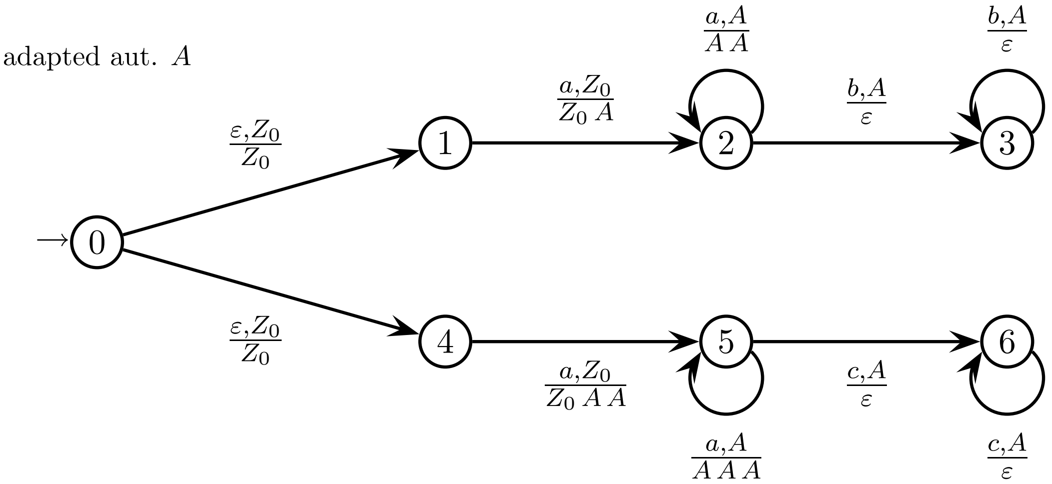


adapted aut. A_2


$\frac{a,A}{A\,A\,A}$
$\frac{c,A}{\varepsilon}$
$\frac{a,Z_0}{Z_0\,A\,A}$
$\frac{c,A}{\varepsilon}$
1
2
3

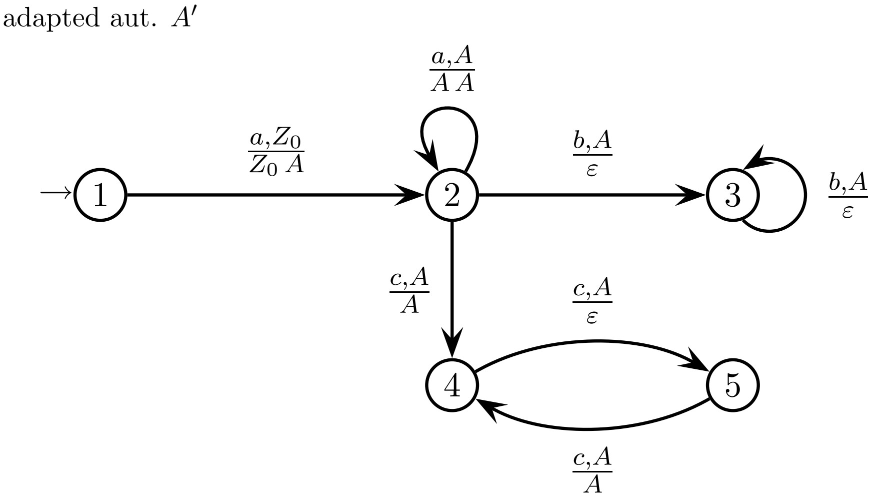
There is not any final state, as recognition is by empty stack. The stack gets empty sooner or later when both component automata are looping in state 3.

To have the union automaton $A = A_1 \cup A_2$ that recognises language L, one proceeds as before with an indeterministic union and obtains what follows:


adapted aut. A
$\frac{\varepsilon, Z_0}{Z_0}$
$\frac{\varepsilon, Z_0}{Z_0}$
$\frac{a, Z_0}{Z_0\ A}$
$\frac{a, A}{A\ A}$
$\frac{b, A}{\varepsilon}$
$\frac{b, A}{\varepsilon}$
$\frac{a, Z_0}{Z_0\ A\ A}$
$\frac{a, A}{A\ A\ A}$
$\frac{c, A}{\varepsilon}$
$\frac{c, A}{\varepsilon}$
0
1
2
3
4
5
6

Automaton A is indeterministic and recognises by empty stack. It looks like simpler and more symmetric than the version recognising by final state, as language L is more naturally suited to such a type of recognition (other languages may behave differently).

It is also possible to adapt the deterministic automaton A' of point (2) to recognise by empty stack, as follows (again without input string terminator):



The idea behind the construction is the same as before: always push one stack symbol A for each input letter a and pop one stack symbol A for every letter b or every two letters c, respectively. Self-loop 3 and loop 4 5 (of length two) count letters b and c, respectively. Sooner or later the stack gets empty in the states 3 or 5, depending on the input string, hence recognition is by empty stack. Automaton A' is deterministic, and is simpler and more symmetric than that recognising by final state (being the union of the two simple adapted automata A_1 and A_2). □

Exercise 31 Consider the following grammar G (axiom S):

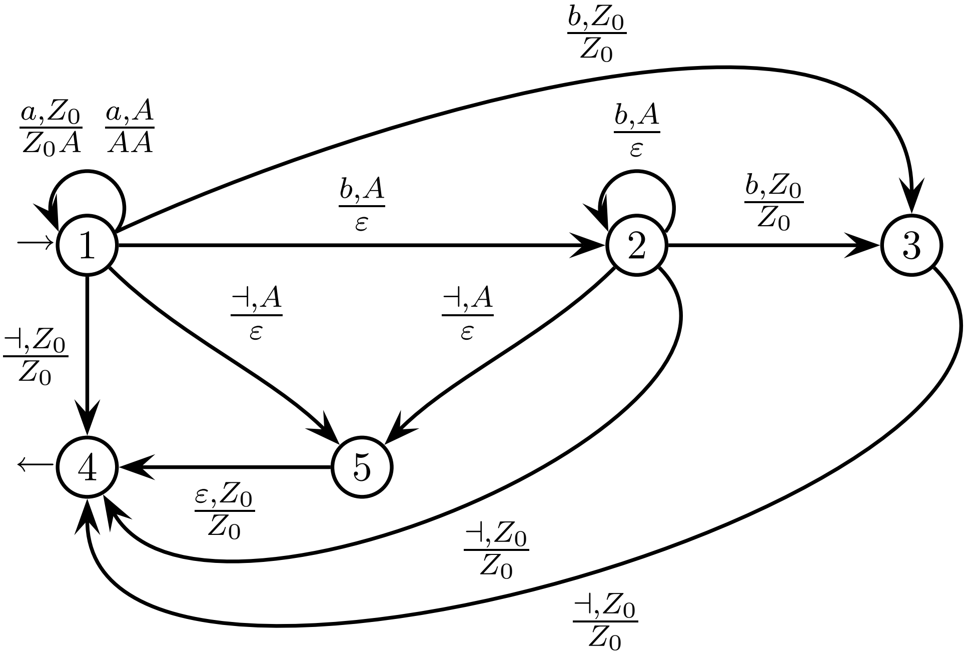
$$G \begin{cases} S \rightarrow a\,X \mid X \\ X \rightarrow a\,X\,b \mid b \mid \varepsilon \end{cases}$$

Answer the following questions:

1. Suppose that the string is terminated by an end-marker $\dashv$ and design a deterministic pushdown automaton A that recognizes language $L(G)$ by final state.
2. Give a formal proof that the pushdown automaton designed at point (1) is correct.
3. Suppose that the string does not have an end-marker and design a deterministic pushdown automaton A that recognizes language $L(G)$ by final state.
4. Give a formal proof that the pushdown automaton designed at point (3) is correct.

Solution of (1)

Here is the state-transition graph of a deterministic pushdown automaton A with five states that recognises language $L(G)$. The input string is supposed to be terminated by the end-marker $\dashv$ and recognition is by final state.



All the transitions labeled with letter a precede those labeled by letter b, hence the recognised input string must be of type $a^*\,b^*$. The automaton counts input letters a by pushing as many stack symbols A and matches such letters with the subsequent letters b by popping symbols A. Moreover one letter a or b is admitted in excess. Recognition occurs only in the final state 4, where however the stack is empty. This justifies sufficiently the correctness of the automaton.

Language $L(G)$ may be deterministically recognisable by empty stack instead of final state. The reader is left the task of investigating this alternative solution. □

Solution of (2)

For completeness, here a more formal proof of the correctness of automaton A is given. First a few accepting computations will help to understand how the automaton works. Here they are:

- accepts string $\varepsilon\ \dashv$ with computation

$$\langle \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{\dashv, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle$$

- accepts string $a\ \dashv$ with computation

$$\langle a\ \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{a, Z_0}{Z_0 A}} \langle \dashv,\, 1,\, Z_0 A \rangle \xRightarrow{\frac{\dashv, A}{\varepsilon}} \langle \varepsilon,\, 5,\, Z_0 \rangle \xRightarrow{\frac{\varepsilon, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle$$

- accepts string $b\ \dashv$ with computation

$$\langle b\ \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{b, Z_0}{Z_0}} \langle \dashv,\, 3,\, Z_0 \rangle \xRightarrow{\frac{\dashv, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle$$

- accepts string $a\,b\ \dashv$ with computation

$$\langle a\,b\ \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{a, Z_0}{Z_0 A}} \langle b\ \dashv,\, 1,\, Z_0 A \rangle \xRightarrow{\frac{b, A}{\varepsilon}} \langle \dashv,\, 2,\, Z_0 \rangle \xRightarrow{\frac{\dashv, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle$$

- accepts string $a\,a\,b\ \dashv$ with computation

$$\begin{aligned}&\langle a\,a\,b\ \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{a, Z_0}{Z_0 A}} \langle a\,b\ \dashv,\, 1,\, Z_0 A \rangle \xRightarrow{\frac{a, A}{AA}} \langle b\ \dashv,\, 1,\, Z_0 AA \rangle \xRightarrow{\frac{b, A}{\varepsilon}} \\ &\langle \dashv,\, 2,\, Z_0 A \rangle \xRightarrow{\frac{\dashv, A}{\varepsilon}} \langle \varepsilon,\, 5,\, Z_0 \rangle \xRightarrow{\frac{\varepsilon, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle\end{aligned}$$

- accepts string $a\,b\,b\ \dashv$ with computation

$$\begin{aligned}&\langle a\,b\,b\ \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{a, Z_0}{Z_0 A}} \langle b\,b\ \dashv,\, 1,\, Z_0 A \rangle \xRightarrow{\frac{b, A}{\varepsilon}} \langle b\ \dashv,\, 2,\, Z_0 \rangle \xRightarrow{\frac{b, Z_0}{Z_0}} \\ &\langle \dashv,\, 3,\, Z_0 \rangle \xRightarrow{\frac{\dashv, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle\end{aligned}$$

- accepts string $a\,a\,b\,b\ \dashv$ with computation

$$\begin{aligned}&\langle a\,a\,b\,b\ \dashv,\, 1,\, Z_0 \rangle \xRightarrow{\frac{a, Z_0}{Z_0 A}} \langle a\,b\,b\ \dashv,\, 1,\, Z_0 A \rangle \xRightarrow{\frac{a, A}{AA}} \langle b\,b\ \dashv,\, 1,\, Z_0 AA \rangle \xRightarrow{\frac{b, A}{\varepsilon}} \\ &\langle b\ \dashv,\, 2,\, Z_0 A \rangle \xRightarrow{\frac{b, A}{\varepsilon}} \langle \dashv,\, 2,\, Z_0 \rangle \xRightarrow{\frac{\dashv, Z_0}{Z_0}} \langle \varepsilon,\, 4,\, Z_0 \rangle\end{aligned}$$

From the above computations it should be evident how automaton A performs the last move:

- if the number of letters b is one more than that of letters a, A runs transition $3 \xrightarrow{\frac{\dashv, Z_0}{Z_0}} 4$
- if the number of letters b is equal to that of letters a, A runs transition $2 \xrightarrow{\frac{\dashv, Z_0}{Z_0}} 4$
- if the number of letters b is one less than that of letters a, A runs transition $5 \xrightarrow{\frac{\varepsilon, Z_0}{Z_0}} 4$
- the three short acceptable strings $\varepsilon \dashv$, $a \dashv$ and $b \dashv$ are dealt with as exceptions

The above successful computations cover all the cases $a^h\, b^k \dashv$ with $0 \leq h, k \leq 2$ and $0 \leq |h - k| \leq 1$, and are sufficiently representative of the general case where $h, k > 2$. Therefore automaton A reasonably (if not at all surely) accepts all the strings of language $L(G)$.

On the other side, a computation of pushdown automaton A fails in the two following cases:

- if in the input string a letter b occurs before a letter a, because in the state-transition graph all the transitions labeled with a precede those labeled with b
- if the difference of the numbers of letters a and b is larger than one (that is if the input string is $a^h\, b^k \dashv$ but $|h - k| > 1$), because of the following reasons:
  - first notice that in the state-transition graph every transition labeled with a or b pushes or pops a stack symbol A, respectively, with the only exception of the two transitions labeled with b that enter state 3
  - now examine separately the two sub-cases:
    - if letters a exceed letters b, after reading all the letters b the stack still contains two or more symbols A, but it is impossible to pop more than one symbol A when letters b are finished (check the two transitions entering state 5), while all the final transitions (those that enter final state 4) want the stack to be empty
    - if letters b exceed letters a, the stack gets empty before letters b are finished, but it is impossible to read more than one letter b when the stack is already empty (check the two transitions entering state 3)

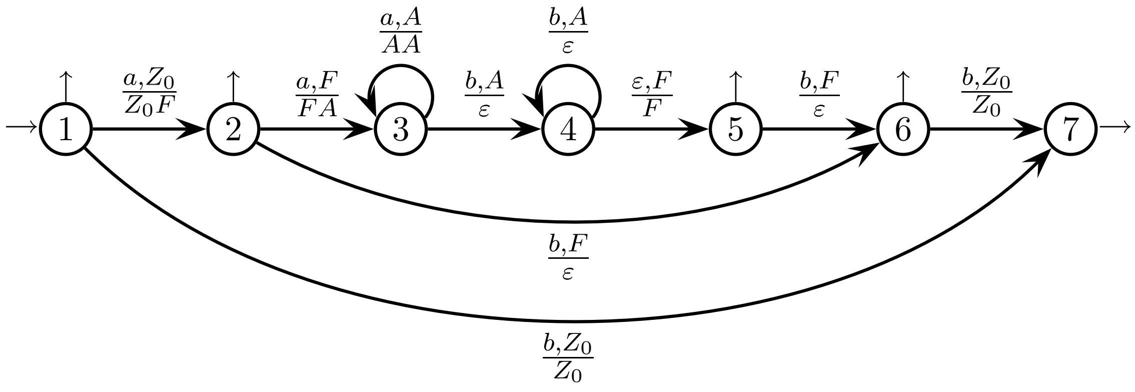
  hence in both sub-cases the computation is prevented of reaching final state 4

Uniting these two cases yields the set of all the strings that are not of type $a^h\, b^k$ $(h, k \geq 0)$ or that are of type $a^h\, b^k$ $(h, k \geq 0)$ but with $|h - k| > 1$, which is the complement of language $L(G)$. Therefore automaton A rejects all the strings that do not belong to $L(G)$.

Since automaton A accepts all the strings that belong to language $L(G)$ and rejects all those that do not, in conclusion it recognises exactly $L(G)$. □

Solution of (3)

Here is a deterministic pushdown automaton A' that recognises language $L(G)$ by final state, but without resorting to the end-marker $\dashv$:


1
2
3
4
5
6
7
$\frac{a,Z_0}{Z_0F}$
$\frac{a,F}{FA}$
$\frac{a,A}{AA}$
$\frac{b,A}{\varepsilon}$
$\frac{b,A}{\varepsilon}$
$\frac{\varepsilon,F}{F}$
$\frac{b,F}{\varepsilon}$
$\frac{b,Z_0}{Z_0}$
$\frac{b,F}{\varepsilon}$
$\frac{b,Z_0}{Z_0}$

The stack alphabet contains the two symbols F (which stays for "first") and A, which are used to count the first letter a (if any) and the subsequent letters a (if any) of the input string, respectively. There are seven states, two more than in the solution with end-marker.

The idea behind such a construction is that after reading a^n in the input, the stack contains $F\,A^{n-1}$ ($n \geq 1$) and the subsequent moves of automaton A' are arranged so as to match the letters a with the letters b that follow in the input tape, and to admit a difference of one. □

Solution of (4)

In order to analyse formally the behaviour of automaton A', it is convenient to specify for each state what type of input string leads to such state and what the stack contents are:

state	**final ?**	**input string**	**stack contents**	**condition**
1	yes	ε	Z_0	
2	yes	a	$Z_0\,F$	
3	no	a^h	$Z_0\,F\,A^{h-1}$	$h \geq 2$
4	no	$a^h\,b^k$	$Z_0\,F\,A^{h-1-k}$	$h \geq 2$ and $1 \leq k < h$
5	yes	$a^h\,b^k$	$Z_0\,F$	$h \geq 2$ and $k = h - 1$
6	yes	$a^h\,b^k$	Z_0	$h \geq 1$ and $k = h$
7	yes	$a^h\,b^k$	Z_0	$h \geq 0$ and $k = h + 1$

Supposed the input string is of type $a^h\,b^k$ ($h, k \geq 0$), based on such table the behaviour and meaning of the various final states of automaton A' are the following:

- state 1 accepts string ε, that is $h = k = 0$
- state 2 accepts string a, that is $h = 1$ and $k = 0$
- state 5 accepts all the strings $a^h\,b^k$ with $h \geq 2$ and $k = h - 1$
- state 6 accepts all the strings $a^h\,b^k$ with $h \geq 1$ and $k = h$
- state 7 accepts all the strings $a^h\,b^k$ with $h \geq 0$ and $k = h + 1$

Notice that when the computation reaches final states 2 and 5, the stack is not empty and contains precisely symbol F, as there is exactly one letter a in excess with respect to letters b.

Uniting all these cases yields the set of all the strings of type $a^h\, b^k$ with $h, k \geq 0$ and $|h - k| \leq 1$, which is language $L(G)$. Thus automaton A' accepts all the strings of $L(G)$.

On the other side, automaton A' rejects the input string in the following two cases:

- if a letter b precedes a letter a, because in the graph all the transitions labeled with a precede those labeled with b

- if the input string is of type $a^h\, b^k$ with $h, k \geq 0$ but $|h - k| > 1$ holds, because (split into two sub-cases depending on whether $h - k \gtrless 0$):

  - if $h > k + 1$ then the computation is still in the state 3 or 4 when the input string is finished, and neither such state is final
  - if $k > h + 1$ then the computation reaches state 7 before the input string is finished, and such a state does not have any outgoing arc to continue and finish the string

  hence in both sub-cases the string is rejected

Uniting these two cases yields the set of all the strings that are not of type $a^h\, b^k$ $(h, k \geq 0)$ or that are of type $a^h\, b^k$ $(h, k \geq 0)$ but with $|h - k| > 1$, which is the complement of language $L(G)$. Therefore automaton A' rejects all the strings that do not belong to $L(G)$.

Since automaton A' accepts all the strings that belong to language $L(G)$ and rejects all those that do not, in conclusion it recognises exactly $L(G)$.

As before, the reader may wish to investigate by himself whether it is possible to recognise deterministically language $L(G)$ by empty stack and without end-marker. □

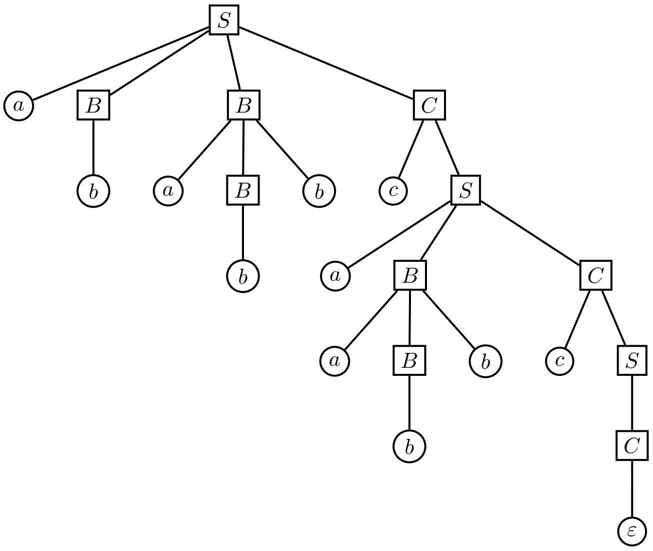
Chapter 4

Syntax Analysis

4.1 Recursive Descent

Exercise 32 Consider the following grammar G over alphabet $\{a, b, c\}$, in extended form (EBNF), with axiom S (a sample syntax tree is given aside):

$$G \begin{cases} S \rightarrow a\, B^*\, C \mid C \\ B \rightarrow a\, B\, b \mid b \\ C \rightarrow c\, S \mid \varepsilon \end{cases}$$



sample string: $a\, b\, a\, b\, b\, c\, a\, a\, b\, b\, c$

Answer the following questions:

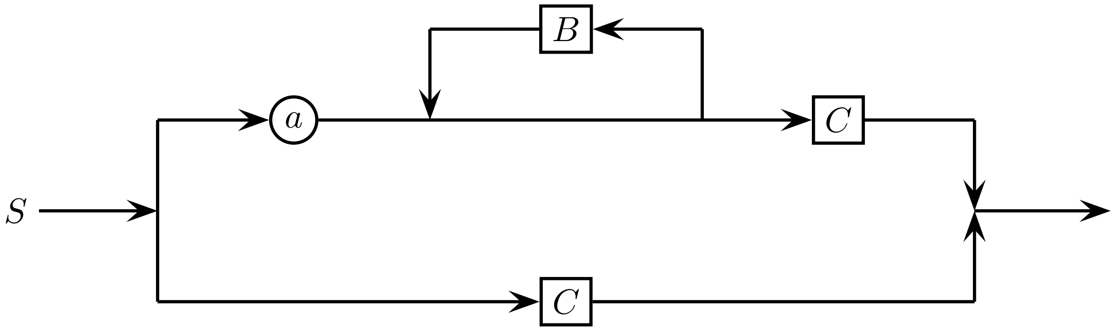
1. Represent grammar G in the form of syntax diagrams.

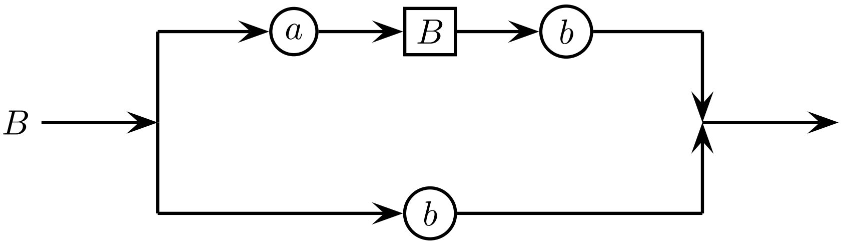
2. Represent grammar G as a network of recursive finite state automata.

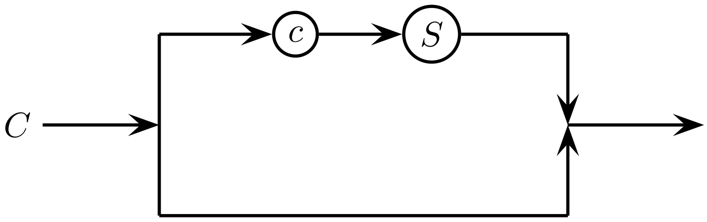
3. Determine whether grammar G is of type $LL(k)$, for some $k \geq 1$.
4. Show the sequence of (recursive) transitions that the automata network representing grammar G follows to recognise the sample string $a\,b\,a\,b\,b\,c\,a\,a\,b\,b\,c \in L(G)$ (the corresponding syntax tree is shown above).
5. Label the syntax diagrams of grammar G obtained before by means of the lookahead sets (for the value of k determined before).
6. Program (in pseudocode) the LL deterministic syntax analyser of language $L(G)$ (for the value of k determined before).
7. Show a simulation of the syntax analyser programmed before, for the sample string $a\,b\,a\,b\,b\,c\,a\,a\,b\,b\,c \in L(G)$ (the corresponding syntax tree is shown above).

Solution of (1)

Here are the three syntax diagrams that expand nonterminals S, B and C:



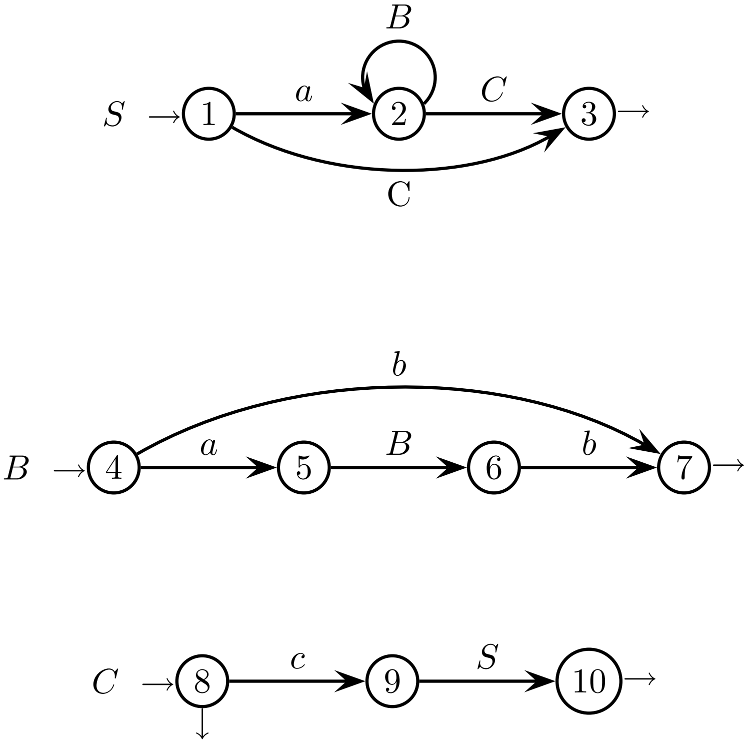




Of course, since grammar G is in extended form (EBNF) and contains a Kleene star operator, the syntax diagrams (namely that expanding axiom S) contain loops (here one self-loop). □

Solution of (2)

Here is the representation of grammar G as a network of recursive finite stata automata, deterministic over the total alphabet $\{a, b, c, B, C, S\}$ (union of terminals and nonterminals):

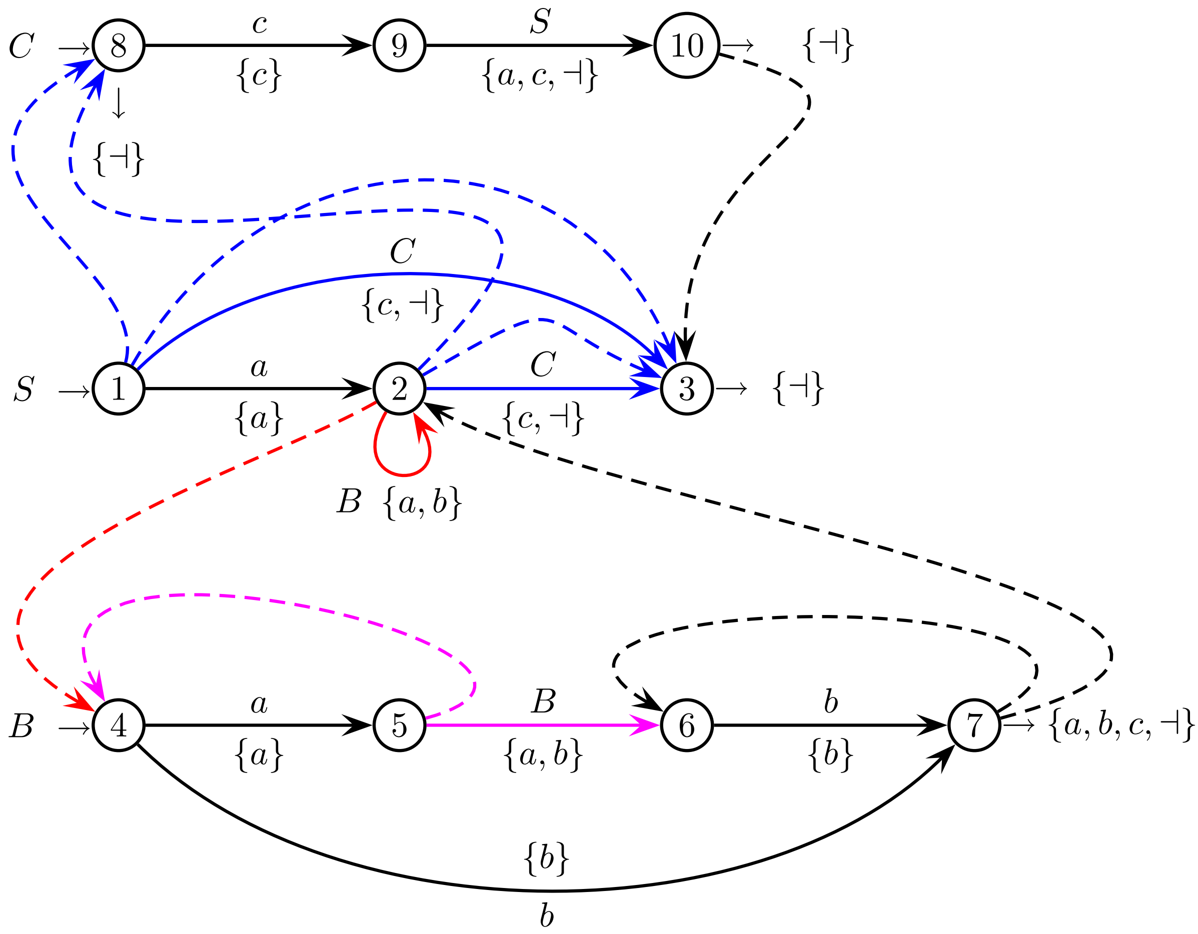


Of course, these state-transition graphs are the dual versions of the syntax diagrams shown before: arcs become nodes and nodes become arcs. Notice that the state 8 of the automaton that expands nonterminal C is both initial and final, due to the presence of the null alternative rule $C \to \varepsilon$. Since grammar G contains a Kleene star in the rule $S \to a\, B^*\, C$, the automaton expanding nonterminal S contains a loop (namely a self-loop). □

Solution of (3)

In order to check whether grammar G is of type $LL(k)$, for some $k \geq 1$, it is necessary to determine the lookahead sets at the bifurcation points in the recursive network of automata representing G, for each tested value of k from one onwards.

Here are the lookahead sets on all the arcs of the automata (not only at the bifurcation points), for $k = 1$. Suppose as usual that there is the additional axiomatic rule $S_0 \to S\ \dashv$, to have a terminator "⊣" at the end of each generated string (such a rule is however purely formal and is not represented as an automaton):



The dashed arcs do not belong to the automata, but are a visual facility here drawn to help the reader about how to determine the various lookahead sets. For better clarity such dashed arcs are coloured in the same way as the normal automaton transition the lookahead set of which they help to determine. More precisely, the following considerations apply:

- the red dashed arc 2 → 4 helps to compute the lookahead set of the normal transition $2 \xrightarrow{B} 2$ (actually a self-loop): imagine to replace nonterminal B by the automaton that expands B itself, hence follow 2 → 4 and go to the initial state 4 of automaton B

- the magenta dashed arc 5 → 4 helps to compute the lookahead set of the normal transition $5 \xrightarrow{B} 6$: imagine to replace nonterminal B by the automaton that expands B itself, hence follow 5 → 4 and go to the initial state 4 of automaton B (of course this will yield the same lookahead set as before)

- the blue dashed arc 2 → 8 helps to compute the lookahead set of the normal transition $2 \xrightarrow{C} 3$: imagine to replace nonterminal C by the automaton that expands C itself, hence follow 2 → 8 and go to the initial state 8 of automaton C

- the blue dashed arc $2 \rightarrow 3$ contributes more to the lookahead set of transition $2 \xrightarrow{C} 3$, because nonterminal C can expand to ε: imagine to skip C, to proceed with the transition(s) that follow C itself and hence to use the corresponding lookahead sets (here simply the final one)
- a similar reasoning applies for determining the lookahead set of the normal transition $1 \xrightarrow{C} 3$ (and the lookahead set is the same as for transition $2 \xrightarrow{C} 3$)
- moreover, as for the determination of the lookahead sets at the exit points of the automata, the following holds (here colours do not matter):
  - the dashed arcs $7 \rightarrow 2$ and $7 \rightarrow 6$ help to determine the lookahead set at the exit of the automaton that expands nonterminal B: imagine to go to the transitions that follow the two occurrences of B in the whole automaton network and use the related lookahead sets
  - the dashed arc $10 \rightarrow 3$ helps to determine the lookahead set at the exit of the automaton that expands nonterminal C: imagine to go to the transitions that follow the two occurrences of C in the whole automaton network and use the related lookahead set
- to conclude, first remember that by definition the lookahead set at the exit of the automaton that expands axiom S must contain the terminator, and then notice that since here S occurs recursively only at the end of the automaton that expands nonterminal C, the lookahead set does not contain anything else (but the terminator)

The remaining lookahead sets are associated with the transitions labeled by terminals, therefore they are immediate and do not deserve any particular explanation. Since the lookahead sets of all the arcs outgoing from each bifurcation point, namely states 2, 4 and 8, are disjoint of one another, grammar G is of type $LL(1)$ (i.e. it holds $k = 1$). □

Observation

If one wishes one can associate the lookahead sets (of order $k = 1$) with the alternative production rules of grammar G, as follows:

rule of G	**lookahead set** $(k = 1)$
$S \rightarrow a\, B^*\, C$	a
$S \rightarrow C$	$c\ \dashv$
$B \rightarrow a\, B\, b$	a
$B \rightarrow b$	b
$C \rightarrow c\, S$	c
$C \rightarrow \varepsilon$	$\dashv$

These lookahead sets are simply obtained by following on the automata the paths corresponding to the grammar rules. Of course one reobtains the same result as before: since all the alternative lookahead sets are disjoint of each other, grammar G is of type $LL(1)$.

Solution of (4)

Here is the (recursive) transition sequence to recognise the sample string $a\,b\,a\,b\,b\,c\,a\,a\,b\,b\,c$:

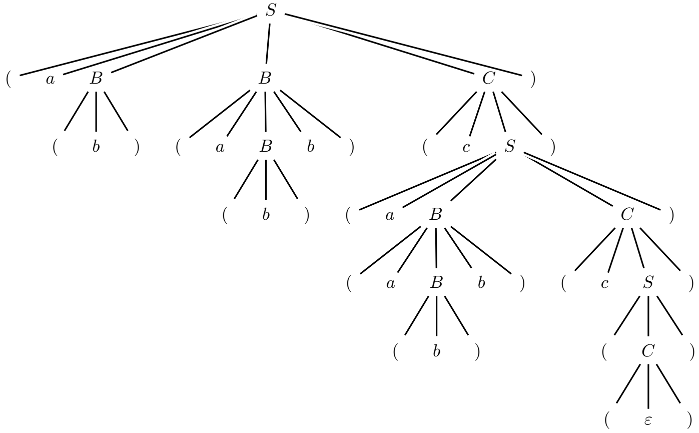
$$\overbrace{1 \xrightarrow{a} 2 \rightarrow \overbrace{\left(4 \xrightarrow{b} 7\right)}^{B} \rightarrow 2 \rightarrow \overbrace{\left(4 \xrightarrow{a} 5 \rightarrow \overbrace{\left(4 \xrightarrow{b} 7\right)}^{B} \rightarrow 6 \xrightarrow{b} 7\right)}^{B} \rightarrow 2 \rightarrow \overbrace{\left(8 \xrightarrow{c} 9 \rightarrow \overbrace{\left(1 \xrightarrow{a} 2 \rightarrow \overbrace{\left(4 \xrightarrow{a} 5 \rightarrow \overbrace{\left(4 \xrightarrow{b} 7\right)}^{B} \rightarrow 6 \xrightarrow{b} 7\right)}^{B} \rightarrow 2 \rightarrow \overbrace{\left(8 \xrightarrow{c} 9 \rightarrow \overbrace{\left(1 \rightarrow \overbrace{\left(8 \xrightarrow{\varepsilon} 8\right)}^{C} \rightarrow 3\right)}^{S} \rightarrow 10\right)}^{C} \rightarrow 3\right)}^{S} \rightarrow 10\right)}^{C} \rightarrow 3}^{S}$$

The sequence of input alphabet letters $\{a, b, c\}$ is the sample string to recognise. Each pair of round brackets embraces a computation segment that takes place in the automaton named on the upper graph bracket. Thus one sees that the computation passes recursively through various automata of the network. □

Observation

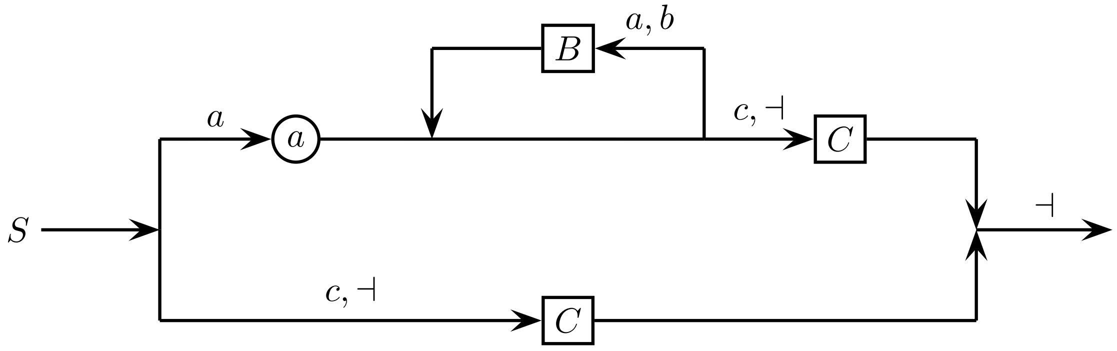
Notice that the above computation in practice reduces to an encoding of the syntax tree as a parenthesised string (imagine to remove the states and apply the superscript nonterminal as a pedex to each parenthesis pair):

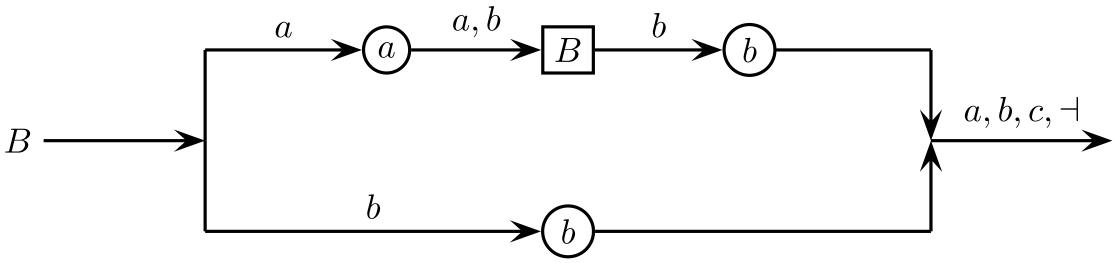
$$\Big(a\ (b)_B\ (a\ (b)_B\ b)_B\ \Big(c\ \Big(a\ (a\ (b)_B\ b)_B\ \big(c\ ((\varepsilon)_C)_S\big)_C\Big)_S\Big)_C\Big)_S$$

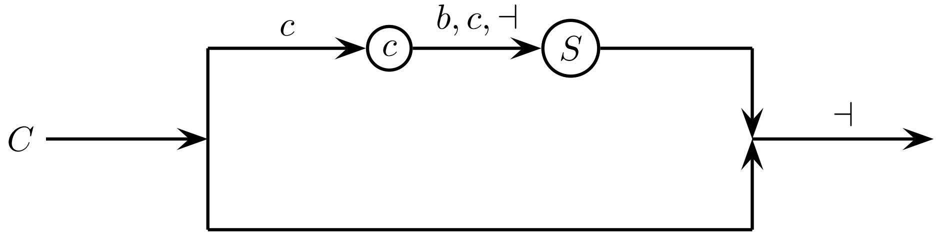


Solution of (5)

One can decorate the syntax diagrams shown before by labeling them (in particular at the bifurcation points) with the lookahead characters. Here they are (with $k = 1$):







The last diagram has an empty branch because nonterminal C is directly nullable due to rule $C \to \varepsilon$; also axiom S is nullable, though indirectly trough rule $S \to C$, while nonterminal B is not nullable. Of course, the lookahead sets are relevant especially at the bifurcation points of the syntax diagrams; however elsewhere they can be used to detect errors. □

Solution of (6)

Here is the recursive descent syntax analyser of type $LL(1)$, obtained from grammar G. An easy way to design the analyser is to resort to the labeled syntax diagrams obtained before and to use them as a programming guideline for encoding the syntax procedures of the analyser:

Recursive descent $LL(1)$ syntax analyser of grammar G

Main program and nonterminal procedures:

```
char c          – global variable

program SYNTAX_ANALYSER
    – formal rule $S_0 \to S \dashv$
    read (c)        – read the initial char.
    call S          – call axiomatic proc.
    call END        – check for end of file
end program
```

```
procedure S
    if (c == $a$) then
        – rule $S \to a\ B^*\ C$
        call a
        while (c $\in \{a, b\}$) do
            call B
        end while
        call C
    else if (c == $c$ or c ==$\dashv$) then
        – rule $S \to C$
        call C
    else error
    end if
end procedure
```

```
procedure B
    if (c == $a$) then
        – rule $B \to a\ B\ b$
        call a
        call B
        call b
    else if (c == $b$) then
        – rule $B \to b$
        call b
    else error
    end if
end procedure
```

```
procedure C
    if (c == $c$) then
        – rule $C \to c\ S$
        call c
        call S
    else if (c ==$\dashv$) then
        – rule $C \to \varepsilon$
        null
    else error
    end if
end procedure
```

Terminal procedures and procedure that checks the end of the of the program string:

```
procedure a
    if (c == $a$) then
        read (c)
    else error
    end if
end procedure
```

```
procedure b
    if (c == $b$) then
        read (c)
    else error
    end if
end procedure
```

```
procedure c
    if (c == $c$) then
        read (c)
    else error
    end if
end procedure
```

```
procedure END
    if (c ! =$\dashv$) then error
    end if
end procedure
```

Global variable “c” contains the current character, corresponding to the position of the read head on the input tape of the analyser. Keyword “null” is simply a formal way to specify that the procedure does not undertake any action. It is assumed that reading and shifting the input head over the end of the input string causes variable “c” to contain the string terminator “⊣”.

The pseudocode of the syntax analyser can be optimised in various ways: for instance one could incorporate terminal procedures “a”, “b” and “c” (which simply check the current character and shift the input head of the analyser) directly into the nonterminal ones, to speed up activation and thus save execution time; more compactions are possible. □

Solution of (7)

Here is the simulation of the execution of the syntax analyser for the sample input string $a\,b\,a\,b\,b\,c\,a\,a\,b\,b\,c$. The simulation is shown in tabular form, and the contents of the implicit execution stack are shown at each step. For simplicity only the “call”, “end” and “read” (one might include “error” if it should be the case) statements are listed, while the “if” and “while” conditions are left implicit (the reader can easily trace them by himself).

The heading of each column specifies the time instant the contents of the column cells refer to. More precisely:

- the active procedure column lists the procedure that executes the statement
- the statement column lists the statement to be executed currently
- the stack column lists the contents of the stack soon after executing the statement
- the input tape column lists the contents of the input tape soon after reading and shifting the input head
- the current character column lists the input character currently read and available for checking

Finally comments simply add some more information, mainly the correspondence between “end” and “call” statements, and few other useful indications.

#	**act. proc.** (before)	**statement**	**stack** (after)	**input tape** (after)	**cur. char** (after)	**comment**
0	none	start program	empty	*a b a b b c a a b b c*	none	
1	program	read (c)	empty	*b a b b c a a b b c*	*a*	
2	program	call S	S	*b a b b c a a b b c*	*a*	
3	S	call a	S a	*b a b b c a a b b c*	*a*	
4	a	read (c)	S a	*a b b c a a b b c*	*b*	
5	a	end	S	*a b b c a a b b c*	*b*	call at 3
6	S	call B	S B	*a b b c a a b b c*	*b*	while
7	B	call b	S B b	*a b b c a a b b c*	*b*	
8	b	read (c)	S B b	*b b c a a b b c*	*a*	
9	b	end	S B	*b b c a a b b c*	*a*	call at 7
10	B	end	S	*b b c a a b b c*	*a*	call at 6
11	S	call B	S B	*b b c a a b b c*	*a*	
12	B	call a	S B a	*b b c a a b b c*	*a*	
13	a	read (c)	S B a	*b c a a b b c*	*b*	
14	a	end	S B	*b c a a b b c*	*b*	call at 12
15	B	call B	S B B	*b c a a b b c*	*b*	
16	B	call b	S B B b	*b c a a b b c*	*b*	
17	b	read (c)	S B B b	*c a a b b c*	*b*	
18	b	end	S B B	*c a a b b c*	*b*	call at 16
19	B	end	S B	*c a a b b c*	*b*	call at 15
20	B	call b	S B b	*c a a b b c*	*b*	
21	b	read (c)	S B b	*a a b b c*	*c*	
22	b	end	S B	*a a b b c*	*c*	call at 20
23	B	end	S	*a a b b c*	*c*	call at 11
24	S	call C	S C	*a a b b c*	*c*	
25	C	call c	S C c	*a a b b c*	*c*	
26	c	read (c)	S C c	*a b b c*	*a*	
27	c	end	S C	*a b b c*	*a*	call at 25
28	C	call S	S C S	*a b b c*	*a*	
29	S	call a	S C S a	*a b b c*	*a*	
30	a	read (c)	S C S a	*b b c*	*a*	
31	a	end	S C S	*b b c*	*a*	call at 29
31	a	end	S C S	*b b c*	*a*	call at 29
32	S	call B	S C S B	*b b c*	*a*	while
33	B	call a	S C S B a	*b b c*	*a*	
34	a	read (c)	S C S B a	*b c*	*b*	
35	a	end	S C S B	*b c*	*b*	call at 33
36	B	call B	S C S B B	*b c*	*b*	
37	B	call b	S C S B B b	*b c*	*b*	
38	b	read (c)	S C S B B b	*c*	*b*	
39	b	end	S C S B B	*c*	*b*	call at 37
40	B	end	S C S B	*c*	*b*	call at 36
41	B	call b	S C S B b	*c*	*b*	
42	b	read (c)	S C S B b	⊣	*c*	
43	b	end	S C S B	⊣	*c*	call at 41
44	B	end	S C S	⊣	*c*	call at 32
45	S	call C	S C S C	⊣	*c*	
46	C	call c	S C S C c	⊣	*c*	
47	c	read (c)	S C S C c	none	⊣	
48	c	end	S C S C	none	⊣	call at 46
49	C	call S	S C S C S	none	⊣	
50	S	call C	S C S C S C	none	⊣	
51	C	null	S C S C S C	none	⊣	
52	C	end	S C S C S	none	⊣	call at 50
53	S	end	S C S C	none	⊣	call at 49
54	C	end	S C S	none	⊣	call at 45
55	S	end	S C	none	⊣	call at 28
56	C	end	S	none	⊣	call at 24
57	S	end	empty	none	⊣	call at 2
58	program	end program	empty	none	⊣	

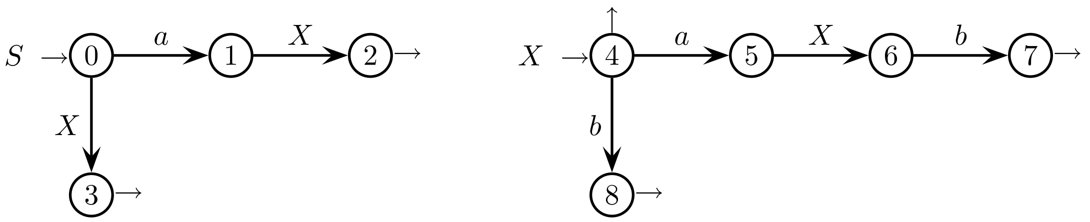
(continues and finishes on the right column)

(end of simulation)

For analysing completely the sample string, 57 statements are necessary (excluding the initial and final ones which are purely formal). The sample string is accepted by empty stack, as it is expected to happen.

Notice that the syntax analyser makes one procedure call that is void, as the procedure does not shift the input head, and other calls that simply perform one read and shift; this behaviour could be accelerated by optimising the pseudocode, as explained before. □

Exercise 33 The following network of recursive finite automata represents a grammar G:



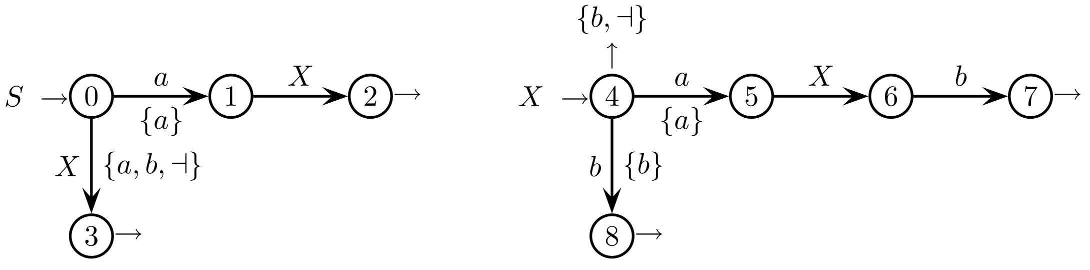
Answer the following questions:

1. Compute the lookahead sets in the relevant points of the network (bifurcation points) and check whether grammar G is of type $LL(1)$.

2. If necessary, check whether grammar G is of type $LL(k)$, for some $k > 1$.

Solution of (1)

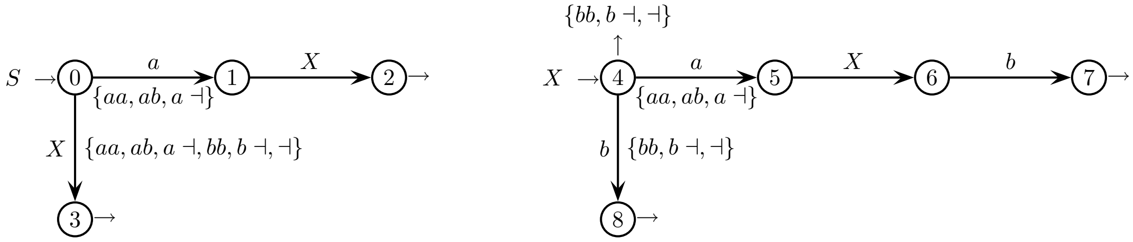
The relevant lookahead sets (only where there are bifurcations) of order $k = 1$ are shown on the state-transition graphs of the two grammar automata, as follows:



States 0 and 4 violate condition $LL(1)$, the former state because of the common lookahead a on both outgoing arcs, the latter state because of the common lookahead b both on arc $4 \rightarrow 8$ and on the final marker dart. Therefore grammar G is not of type $LL(1)$. □

Solution of (2)

To start, examine case $LL(2)$. Here are the lookahead sets of order $k = 2$:



One sees soon that on states 0 and 4 lookaheads aa and bb are common to two outgoing arcs. Therefore grammar G is not of type $LL(2)$.

It is somewhat evident that the violation of the LL property in the state 0 persists for all values $k > 2$, because bath paths labeled $a\,X$ and X leading to states 2 and 3, respectively, can generate a string with a prefix consisting of an arbitrary number of letters a, that is a^k. A similar phenomenon occurs in the state 4, though with a prefix of type b^k ($k \geq 2$). The conclusion is that grammar G is not of type $LL(k)$, for any $k \geq 1$. □

Observation

Another way for understanding the origin of the conflict, consists of examining ambiguity. In fact phrase $a\,b \in L(G)$ is ambiguous, because it can be generated by grammar G with two different computations of the automaton representation of G. Here are the two computations:

$$0 \xrightarrow{a} 1 \rightarrow \overbrace{\left(4 \xrightarrow{b} 8\right)}^{X} \rightarrow 2$$

$$0 \rightarrow \overbrace{\left(4 \xrightarrow{a} 5 \rightarrow \overbrace{\left(4 \xrightarrow{\varepsilon} 4\right)}^{X} \rightarrow 6 \xrightarrow{b} 7\right)}^{X} \rightarrow 3$$

Both computations generate string $a\,b$, but are different. Therefore grammar G is ambiguous and the immediate consequence is that G is not of type $LL(k)$, for any $k \geq 1$.

Exercise 34 Consider the following grammar G (axiom S):

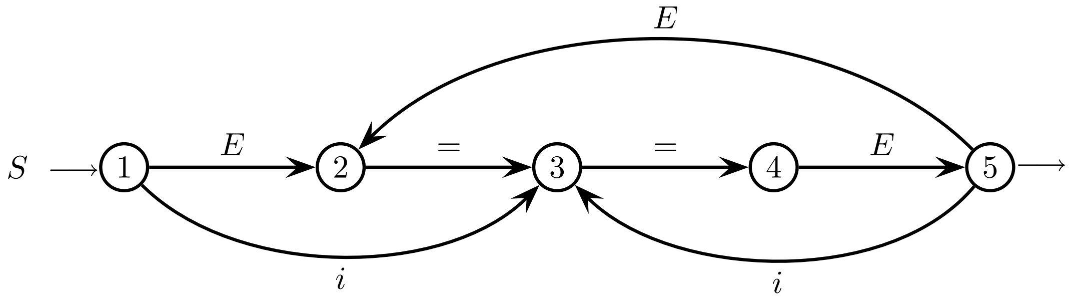
$$G \begin{cases} S \rightarrow & (\; E \text{ ‘}=\text{’‘}=\text{’ } E \mid \text{‘}i\text{’ ‘}=\text{’ } E \;)^{+} \\ E \rightarrow & \text{‘}i\text{’} \; (\; \text{‘}+\text{’ ‘}i\text{’} \;)^{*} \end{cases}$$

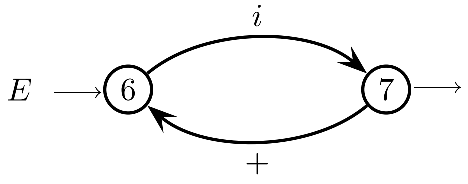
Answer the following questions:

1. Represent grammar G as a network of recursive finite state automata.
2. Check whether grammar G (in the network form) is of type $LL(k)$, for some $k \geq 1$.
3. If necessary, modify grammar G and obtain an equivalent grammar G' of type $LL(1)$.

Solution of (1)

Here is grammar G represented as a network of recursive finite state automata over the total alphabet, fully equivalent to the representation of G as a set of rules:





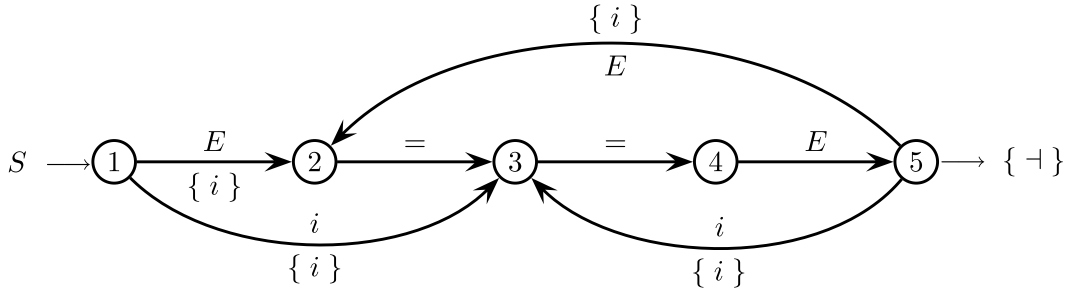
Both automata are deterministic and in reduced form (all the states are both reachable and definite), as it is required to represent a grammar. Moreover they are in minimal form; this is not strictly required to represent a grammar but of course is useful. The latter automaton is not recursive either, as it does not have any arc labeled by means of nonterminals. □

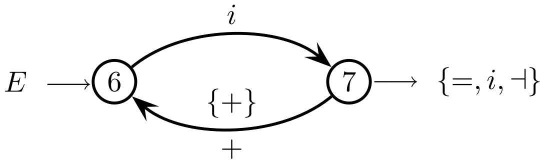
Observation

It is also possible to represent grammar G by means of a network of recursive indeterministic finite state automata (with multiple labeled arcs or spontaneous transitions), but here such a form does not seem likely to help to minimise the number of states, while it would surely make $LL(1)$ analysis more difficult.

Solution of (2)

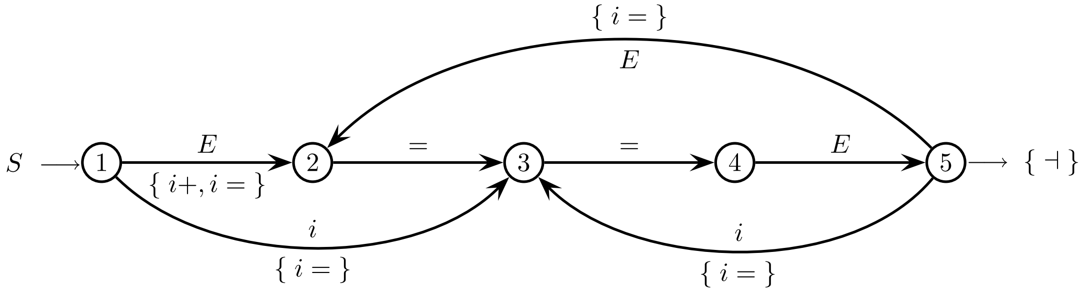
Verification of the $LL(1)$ property (for both automata):





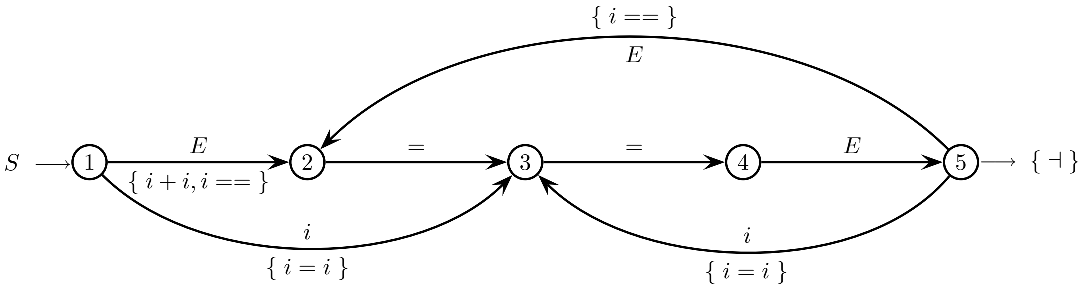
States 1 and 5 violate the LL condition: grammar G is not of type $LL(1)$.

Verification of the $LL(2)$ property (only for the automaton that expands nonterminal S):



States 1 and 5 still violate the LL condition: grammar G is not of type $LL(2)$.

Verification of the $LL(3)$ property (only for the automaton that expands nonterminal S):

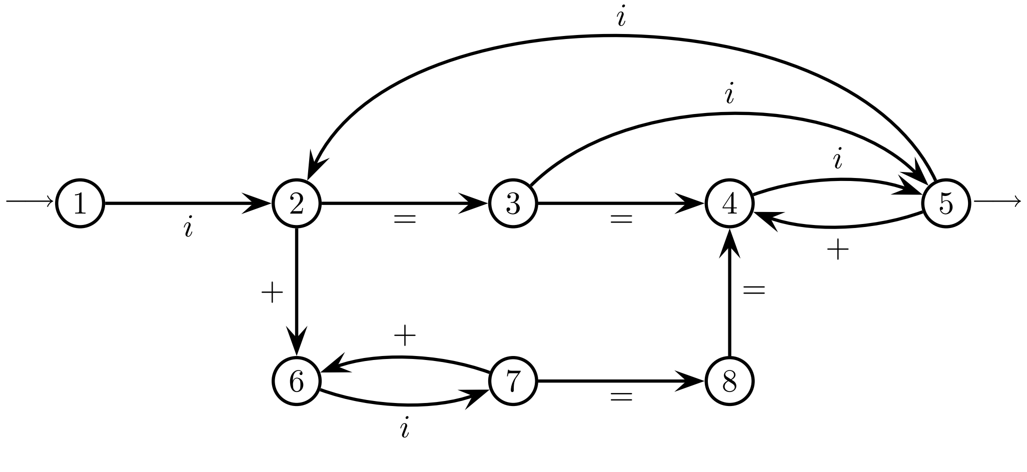


No state violates the LL condition any longer: grammar G is of type $LL(3)$. □

Solution of (3)

A phrase of language $L(G)$ is a list, not empty and without separator, of relations, each of which contains one or two characters "=" and two addition expressions of variables i on the left and right side, but with the constraint that on the left side of a relation containing only one character "=", there may be only one variable i, not an addition expression. Therefore language $L(G)$ is regular and can be generated by a grammar of type $LL(1)$.

To this purpose it suffices to design the (minimal) deterministic finite state automaton that recognises language $L(G)$, and the corresponding right-linear grammar will certainly be of type $LL(1)$, as it is well known. Here is the recogniser automaton of $L(G)$, obtained in an intuitive way (it is easy to check that the automaton is in reduced and minimal form):



Here is the right-linear grammar G' of type $LL(1)$ that generates language $L(G)$ (axiom $\langle 1 \rangle$):

$$G' \begin{cases} \langle 1 \rangle & \rightarrow & \text{`}i\text{'} \ \langle 2 \rangle \\ \langle 2 \rangle & \rightarrow & \text{`}=\text{'} \ \langle 3 \rangle \ \mid \ \text{`}+\text{'} \ \langle 6 \rangle \\ \langle 3 \rangle & \rightarrow & \text{`}=\text{'} \ \langle 4 \rangle \ \mid \ \text{`}i\text{'} \ \langle 5 \rangle \\ \langle 4 \rangle & \rightarrow & \text{`}i\text{'} \ \langle 5 \rangle \\ \langle 5 \rangle & \rightarrow & \varepsilon \ \mid \ \text{`}i\text{'} \ \langle 2 \rangle \ \mid \ \text{`}+\text{'} \ \langle 4 \rangle \\ \langle 6 \rangle & \rightarrow & \text{`}i\text{'} \ \langle 7 \rangle \\ \langle 7 \rangle & \rightarrow & \text{`}+\text{'} \ \langle 6 \rangle \ \mid \ \text{`}=\text{'} \ \langle 8 \rangle \\ \langle 8 \rangle & \rightarrow & \text{`}=\text{'} \ \langle 4 \rangle \end{cases}$$

Clearly all the lookahead sets of alternative rules are disjoint (and of course that of the null rule $\langle 5 \rangle \to \varepsilon$ contains the terminator as well). □

Observation

Since in the representation of grammar G as a network of two recursive finite state automata, one of them is actually not recursive, it would be possible to obtain algorithmically the automaton that recognises language $L(G)$, in the following way:

- consider the recursive automaton of axiom S and expand the arcs labeled with nonterminal E by means of the non-recursive automaton of E, and thus obtain an ordinary automaton, not recursive and (in general) indeterministic
- determinise the obtained automaton by means of the subset construction and then possibly minimise the number of states

The resulting automaton is deterministic (and possibly minimal) and recognises language $L(G)$. It is not necessarily identical to the automaton obtained before, unless it is in minimal form as well; of course these two automata are equivalent.

As an alternative one could design directly (in an intuitive way) a grammar G' of type $LL(1)$ in extended form (EBNF). Here it is, for instance:

$$G' \begin{cases} S \;\to\; (\,\text{‘}i\text{’}\;(\,\text{‘}=\text{’}\;[\,\text{‘}=\text{’}\,]\;\mid\;\text{‘}+\text{’}\;E\;\text{‘}=\text{’}\;\text{‘}=\text{’})\;E\,)^{+} \\ E \;\to\; \text{‘}i\text{’}\;(\,\text{‘}+\text{’}\;\text{‘}i\text{’}\,)^{*} \end{cases}$$

One can easlily realise that such a grammar G' is of type $LL(1)$: it suffices to represent it as a network of recursive finite state automata and check the $LL(1)$ property. Actually grammar G' above corresponds at a quick glance directly to the before given automaton that recognises language $L(G)$. Alternatively, try to design the syntax analyser of type $LL(1)$ of grammar G' above and write the syntax procedure of nonterminals S and E. The reader is left the task of verifying the correctness of grammar G' (in either way).

Exercise 35 Consider the following grammar G, in extended form (EBNF), over terminal alphabet $\{a, b, c, d\}$ and with axiom S:

$$G \begin{cases} S \;\to\; A \dashv \\ A \;\to\; (\,a\,A^{*}\,A \;\mid\; B\,)\;b \\ B \;\to\; a\,C \;\mid\; c \\ C \;\to\; d\,C \;\mid\; d \end{cases}$$

Answer the following questions:

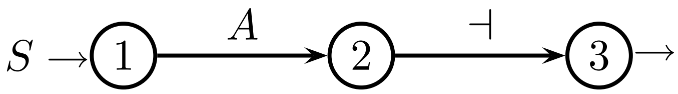
1. Represent grammar G as a network of recursive finite state automata.

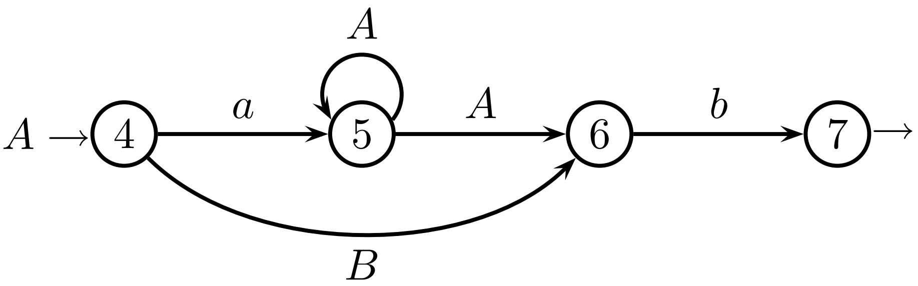
2. Check each of the four nonterminals S, A, B and C separately, and find the smallest integer $k \geq 1$ such that the considered nonterminal is of type $LL(k)$.

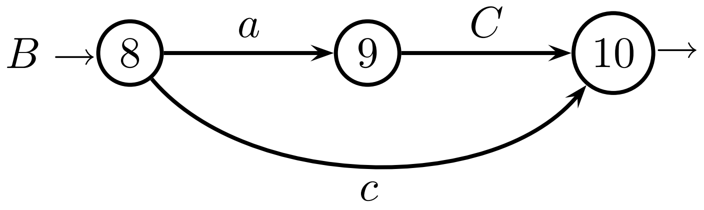
3. If necessary, design a new grammar G' equivalent to grammar G, such that it is of type $LL(k)$ for some lower integer k (than that previously found).

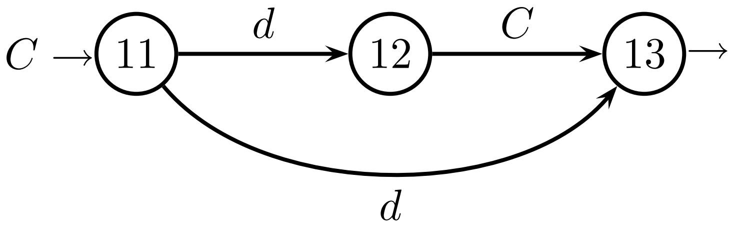
Solution of (1)

Here is the network of recursive finite state (indeterministic) automata that correspond directly to the rules of grammar G in the most straightforward way, that is according to the Thompson or modular construction, applied in such a way as to avoid to use spontaneous transitions (ε-transitions):

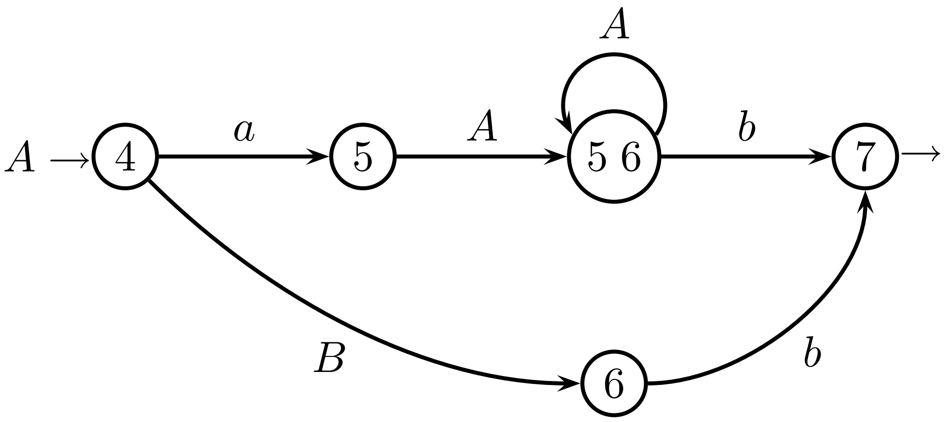


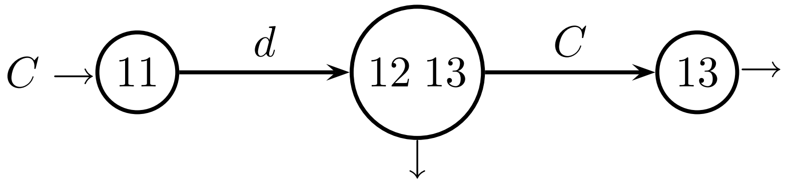






The two automata that expand nonterminals A and C are indeterministic. To perform LL analysis it is necessary to determinise these two automata by means of the subset construction, which here can be applied in an almost intuitive way, with respect to the total alphabet of grammar G, that is $\Sigma \cup V_N$ (union of terminal and nonterminal symbols). Here they are:





These two automata correspond to reformulate the extended rules as follows:

$$
\begin{array}{lll}
A & \rightarrow & a\ A\ A^{*}\ b \ \mid\ B\ b \\
C & \rightarrow & d\ (C \ \mid\ \varepsilon\ )
\end{array}
$$

Here indeterminism (with respect to the total alphabet) is eliminated.
If one prefers one can proceed as follows, by using the cross operator:

$$
\begin{array}{lll}
A & \rightarrow & a\ A^{+}\ b \ \mid\ B\ b \\
C & \rightarrow & d\ (C \ \mid\ \varepsilon\ )
\end{array}
$$

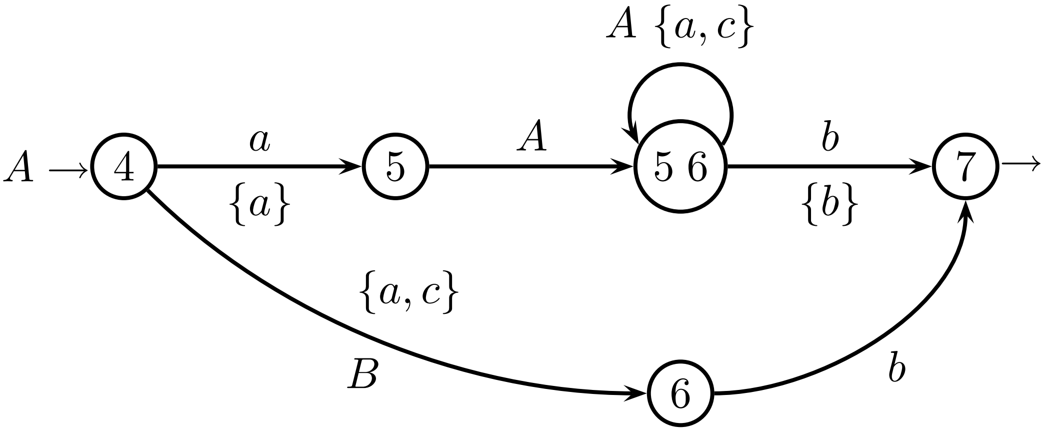
Of course one can obtain directly the deterministic versions of such automata, but here the design has been carried out step by step intentionally. □

Solution of (2)

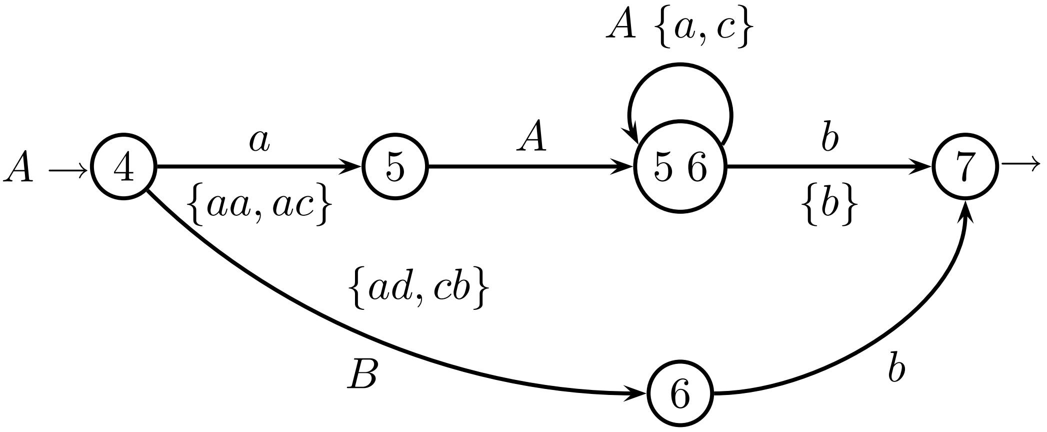
Now the $LL(k)$ analysis is carried out, for each nonterminal separately.

Clearly nonterminal (axiom) S is of type $LL(1)$, as there are not any bifurcations at the nodes of the corresponding automaton.

Here are the lookahead sets of nonterminal A (only on the bifurcation at nodes 4 and 5 6), with $k = 1$:

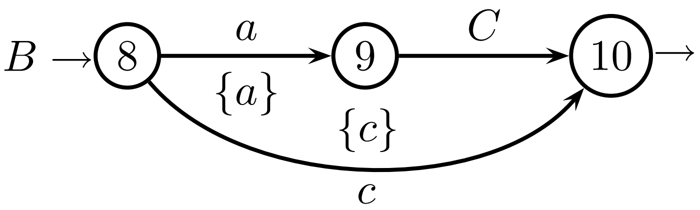


Since at node 4 the lookahead sets are not disjoint yet, nonterminal A is not of type $LL(1)$. For $k = 2$ such sets are the following. Here they are computed only where it is necessary, that is on the bifurcation at node 4:



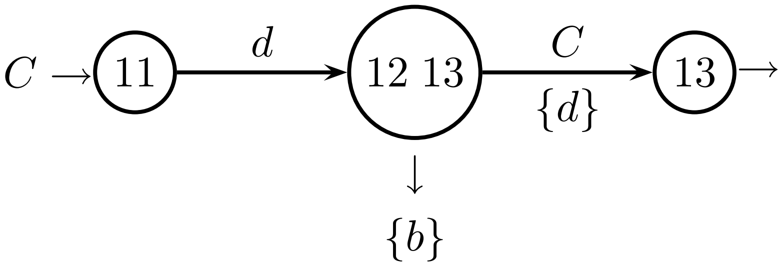
Since now the lookahead sets are disjoint, nonterminal A is of type $LL(2)$.

Here are the lookahead sets of nonterminal B (on the bifurcation at node 8), for $k = 1$ (actually such sets here are trivial):



Since the lookahead sets are disjoint, nonterminal B is of type $LL(1)$.

Here are the lookahead sets of nonterminal C (on the bifurcation at node 12 13), with $k = 1$:



Since the lookahead sets are disjoint, nonterminal C is of type $LL(1)$.

In conclusion, grammar G happens to be of type $LL(2)$. □

Solution of (3)

As grammar G is already $LL(k)$ for $k = 2$, at the best one could search a new grammar G', equivalent to G, that is $LL(k)$ for $k = 1$. To begin, notice that nonterminal B generates a regular language, that is the following:

$$L(B) = a\ d^+\ \mid\ c$$

because the rules that expand nonterminal B are either terminal or right-linear. Then notice that the alternative rule $A \to B\ b$ generates the following language (still regular):

$$L(B\ b) = \big(\ a\ d^+\ \mid\ c\ \big)\ b = a\ d^+\ b\ \mid\ c\ b$$

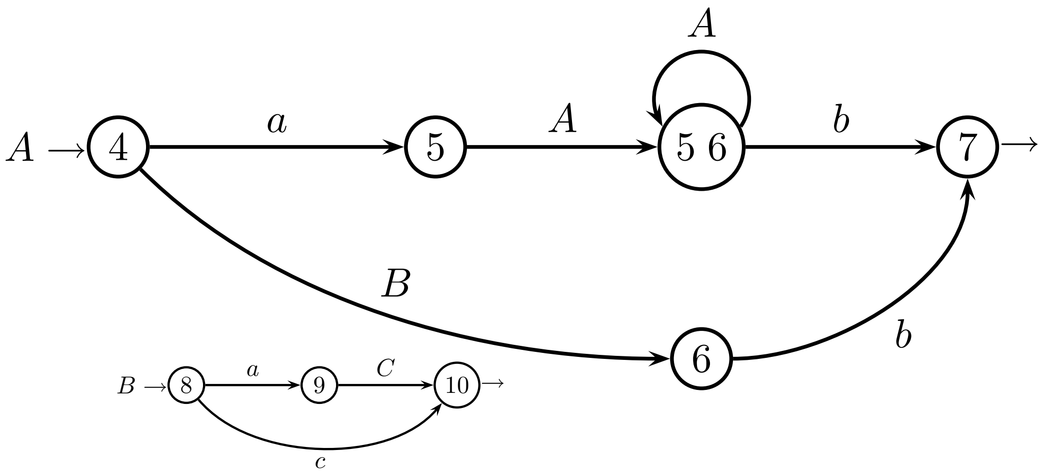
By examining again the rules that expand nonterminal A, one sees soon that value $k = 2$ derives from the need of distinguishing and selecting between the first two members of $A \to (\ a\ A^*\ A\ \mid\ B\ )\ b = a\ A^+\ b\ \mid\ B\ b = \underbrace{a\ A^+\ b}_{1}\ \mid\ \underbrace{a\ d^+\ b}_{2}\ \mid\ \underbrace{c\ b}_{3}$, as both begin with letter a (the third one begins with c and is distinguishable). It is therefore natural to think of joining and unifying such two members. By factoring letters a and b on the left and right side, respectively, both in the former and latter member, one obtains soon the following grammar G' (axiom S):

$$G' \begin{cases} S & \to & A\ \dashv \\ A & \to & a\ (\ A^+\ \mid\ d^+\ )\ b\ \mid\ c\ b \end{cases}$$

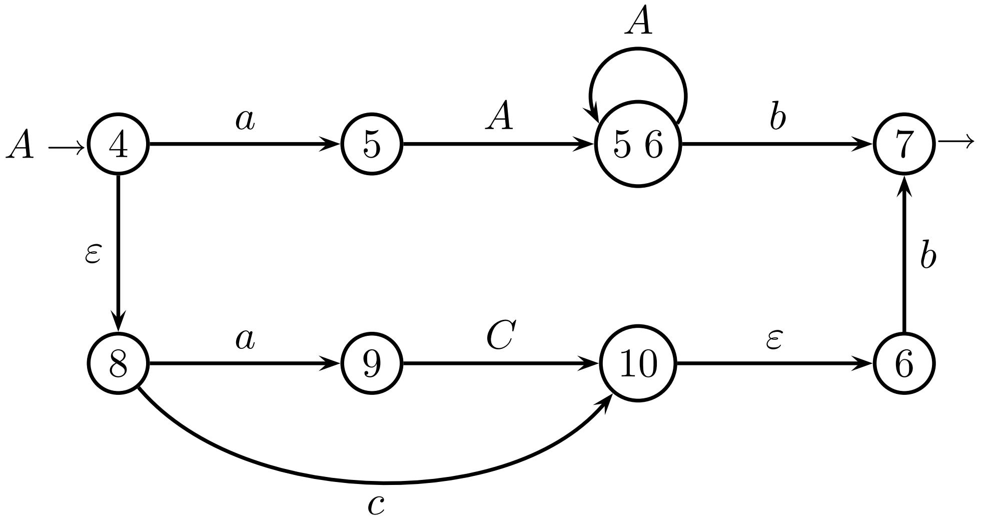
The new grammar G' is in extended form (EBNF) and is equivalent to the original one (check by generating a few sample strings). Moreover, clearly it is of type $LL(1)$ as nonterminal A is not nullable and begins with letter a, such that the alternative in the rule are all separable by resorting to a lookahead window of size $k = 1$. □

Observation

If one wishes, one could obtain more or less the same result as before by substituting directly in the automata. Consider the automaton that expands nonterminal A and replace to the transition labeled with nonterminal B, which is part of the bifurcation responsible for the grammar to be of type $LL(2)$, the whole automaton that expands B, in the following way:

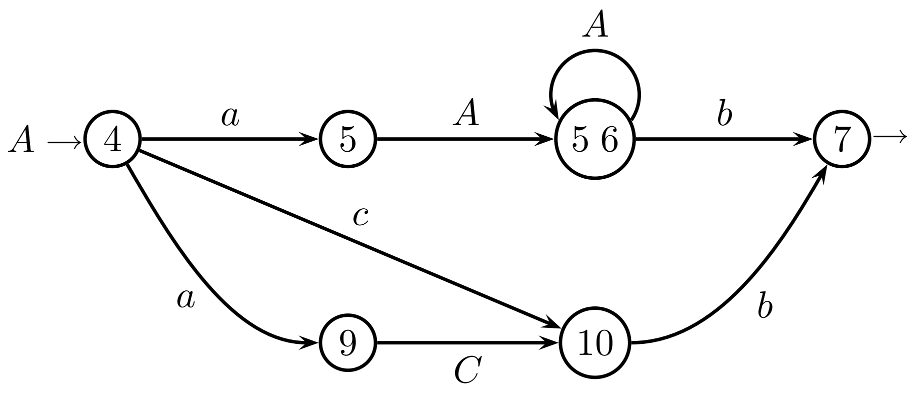


Here is the resulting automaton (notice that the automaton of nonterminal B is inserted and connected by means of spontaneous transitions):

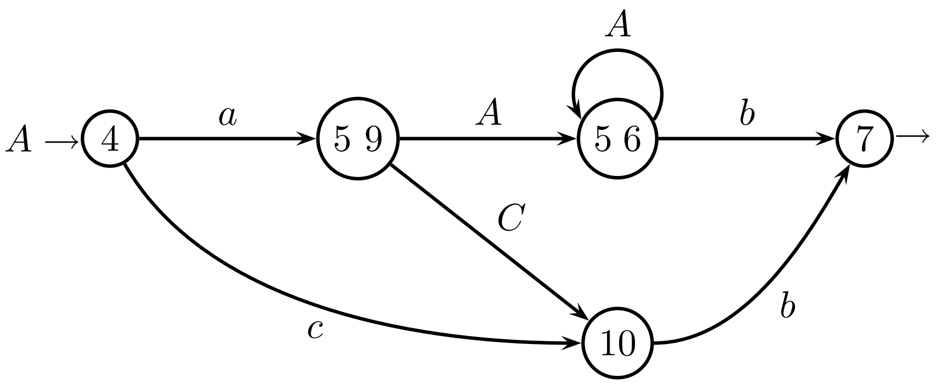


After the replacement, the automaton is indeterministic precisely due to the presence of spontaneous transitions (ε-transition).

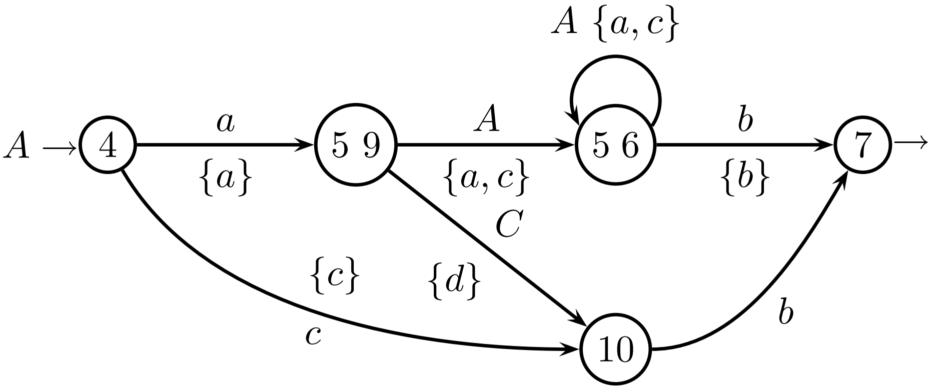
Now determinise the automaton with respect to the total alphabet of the grammar (union of terminal and nonterminal symbols). First cut spontaneous transitions:



After cutting spontaneous transitions and moving backwards the arcs outgoing from nodes 6 and 8, these two nodes are unreachable and can be removed from the graph. Then apply the subset construction (which here is at all trivial):

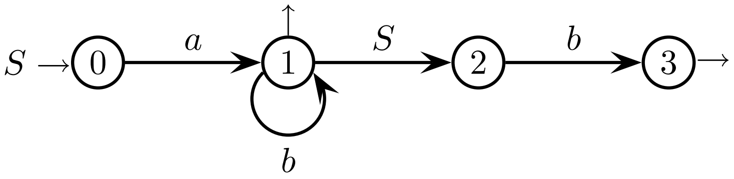


Thus, the automaton is deterministic again. Now recompute the lookahead sets on the bifurcations (nodes 4, 5 9 and 5 6), for $k = 1$. This yields what follows:



One realises soon that the automaton expanding nonterminal A is of type $LL(1)$. If such an automaton is written back as a set of rules, one has about the same result as before (actually to have an identical result one should replace also nonterminal C though it is not strictly necessary).

Exercise 36 The following grammar G is given, presented as an automaton:

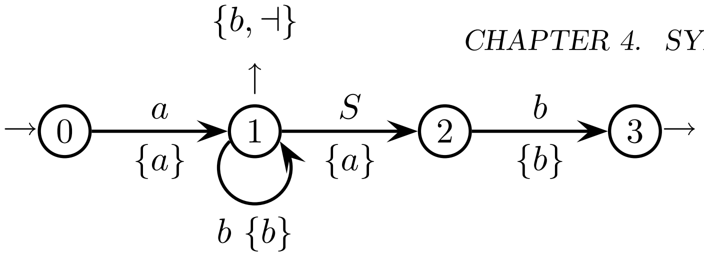


Answer the following questions:

1. Compute the lookahead sets on all the arcs of the graph and say whether grammar G is of type $LL(k)$ for some $k \geq 1$.
2. If necessary, examine whether it is possible to find a grammar G' equivalent to G with the $LL(1)$ property.

Solution of (1)

Here are the lookahead sets of order one of grammar G (with $k = 1$)::



Clearly grammar G is not of type $LL(1)$ in the state 1 (which is a bifurcation point), as the lookahead sets of the self-loop and of the terminal outgoing arc share the same string b. It is not difficult to convince oneself that G is not of type $LL(k)$ either, for any $k > 1$, as both the lookahead set of order k of the self-loop on state 1 and that of the terminal arc outgoing from state 1 contain the same string b^k, for any $k > 1$, and therefore are not disjoint. □

Solution of (2)

The language generated by grammar G contains phrases of the following type:

$$\underbrace{a\,b^*}_{1}\ \underbrace{a\,b^*}_{2}\ \dots\ \underbrace{a\,b^*}_{n}\ \underbrace{a\,b^*}_{n+1}\ \underbrace{b}_{n}\ \dots\ \underbrace{b}_{2}\ \underbrace{b}_{1} \qquad\qquad n \geq 0$$

Such a language is a variant of language $a^n b^n$ ($n \geq 0$), well known to be of type $LL(1)$. However here a recursive descent syntax analyser with a lookahead window of fixed size $k \geq 1$ could never distinguish, only by examining a sequence of k consecutive input characters, whether the input tape contains one of the factors (substrings) of type b^*, which are purely regular and need not use the pushdown stack, or part of the suffix of the n final characters b, which match as many characters a and clearly imply the use of the pushdown stack. □

Exercise 37 One wishes one modified the below given grammar G so as to obtain an equivalent grammar G' suited to recursive descent deterministic syntax analysis (that is suited to $LL(k)$ analysis for some $k \geq 1$).

$$G \begin{cases} S \rightarrow S\,A \mid A \\ A \rightarrow a\,A\,b \mid \varepsilon \end{cases}$$

Answer the following questions:

1. Design grammar G' and explain why it is equivalent to the original grammar G.
2. Determine the lookahead sets of grammar G' and check whether G' is of type $LL(1)$ or $LL(k)$, for some $k > 1$.

Solution of (1)

Grammar G has two defects that cause the loss of the $LL(k)$ property: it is recursive on the left and moreover ambiguous; for instance, two different modes of generating ε are the following:

$$S \Rightarrow A \Rightarrow \varepsilon \qquad \text{and} \qquad S \Rightarrow S\,A \Rightarrow A\,A \Rightarrow \varepsilon\,A \Rightarrow \varepsilon\,\varepsilon = \varepsilon$$

As axiom S generates the sentential form A^+ and nonterminal A generates language $L_A = \{\, a^n\, b^n \mid n \geq 0 \,\}$, the language generated by grammar G is the following:

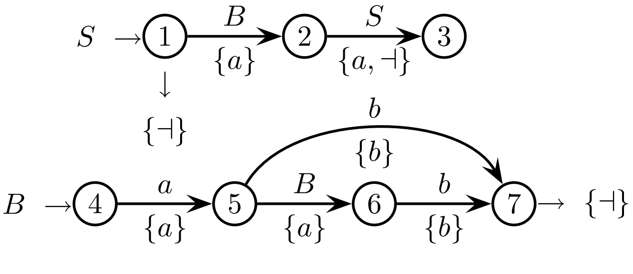
$$L(G) = x^+ \qquad \text{where } x \in L_A$$

Actually $L(G)$ is a subset of the Dyck language over alphabet $\{a, b\}$. □

Solution of (2)

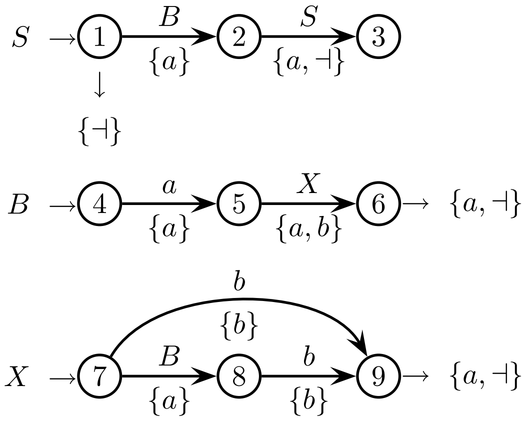
Here is a solution, not in extended form (BNF). It is easy to write a grammar G_1 equivalent to G, but not recursive on the left and not ambiguous. Here it is, with lookahead:

grammar G_1	lookahead set
$S \;\rightarrow\; B\,S$	a
$S \;\rightarrow\; \varepsilon$	$\dashv$
$B \;\rightarrow\; a\,B\,b$	a
$B \;\rightarrow\; a\,b$	a



Notice that to eliminate ambiguity, the empty string ε is generated directly and solely by axiom S. As the alternative rules that expand nonterminal B share a prefix (namely character a), they violate the $LL(1)$ condition. However grammar G_1 turns out to be of type $LL(2)$, as one sees soon since the lookahead sets of order 2 of the two rules expanding B are $a\,a$ and $a\,b$ (check on the automaton that models the rules of B). By factoring prefix a on the left one obtains the following grammar G_2, which is of type $LL(1)$. Here it is, with lookahead:

grammar G_2	lookahead set
$S \rightarrow B\,S$	a
$S \rightarrow \varepsilon$	$\dashv$
$B \rightarrow a\,X$	a
$X \rightarrow B\,b$	a
$X \rightarrow b$	b



Therefore grammar G_2 is the grammar G' one is looking for. □

4.2 Shift and Reduction

Exercise 38 Consider the following grammar G (axiom S):

$$G \begin{cases} S \rightarrow X\,b \mid X \\ X \rightarrow a\,X\,b \mid a \mid \varepsilon \end{cases}$$

Answer the following questions:

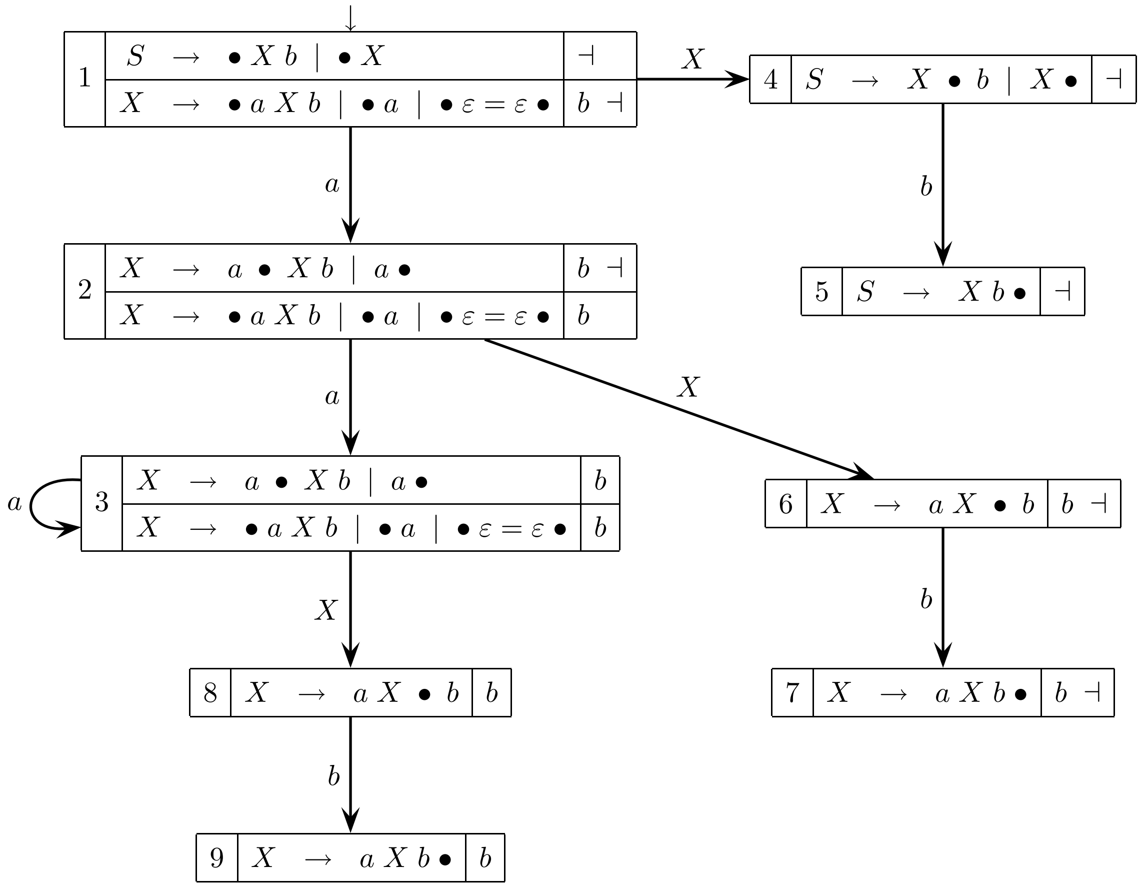
1. Design the driver graph of type $LR(1)$ (the prefix recogniser) of grammar G.

2. For each macrostate of the driver graph of type $LR(1)$ of grammar G, show whether there are conflicts of type reduction-shift or reduction-reduction.

3. Design again the driver graph of type $LR(1)$ of grammar G, but for the grammar represented as a network of recursive finite state automata.

Solution of (1)

Here is the driver graph of type $LR(1)$ of the syntax analyser of grammar G:


1
$S \rightarrow \bullet X b \mid \bullet X$ | $\dashv$
$X \rightarrow \bullet a X b \mid \bullet a \mid \bullet \varepsilon = \varepsilon \bullet$ | $b \dashv$
X
4
$S \rightarrow X \bullet b \mid X \bullet$ | $\dashv$
a
b
2
$X \rightarrow a \bullet X b \mid a \bullet$ | $b \dashv$
$X \rightarrow \bullet a X b \mid \bullet a \mid \bullet \varepsilon = \varepsilon \bullet$ | b
5
$S \rightarrow X b \bullet$ | $\dashv$
a
X
a
3
$X \rightarrow a \bullet X b \mid a \bullet$ | b
$X \rightarrow \bullet a X b \mid \bullet a \mid \bullet \varepsilon = \varepsilon \bullet$ | b
6
$X \rightarrow a X \bullet b$ | $b \dashv$
X
b
8
$X \rightarrow a X \bullet b$ | b
7
$X \rightarrow a X b \bullet$ | $b \dashv$
b
9
$X \rightarrow a X b \bullet$ | b

For simplicity, the candidates of alternative productions are grouped in line. The driver graph has nine macrostates. □

Solution of (2)

Here follows the analysis of the macrostates of the driver graph above. All the macrostates containing one shift and one reduction candidate, or two reduction candidates, are examined.

macrostate	**potential conflict**
1	the choice between the reduction $X \to \varepsilon \ \bullet \ \{b \ \dashv\}$ and the shift over a is deterministic - there is not any conflict
2	the choice between reductions $X \to a \ \bullet \ \{b \ \dashv\}$ and $X \to \varepsilon \ \bullet \ \{b\}$ is not deterministic, while the choice between the same reductions and the shift over a is deterministic - there is a reduction-reduction conflict
3	the choice between reductions $X \to a \ \bullet \ \{b\}$ and $X \to \varepsilon \ \bullet \ \{b\}$ is not deterministic - there is a reduction-reduction conflict
4	the choice between the reduction $S \to X \ \bullet \ \{\dashv\}$ and the shift over b is deterministic - there is not any conflict

In conclusion, the driver graph of type $LR(1)$ of grammar G has two reduction-reduction conflicts, macrostates 2 and 3 are inadequate and therefore G is not of type $LR(1)$. □

Observation

One may notice that grammar G is clearly ambiguous, as the strings of type $a^n\, b^n$ $(n \geq 1)$ are generated in two different ways. For instance (with $n = 2$):

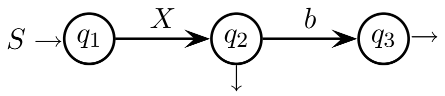
$$S \Rightarrow X \Rightarrow a\ X\ b \Rightarrow a\ a\ X\ b\ b \Rightarrow a\ a\ b\ b$$

$$S \Rightarrow X\ b \Rightarrow a\ X\ b\ b \Rightarrow a\ a\ b\ b$$

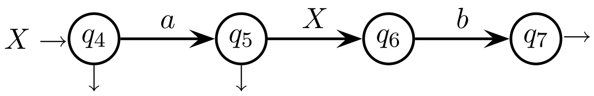
Since an ambiguous grammar cannot be analysed deterministically, grammar G is not of type $LR(k)$, for any $k \geq 1$.

Solution of (3)

One may wish to examine grammar G by modeling it as a network of recursive finite state automata. Here is G modeled as a network of two such (deterministic) automata:

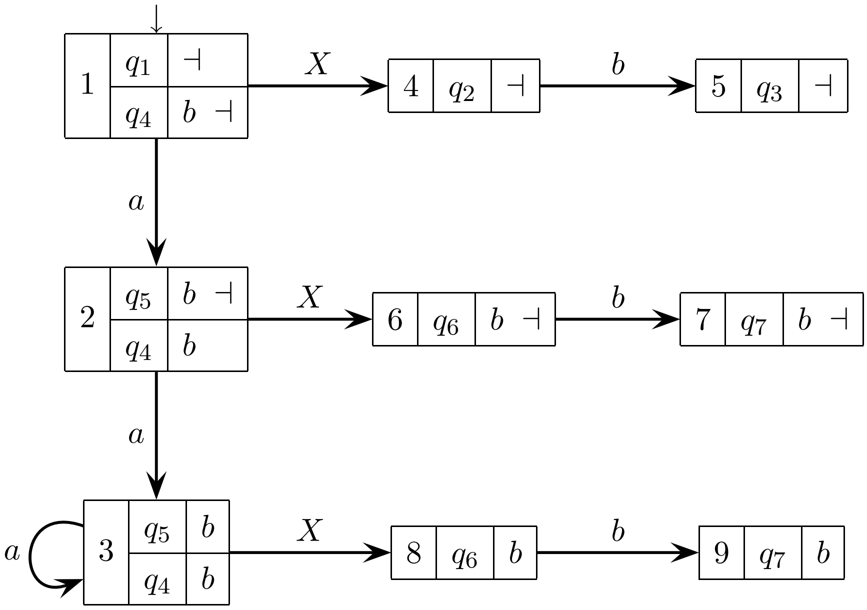


automaton of $S \to X\ b \ \mid\ X$



automaton of $X \to a\ X\ b \ \mid\ a \ \mid\ \varepsilon$

The driver graph of type $LR(1)$ of grammar G in the form of automata is the following:



States q_2, q_3, q_4, q_5 and q_7 are final for the automata modeling grammar G, therefore the macrostates of the driver graph that contain them may have a conflict of type shift-reduction or reduction-reduction. Here is the list of potential conflicts:

macrostate	**potential conflict**
1	the choice between the reduction q_1 $\{b\ \dashv\}$ and the shift over a is deterministic - there is not any conflict
2	the choice between reductions q_4 $\{b\ \dashv\}$ and q_5 $\{b\}$ is not deterministic, while the choice between the same reductions and the shift over a is deterministic - there is a reduction-reduction conflict
3	the choice between reductions q_5 $\{b\}$ and q_4 $\{b\}$ is not deterministic - there is a reduction-reduction conflict
4	the choice between the reduction q_2 $\{\dashv\}$ and the shift over b is deterministic - there is not any conflict

As before, there are two conflicts of type reduction-reduction in the inadequate macrostates 2 and 3, therefore grammar G is not of type $LR(1)$. □

Exercise 39 Consider the following grammar G (axiom S):

$$G \begin{cases} S \rightarrow E \text{ ‘} \dashv \text{’} \\ E \rightarrow T \text{ ‘}+\text{’} E \mid T \\ T \rightarrow \text{‘}i\text{’} \mid \text{‘}i\text{’ ‘}+\text{’} T \mid \text{‘(’} E \text{ ‘)’} \end{cases}$$

Answer the following questions:

1. Check whether grammar G is of type $LR(1)$ (use a method of choice).
2. If not yet done, draw the driver graph of type $LR(1)$ of grammar G.
3. If necessary, modify grammar G and obtain an equivalent grammar G' of type $LR(1)$.
4. Discuss briefly whether language $L(G)$ has a grammar G'' of type $LR(0)$.

Solution of (1)

The answer is easily found almost immediately: grammar G is not of type $LR(1)$, because it is ambiguous. In fact the phrases of language $L(G)$ are of the following type, with nested parenthesis pairs:

$$i \; + \; \ldots \; + \; (\; i \; + \; \ldots \; + \; (\; i \; + \; \ldots \;) \; + \; \ldots \;) \; + \; \ldots \; \dashv$$

Such phrases may contain several concatenated parenthesis pairs (at any depth level), like for instance:

$$\cdots + (\ldots) + \cdots + (\ldots) + \ldots$$

besides containing nested parenthesis pairs. But the series of additions is generated in an ambiguous way by both nonterminal E and T, as one sees soon by inspecting the rules of grammar G. □

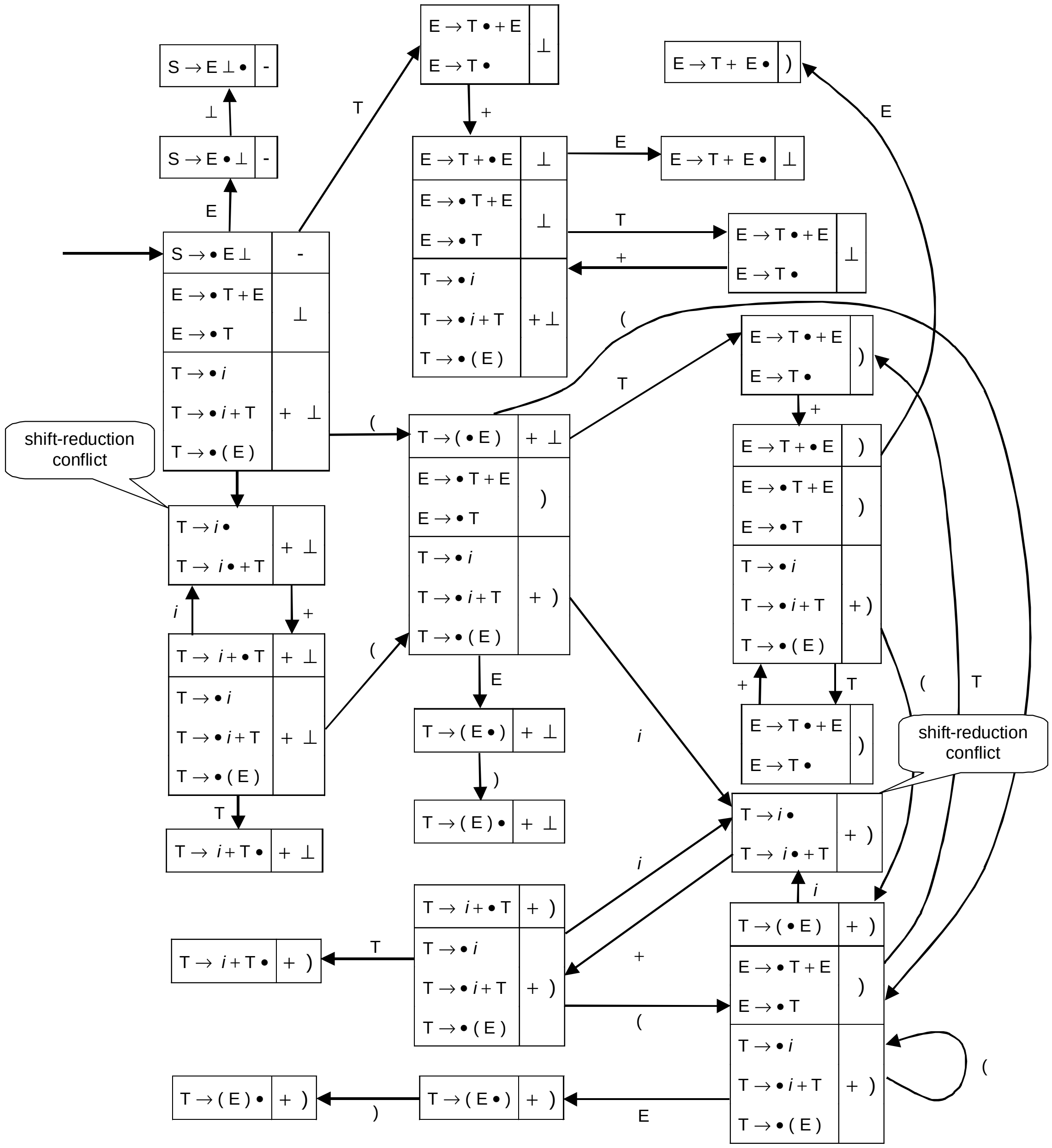
Observation

One may also reason by noticing that language $L(G)$ is defined by grammar G as a two-level list, but with the same separator "+" in the upper and lower level. Addition at the upper level is generated by rule $E \rightarrow T \; + \; E$, while at the lower level the working rule is $T \rightarrow i \; + \; T$. This causes language $L(G)$ to be defined ambiguously, as it is unclear which level the elements of the list (variables "i") belong to, whether the upper or lower one.

Moreover, language $L(G)$ (without terminator "$\dashv$") coincides with its mirror image, that is equality $L(G) = L(G)^R$ holds. This observation will be useful in the following, in answering the third question.

Solution of (2)

It is not necessary to draw the driver graph of type $LR(1)$ of grammar G, as observing that G is ambiguous suffices to conclude it is not deterministic. However, here it is for completeness (for simplicity macrostates are not numbered and "⊥" is the terminator):



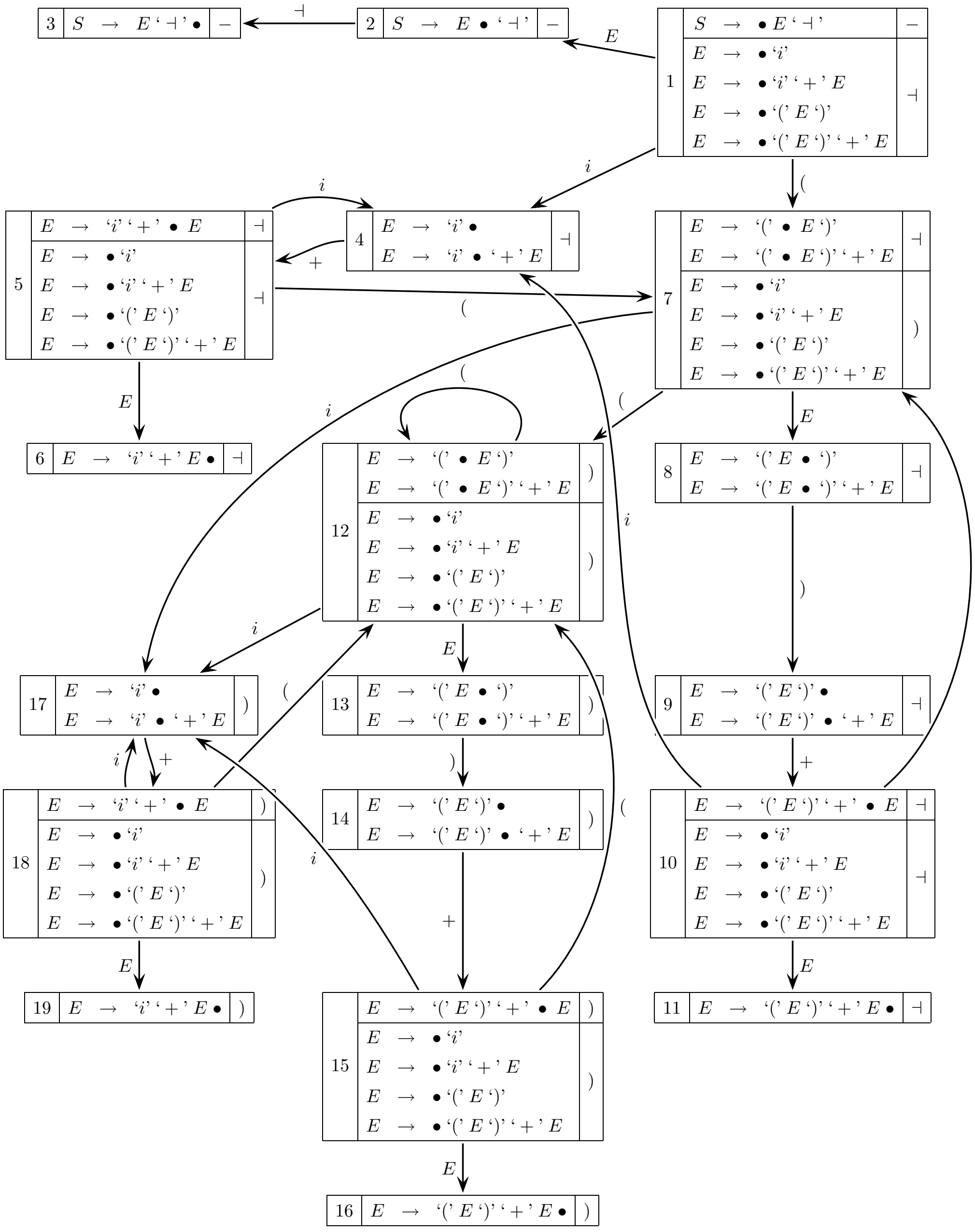
In the $LR(1)$ driver graph of grammar G there are two inadequate macrostates, which correspond to the reduction candidate $T \to i \bullet$ with lookahead including terminal "+". Of course such a candidate conflicts with the shift candidate $T \to i \bullet + T$, when the input tape of the syntax analyser contains precisely terminal "+". Actually the existence of the conflict is somewhat obvious: as said before, it is unclear whether the chain of additions under examination is generated by the current term T (this corresponds to execute a shift move) rather than whether the current term T is finished (this corresponds to execute a reduction move) and the analyser has to proceed and expand a new term T ambiguously generated by the upper level rule $E \rightarrow T\ +\ E$. Therefore grammar G is not of type $LR(1)$. □

Solution of (3)

In order to design a deterministic bottom-up analyser, it is necessary to remove ambiguity from grammar G and thus obtain grammar G'. One can do as follows and eliminate rule $E \rightarrow T$ from G (and hence eliminate the nonterminal T itself), but without introducing null rules:

$$G' \begin{cases} S \rightarrow E\ `\dashv\text{'} \\ E \rightarrow \text{`}i\text{'} \mid \text{`}i\text{'}\ \text{`}+\text{'}\ E \mid \text{`(' } E \text{ `)'} \mid \text{`(' } E \text{ `)'}\ \text{`}+\text{'}\ E \end{cases}$$

Clearly grammars G' and G are equivalent. But G' is of type $LR(1)$, as lookahead allows to choose between shift and reduction candidates. Here is the driver graph of type $LR(1)$ of G':



All the macrostates of the driver graph are adequate, therefore grammar G' is of type $LR(1)$. □

Observation

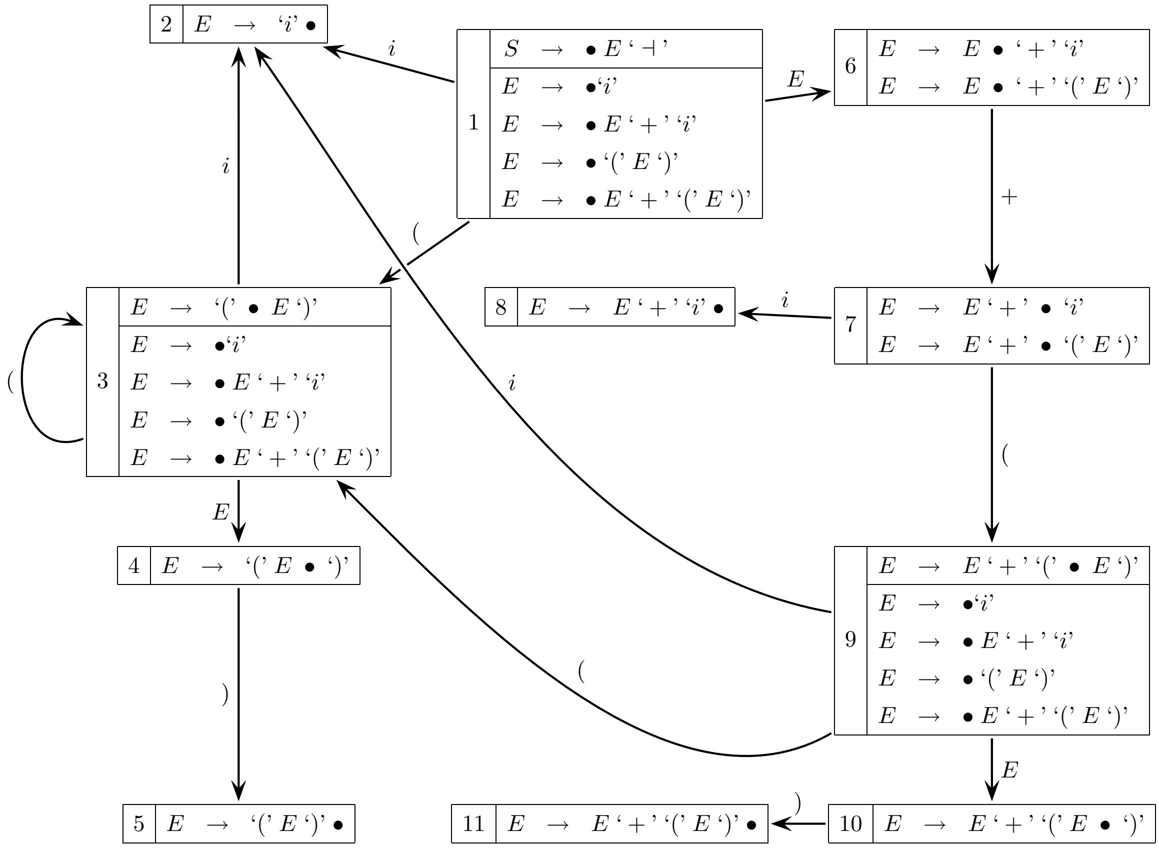
As an alternative one can try to obtain a grammar G', equivalent to grammar G, that is of type $LL(1)$ and consequently of type $LR(1)$. There are a few possible solutions, structurally inelegant but working. The reader is left the task of finding by himself such formulations of G'.

Solution of (4)

It is possible to find a grammar G'' equivalent to grammar G and of type $LR(0)$. Notice that the grammar G' given before is not $LR(0)$, as it contains rules that are prefixes of other rules, and this fact necessarily originates inadequate macrostates in the driver graph of type $LR(0)$. Moreover, it is not granted in general that a grammar of type $LR(1)$ would be of type $LR(0)$ as well. However, here to obtain a working grammar G'' it suffices to transform grammar G' and change right recursion into left recursion. Here is grammar G'':

$$G'' \begin{cases} S \rightarrow E\,`\dashv\text{'} \\ E \rightarrow `i\text{'} \mid E\,`+\text{'}\,`i\text{'} \mid `(\text{'}\,E\,`)\text{'} \mid E\,`+\text{'}\,`(\text{'}\,E\,`)\text{'} \end{cases}$$

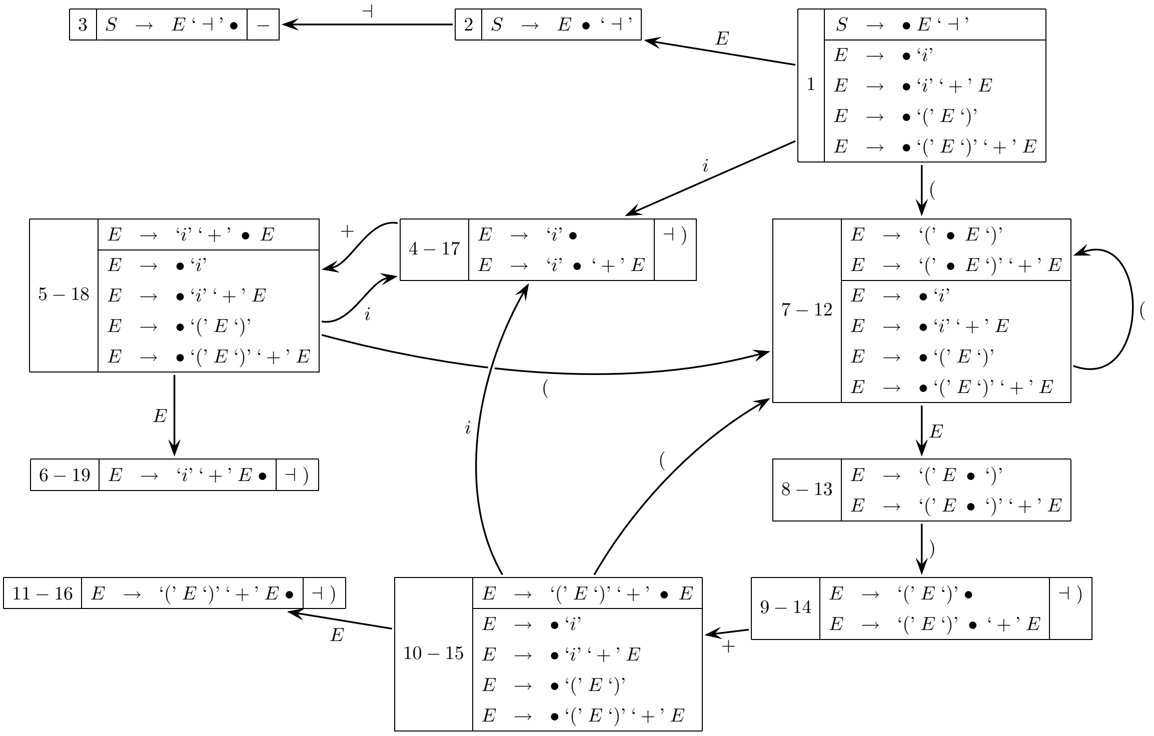
To obtain grammar G'' from G', right recursion has been changed into left recursion, and in this way there are not any more rules that are prefixes of other rules. As observed before, language $L(G)$ is invariant with respect to mirroring (and hence also language $L(G')$). It follows that grammars G' and G'' are equivalent, and therefore G'' is equivalent to grammar G, too. It is somewhat evident that in the driver graph of type $LR(0)$ of G, reduction candidates are contained in isolated macrostates, because the sequences of shift moves that lead to reduction moves are all different from one another. Here is the driver graph of type $LR(0)$ of G:



All the macrostates of the driver graph are adequate, therefore grammar G'' is of type $LR(0)$.
□

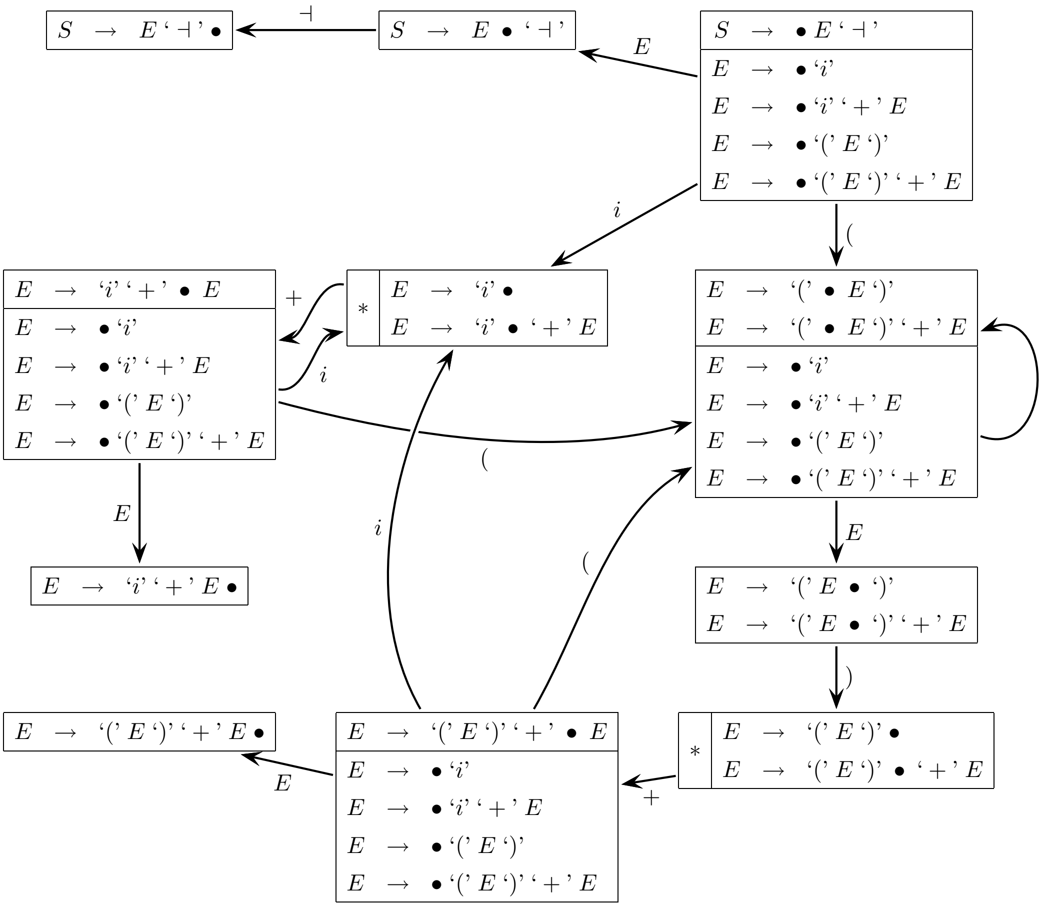
Observation

Incidentally, one may notice that grammar G' is of type $LALR(1)$ (though as said before it is not of type $LR(0)$). To prove this fact, it suffices to draw the driver graph of type $LALR(1)$ of G': overlap the macrostates of the driver graph of type $LR(1)$ of G' that have the same base (that is the macrostates containing the same candidates marked in identical way) and join the lookahead sets of identical reduction candidates. Then one obtains what follows (numerical labels indicate overlapped macrostates):



For convenience only the lookahead sets of reduction candidates are left on show, not those of shift candidates (the latter ones in fact only help to draw the graph but not to use it as the driver of the syntax analyser). One sees soon that all the macrostates of the driver graph of type $LALR(1)$ of grammar G' are adequate. Therefore G' is of type $LALR(1)$.

However grammar G' is not of type $LR(0)$, as it has already been sufficiently argued before. This fact can be proved rigorously here by drawing the driver graph of type $LR(0)$ of G'. As it is well known, such a graph has the same topology of that of type $LALR(1)$, with the only difference that lookahead sets are completely removed. It is therefore immediate to derive it from the $LALR(1)$ graph of G' drawn before. Here it is:



For convenience macrostates are not numbered. One sees soon that there are inadequate macrostates, that is those containing reduction and shift candidates (they are marked with an asterisk). Such macrostates are exactly those containing a rule that is prefix of another rule (according to the observation above). Clearly the prefix rule, as it is shorter, becomes a reduction candidate while the other rule, which is longer, is still a shift candidate, and without resorting to lookahead the syntax analyser cannot solve the shift-reduction conflict. Remember however that, as observed before, grammar G'', which is not right-recursive, is of type $LR(0)$.

Exercise 40 Consider the following grammar G (not in extended form) over the terminal alphabet $\{a, b\}$ and with axiom S:

$$G \begin{cases} S \ \rightarrow \ A \ \dashv \\ A \ \rightarrow \ b\, B\, a \\ B \ \rightarrow \ b\, B \ | \ a\, A \ | \ a \end{cases}$$

Answer the following questions:

1. Discuss briefly whether language $L(G)$ is regular or not.

2. Prove formally that grammar G is of type $LR(1)$.
3. Determine formally whether grammar G is of type $LR(0)$ or not.
4. Discuss briefly whether grammar G is of type $LALR(1)$ or not.

Solution of (1)

If one observes the following derivation of grammar G, which is of self-embedding type in the nonterminal A and B:

$$S \overset{*}{\Rightarrow} b^+Ba \Rightarrow b^+aAa \overset{*}{\Rightarrow} b^+ab^+Baa \Rightarrow b^+ab^+aAaa \Rightarrow b^+ab^+a\dots a\dots aa$$

one concludes soon that language $L(G)$ can be expressed as follows:

$$L(G) = \left\{ \underbrace{b^+a}_{1} \; \underbrace{b^+a}_{2} \; \dots \; \underbrace{b^+a}_{n} \; a \; \underbrace{a}_{n} \; \dots \; \underbrace{a}_{2} \; \underbrace{a}_{1} \;\middle|\; n \geq 1 \right\}$$

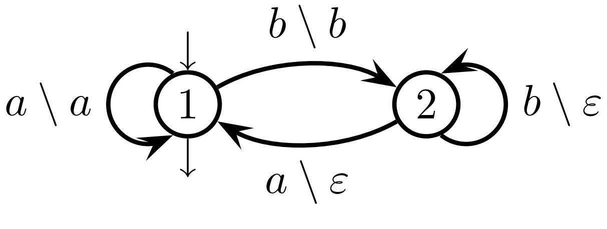
or more formally:

$$L(G) = \{ \; x \mid \quad \forall n \geq 1 \quad x = \left(b^+ \, a\right)^n \; a^n \; \}$$

Clearly a regular expression cannot grant that the number of factors of type b^+a equals the number of final letters a (excluding the middle letter a); therefore language $L(G)$ is not regular. Such a reasoning is not fully formal but is somewhat convincing and suffices for answering reasonably the question. □

Observation

A fully rigorous reasoning is the following. A generic string of language $L(G)$ should be input to a finite state transducer automaton τ, suitably defined, that emits onto the output tape exactly one letter b for each factor of type b^+a it finds in the input tape. A deterministic transducer τ that works as required is the following (may not be the only possible solution):



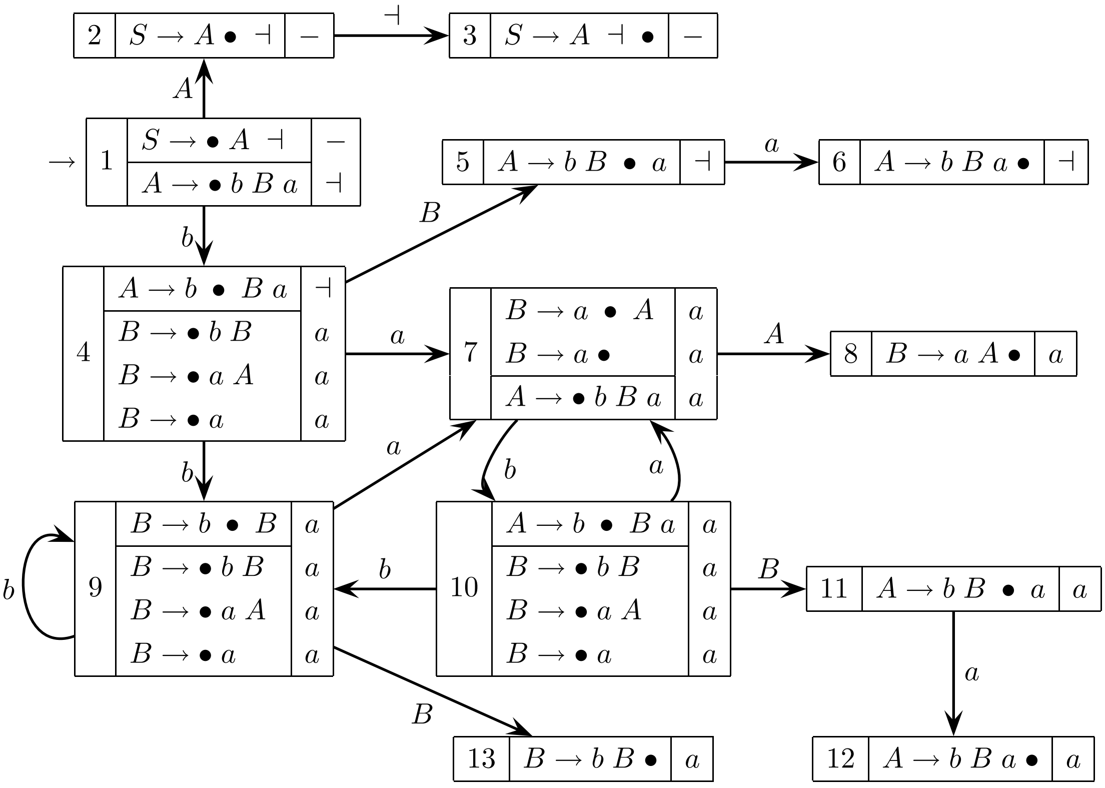
Given an input factor of type b^+a, such an automaton translates the prefix b^+ into exactly one letter b and cancels the suffix a. The final sequence of letters a (including the middle letter a) in the input tape, which are not preceded by any letter b, is output unaltered.

Notice that the source language of transducer τ is the regular language $L' = (\, b^+ a \mid a^* \,)^*$, which is different from language $L(G)$ though obviously it holds $L(G) \subsetneq L'$. This is however not relevant, what matters is that τ translates the strings of $L(G)$ as described above.

Therefore it holds $\tau\,(L(G)) = \{b^n\, a\, a^n \mid \; n \geq 1\}$, which is a notoriously non-regular language (this can be proved for instance by means of the so-called "pumping lemma"). But since the family of regular languages is closed with respect to rational (finite state) transduction, language $L(G)$ cannot be regular, otherwise a contradiction would arise.

Solution of (2)

Grammar G is really $LR(1)$, as the exercise itself states. To prove the statement rigorously, it is necessary to design the driver graph of type $LR(1)$ of the syntx analyser. Here it is:



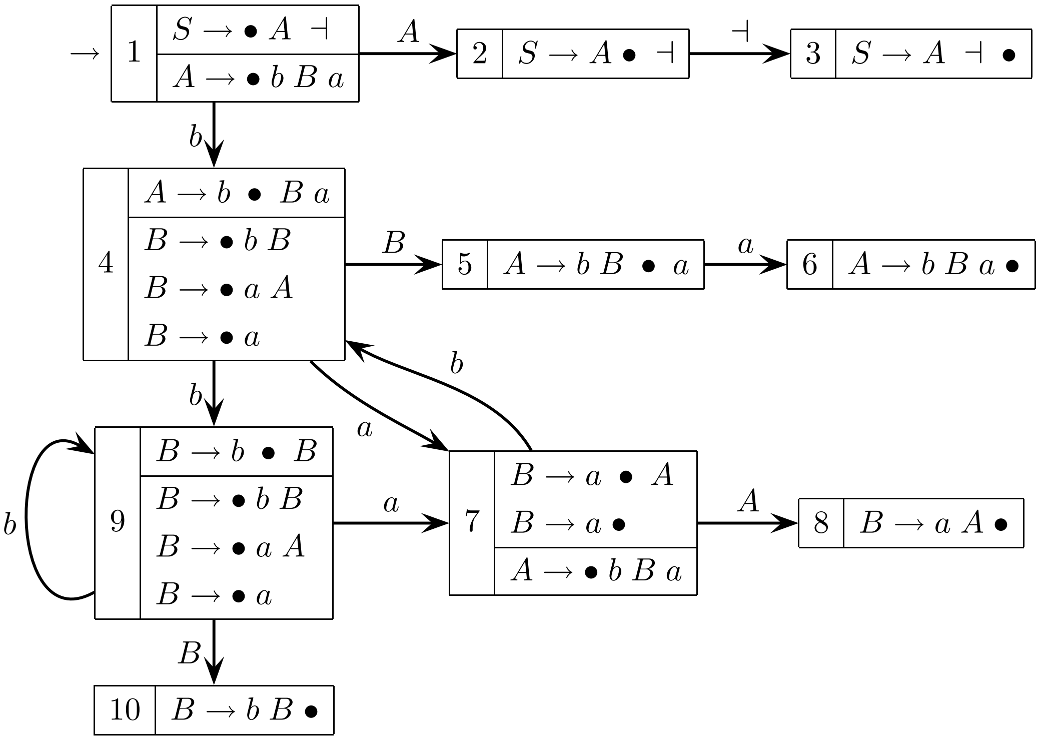
2 | S → A • ⊣ | –
3 | S → A ⊣ • | –
1 | S → • A ⊣ | – ; A → • b B a | ⊣
5 | A → b B • a | ⊣
6 | A → b B a • | ⊣
4 | A → b • B a | ⊣ ; B → • b B | a ; B → • a A | a ; B → • a | a
7 | B → a • A | a ; B → a • | a ; A → • b B a | a
8 | B → a A • | a
9 | B → b • B | a ; B → • b B | a ; B → • a A | a ; B → • a | a
10 | A → b • B a | a ; B → • b B | a ; B → • a A | a ; B → • a | a
11 | A → b B • a | a
12 | A → b B a • | a
13 | B → b B • | a
⊣
A
a
B
b

All the states are adequate, hence grammar G is actually of type $LR(1)$. □

Solution of (3)

By observing the driver graph of type $LR(1)$ and noticing that there are mixed shift-reduction macrostates, one can conclude soon that grammar G is not of type $LR(0)$.

By designing the driver graph of type $LR(0)$ of the syntax analyser, one sees in a more rigorous and extensive way that grammar G is not of type $LR(0)$. Here is the driver graph:

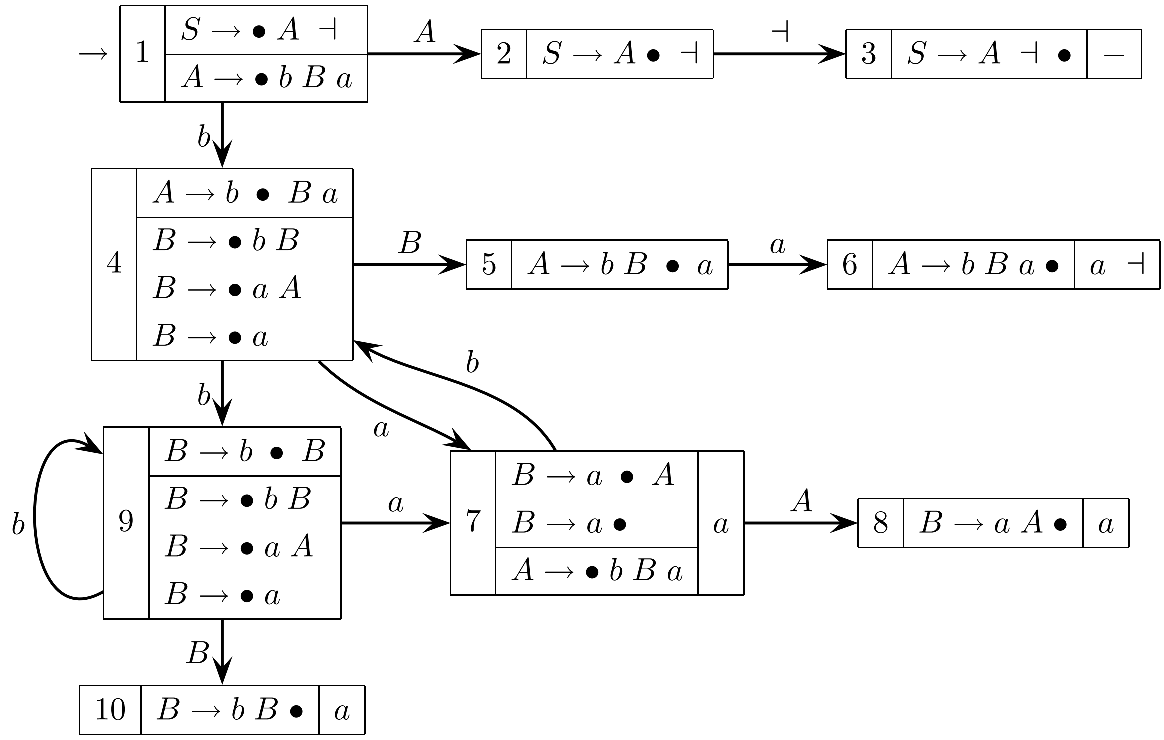

1
$S \rightarrow \bullet A \dashv$
$A \rightarrow \bullet b B a$
A
2
$S \rightarrow A \bullet \dashv$
$\dashv$
3
$S \rightarrow A \dashv \bullet$
b
4
$A \rightarrow b \bullet B a$
$B \rightarrow \bullet b B$
$B \rightarrow \bullet a A$
$B \rightarrow \bullet a$
B
5
$A \rightarrow b B \bullet a$
a
6
$A \rightarrow b B a \bullet$
b
b
a
9
$B \rightarrow b \bullet B$
$B \rightarrow \bullet b B$
$B \rightarrow \bullet a A$
$B \rightarrow \bullet a$
b
a
7
$B \rightarrow a \bullet A$
$B \rightarrow a \bullet$
$A \rightarrow \bullet b B a$
A
8
$B \rightarrow a A \bullet$
B
10
$B \rightarrow b B \bullet$

There is an inadequate macrostate, namely state 7, with a shift-reduction conflict; therefore grammar G is not of type $LR(0)$.

The conflicting reduction candidate is $B \rightarrow a \bullet$, which conflicts with the shift candidate $B \rightarrow a \bullet A$. In fact both candidates expand the same nonterminal B, but the former rule is a prefix of the latter one. Therefore they lead to the same macrostate (as they expand the same nonterminal and are analysed in parallel). However, without lookahead it is impossible to decide deterministically which of reduction and shift is the right move to execute. □

Solution of (4)

It proves that grammar G is of type $LALR(1)$. It suffices to fold the driver graph of type $LR(1)$ and realise that the thus obtained $LALR(1)$ driver graph does not contain any inadequate macrostate, as lookahead always succeeds in solving the shift-reduction conflict identified above. Here is the driver graph (lookahead is displayed only for reduction candidates):



All the macrostates are adequate, therefore grammar G is of type $LALR(1)$. Notice that in the state 7 the reduction candidate $B \to a\,\bullet$ has lookahead a and hence can be separated from the two shift candidates, which have outgoing arcs labeled with A and b. □

Exercise 41 The following grammar G (not in extended form) is given:

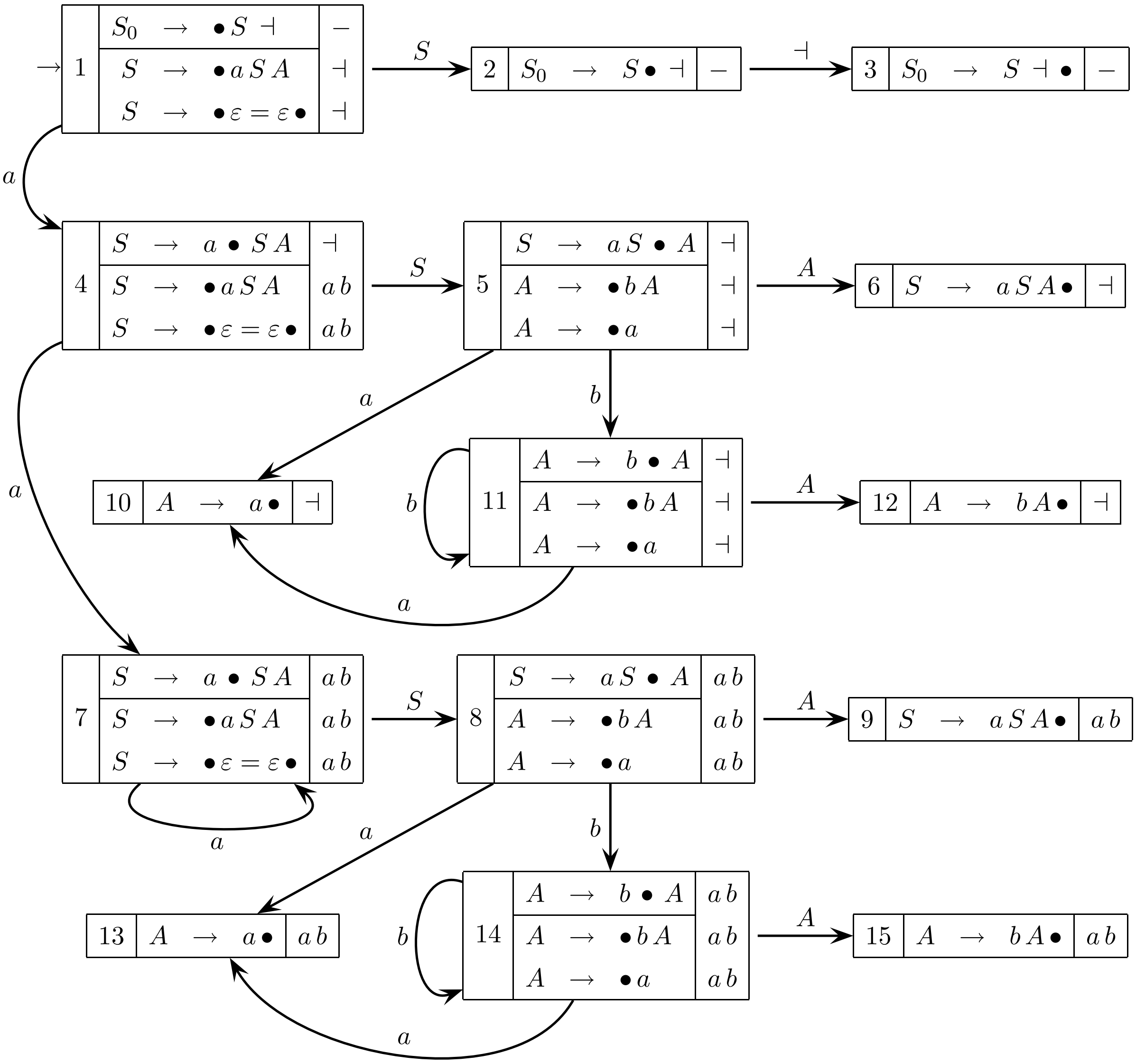
$$G \begin{cases} S \;\to\; a\,S\,A \;\mid\; \varepsilon \\ A \;\to\; b\,A \;\mid\; a \end{cases}$$

Answer the following questions:

1. Draw the $LR(1)$ driver graph (the bottom-up recognizer) of grammar G.
2. Check whether the drawn driver graph satisfies the conditions $LR(1)$, $LALR(1)$ or $LR(0)$, and explain why.

Solution of (1)

Here is the driver graph of grammar G, of type $LR(1)$ (here G is completed by adding the terminator ⊣ and the new axiom S_0):


1
$S_0 \to \bullet S \dashv$ | –
$S \to \bullet a S A$ | ⊣
$S \to \bullet \varepsilon = \varepsilon \bullet$ | ⊣
S
2 $S_0 \to S \bullet \dashv$ | –
⊣
3 $S_0 \to S \dashv \bullet$ | –
a
4
$S \to a \bullet S A$ | ⊣
$S \to \bullet a S A$ | $a\,b$
$S \to \bullet \varepsilon = \varepsilon \bullet$ | $a\,b$
S
5
$S \to a S \bullet A$ | ⊣
$A \to \bullet b A$ | ⊣
$A \to \bullet a$ | ⊣
A
6 $S \to a S A \bullet$ | ⊣
a
b
10 $A \to a \bullet$ | ⊣
b
11
$A \to b \bullet A$ | ⊣
$A \to \bullet b A$ | ⊣
$A \to \bullet a$ | ⊣
A
12 $A \to b A \bullet$ | ⊣
a
a
7
$S \to a \bullet S A$ | $a\,b$
$S \to \bullet a S A$ | $a\,b$
$S \to \bullet \varepsilon = \varepsilon \bullet$ | $a\,b$
S
8
$S \to a S \bullet A$ | $a\,b$
$A \to \bullet b A$ | $a\,b$
$A \to \bullet a$ | $a\,b$
A
9 $S \to a S A \bullet$ | $a\,b$
a
a
b
13 $A \to a \bullet$ | $a\,b$
b
14
$A \to b \bullet A$ | $a\,b$
$A \to \bullet b A$ | $a\,b$
$A \to \bullet a$ | $a\,b$
A
15 $A \to b A \bullet$ | $a\,b$
a

In total the graph has 15 nodes (the first three nodes are essentially meant for the terminator and could be omitted). □

Solution of (2)

Grammar G cannot be of type $LR(0)$, as it contains a null rule. One sees soon however that the driver graph of G does not satisfy condition $LR(1)$, because it contains two states (namely 4 and 7) with a shift-reduction conflict. Therefore grammar G is not of type $LR(1)$ and hence not of type $LALR(1)$ either. □

Exercise 42 For the following language L over alphabet $\{a, b, c, d\}$:

$$L = \{\ a^n\, b\, c^n \ \mid \quad n \geq 0\ \} \ \cup \ \{\ a^n\, c^n\, d \ \mid \quad n \geq 0\ \}$$

one must design a grammar G suited to deterministic syntax analysis. One is free to choose which determinism condition of $LL(.)$, $LR(.)$ or $LALR(.)$ one prefers to use.

Answer the following questions:

1. Write the requested grammar G (and say which condition is assumed).
2. Check formally whether grammar G satisfies the chosen determinism condition and explain why it works.

Solution of (1)

Here it is chosen to give a simple and compact grammar G that is of type $LR(1)$. Here it is (axiom S):

$$G \begin{cases} S \rightarrow B & \mathcal{L}'_1 = \{a, b\} & \mathcal{L}'_2 = \{aa, ab, b \dashv\} & \ldots \\ S \rightarrow D\, d & \mathcal{L}''_1 = \{a, d\} & \mathcal{L}''_2 = \{aa, ac, d \dashv\} & \ldots \\ B \rightarrow a\, B\, c & \mathcal{L}'_1 = \{a\} & & \\ B \rightarrow b & \mathcal{L}''_1 = \{b\} & & \\ D \rightarrow a\, D\, c & \mathcal{L}'_1 = \{a\} & & \\ D \rightarrow \varepsilon & \mathcal{L}''_1 = \{c\} & & \end{cases}$$

Intuitively grammar G generates, by means of the former and latter axiomatic rule, the former and latter component of language L, respectively.

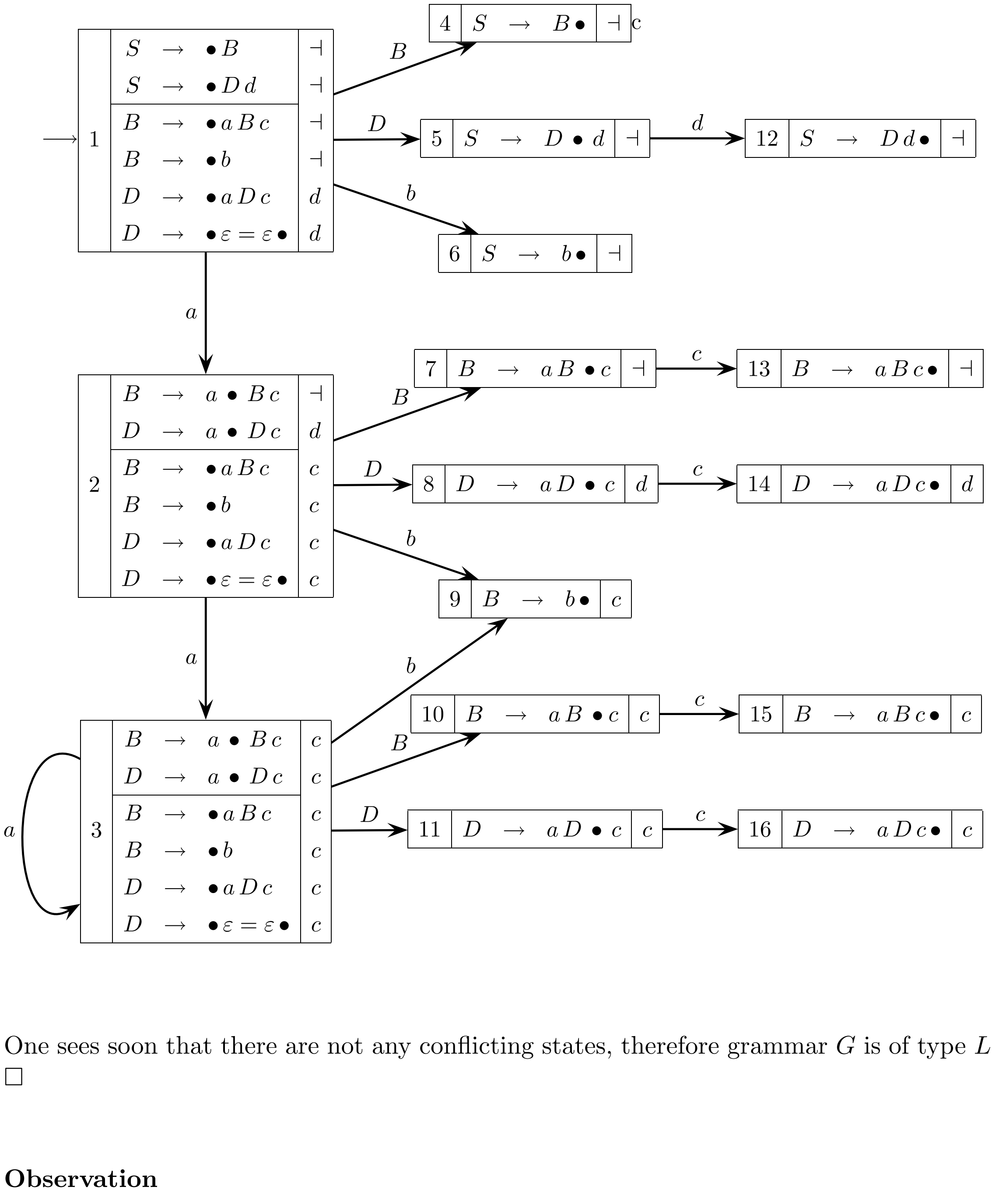
However grammar G is not $LL(k)$, for any $k \geq 1$: the two alternative axiomatic rules have lookahead sets $\mathcal{L}'_k$ and $\mathcal{L}''_k$ (of order $k \geq 1$) both containing string a^k, for every $k \geq 1$; therefore they are not disjoint. For instance, for two such rules it holds $\mathcal{L}'_1 = \{a, b\}$ and $\mathcal{L}''_1 = \{a, d\}$, $\mathcal{L}'_2 = \{aa, ab, b \dashv\}$ and $\mathcal{G}''_2 = \{aa, ac, d \dashv\}$, and so on. The other rules, not axiomatic, do not cause any problem.

It seems impossible to find a grammar that is of type $LL(k)$ for some $k \geq 1$. In fact the recursive descent syntax analyser should read an arbitrary number of letters a before realising whether the self-embedding structure $a^n\, c^n$ contains a letter b or is followed by a letter d, and on the other side such two choices are mutually exclusive and use rules that are necessarily different, and therefore not unifiable.

It is however easy to realise that grammar G is $LR(1)$ (but not $LR(0)$ due to the null rule $D \rightarrow \varepsilon$). Intuitively, first one has to push letters a and then to pop them for counting letters c. If letters a and c are separated by letter b, one has to read such a letter and then to go into a state that does not imply the presence of letter d at the end; otherwise one has to go into a state that implies the presence of letter d at the end. Such a behaviour is compliant with the competence of a deterministic syntax analyser of type $LR(1)$, as such a machine can use states with some freedom. □

Solution of (2)

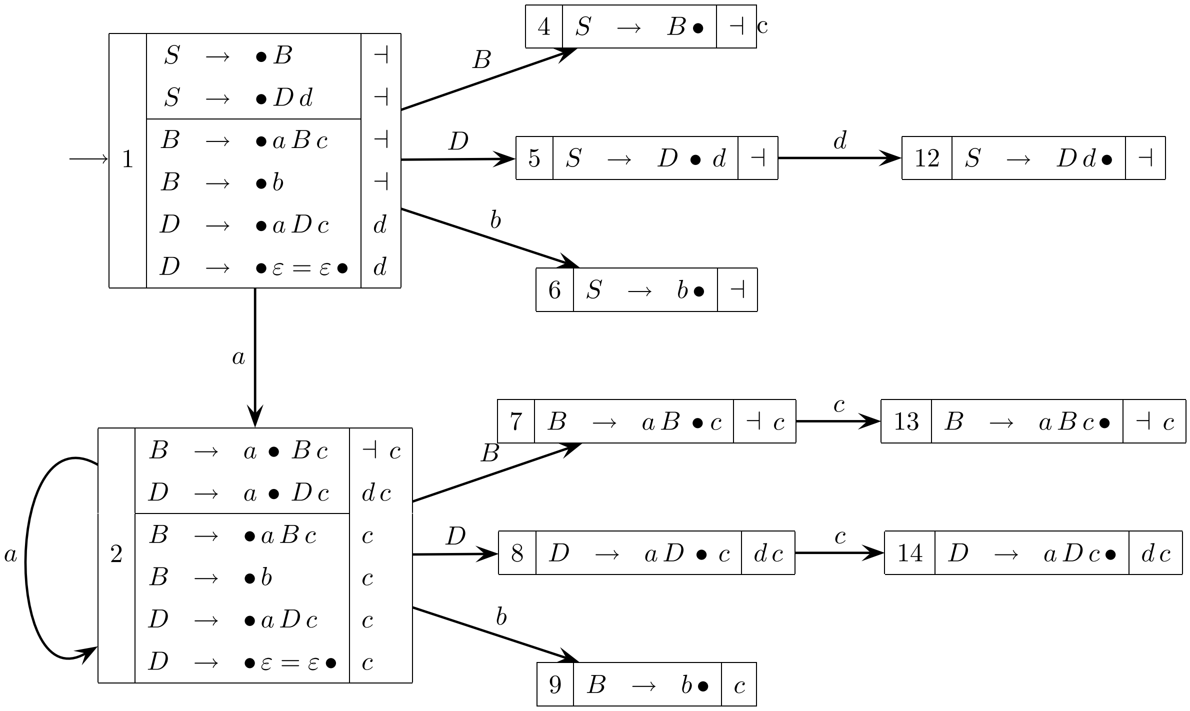
However the formal verification that the grammar G given before is of type $LR(1)$ must be done by drawing and analysing the driver graph of the $LR(1)$ syntax analyser associated with G. Here is the driver graph (the new axiomatic rule $S_0 \to S \dashv$ is assumed):



One sees soon that there are not any conflicting states, therefore grammar G is of type $LR(1)$. □

Observation

It is also easy to realise that grammar G is of type $LALR(1)$: if the driver graph is folded to merge states with the same base, there are not any conflicting states anyway (but remember that as said before G is not of type $LR(0)$). Here is the $LALR(1)$ driver graph:



As one sees soon, there are not any conflicts. Topologically the driver graph of type $LALR(1)$ coincides with that of type $LR(0)$, but if the lookahead sets were removed (and thus the driver graph were reduced to the pure $LR(0)$ one), then states 1 and 2 would contain shift-reduction conflicts (mainly due to the null rule $D \to \varepsilon$).

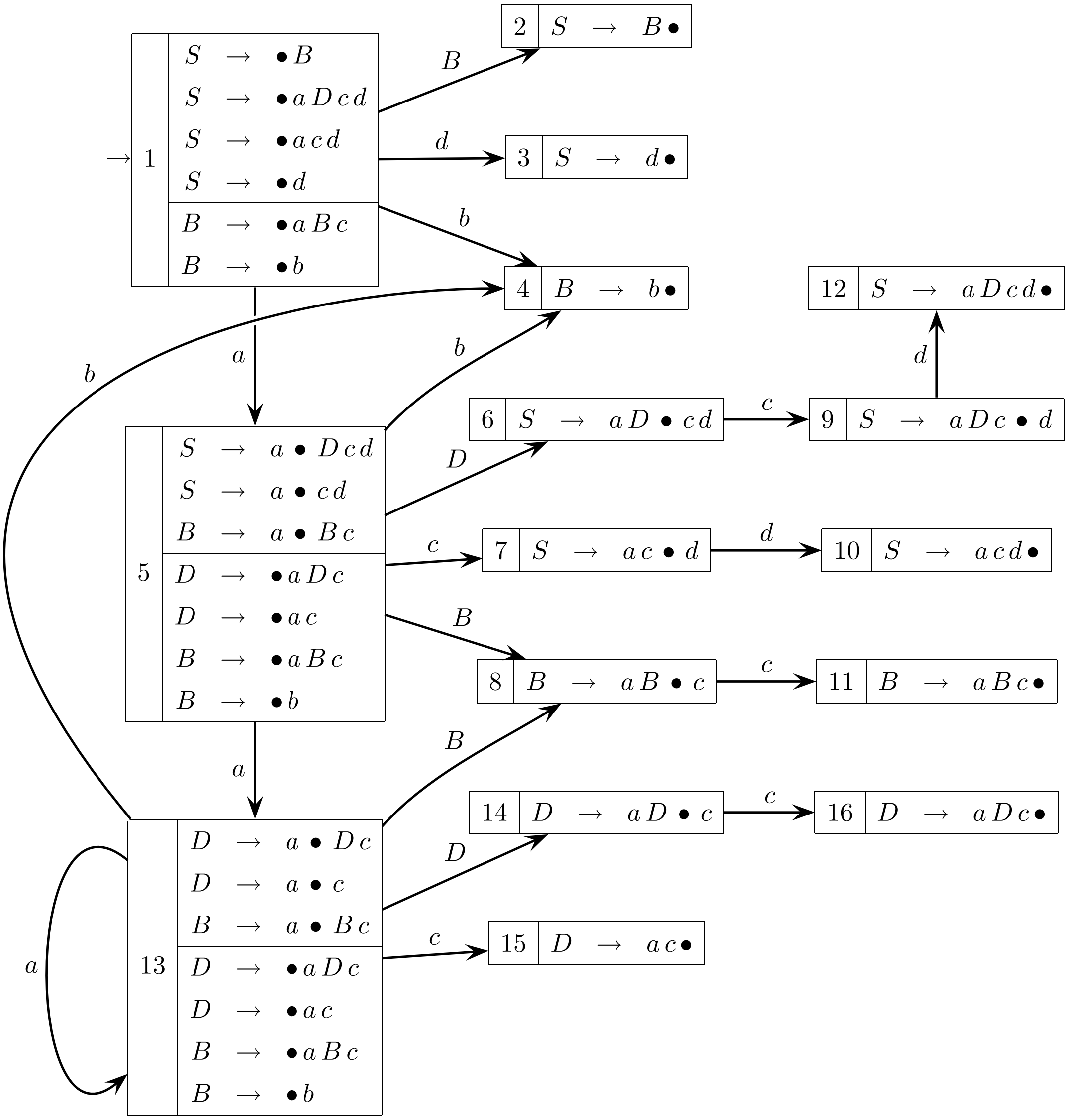
Observation

With some more effort, it is not difficult to design a grammar G', equivalent to grammar G, that is of type $LR(0)$, though not as simple and compact as the version before. Here it is:

$$G' \left\{ \begin{array}{lll rcl rcl rcl}
S & \to & B & \mathcal{L}_1^I & = & \{a, b\} & \mathcal{L}_2^I & = & \{aa, ab, b \dashv\} & \mathcal{L}_3^I & = & \{aaa, \ldots\} \\
S & \to & a\,D\,c\,d & \mathcal{L}_1^{II} & = & \{a\} & \mathcal{L}_2^{II} & = & \{aa\} & \mathcal{L}_3^{II} & = & \{aaa, \ldots\} \\
S & \to & a\,c\,d & \mathcal{L}_1^{III} & = & \{a\} & \mathcal{L}_2^{III} & = & \{ac\} & & & \\
S & \to & d & \mathcal{L}_1^{IV} & = & \{d\} & & & & & & \\
B & \to & a\,B\,c & \mathcal{L}_1^I & = & \{a\} & & & & & & \\
B & \to & b & \mathcal{L}_1^{II} & = & \{b\} & & & & & & \\
D & \to & a\,D\,c & \mathcal{L}_1^I & = & \{a\} & \mathcal{L}_2^I & = & \{aa\} & & & \\
D & \to & a\,c & \mathcal{L}_1^{II} & = & \{a\} & \mathcal{L}_2^{II} & = & \{ac\} & & &
\end{array} \right.$$

First notice that grammar G' is neither of type $LL(1)$, nor $LL(2)$ nor $LL(3)$, as the lookahead sets aside (computed intuitively) prove, because the alternative axiomatic rules do not have disjoint lookahead. Actually G' is not of type $LL(k)$ either, for any $k \geq 4$, because both the lookahead sets $\mathcal{L}_k^I$ and $\mathcal{L}_k^{II}$ of order $k \geq 1$ of nonterminal S contain string a^k, as one sees easily. This is the same behaviour as already observed for the previous version G, as it has been argued before that such a phenomenon is inherent to language $L(G)$.

However grammar G is of type $LR(0)$ and this is proved formally by drawing the driver graph of type $LR(0)$, as follows:



All the states of the graph are adequate (there are not any conflict), therefore the proposed new grammar G' is of type $LR(0)$.

4.3 Earley Algorithm

Exercise 43 There are given the following grammar G (with axiom S), not in extended form:

$$G \left\{ \begin{array}{lcl} S & \rightarrow & a\ B\ S\ b \\ S & \rightarrow & a\ B \\ B & \rightarrow & b\ B \\ B & \rightarrow & \varepsilon \end{array} \right.$$

and the following sample string, which is a phrase of language $L(G)$:

$$a\ b\ a\ b$$

Answer the following questions:

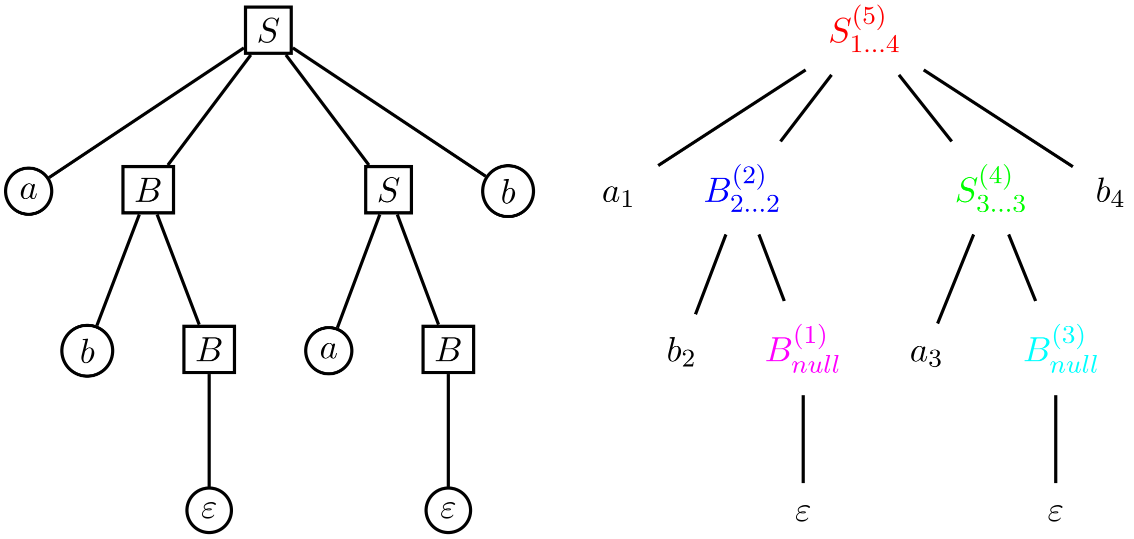
1. Simulate the recognition process of the sample string by means of the Earley algorithm.
2. Draw the syntax tree of the recognised sample string.

Solution of (1)

The simulation of the recognition process of the sample string $a\,b\,a\,b$ by means of the Earley algorithm is shown in the next page.

Solution of (2)

The syntax tree of the sample string $a\,b\,a\,b$ is the following (on the left side):



By resorting to the colours and the denumeration of the reduction candidates in the simulation table shown above, one sees soon the relation between the nodes of the syntax tree and the reduction candidates in the table (tree on the right side).

It would not be difficult to verify that grammar G is of type $LR(1)$ (obviously G is not of type $LR(0)$ due to the null rule $B \to \varepsilon$). Therefore G is not ambiguous and the syntax tree of the recognised sample string is certainly unique. $\square$

Simulation scheme of the Earley algorithm

state 0	pos. a	state 1	pos. b	state 2	pos. a	state 3	pos. b	state 4	pos. $\dashv$
$S \to \bullet\, a\, B\, S\, b$	0	$S \to a \bullet B\, S\, b$	0	$B \to b \bullet B$	1	$S \to a \bullet B\, S\, b$	2	$B \to b \bullet B$	3
$S \to \bullet\, a\, B$	0	$S \to a \bullet B$	0	$B \to \bullet\, b\, B$	2	$S \to a \bullet B$	2	$\mathbf{S} \to a\, B\, S\, b \bullet^{(5)}$	0
		$B \to \bullet\, b\, B$	1	$\mathbf{B} \to \bullet\, \varepsilon = \varepsilon \bullet^{(1)}$	2	$B \to \bullet\, b\, B$	3	$B \to \bullet\, b\, B$	4
		$B \to \bullet\, \varepsilon = \varepsilon \bullet$	1	$\mathbf{B} \to b\, B \bullet^{(2)}$	1	$\mathbf{B} \to \bullet\, \varepsilon = \varepsilon \bullet^{(3)}$	3	$B \to \bullet\, \varepsilon = \varepsilon \bullet$	4
		$S \to a\, B \bullet S\, b$	0	$S \to a\, B \bullet S\, b$	0	$S \to a\, B \bullet S\, b$	2	$B \to b\, B \bullet$	3
		$S \to a\, B \bullet$	0	$S \to a\, B \bullet$	0	$\mathbf{S} \to a\, B \bullet^{(4)}$	2	$S \to a\, B \bullet S\, b$	2
		$S \to \bullet\, a\, B\, S\, b$	1	$S \to \bullet\, a\, B\, S\, b$	2	$S \to \bullet\, a\, B\, S\, b$	3	$S \to a\, B \bullet$	2
		$S \to \bullet\, a\, B$	1	$S \to \bullet\, a\, B$	2	$S \to \bullet\, a\, B$	3	$S \to \bullet\, a\, B\, S\, b$	4
						$S \to a\, B\, S \bullet b$	0	$S \to \bullet\, a\, B$	4
								$S \to a\, B\, S \bullet b$	0

In the state 4 there is an axiomatic reduction candidate (number 5 coloured in red), labeled with starting state 0.
Such a candidate causes the sample string $a\, b\, a\, b$ to be recognised.
The colours and numeration of some reduction candidates refer to the nodes of the syntax trees (see the following question). □

Exercise 44 The following grammar G is given (axiom S), not in extended form:

$$G \begin{cases} S \rightarrow a\,S\,B\,b\,B \\ S \rightarrow c \\ B \rightarrow b\,B \\ B \rightarrow \varepsilon \end{cases}$$

Answer the following questions:

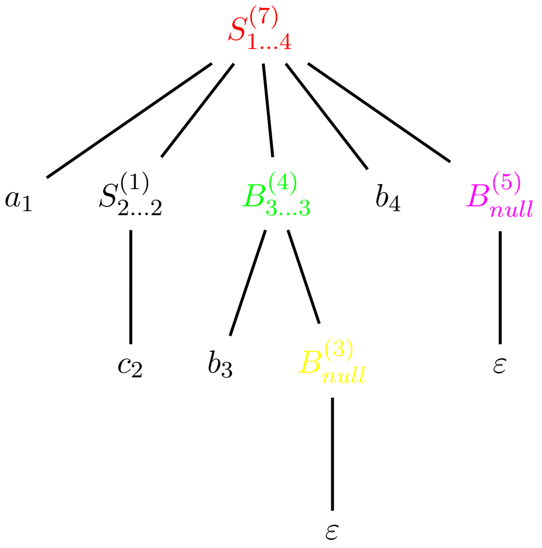
1. By means of the Earley parsing algorithm, simulate the recognition process of the following sample string (which belongs to language $L(G)$): $a\,c\,b\,b$.
2. Draw all the syntax trees of the sample string given before and show the correspondence between tree nodes and the reduction candidates identified in the recognition process.

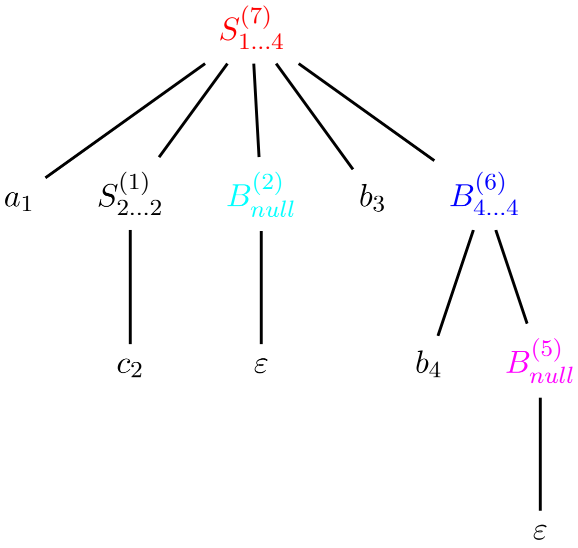
Solution of (1)

The simulation of the recognition process of string $a\,c\,b\,b$ is shown in the next page.

Solution of (2)

Clearly grammar G is ambiguous and the sample string can be generated in two different ways. Therefore there are two syntax trees. Here they are:





The correspondence between nodes and the reduction candidates shown in the simulation table before is rather evident, and it is also highlighted by colouring and numbering the nodes and the reduction candidates referred to by the nodes. □

Simulation scheme of the Earley algorithm									
state 0	pos. a	state 1	pos. c	state 2	pos. b	state 3	pos. b	state 4	pos. $\dashv$
$S \rightarrow \bullet a\,S\,B\,b\,B$	0	$S \rightarrow a \bullet S\,B\,b\,B$	0	$\mathbf{S} \rightarrow c\,\bullet^{(1)}$	1	$B \rightarrow b \bullet B$	2	$B \rightarrow b \bullet B$	3
$S \rightarrow \bullet c$	0	$S \rightarrow \bullet a\,S\,B\,b\,B$	1	$S \rightarrow a\,S \bullet B\,b\,B$	0	$S \rightarrow a\,S\,B\,b \bullet B$	0	$S \rightarrow a\,S\,B\,b \bullet B$	0
		$S \rightarrow \bullet c$	1	$B \rightarrow \bullet b\,B$	2	$B \rightarrow \bullet b\,B$	3	$B \rightarrow \bullet b\,B$	4
				$\mathbf{B} \rightarrow \bullet \varepsilon = \varepsilon \bullet^{(2)}$	2	$\mathbf{B} \rightarrow \bullet \varepsilon = \varepsilon \bullet^{(3)}$	3	$\mathbf{B} \rightarrow \bullet \varepsilon = \varepsilon \bullet^{(5)}$	4
				$S \rightarrow a\,S\,B \bullet b\,B$	0	$\mathbf{B} \rightarrow b\,B \bullet^{(4)}$	2	$\mathbf{B} \rightarrow b\,B \bullet^{(6)}$	3
						$S \rightarrow a\,S\,B\,b\,B \bullet$	0	$\mathbf{S} \rightarrow a\,S\,B\,b\,B \bullet^{(7)}$	0
						$S \rightarrow a\,S\,B \bullet b\,B$	0	$B \rightarrow b\,B \bullet$	2
								$S \rightarrow a\,S\,B \bullet b\,B$	0

In the state 4 there is an axiomatic reduction candidate (number 7 coloured in red), labeled with starting state 0.
Such a candidate causes the sample string $a\,c\,b\,b$ to be recognised.
The colours and numeration of some reduction candidates refer to the nodes of the syntax trees (see the following question). □

Chapter 5

Transduction and Semantic

5.1 Syntax Transduction

Exercise 45 Consider the following regular expression R, which generates a regular language L over alphabet $\{a, b, c\}$:

$$R = a^+ \ b \ a^+ \ \cup \ a^+ \ c \ a^+$$

and the following transduction function $\tau \colon L \to \{d\}^+$:

$$\tau(a^m \ b \ a^n) = d^m \qquad \tau(a^m \ c \ a^n) = d^n \qquad m, n \geq 1$$

The behaviour of transduction τ on the source language L is split into two cases.

Examples: $\tau(a \ a \ a \ b \ a \ a) = d \ d \ d$ $\qquad$ $\tau(a \ a \ a \ c \ a \ a) = d \ d$

Answer the following questions:

1. Write the regular transduction expression R_τ of the transduction τ described above.
2. Design a finite state transducer T that computes transduction τ.
3. Determine whether the finite state transducer T is deterministic or not.
4. Discuss whether transduction τ could be computed deterministically in some way, for instance by means of a finite state or pushdown transducer, or by means of a deterministic LL or LR syntactic transduction scheme.

Solution of (1)

Here is the requested regular transduction expression R_τ, obtained in an intuitive way from the specifications of language L and transduction τ:

$$R_\tau = \left(\frac{a}{d}\right)^+ \ \frac{b}{\varepsilon} \ \left(\frac{a}{\varepsilon}\right)^+ \ \cup \ \left(\frac{a}{\varepsilon}\right)^+ \ \frac{c}{\varepsilon} \ \left(\frac{a}{d}\right)^+$$

The usual (but in general less compact) formulation of the regular transduction expression R_τ is the following:

$$R_\tau = (\, a \ \{d\} \,)^+ \ b \ a^+ \ \cup \ a^+ \ c \ (\, a \ \{d\} \,)^+$$

And here are the two derivations of expression R_τ that prove the two examples of transduction τ given in the exercise ($\tau(a\ a\ a\ b\ a\ a) = d\ d\ d$ and $\tau(a\ a\ a\ c\ a\ a) = d\ d$):

$$\left(\tfrac{a}{d}\right)^+ \tfrac{b}{\varepsilon} \left(\tfrac{a}{\varepsilon}\right)^+ \cup \left(\tfrac{a}{\varepsilon}\right)^+ \tfrac{c}{\varepsilon} \left(\tfrac{a}{d}\right)^+ \Rightarrow \left(\tfrac{a}{d}\right)^+ \tfrac{b}{\varepsilon} \left(\tfrac{a}{\varepsilon}\right)^+ \Rightarrow \tfrac{aaa}{ddd} \tfrac{b}{\varepsilon} \left(\tfrac{a}{\varepsilon}\right)^+ \Rightarrow \tfrac{aaa}{ddd} \tfrac{b}{\varepsilon} \tfrac{aa}{\varepsilon\varepsilon} = \tfrac{aaabaa}{ddd}$$

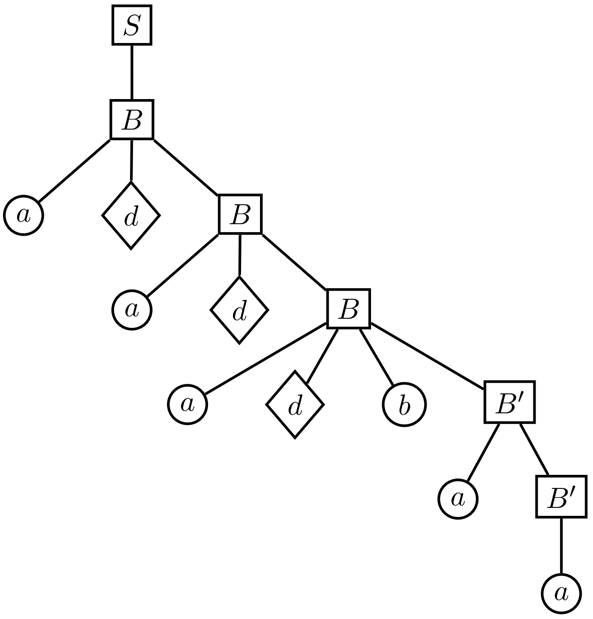
$$\left(\tfrac{a}{d}\right)^+ \tfrac{b}{\varepsilon} \left(\tfrac{a}{\varepsilon}\right)^+ \cup \left(\tfrac{a}{\varepsilon}\right)^+ \tfrac{c}{\varepsilon} \left(\tfrac{a}{d}\right)^+ \Rightarrow \left(\tfrac{a}{\varepsilon}\right)^+ \tfrac{c}{\varepsilon} \left(\tfrac{a}{d}\right)^+ \Rightarrow \tfrac{aaa}{\varepsilon\varepsilon\varepsilon} \tfrac{c}{\varepsilon} \left(\tfrac{a}{d}\right)^+ \Rightarrow \tfrac{aaa}{\varepsilon\varepsilon\varepsilon} \tfrac{c}{\varepsilon} \tfrac{aa}{dd} = \tfrac{aaacaa}{dd}$$

These two derivation examples prove sufficiently that expression R_τ computes transduction τ really and therefore is correct.

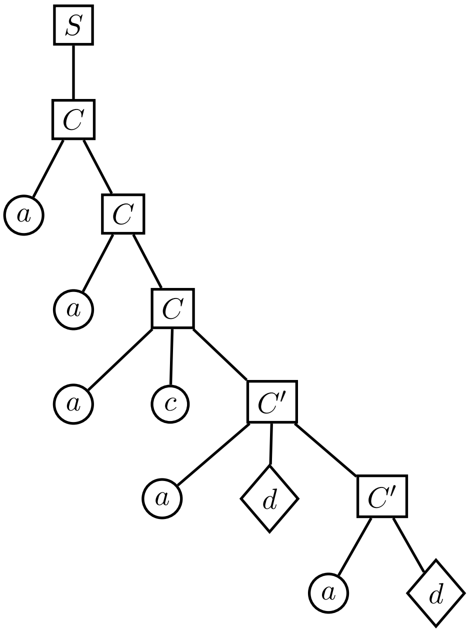
Proceed as follows to obtain a valid transduction expression in a more systematic way. First write the right-linear grammar G_s that generates source language L, then transform G_s into a syntax transduction scheme G_τ (again of right-linear type) and finally obtain from G_τ the requested regular transduction expression. Here is how to do (axiom S):

$$G_s \begin{cases} S \ \rightarrow \ B \ \mid \ C \\ B \ \rightarrow \ a\, B \ \mid \ a\, b\, B' \\ C \ \rightarrow \ a\, C \ \mid \ a\, c\, C' \\ B' \ \rightarrow \ a\, B' \ \mid \ a \\ C' \ \rightarrow \ a\, C' \ \mid \ a \end{cases} \qquad G_\tau \begin{cases} S \ \rightarrow \ B \ \mid \ C \\ B \ \rightarrow \ a\, \{\, d \,\}\, B \ \mid \ a\, \{\, d \,\}\, b\, B' \\ C \ \rightarrow \ a\, C \ \mid \ a\, c\, C' \\ B' \ \rightarrow \ a\, B' \ \mid \ a \\ C' \ \rightarrow \ a\, \{\, d \,\}\, C' \ \mid \ a\, \{\, d \,\} \end{cases}$$

Source grammar G_s (shown above on the left) distinguishes between nonterminals B, C and between nonterminals B', C', in order to allow different translations of the sequence of terminal a that precedes and follows the letter b or c occurring in the middle of the source string, respectively. Then transduction scheme G_τ is obtained easily from G_s (shown above on the right). Letter a is soon translated into letter d in the rules expanding nonterminals B and C'. Here are the two syntax trees of the transduction examples given in the exercise:



$\tau(a\ a\ a\ b\ a\ a) = d\ d\ d$



$\tau(a\ a\ a\ c\ a\ a) = d\ d$

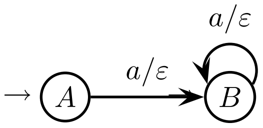
These two trees justify sufficiently the correctness of syntax transduction scheme G_τ. Finally one obtains from G_τ the following formulation of the regular transduction expression R'_τ, for instance by using the method of the linear language equations:

$$R'_\tau = (a\ \{\ d\ \})^+\ b\ a^+\ \mid\ a^+\ c\ (a\ \{\ d\ \})^+$$

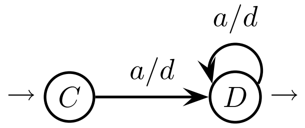
Expression R'_τ is actually identical to the regular transduction expression R_τ given before, though it is written in a slightly different notational style. □

Solution of (2)

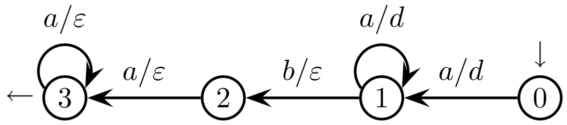
A finite state transducer T that computes transduction τ is readily obtained form regular transduction expression R_τ by applying the Thompson or modular construction. Here it is, constructed step by step:



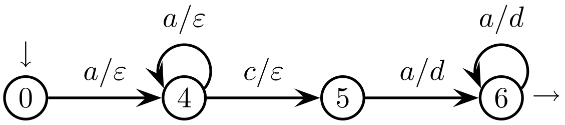
$$\left(\frac{a}{\varepsilon}\right)^{+}$$



$$\left(\frac{a}{d}\right)^{+}$$

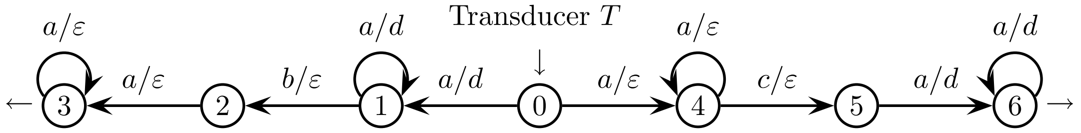


$$\left(\frac{a}{d}\right)^{+}\ \frac{b}{\varepsilon}\ \left(\frac{a}{\varepsilon}\right)^{+}$$



$$\left(\frac{a}{\varepsilon}\right)^{+}\ \frac{c}{\varepsilon}\ \left(\frac{a}{d}\right)^{+}$$

Transducer T



$$\left(\frac{a}{d}\right)^{+}\ \frac{b}{\varepsilon}\ \left(\frac{a}{\varepsilon}\right)^{+}\ \cup\ \left(\frac{a}{\varepsilon}\right)^{+}\ \frac{c}{\varepsilon}\ \left(\frac{a}{d}\right)^{+}$$

Here the Thompson (modular) construction is applied with some shortcut, for simplicity. Transducer T has seven states in total. □

Solution of (3)

One sees soon that the recogniser automaton underlying to the above designed finite state transducer T is indeterministic in the state 0. In fact there are two outgoing arcs with the same input label a and hence the transducer T itself is indeterministic. One should remember that in general the Thompson or modular construction produces indeterministic automata. □

Observation

It seems very difficult (if not impossible) to compute transduction τ by means of a finite state deterministic transducer, as the letters b and c that discriminate between the two behaviours of τ (whether to output m rather than n letters d) are found inside of the input string, while for a deterministic finite state transducer to work correctly it would be necessary to read this piece of information at the beginning of the operation, before outputting anything.

First solution of (4)

It is not difficult to convince oneself that transduction τ can be computed deterministically by means of a pushdown transducer, with recognition by final state. Such a transducer works according to the following strategy:

- First, given the input string $a^m\, e\, a^n$ where $e \in \{b, c\}$ $(m, n \geq 1)$, the transducer reads from the input tape m letters a and pushes them onto the stack, each encoded by means of the stack symbol A; while doing so the transducer does not output anything.

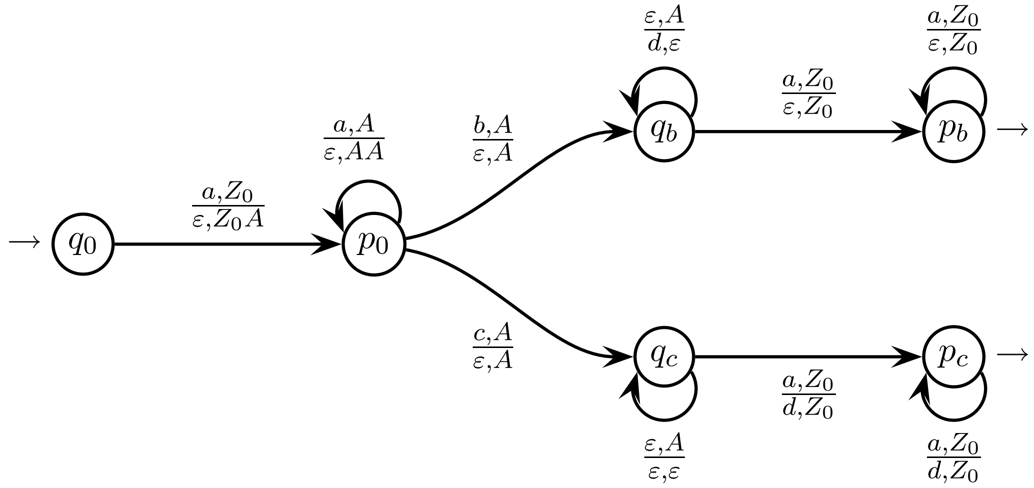
- Then the transducer enters state q_b or q_c depending on whether it reads letter b or c in the input tape (that is depending on whether $e = b$ or $e = c$), respectively, and in both cases it does not output anything.

- Finally the transducer splits its behaviour as follows:

  - if it is in the state q_b, flushes the stack and outputs one letter d for each stack symbol A, then reads from the input tape n letters a without outputting anything
  - if it is in the state q_c, flushes the stack without outputting anything, then reads from the input tape n letters a and outputs one letter d for each of them

  In both cases, at the end the transducer recognises by final state and terminates.

More precisely, the transducer should check also that the input tape contains at least one letter a before and after character e (that is b or c). Here is a state-transition graph of a deterministic pushdown automaton that implements the above described strategy:


$\frac{\varepsilon, A}{d, \varepsilon}$
$\frac{a, Z_0}{\varepsilon, Z_0}$
$\frac{a, Z_0}{\varepsilon, Z_0}$
$\frac{a, A}{\varepsilon, AA}$
$\frac{b, A}{\varepsilon, A}$
q_b
p_b
$\frac{a, Z_0}{\varepsilon, Z_0 A}$
q_0
p_0
$\frac{c, A}{\varepsilon, A}$
q_c
p_c
$\frac{a, Z_0}{d, Z_0}$
$\frac{\varepsilon, A}{\varepsilon, \varepsilon}$
$\frac{a, Z_0}{d, Z_0}$

Here are the simulations of the recognition process of the two sample transductions given in the exercise ($\tau(a\ a\ a\ b\ a\ a) = d\ d\ d$ and $\tau(a\ a\ a\ c\ a\ a) = d\ d$). Remember that a pushdown transducer configuration is denoted as $\left\langle \frac{\text{input tape portion to be read}}{\text{written output tape portion}}, \text{state, stack contents} \right\rangle$:

$$\left\langle \frac{aaabaa}{\varepsilon},\, q_0,\, Z_0 \right\rangle \overset{\frac{a,Z_0}{\varepsilon,Z_0A}}{\Longrightarrow} \left\langle \frac{aabaa}{\varepsilon},\, p_0,\, Z_0A \right\rangle \overset{\frac{a,A}{\varepsilon,AA}}{\Longrightarrow} \left\langle \frac{abaa}{\varepsilon},\, p_0,\, Z_0AA \right\rangle \overset{\frac{a,A}{\varepsilon,AA}}{\Longrightarrow}$$

$$\left\langle \frac{baa}{\varepsilon},\, p_0,\, Z_0AAA \right\rangle \overset{\frac{b,A}{\varepsilon,A}}{\Longrightarrow} \left\langle \frac{aa}{\varepsilon},\, q_b,\, Z_0AAA \right\rangle \overset{\frac{\varepsilon,A}{d,\varepsilon}}{\Longrightarrow} \left\langle \frac{aa}{d},\, q_b,\, Z_0AA \right\rangle \overset{\frac{\varepsilon,A}{d,\varepsilon}}{\Longrightarrow}$$

$$\left\langle \frac{aa}{dd},\, q_b,\, Z_0A \right\rangle \overset{\frac{\varepsilon,A}{d,\varepsilon}}{\Longrightarrow} \left\langle \frac{aa}{ddd},\, q_b,\, Z_0 \right\rangle \overset{\frac{a,Z_0}{\varepsilon,Z_0}}{\Longrightarrow} \left\langle \frac{a}{ddd},\, p_b,\, Z_0 \right\rangle \overset{\frac{a,Z_0}{\varepsilon,Z_0}}{\Longrightarrow}$$

$$\left\langle \frac{\varepsilon}{ddd},\, p_b,\, Z_0 \right\rangle \text{ – recognise and terminate}$$

$$\left\langle \frac{aaacaa}{\varepsilon},\, q_0,\, Z_0 \right\rangle \overset{\frac{a,Z_0}{\varepsilon,Z_0A}}{\Longrightarrow} \left\langle \frac{aacaa}{\varepsilon},\, p_0,\, Z_0A \right\rangle \overset{\frac{a,A}{\varepsilon,AA}}{\Longrightarrow} \left\langle \frac{acaa}{\varepsilon},\, p_0,\, Z_0AA \right\rangle \overset{\frac{a,A}{\varepsilon,AA}}{\Longrightarrow}$$

$$\left\langle \frac{caa}{\varepsilon},\, p_0,\, Z_0AAA \right\rangle \overset{\frac{c,A}{\varepsilon,A}}{\Longrightarrow} \left\langle \frac{aa}{\varepsilon},\, q_c,\, Z_0AAA \right\rangle \overset{\frac{\varepsilon,A}{\varepsilon,\varepsilon}}{\Longrightarrow} \left\langle \frac{aa}{\varepsilon},\, q_c,\, Z_0AA \right\rangle \overset{\frac{\varepsilon,A}{\varepsilon,\varepsilon}}{\Longrightarrow}$$

$$\left\langle \frac{aa}{\varepsilon},\, q_c,\, Z_0A \right\rangle \overset{\frac{\varepsilon,A}{\varepsilon,\varepsilon}}{\Longrightarrow} \left\langle \frac{aa}{\varepsilon},\, q_c,\, Z_0 \right\rangle \overset{\frac{a,Z_0}{d,Z_0}}{\Longrightarrow} \left\langle \frac{a}{d},\, p_c,\, Z_0 \right\rangle \overset{\frac{a,Z_0}{d,Z_0}}{\Longrightarrow}$$

$$\left\langle \frac{\varepsilon}{dd},\, p_c,\, Z_0 \right\rangle \text{ – recognise and terminate}$$

These two simulations and the explanation above justify sufficiently that the designed pushdown automaton satisfies the specification and therefore is correct.

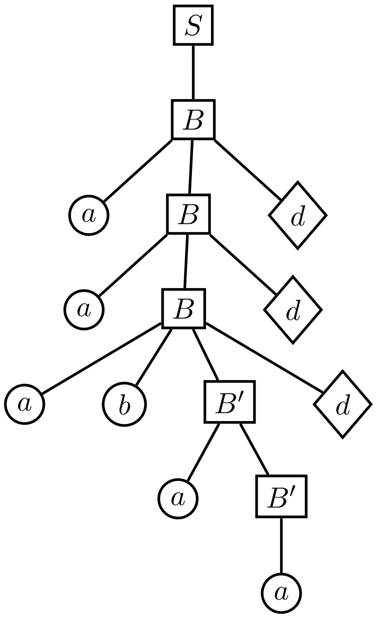
As an alternative, one can compute transduction τ by means of a deterministic syntax transduction scheme G'_τ of type $LR(1)$, in the postfix form where a destination element may occur only on the right end of the rule. Remember that the postfix form is necessary for the syntax transducer to be able to output a letter d only when a reduction move is executed, and not a shift move (as a macrostate may contain two or more different shift candidates). □

Second solution of (4)

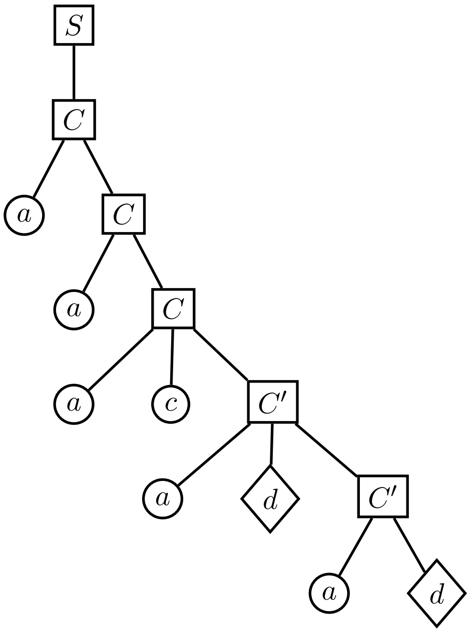
The previous syntax transduction scheme G_τ is not in the postfix form and cannot be used to this purpose as it is, but it is immediate to transform G_τ and obtain a postfix scheme G'_τ by moving the destination elements $\{d\}$ to the right end of the rules. Of course the source part of G'_τ is the same as grammar G_s given before. Here is scheme G'_τ (in the combined form):

$$G'_\tau \left\{ \begin{array}{lcll} S & \to & B \mid C & \\ B & \to & a\, B\ \{d\} \mid a\, b\, B'\ \{d\} & \text{–element } \{d\} \text{ is moved to the right} \\ C & \to & a\, C \mid a\, c\, C' & \\ B' & \to & a\, B' \mid a & \\ C' & \to & a\, C'\ \{d\} \mid a\ \{d\} & \text{–element } \{d\} \text{ is moved to the right} \end{array} \right.$$

Notice that schemes G_τ and G'_τ are almost identical, except in that the destination elements of G'_τ are in the postfix position. Since G'_τ is a slight modification of G_τ, reasonably it performs transduction τ really and therefore is correct. Here are the two syntax trees of the same transduction examples as before, rearranged for G'_τ:



$\tau(a\ a\ a\ b\ a\ a) = d\ d\ d$

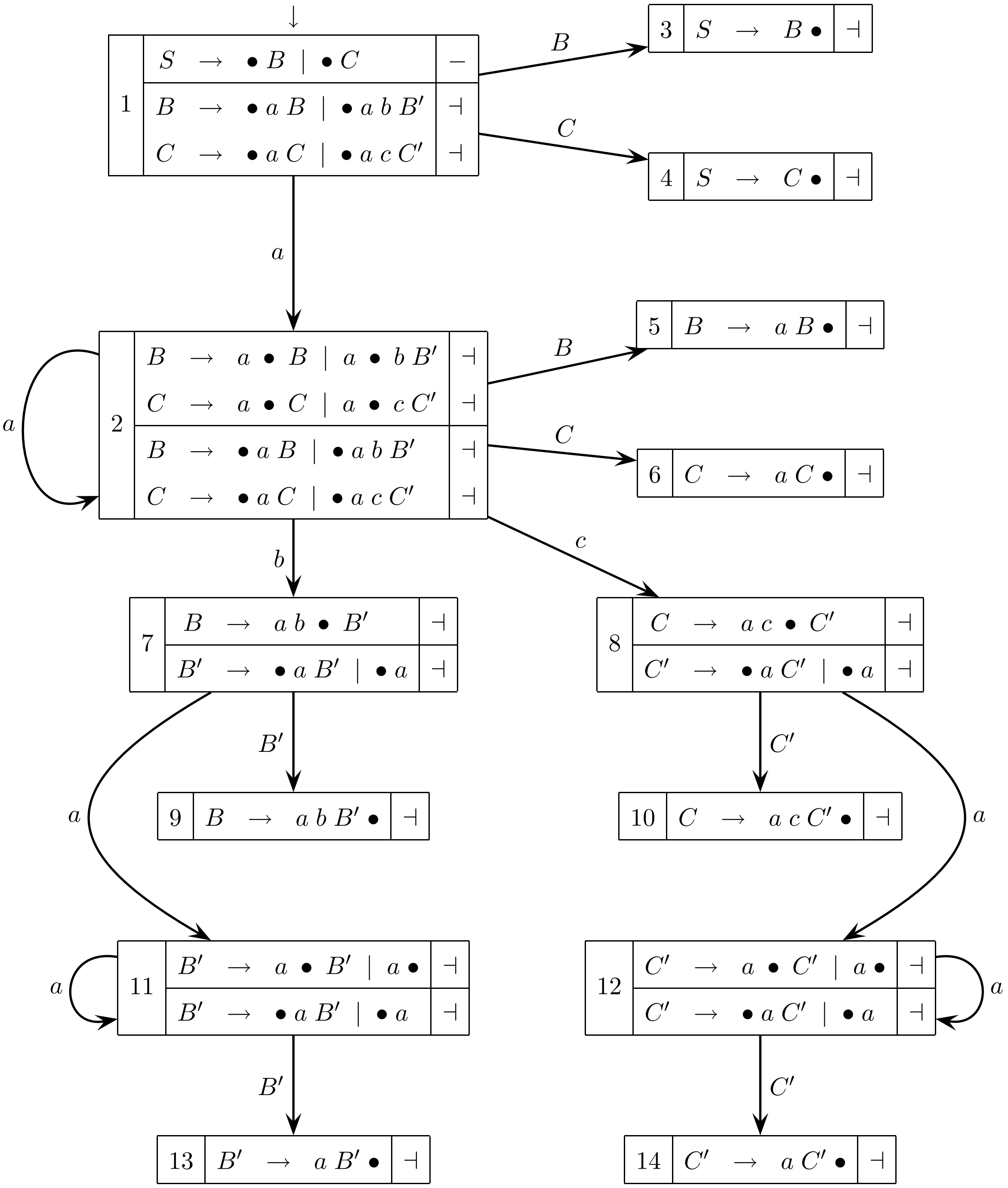


$\tau(a\ a\ a\ c\ a\ a) = d\ d$

To prove that scheme G'_τ is of type $LR(2)$, see the observation below. □

Observation

To prove formally that syntax transduction scheme G'_τ is of type $LR(1)$, here the driver graph of type $LR(1)$ of G'_τ is drawn and the $LR(1)$ condition is checked. Of course to obtain the driver graph of G'_τ one need only consider the source part (which is grammar G_s), hence in the graph the destination elements are omitted for simplicity:



There are not any conflicts of type shift-reduction or reduction-reduction, all the macrostates are adequate and therefore syntax scheme G'_τ is deterministic of type $LR(1)$. Therefore a deterministic syntax transducer exists and could be fully designed.

It seems reasonable to think that the $LR(1)$ deterministic syntax transducer behaves essentially as the above described deterministic pushdown transducer. Here is the simulation of the syntax transducer for the two transduction examples of the exercise ($\tau(a\ a\ a\ b\ a\ a) = d\ d\ d$ and $\tau(a\ a\ a\ c\ a\ a) = d\ d$). Remember that a syntax transducer configuration is specified as:

$$\left\langle \frac{\text{input tape portion to be read}}{\text{output tape portion}}, \text{driver macrostate, stack contents} \right\rangle$$

that the stack alphabet consists of the macrostates of the driver graph, and that the initial and final configurations are the following:

$$\left\langle \frac{\text{input string}}{\varepsilon}, \text{initial macrostate, initial macrostate} \right\rangle$$

$$\left\langle \frac{\text{axiom } S}{\text{output string}}, \text{initial macrostate, initial macrostate} \right\rangle$$

In practice the initial macrostate plays the same role as the initial stack symbol Z_0. The syntax transducer recognises and terminates if the input string is reduced to the axiom S and the stack contains only the initial macrostate, that is if it is empty. And now the two simulations:

$$\left\langle \tfrac{aaabaa}{\varepsilon}, 1, 1 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{aabaa}{\varepsilon}, 2, 1\ 2 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{abaa}{\varepsilon}, 2, 1\ 2\ 2 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{baa}{\varepsilon}, 2, 1\ 2\ 2\ 2 \right\rangle \xRightarrow{\text{shift } b}$$

$$\left\langle \tfrac{aa}{\varepsilon}, 7, 1\ 2\ 2\ 2\ 7 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{a}{\varepsilon}, 11, 1\ 2\ 2\ 2\ 7\ 11 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{\varepsilon}{\varepsilon}, 11, 1\ 2\ 2\ 2\ 7\ 11\ 11 \right\rangle \xRightarrow{\substack{\text{reduce} \\ B' \to a}}$$

$$\left\langle \frac{B'}{\varepsilon}, 11, 1\ 1\ 2\ 2\ 7\ 11 \right\rangle \xRightarrow{\text{shift } B'} \left\langle \tfrac{\varepsilon}{\varepsilon}, 13, 1\ 2\ 2\ 2\ 7\ 11\ 13 \right\rangle \xRightarrow{\substack{\text{reduce} \\ B' \to aB'}} \left\langle \frac{B'}{\varepsilon}, 7, 1\ 2\ 2\ 2\ 7 \right\rangle \xRightarrow{\text{shift } B'}$$

$$\left\langle \tfrac{\varepsilon}{\varepsilon}, 9, 1\ 2\ 2\ 2\ 7\ 9 \right\rangle \xRightarrow[\text{output } d]{\substack{\text{reduce} \\ B \to abB'}} \left\langle \tfrac{B}{d}, 2, 1\ 2\ 2 \right\rangle \xRightarrow{\text{shift } B} \left\langle \tfrac{\varepsilon}{d}, 5, 1\ 2\ 2\ 5 \right\rangle \xRightarrow[\text{output } d]{\substack{\text{reduce} \\ B \to aB}}$$

$$\left\langle \tfrac{B}{dd}, 2, 1\ 2 \right\rangle \xRightarrow{\text{shift } B} \left\langle \tfrac{\varepsilon}{dd}, 5, 1\ 2\ 5 \right\rangle \xRightarrow[\text{output } d]{\substack{\text{reduce} \\ B \to aB}} \left\langle \tfrac{B}{ddd}, 1, 1 \right\rangle \xRightarrow{\text{shift } B}$$

$$\left\langle \tfrac{\varepsilon}{ddd}, 3, 1\ 3 \right\rangle \xRightarrow{\substack{\text{reduce} \\ S \to B}} \left\langle \tfrac{S}{ddd}, 1, 1 \right\rangle \text{ – recognise and terminate}$$

$$\left\langle \tfrac{aaacaa}{\varepsilon}, 1, 1 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{aacaa}{\varepsilon}, 2, 1\ 2 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{acaa}{\varepsilon}, 2, 1\ 2\ 2 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{caa}{\varepsilon}, 2, 1\ 2\ 2\ 2 \right\rangle \xRightarrow{\text{shift } c}$$

$$\left\langle \tfrac{aa}{\varepsilon}, 8, 1\ 2\ 2\ 2\ 8 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{a}{\varepsilon}, 12, 1\ 2\ 2\ 2\ 8\ 12 \right\rangle \xRightarrow{\text{shift } a} \left\langle \tfrac{\varepsilon}{\varepsilon}, 12, 1\ 2\ 2\ 2\ 8\ 12\ 12 \right\rangle \xRightarrow[\text{output } d]{\substack{\text{reduce} \\ C' \to a}}$$

$$\left\langle \frac{C'}{d}, 12, 1\ 2\ 2\ 2\ 8\ 12 \right\rangle \xRightarrow{\text{shift } C'} \left\langle \tfrac{\varepsilon}{d}, 14, 1\ 2\ 2\ 2\ 8\ 12\ 14 \right\rangle \xRightarrow[\text{output } d]{\substack{\text{reduce} \\ C' \to aC'}} \left\langle \frac{C'}{dd}, 8, 1\ 2\ 2\ 2\ 8 \right\rangle \xRightarrow{\text{shift } C'}$$

$$\left\langle \tfrac{\varepsilon}{dd}, 10, 1\ 2\ 2\ 2\ 8\ 10 \right\rangle \xRightarrow{\substack{\text{reduce} \\ C \to acC'}} \left\langle \tfrac{C}{dd}, 2, 1\ 2\ 2 \right\rangle \xRightarrow{\text{shift } C} \left\langle \tfrac{\varepsilon}{dd}, 6, 1\ 2\ 2\ 6 \right\rangle \xRightarrow{\substack{\text{reduce} \\ C \to aC}}$$

$$\left\langle \tfrac{C}{dd}, 2, 1\ 2 \right\rangle \xRightarrow{\text{shift } C} \left\langle \tfrac{\varepsilon}{dd}, 6, 1\ 2\ 6 \right\rangle \xRightarrow{\substack{\text{reduce} \\ C \to aC}} \left\langle \tfrac{C}{dd}, 1, 1 \right\rangle \xRightarrow{\text{shift } C}$$

$$\left\langle \tfrac{\varepsilon}{dd}, 4, 1\ 4 \right\rangle \xRightarrow{\substack{\text{reduce} \\ S \to C}} \left\langle \tfrac{S}{dd}, 1, 1 \right\rangle \text{ – recognise and terminate}$$

Both translations succeed as expected. In both cases the syntax transducer uses the stack to count the three letters a that precede letter b or c (notice that one stack symbol 2 is pushed for each letter a and that therefore in total three stack symbols 2 are pushed). But in the former example the transducer uses the stack as a memory storage to count the number of letters d

to be output (notice that one stack symbol 2 is popped for each letter d), while in the latter one it simply flushes the stack without using it to count letters d (stack symbols 2 are thrown away); thus the syntax transducer behaves similarly to the above given pushdown transducer. The reconstructed rightmost derivations of transduction scheme G'_τ, obtained from the two computations above, are the following:

$$S \overset{S \to B}{\Longrightarrow} B \overset{B \to a\,B\,\{d\}}{\Longrightarrow} a\;B\;\{d\} \overset{B \to a\,B\,\{d\}}{\Longrightarrow} a\;a\;B\,\{d\;d\} \overset{B \to a\,b\,B'\,\{d\}}{\Longrightarrow} a\;a\;a\;b\;B'\;\{d\;d\;d\}$$
$$\overset{B' \to a\,B'}{\Longrightarrow} a\;a\;a\;b\;a\;B'\;\{d\;d\;d\} \overset{B' \to a}{\Longrightarrow} a\;a\;a\;b\;a\;a\;\{d\;d\;d\}$$

$$S \overset{S \to C}{\Longrightarrow} C \overset{C \to a\,C}{\Longrightarrow} a\;C \overset{C \to a\,C}{\Longrightarrow} a\;a\;C \overset{C \to a\,c\,C'}{\Longrightarrow} a\;a\;a\;c\;C' \overset{C' \to a\,C'\,\{d\}}{\Longrightarrow} a\;a\;a\;c\;a\;C'\;\{d\}$$
$$\overset{C' \to a\,\{d\}}{\Longrightarrow} a\;a\;a\;c\;a\;a\;\{d\;d\}$$

These are obtained by reading backwards the two computations and concatenating the reduction moves (those labeled by the "reduce" keyword), completed with the destination elements.

Observation

On the other side, it seems very difficult (if not at all impossible) to compute transduction τ by means of a deterministic transduction scheme of type $LL(k)$, for any $k \geq 1$. The reason should be the same as that given before against the existence of a deterministic finite state transducer: namely that the piece of information provided by letter b or c in the source string is read too late to be useful for the recursive descent deterministic transducer. □

Exercise 46 A series of multiplications and additions of integers, without parentheses, should be translated from infix to prefix (polish) notation. Multiplication precedes addition and operands are one-digit (decimal) numbers (from 0 to 9). The source language L_s is defined by the following (source) grammar G_s:

$$G_s \begin{cases} E & \to \quad T \text{ `} + \text{'} E \mid T \\ T & \to \quad \text{`0' `} \times \text{'} T \mid \text{`1' `} \times \text{'} T \mid \ldots \mid \text{`9' `} \times \text{'} T \\ T & \to \quad \text{`0'} \mid \text{`1'} \mid \ldots \mid \text{`9'} \end{cases}$$

Here follows a transduction example:

$$3 + 1 \times 5 \times 4 + 2 \quad \xrightarrow{\tau} \quad \mathsf{add}\ 3\ \mathsf{add}\ \mathsf{mul}\ 1\ \mathsf{mul}\ 5\ 4\ 2$$

Answer the following questions:

1. Complete the transduction grammar G drafted before and write the destination grammar G_p that generates transduction τ.

2. Without changing the source language L_s, one wishes one optimised transduction τ as much as possible, in order not to emit dummy operations such as:

- multiplication by 0
- multiplication by 1

Here follow two examples of the optimised transduction τ':

$$3 + 1 \times 5 + 2 \quad \xrightarrow{\tau'} \quad \mathsf{add}\ 3\ \mathsf{add}\ 5\ 2$$

$$3 \times 5 \times 0 + 4 + 0 \quad \xrightarrow{\tau'} \quad \mathsf{add}\ 0\ \mathsf{add}\ 4\ 0$$

Design a new transduction grammar G' that computes the optimised transduction τ' (write both new source and destination components), and draw the new source and destination syntax trees (or both superposed) of the latter example.

3. Discuss briefly how to improve more transduction τ' and grammar G', in order not to emit the addition operation in the case either summand is null.

Solution of (1)

Here is the complete grammar G of transduction τ. It is a classical infix to postfix translation and does not have any difficulty (axiom E):

G_s	G_p
$E \rightarrow T$ ‘ + ’ E	$E \rightarrow$ ‘add’ T E
$E \rightarrow T$	$E \rightarrow T$
$T \rightarrow$ ‘0’ ‘ × ’ T	$T \rightarrow$ ‘mul’ ‘0’ T
$T \rightarrow$ ‘1’ ‘ × ’ T	$T \rightarrow$ ‘mul’ ‘1’ T
...	...
$T \rightarrow$ ‘9’ ‘ × ’ T	$T \rightarrow$ ‘mul’ ‘9’ T
$T \rightarrow$ ‘0’	$T \rightarrow$ ‘0’
$T \rightarrow$ ‘1’	$T \rightarrow$ ‘1’
...	...
$T \rightarrow$ ‘9’	$T \rightarrow$ ‘9’

Solution of (2)

In order to refine transduction τ as requested, it is necessary to modify source grammar G_s (but without changing source language L_s) to discriminate between the various indicated cases. In fact the syntax class (nonterminal) T, which generates a product as a chain of factors, should be spilt into subclasses T_0, T_N and T_1 as follows:

- nonterminal T_0 generates a product containing at least one factor 0 in any position (beginning middle or end):

$$T_0 \stackrel{+}{\Rightarrow} \ldots \times 0 \times \ldots$$

- nonterminal T_N generates a product that does not contain any factor 0, but that is not made only of factors 1

$$T_N \stackrel{+}{\not\Rightarrow} \ldots \times 0 \times \ldots \qquad \text{and} \qquad T_N \stackrel{+}{\not\Rightarrow} 1 \times 1 \times \ldots \times 1$$

- nonterminal T_1 generates a product containing only factors 1

$$T_1 \stackrel{+}{\Rightarrow} 1 \times 1 \times \ldots \times 1$$

And here is the grammar G' that realises the optimised transduction τ'. Such a grammar distinguishes the generation and translation of the various product types, as above classified:

- in the cases of nonterminal T_0 or T_1 grammar G' outputs the result 0 or 1 in advance, respectively, before finishing the chain of factors

- in the case of nonterminal T_N factors are output unaltered, with the exception of suppressing those with value 1 and the related multiplications (remember that T_N must contain at least one factor different from both 0 and 1)

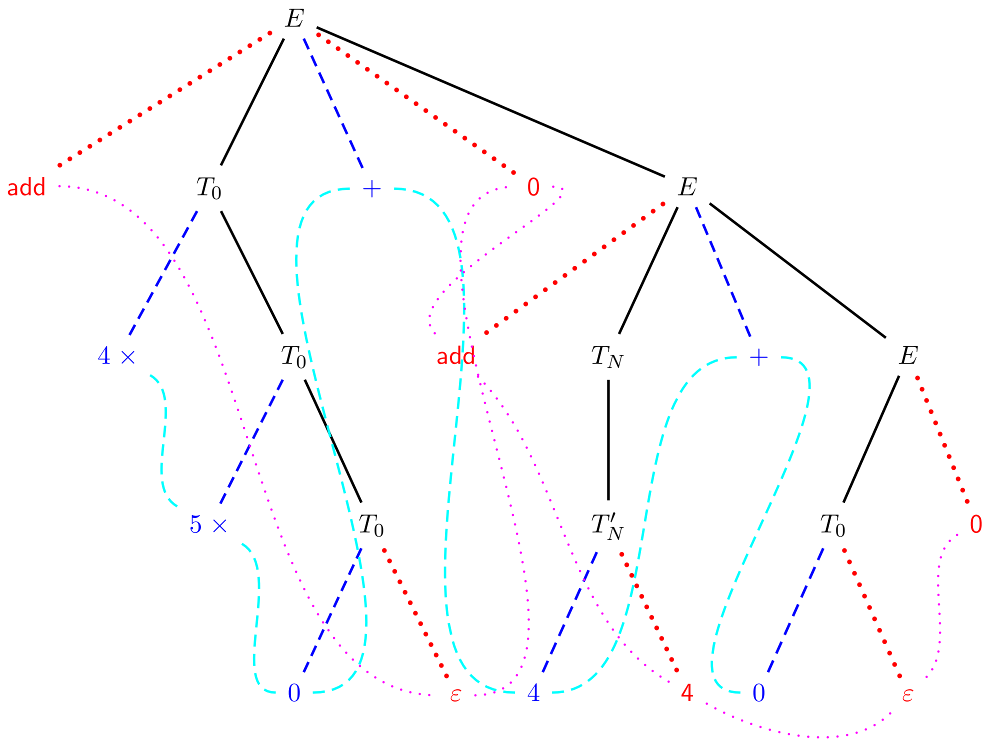
Grammar G' is represented in a somewhat modular form, where the rule subsets generating the complete expression and the three above described product types are separated. It would perhaps be possible to merge partially the modules and minimise the number of rules, at the expense however of having a less readable grammar.

G'_s	G'_p	
$E \rightarrow T_0$ ‘ $+$ ’ E	$E \rightarrow$ ‘add’ T_0 ‘0’ E	
$E \rightarrow T_N$ ‘ $+$ ’ E	$E \rightarrow$ ‘add’ T_N E	
$E \rightarrow T_1$ ‘ $+$ ’ E	$E \rightarrow$ ‘add’ T_1 ‘1’ E	
$E \rightarrow T_0$	$E \rightarrow T_0$ ‘0’	
$E \rightarrow T_N$	$E \rightarrow T_N$	
$E \rightarrow T_1$	$E \rightarrow T_1$ ‘1’	
$T_0 \rightarrow$ ‘x’ ‘ $\times$ ’ T_0	$T_0 \rightarrow T_0$	$x = 1, \ldots, 9$
$T_0 \rightarrow$ ‘0’ ‘ $\times$ ’ T'_0	$T_0 \rightarrow T'_0$	
$T_0 \rightarrow$ ‘0’	$T_0 \rightarrow \varepsilon$	
$T'_0 \rightarrow$ ‘y’ ‘ $\times$ ’ T'_0	$T'_0 \rightarrow T'_0$	$y = 0, \ldots, 9$
$T'_0 \rightarrow$ ‘y’	$T'_0 \rightarrow \varepsilon$	$y = 0, \ldots, 9$
$T_N \rightarrow T_1$ ‘ $\times$ ’ T'_N	$T_N \rightarrow T_1\ T'_N$	
$T_N \rightarrow T'_N$	$T_N \rightarrow T'_N$	
$T'_N \rightarrow$ ‘z’ ‘ $\times$ ’ T_N	$T'_N \rightarrow$ ‘mul’ ‘z’ T_N	$z = 2, \ldots, 9$
$T'_N \rightarrow$ ‘z’ ‘ $\times$ ’ T_1	$T'_N \rightarrow$ ‘z’ T_1	$z = 2, \ldots, 9$
$T'_N \rightarrow$ ‘z’	$T'_N \rightarrow$ ‘z’	$z = 2, \ldots, 9$
$T_1 \rightarrow$ ‘1’ ‘ $\times$ ’ T_1	$T_1 \rightarrow T_1$	
$T_1 \rightarrow$ ‘1’	$T_1 \rightarrow \varepsilon$	

Of course there may be other solutions, which realise transduction τ' with a different syntactic mechanism, and also partial solutions that do not realise the complete optimisation of τ' but only some aspects thereof, for instance when not all factors 1 are removed and other similar simplifications.

Finally here is the syntax tree of the second transduction example τ':

$$3 \times 5 \times 0 + 4 + 0 \quad \xrightarrow{\tau'} \quad \mathsf{add}\ 0\ \mathsf{add}\ 4\ 0$$



The nonterminal skeleton of the syntax tree is coloured in black. By definition the skeleton is common to source and destination. The source and destination parts of the tree are coloured in blue (dashed) and red (dotted), respectively. Similarly, the source and destination fronteers are coloured in cyan and magenta, respectively. This syntax tree justifies sufficiently the correctness of the designed transduction grammar G'.

The reader may exercise by himself in drawing the syntax tree related to the first transduction example τ', which involves also nonterminal T_1.

Solution of (3)

After the structure of grammar G', it is easy to suppress completely the emission of a product the value of which will certainly be 0. Actually it suffices to modify the destination part of the first rule pair of G':

$$E \rightarrow T_0 \text{ `} + \text{'} E \qquad\qquad E \rightarrow \text{`add'}\ T_0 \text{ `0'} E$$

as follows:

$$E \rightarrow T_0 \text{ `} + \text{'} E \qquad\qquad E \rightarrow T_0\ E$$

Here the emission of the summand 0 of operator "add" is suppressed, as such a summand is completely unuseful.

Observation

In one prefers one can also manage the case when all the summands have vale 0, as in such a situation the whole expression has value 0. In fact a translation different from the empty string ε exists in any case, because it would not make any sense to say that the source expression, even if it has value 0, should be translated into ε. This new case can be however dealt with similarly to what has been done for multiplication, by splitting the syntax class (nonterminal) E and creating a subclass (again a nonterminal) E_0 that generates a chain of additions the terms of which are all doomed to take a value 0 (syntax class T_0 can already serve for generating such terms), and by making E_0 emit only one value 0. Clearly there remains a syntax subclass E_1 where at least one term may not be 0, and such a subclass need be dealt with in the appropriate way as well. This refinement of the grammar is relatively simple and the reader is left the task of completing it by himself.

Exercise 47 The source phrases are strings, not empty, over alphabet $\{a, c\}$, where letters a and c correspond to open and closed parentheses, respectively. Such phrases may be well or ill parenthesised. In the latter case (ill) there may be one or more letters c in excess (like for instance in the phrase $ac\,\overset{\downarrow}{c}\,ac$ where the little vertical dart points to the exceeding c), while having letters a in excess is not permitted (for instance a string like $a\,a\,c$ is excluded). Caution: the vertical dart pointing to the exceeding c is not part of the language; here it is included only as a visual facility for the reader.

One wishes one had a purely syntactic translator (that is without attributes) that can correct a source phrase and translate it into a well parenthesised string over alphabet $\{begin,\ end\}$. Correction is carried out by inserting in the appropriate positions more letters a, to match the letters c in excess (and then of course both are transilterated to *begin* and *end*). Here follow a few examples, which demonstrate how correction works (letters b and e are short forms for *begin* and *end*, respectively):

source string	translated and possibly corrected string (and explanation)	
$a\,ac\,ac\,c$	$b\,be\,be\,e$	the source string is already correct and therefore is simply transliterated to the destination alphabet
$\overset{\downarrow}{c}\,ac$	$b\overset{\downarrow}{e}\,be$	the source string is wrong and the missing letter a is inserted by the translator encoded as one b
$a\,ac\,c\,\overset{\downarrow}{c}\,\overset{\downarrow}{c}\,ac$	$b\,be\,e\,b\overset{\downarrow}{e}\,b\overset{\downarrow}{e}\,be$	the source string is wrong and the two missing letters a are inserted by the translator encoded as two b's

Answer the following questions:

1. Design the state-transition graph of a pushdown IO-automaton T (an automaton with one input and one output tape), preferably deterministic, that computes the above described transduction; freely suppose that the source string has a terminator '$\dashv$'. Explain informally and briefly how automaton T works.

2. Write a transduction grammar or syntax scheme G that computes the above described transduction.

Solution of (1)

The requested pushdown IO-automaton T is a variant of the recogniser automaton of the Dyck language and behaves as follows:

- reads in the input tape the open parenthesis a (independently of the contents of the pushdown stack), pushes it encoded with the stack symbol A and writes letter b onto the output tape
- if the input tape contains the closed parenthesis c and
  - the pushdown stack is not empty, writes immediately letter e onto the output tape and pops one symbol A
  - the pushdown stack is empty, instead of entering an error state as it would happen in the recognition of the standard Dyck language, writes the missing open parenthesis onto the output tape, immediately followed by the closed corresponding one, and then proceeds normally (without operating the pushdown stack)
- if the input tape is at the end (because the terminator '$\dashv$' is found) and
  - the pushdown stack is empty, accepts the input string, validates the emitted corresponding translation (which is the corrected version of the input string) and terminates
  - the pushdown stack is not empty, enters an error state because there are too many letters a (which is excluded by hypothesis)

The only memory symbol is A, and if recognition is by empty stack then the pushdown IO-automaton T is deterministic and one-state. Under such a hypothesis, here is the list of the transitions of automaton T (q is the only state of T and Z_0 is the initial stack element):

$$\begin{array}{rrcl}
1: & \delta\,(q, a, Z_0) & = & (q, b, Z_0 A) \\
2: & \delta\,(q, a, A) & = & (q, b, AA) \\
3: & \delta\,(q, c, A) & = & (q, e, \varepsilon) \\
4: & \delta\,(q, c, Z_0) & = & (q, b\,e, Z_0) \\
5: & \delta\,(q, \dashv, Z_0) & = & \text{do not output anything, } \underline{\text{accept}} \text{ and terminate} \\
6: & \delta\,(q, \dashv, A) & = & \text{do not output anything, } \underline{\text{reject}} \text{ and terminate (too many letters } a\text{)}
\end{array}$$

Clearly the designed pushdown IO-automaton T is deterministic and, as such, the transduction is a function (one-valued), that is the performed correction is unique. Here is a sample computation, related to the third sample string above $a\,a\,c\,c\,c\,c\,a\,c\ \dashv$:

$$\begin{array}{l}
\left\langle\, \frac{aaccccac\dashv}{\varepsilon},\ q,\ Z_0 \,\right\rangle \overset{1}{\Rightarrow} \left\langle\, \frac{accccac\dashv}{b},\ q,\ Z_0 A \,\right\rangle \overset{2}{\Rightarrow} \left\langle\, \frac{ccccac\dashv}{bb},\ q,\ Z_0 AA \,\right\rangle \overset{3}{\Rightarrow} \left\langle\, \frac{cccac\dashv}{bbe},\ q,\ Z_0 A \,\right\rangle \overset{3}{\Rightarrow} \\
\overset{3}{\Rightarrow} \left\langle\, \frac{ccac\dashv}{bbee},\ q,\ Z_0 \,\right\rangle \overset{4}{\Rightarrow} \left\langle\, \frac{cac\dashv}{bbeebe},\ q,\ Z_0 \,\right\rangle \overset{4}{\Rightarrow} \left\langle\, \frac{ac\dashv}{bbeebebe},\ q,\ Z_0 \,\right\rangle \overset{1}{\Rightarrow} \left\langle\, \frac{c\dashv}{bbeebebeb},\ q,\ Z_0 A \,\right\rangle \overset{3}{\Rightarrow} \\
\overset{3}{\Rightarrow} \left\langle\, \frac{\dashv}{bbeebebebe},\ q,\ Z_0 \,\right\rangle \overset{5}{\Rightarrow} \text{accept and terminate}
\end{array}$$

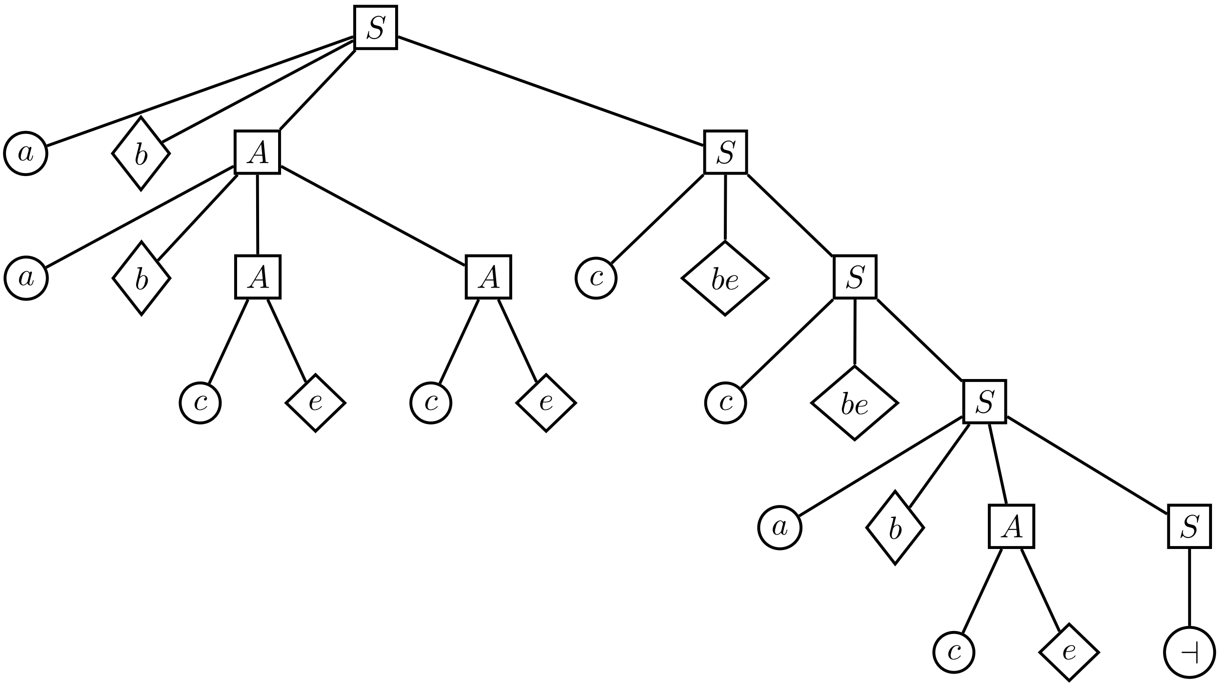
The computation shows that the IO-automaton T performs correctly the correction as required by the specification and outputs string $b\,b\,e\,e\,b\,e\,b\,e\,b\,e$. □

Solution of (2)

As there is a one-state pushdown IO-automaton, essentially it already provides the requested syntax transduction scheme and requires only to be rewritten in the form of a transduction grammar G. Here it is, in the combined form (axiom S corresponds to the memory symbol Z_0):

$$G \begin{cases} S \rightarrow a\,\{b\}\,A\,S & \text{– transliterate } a \text{ (not nested) into } b \\ A \rightarrow a\,\{b\}\,A\,A & \text{– transliterate } a \text{ (nested) into } b \\ A \rightarrow c\,\{e\} & \text{– transliterate } c \text{ (nested or not) into } e \\ S \rightarrow c\,\{b\,e\}\,S & \text{– correct by inserting } b \text{ and transliterate } c \text{ into } e \\ S \rightarrow \dashv & \text{– terminate the derivation} \end{cases}$$

The rules of grammar G are orderly put in correspondence with the moves (transitions) of IO-automaton T. Obviously here the bottom move of T, which performs rejection, does not figure any longer, as simply the source part of G does not generate strings with exceeding letters a. Here is a sample syntax tree, related to the third sample string $a\,a\,c\,c\,c\,c\,a\,c\ \dashv$:



Once again the correction is performed as required. Grammar G is deterministic (of type $LL(1)$), as it can be easily verified and is however somewhat obvious after the way G has been obtained. Therefore one proves again that the transduction is a function (one-valued). □

Observation

Here is another possible solution (in the separate form), derived from the standard (not ambiguous) grammar of the Dyck language (that is from rules $S \rightarrow a\,S\,c\,S \mid \varepsilon$):

G_{source}	$G_{dest.}d$
$S \rightarrow a\ S\ c\ S$	$S \rightarrow b\ S\ e\ S$
$S \rightarrow \varepsilon\ S\ c\ S$	$S \rightarrow b\ S\ e\ S$
$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

This new solution is extremely simple and fully answers the question. Notice that the solution is highly indeterministic and therefore is not of type $LL(k)$, for any $k \geq 1$ (determinism is not mandatory for the exercise anyway). In fact, now the transduction is not one-valued: for instance the wrong source string $a\,c\,c$ is translated and corrected both as $b\,e\,b\,e$ and as $b\,b\,e\,e$, depending on whether one imagines the exceeding letter c to be $a\,c\ \overset{\downarrow}{c}$ rather than $a\ \overset{\downarrow}{c}\ c$, or equivalently whether the corresponding correct source string should be $a\,c\,a\,c$ rather than $a\,a\,c\,c$, respectively. Actually it is not hard to convince oneself that this second solution generates for every wrong source string all the possible translated and corrected destination strings.

Incidentally, such an observation is useful to point out that by transforming the very simple second solution into a one-state pushdown automaton (by applying the standard transformation from a grammar to an indeterministic one-state pushdown automaton as it is described in the textbook of the course), one quickly obtains the answer to the previous question, that is a pushdown IO-automaton that computes the transduction, although this time such an automaton will be indeterministic (however the exercise mentions determinism only as an optional though preferential feature).

Observation

Just to provide some more opportunity to make exercise, here is a third deterministic solution (in the separate form), also obtained as a restriction of the standard (not ambiguous) grammar of the Dyck language (that is rules $S \rightarrow a\,S\,c\,S \mid \varepsilon$):

G_{source}	$G_{dest.}$
$S \rightarrow a\ T\ c\ S \mid \varepsilon \mid c\ C$	$S \rightarrow b\ T\ e\ S \mid \varepsilon \mid b\ e\ C$
$T \rightarrow a\ T\ c\ T \mid \varepsilon$	$T \rightarrow b\ T\ e\ T \mid \varepsilon$
$C \rightarrow c\ C \mid \varepsilon$	$C \rightarrow b\ e\ C \mid \varepsilon$

The restriction here consists of the idea of modifying the Dyck grammar so as to make it generate the additional missing open parenthesis b if the source string contained a closed parenthesis c that were neither nested nor immediately preceded by an open parenthesis a, because only in this case one could deterministically consider letter c as if it were in excess. Such a behaviour can be obtained by differentiating the generation of nested closed parentheses from that of the parentheses concatenated without nesting; the latter parentheses are generated by nonterminal C and the corresponding rule of grammar $G_{dest.}$ performs correction by inserting letter b. The blue alternative are equivalent to the usual Dyck rules, while the red ones perform correction. The thus obtained solution is more verbose than the previous one, but is deterministic of type $LL(1)$, as it can be easily verified. The transduction scheme is certainly a function (one-valued), that is the performed correction is only one of many possible others (as mentioned before).

Exercise 48 A text in natural language is given, consisting of words separated by comma "," or blank $\langle\mathsf{blank}\rangle$, or even by both. The word is roughly modeled as a string of identical terminal characters c. The text may contain parentheses "(" and ")" including a subtext, and parentheses may not be nested. The following (source) grammar G_s defines such a text format (with axiom S):

$$G_s \begin{cases} S \;\rightarrow\; (\; W \mid P \;)\; (\; \text{`,'} \mid \text{`}\langle\mathsf{blank}\rangle\text{'} \mid \text{`},\langle\mathsf{blank}\rangle\text{'}\;)\; S \\ S \;\rightarrow\; (\; W \mid P \;) \\ W \;\rightarrow\; c^{+} \\ P \;\rightarrow\; \text{`('}\; W\; (\; (\; \text{`,'} \mid \text{`}\langle\mathsf{blank}\rangle\text{'} \mid \text{`},\langle\mathsf{blank}\rangle\text{'}\;)\; W\;)^{*}\; \text{`)'} \end{cases}$$

Answer the following questions:

1. Design a syntax transduction scheme G_t (a transduction grammar), by modifying suitably the source grammar G_s, that generates a translated text identical to the source one, with the exception that the words included in the parentheses are separated exactly by one blank, independently of whatever way they are separated in the source text. Here is an example:

   source string:

   ccc $\langle\mathsf{blank}\rangle$ $(cc,$ $cc,$ $\langle\mathsf{blank}\rangle$ $cccc$ $\langle\mathsf{blank}\rangle$ $cc),$ $cc,$ $\langle\mathsf{blank}\rangle$ ccc

   translation:

   ccc $\langle\mathsf{blank}\rangle$ $(cc$ $\langle\mathsf{blank}\rangle$ cc $\langle\mathsf{blank}\rangle$ $cccc$ $\langle\mathsf{blank}\rangle$ $cc),$ $cc,$ $\langle\mathsf{blank}\rangle$ ccc

2. Design a transducer automaton T that computes the above described transduction (spend a short time and think about what transducer model should be used).

3. Show a computation of the transducer automaton T.

Solution of (1)

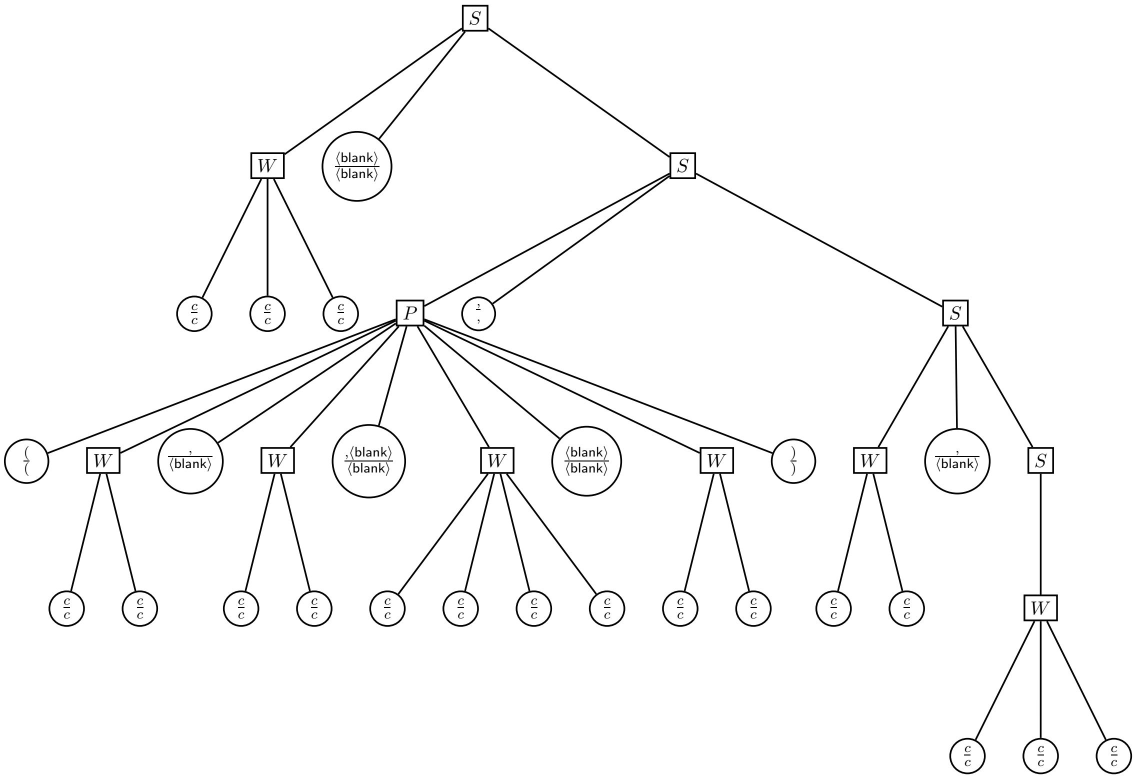
The transduction grammar G_t is the following:

$$G_t \begin{cases} S \;\rightarrow\; (\; W \mid P \;) \left(\frac{\text{`,'}}{\text{`,'}} \mid \frac{\text{`}\langle\mathsf{blank}\rangle\text{'}}{\text{`}\langle\mathsf{blank}\rangle\text{'}} \mid \frac{\text{`},\langle\mathsf{blank}\rangle\text{'}}{\text{`},\langle\mathsf{blank}\rangle\text{'}} \right) S \\ S \;\rightarrow\; (\; W \mid P \;) \\ W \;\rightarrow\; (\frac{c}{c})^{+} \\ P \;\rightarrow\; \frac{\text{`('}}{\text{`('}}\; W \left(\left(\frac{\text{`,'}}{\text{`}\langle\mathsf{blank}\rangle\text{'}} \mid \frac{\text{`}\langle\mathsf{blank}\rangle\text{'}}{\text{`}\langle\mathsf{blank}\rangle\text{'}} \mid \frac{\text{`},\langle\mathsf{blank}\rangle\text{'}}{\text{`}\langle\mathsf{blank}\rangle\text{'}} \right) W \right)^{*} \frac{\text{`)'}}{\text{`)'}} \end{cases}$$

The usual (but in general much less compact) formulation of grammar G_t is the following:

$$G_t \left\{ \begin{array}{lcl} S & \to & (W \mid P) \; (\text{‘,’} \; \{\text{‘,’}\} \mid \text{‘}\langle\mathsf{blank}\rangle\text{’} \; \{\text{‘}\langle\mathsf{blank}\rangle\text{’}\} \mid \text{‘, }\langle\mathsf{blank}\rangle\text{’} \; \{\text{‘, }\langle\mathsf{blank}\rangle\text{’}\} \,) \; S \\ S & \to & (W \mid P) \\ W & \to & (c \; \{c\} \,)^+ \\ P & \to & \text{‘(’} \; \{\text{‘(’}\} \; W \; (\, (\text{‘,’} \; \{\text{‘}\langle\mathsf{blank}\rangle\text{’}\} \mid \text{‘}\langle\mathsf{blank}\rangle\text{’} \; \{\text{‘}\langle\mathsf{blank}\rangle\text{’}\} \mid \text{‘, }\langle\mathsf{blank}\rangle\text{’} \; \{\text{‘}\langle\mathsf{blank}\rangle\text{’}\} \,) \; W \,)^* \; \text{‘)’} \; \{\text{‘)’}\} \end{array} \right.$$

The structural modifications to the source grammar G_s are somewhat few and none of them is relevant. Here is a sample syntax tree, related to the above sample string ccc ⟨blank⟩ (cc, cc, ⟨blank⟩ $cccc$ ⟨blank⟩ cc), cc, ⟨blank⟩ ccc:



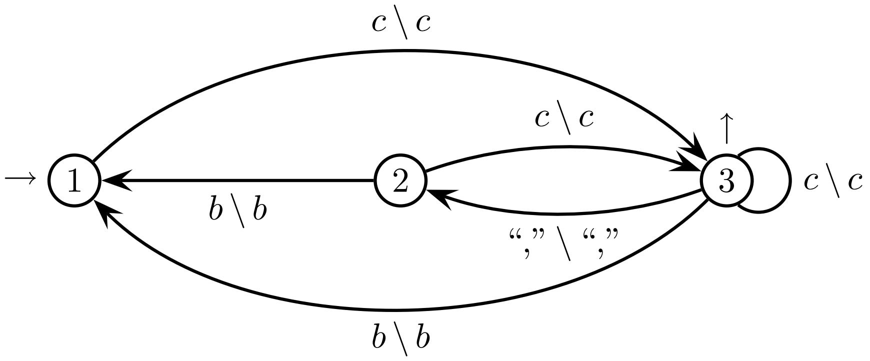
By the way, notice that transduction grammar G_t does not have any recursive rules, but one which is right-linear. Therefore the transduction is rational (regular) and it would be possible to model it by means of a transduction regular expression, instead of a grammar. □

Solution of (2)

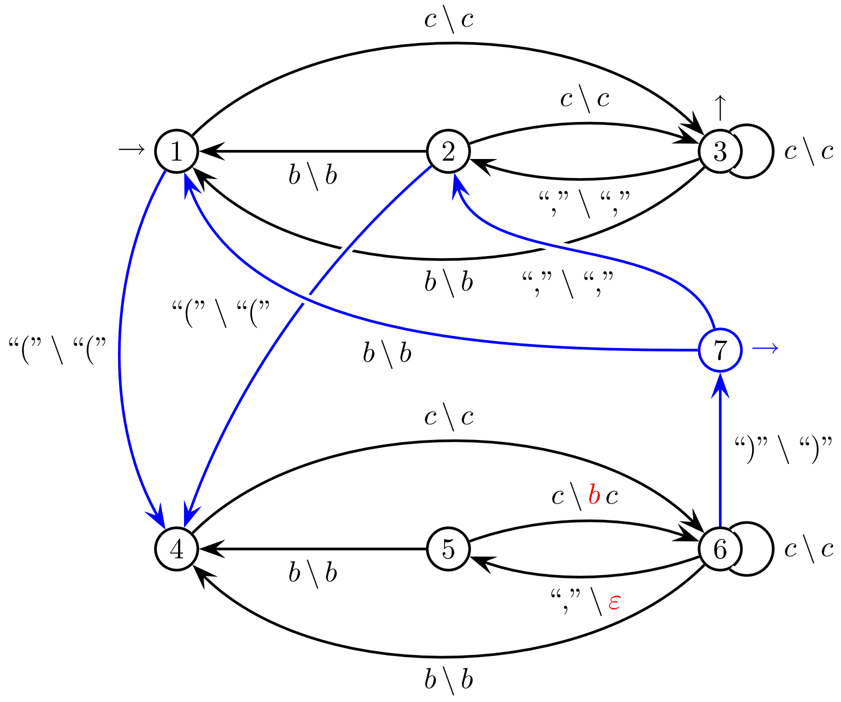
Clearly the above transduction is something very similar to a simple transliteration, to be however done only in the context of parentheses: such a transliteration maps a comma “,” to a blank “⟨blank⟩”, with the exception that the pair “, ⟨blank⟩” is mapped to only one blank “⟨blank⟩”, not into two.

In order to compute such a (pseudo)transliteration a finite state transducer automaton suffices. It has to record the information whether the context is inside of a parentheses pair or outside. If outside, the translated text is identical to the source one, while inside the above described (pseudo)transliteration has to be applied. One can arrange the construction of the transducer

T in two phases. Here is the automaton that translates only the phrases without parentheses (here letter "b" is short for a blank "⟨blank⟩"):



To add parentheses, it suffices in a way to double the automaton, to connect the two halves and to modify the automaton half that computes the transduction inside of parentheses. Here the connections are coloured in blue and the modification to the translation is coloured in red:



In this way the transducer automaton T is complete and does not need any special explanation, but to notice that it is deterministic and requires only six states; fewer states may however suffice, possibly by giving the automaton an indeterministic form. □

Solution of (3)

Here is a sample computation of transducer T, related to the sample string above $ccc\ b\ (cc,\ cc,\ b\ cccc\ b\ cc),\ cc,\ b\ ccc$:

$$\begin{aligned}
&\left\langle \frac{cccb(cc,cc,bccccbcc),cc,bccc}{\varepsilon}, 1 \right\rangle \Rightarrow \left\langle \frac{ccb(cc,cc,bccccbcc),cc,bccc}{c}, 3 \right\rangle \Rightarrow \left\langle \frac{cb(cc,cc,bccccbcc),cc,bccc}{cc}, 3 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{b(cc,cc,bccccbcc),cc,bccc}{ccc}, 3 \right\rangle \Rightarrow \left\langle \frac{(cc,cc,bccccbcc),cc,bccc}{cccb}, 1 \right\rangle \Rightarrow \left\langle \frac{cc,cc,bccccbcc),cc,bccc}{cccb(}, 4 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{c,cc,bccccbcc),cc,bccc}{cccb(c}, 6 \right\rangle \Rightarrow \left\langle \frac{,cc,bccccbcc),cc,bccc}{cccb(cc}, 6 \right\rangle \Rightarrow \left\langle \frac{cc,bccccbcc),cc,bccc}{cccb(cc}, 5 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{c,bccccbcc),cc,bccc}{cccb(ccbc}, 6 \right\rangle \Rightarrow \left\langle \frac{,bccccbcc),cc,bccc}{cccb(ccbcc}, 6 \right\rangle \Rightarrow \left\langle \frac{bccccbcc),cc,bccc}{cccb(ccbcc}, 5 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{ccccbcc),cc,bccc}{cccb(ccbccb}, 4 \right\rangle \Rightarrow \left\langle \frac{cccbcc),cc,bccc}{cccb(ccbccbc}, 6 \right\rangle \Rightarrow \left\langle \frac{ccbcc),cc,bccc}{cccb(ccbccbcc}, 6 \right\rangle \Rightarrow \left\langle \frac{cbcc),cc,bccc}{cccb(ccbccbccc}, 6 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{bcc),cc,bccc}{cccb(ccbccbcccc}, 6 \right\rangle \Rightarrow \left\langle \frac{cc),cc,bccc}{cccb(ccbccbccccb}, 4 \right\rangle \Rightarrow \left\langle \frac{c),cc,bccc}{cccb(ccbccbccccbc}, 6 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{),cc,bccc}{cccb(ccbccbccccbcc}, 6 \right\rangle \Rightarrow \left\langle \frac{,cc,bccc}{cccb(ccbccbccccbcc)}, 7 \right\rangle \Rightarrow \left\langle \frac{cc,bccc}{cccb(ccbccbccccbcc),}, 2 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{c,bccc}{cccb(ccbccbccccbcc),c}, 3 \right\rangle \Rightarrow \left\langle \frac{,bccc}{cccb(ccbccbccccbcc),cc}, 3 \right\rangle \Rightarrow \left\langle \frac{bccc}{cccb(ccbccbccccbcc),cc,}, 2 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{ccc}{cccb(ccbccbccccbcc),cc,b}, 1 \right\rangle \Rightarrow \left\langle \frac{cc}{cccb(ccbccbccccbcc),cc,bc}, 3 \right\rangle \Rightarrow \left\langle \frac{c}{cccb(ccbccbccccbcc),cc,bcc}, 3 \right\rangle \Rightarrow \\
&\Rightarrow \left\langle \frac{\varepsilon}{cccb(ccbccbccccbcc),cc,bccc}, 3 \right\rangle
\end{aligned}$$

□

Observation

Of course a pushdown automaton would work as well (independently of whether deterministic or not), but it would be exceedingly powerful for the simple proposed problem.

Exercise 49 A source language consists of arithmetic expressions with the usual infix addition operator and parentheses. Moreover, on the left of each parenthesised subexpression there is a marker field that specifies whether the translation of the subexpression is prefix or postfix (see the example below). The source language is generated by the following (source) grammar:

$$G_s \left\{ \begin{array}{lcl} S & \rightarrow & (\text{`prefix'} \mid \text{`postfix'})\ \text{`('}\ E\ \text{`)'} \\ E & \rightarrow & T\ (\text{`+'}\ T)^* \\ T & \rightarrow & a \mid (\text{`prefix'} \mid \text{`postfix'})\ \text{`('}\ E\ \text{`)'} \end{array} \right.$$

A sample translation is the following. For better clarity here variable names are differentiated and operators are numbered, but in the source string all the variables are called with the same name a and the operators do not have any numbering:

$$\begin{array}{ll} \text{source:} & \text{postfix}\ (\ a\ +_1\ b\ +_2\ \text{prefix}\ (\ c\ +_3\ \text{prefix}\ (\ d\ )\ )\ ) \\ \text{destination:} & a\ b\ +_1\ +_3\ c\ d\ +_2 \end{array}$$

Notice that the marker field is not preserved in the destination string.

Answer the following questions:

1. Modify if necessary the source grammar G_s shown before and design a syntax transduction scheme G_t (or a transduction grammar), without semantic attributes, that translates the source language and outputs locally the prefix or postfix form of each subexpression, in agreement with the prescription given by the leading marker field.

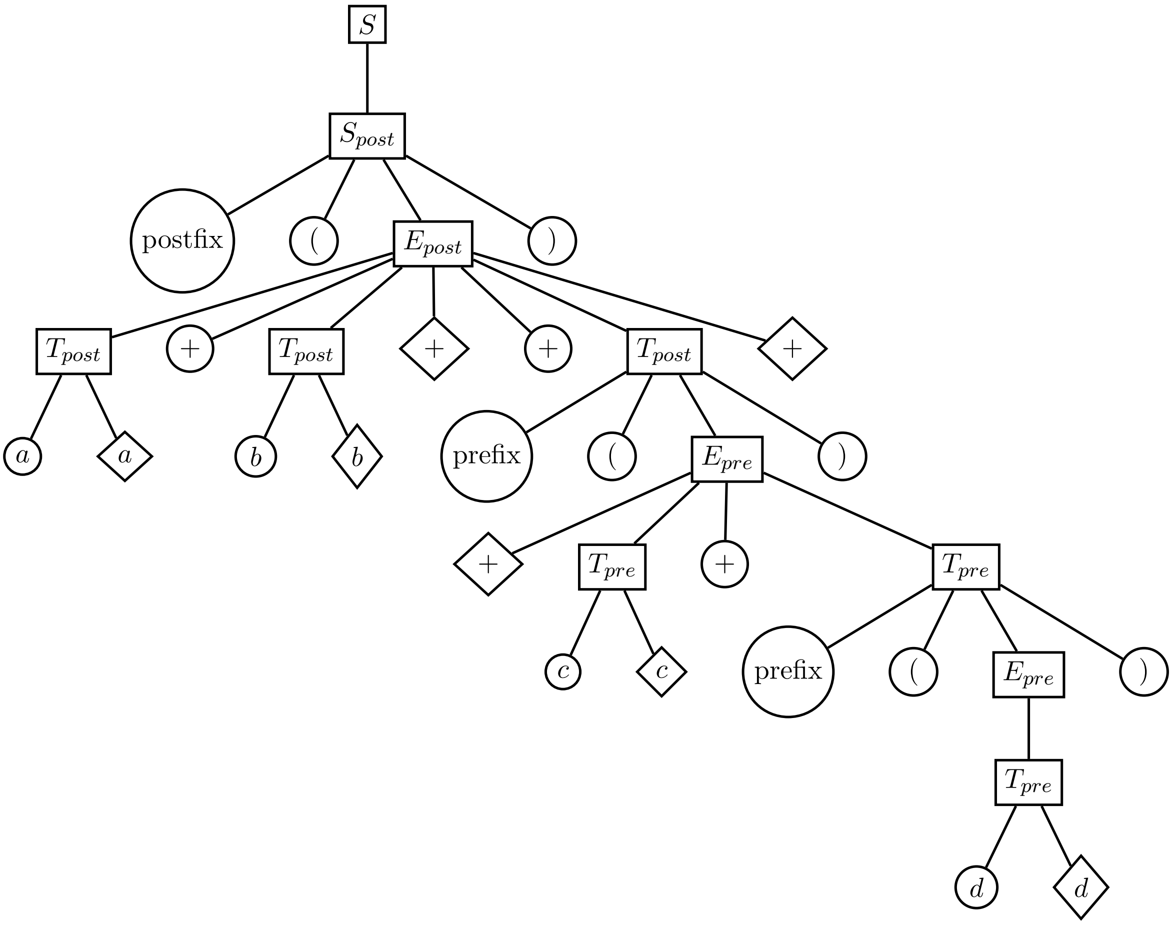
2. Discuss whether the designed transduction scheme G_t (or grammar) is deterministic.

Solution of (1)

One way for solving the problem is that of splitting source grammar G_s into two different transduction schemes G_{pre} and G_{post}, prefix or postfix depending on the marker field, and then of connecting the two schemes when a transition from prefix to postfix takes place, or viceversa. Here is the split scheme, presented in a modular way (axiom S):

$$
G_t \left\{
\begin{array}{l}
\qquad\qquad S \;\rightarrow\; S_{pre} \mid S_{post} \\[4pt]
G_{pre} \left\{
\begin{array}{lcl}
S_{pre} & \rightarrow & \text{‘prefix’ ‘(’ } E_{pre} \text{ ‘)’} \\
E_{pre} & \rightarrow & \left(\{\text{‘+’}\}\; T_{pre}\; \text{‘+’} \right)^{*}\; T_{pre} \\
T_{pre} & \rightarrow & a\; \{a\} \mid \text{‘prefix’ ‘(’ } E_{pre} \text{ ‘)’} \mid \text{‘postfix’ ‘(’ } E_{post} \text{ ‘)’}
\end{array}
\right. \\[20pt]
G_{post} \left\{
\begin{array}{lcl}
S_{post} & \rightarrow & \text{‘postfix’ ‘(’ } E_{post} \text{ ‘)’} \\
E_{post} & \rightarrow & T_{post}\; \left(\text{‘+’}\; T_{post}\; \{\text{‘+’}\} \right)^{*} \\
T_{post} & \rightarrow & a\; \{a\} \mid \text{‘prefix’ ‘(’ } E_{pre} \text{ ‘)’} \mid \text{‘postfix’ ‘(’ } E_{post} \text{ ‘)’}
\end{array}
\right.
\end{array}
\right.
$$

The whole scheme is in the combined form and the graph parentheses “{” and “}” enclose the terminal symbols to output. If one wishes one can separate the source and destination parts of the scheme. Here is a sample syntax tree, related to the above given sample string:



Notice that the rule expanding nonterminal E_{pre} changes the associativity of the expression (as for instance expression $a + b + c$ is translated into expression $+ a + b c$); this however is irrelevant for the exercise, which does not specify anything about such an aspect. □

Solution of (2)

If one examines the source part of the scheme G_{pre}, one sees easily that it is not of type $LL(k)$, for any $k \geq 1$, as the extended rule that expands nonterminal E_{pre} does not allow to realise when the star operator should iterate or finish. As it is, scheme G_{tr} is not deterministic.

Different formulations of the scheme might however happen to be deterministic; the reader may wish to investigate this issue by himself. □

Exercise 50 The following language L is given, over alphabet $\{a, b, c, d\}$ (L is the union of two components 1 and 2):

$$L = \underbrace{\{\, a^n\, b^n\, c \mid \quad n \geq 0 \,\}}_{\text{component 1}} \cup \underbrace{\{\, a^n\, b^n\, d \mid \quad n \geq 1 \,\}}_{\text{component 2}}$$

Consider the following transduction τ (which behaves differently depending on the language component):

$$\underbrace{\tau\,(\,a^n\,b^n\,c\,) = a^{2n}b^n}_{\text{behaviour 1}} \quad n \geq 0 \qquad\qquad \underbrace{\tau\,(\,a^n\,b^n\,d\,) = (a\,b)^n}_{\text{behaviour 2}} \quad n \geq 1$$

Answer the following questions:

1. Design a pushdown transducer T (possibly indeterministic), recognising by final state, that computes transduction τ.
2. Show one or two computations of transducer T.
3. Design a transduction scheme (or grammar) G that realises transduction τ.

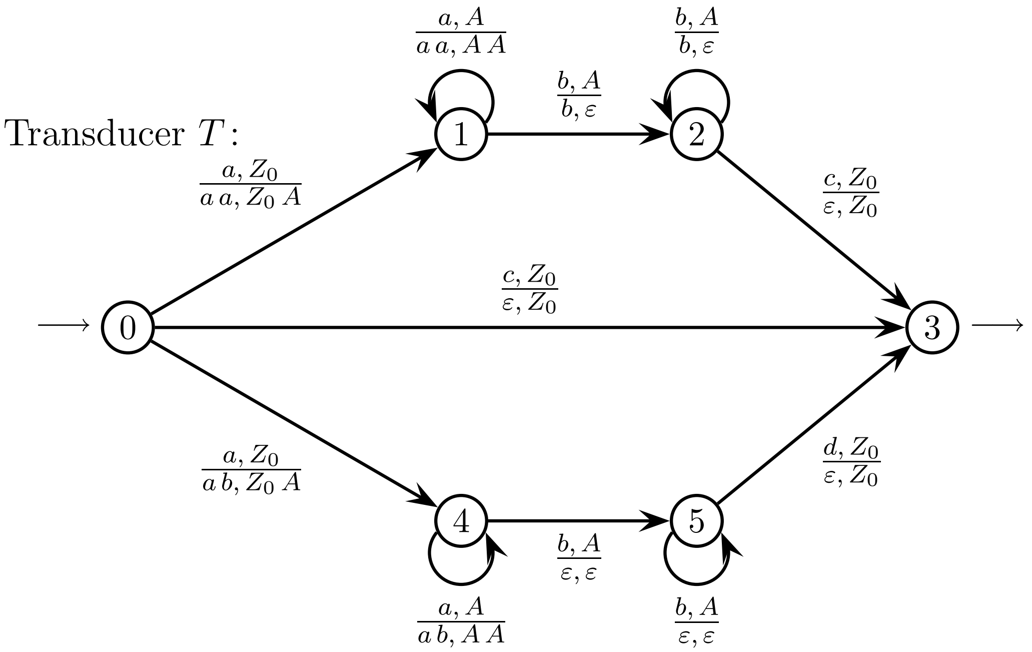
Solution of (1)

Although recognising the source language could be done by means of a deterministic pushdown automaton (recognising by final state), transduction τ is inherently indeterministic and has to be computed by means of an indeterministic pushdown transducer automaton (again recognising by final state). The pushdown transducer T reads and pushes the input letters a (by encoding them in an appropriate way) and acts differently depending on the two possible transduction behaviours (1 or 2):

Behaviour 1 transducer T outputs the corresponding letters a and doubles them; then it pops the letters a to match the input letters b and to output the corresponding letters b

Behaviour 2 transducer T outputs the corresponding pairs $a\,b$; then it pops the letters a to match the input letters b (here it does not output anything)

At the beginning one has to choose indeterministicallly which of the two transductive behaviours 1 or 2 to adopt; the initial choice is validated at the end when either the input letter c or d will be read. In the case 1 the transduction can output the empty string, while in the case 2 this may not happen by definition. Here is the state-transition graph of such a transducer T:


Transducer T:
$\frac{a,\,A}{a\,a,\,A\,A}$
$\frac{b,\,A}{b,\,\varepsilon}$
$\frac{b,\,A}{b,\,\varepsilon}$
1
2
$\frac{a,\,Z_0}{a\,a,\,Z_0\,A}$
$\frac{c,\,Z_0}{\varepsilon,\,Z_0}$
$\frac{c,\,Z_0}{\varepsilon,\,Z_0}$
0
3
$\frac{a,\,Z_0}{a\,b,\,Z_0\,A}$
$\frac{d,\,Z_0}{\varepsilon,\,Z_0}$
4
5
$\frac{b,\,A}{\varepsilon,\,\varepsilon}$
$\frac{a,\,A}{a\,b,\,A\,A}$
$\frac{b,\,A}{\varepsilon,\,\varepsilon}$

The indeterministic aspect is concentrated only in the state 0, as the rest of the graph is fully deterministic. Actually it seems impossible to eliminate such a limited form of indeterminism, as the behaviour at the output is different in the two cases and one does not easily see how one could unify them. Recognition is by final state, though here it happens that in the final state 3 the pushdown stack gets completely empty. □

Solution of (2)

Here is the simulation of two representative computations, of the many possible ones:

$$\left\langle \frac{abc}{\varepsilon},\, 0,\, Z_0 \right\rangle \overset{\frac{a,\,Z_0}{a\,a,\,Z_0\,A}}{\Longrightarrow} \left\langle \frac{bc}{aa},\, 1,\, Z_0 A \right\rangle \overset{\frac{b,\,A}{b,\,\varepsilon}}{\Longrightarrow} \left\langle \frac{c}{aab},\, 2,\, Z_0 \right\rangle \overset{\frac{c,\,Z_0}{\varepsilon,\,Z_0}}{\Longrightarrow} \left\langle \frac{\varepsilon}{aab},\, 3,\, Z_0 \right\rangle$$

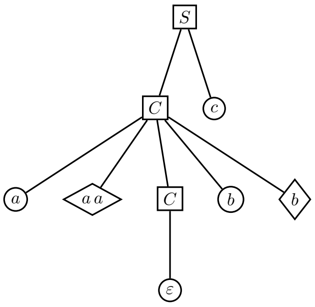
$$\left\langle \frac{abd}{\varepsilon},\, 0,\, Z_0 \right\rangle \overset{\frac{a,\,Z_0}{a\,b,\,Z_0\,A}}{\Longrightarrow} \left\langle \frac{bd}{ab},\, 4,\, Z_0 A \right\rangle \overset{\frac{b,\,A}{\varepsilon,\,\varepsilon}}{\Longrightarrow} \left\langle \frac{d}{ab},\, 5,\, Z_0 \right\rangle \overset{\frac{d,\,Z_0}{\varepsilon,\,Z_0}}{\Longrightarrow} \left\langle \frac{\varepsilon}{ab},\, 3,\, Z_0 \right\rangle$$

Transducer T works properly with computations that call in the self-loops as well. These two simulations justify sufficiently that the designed transducer is correct. □

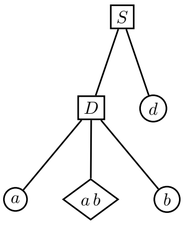
Solution of (3)

The transduction scheme G that realises the transduction relation τ does not have particular difficulties. Here it is, presented in the combined form (axiom S):

$$G\left\{\begin{array}{lcl} S & \to & C\,c \\ S & \to & D\,d \\ C & \to & a\ \{\,a\,a\,\}\ C\,b\ \{\,b\,\} \\ C & \to & \varepsilon \\ D & \to & a\ \{\,a\,b\,\}\,D\,b \\ D & \to & a\ \{\,a\,b\,\}\ b \end{array}\right.$$



$\tau\,(abc) = aab$



$\tau\,(abd) = ab$

Scheme G corresponds in an evident way to the transducer automaton T given before (aside there are shown the syntax trees of the above simulated transduction cases $\tau(abc) = aab$ and $\tau(abd) = ab$). It is easy to realise that such a scheme is not $LL(k)$, for any $k \geq 1$, because of the alternative rules that expand S (the associated lookahead sets of order k share string a^k). As a consequence of what has been said before, it does not seem possible to find a deterministic scheme for transduction τ. □

5.2 Attribute Grammar

Exercise 51 Consider a table containing a list of positive integer numbers, all different from one another. Associated with each element of the list there is a set (not empty) of positive integer numbers, the sum of which equals the former element. Thus the table is a relational data base with two attributes: the former is a search key NUM and the latter is a set ADD of numbers.

It is requested to design an attribute grammar G that, for each key-set pair of the relation, checks whether the sum of the numbers contained in the set ADD (described as a list) equals the associated search key NUM and that, for those pairs that pass the check (and only for them), translates the relation into first normal form, as follows:

source table

NUM	ADD
12	$\{2, 10\}$
6	$\{2, 4\}$
9	$\{2\}$
30	$\{2, 3, 25\}$

$\Longrightarrow$

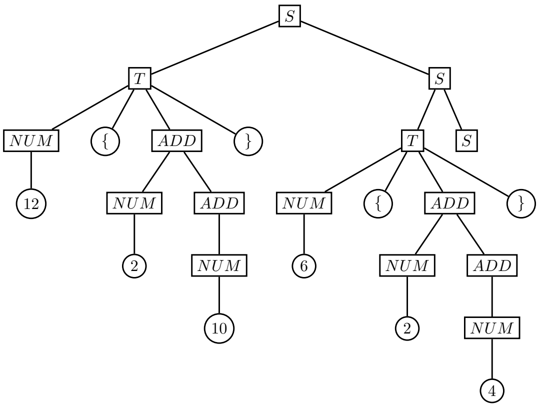
translated table (in 1^{st} normal form)

NUM	$ELEM$
12	2
12	10
6	2
6	4
30	2
30	3
30	25

Notice that the entry with key 9 of the source table does not pass the check and hence does not show up in the normalised table.

The syntactic support G that generates the source table is given below on the left (axiom S). Nonterminal NUM generates an integer, but here it is dealt with as a terminal. The partial syntax tree that generates the two top entries (those with keys 12 and 6) of the source table above is shown on the right (the rest of the tree is omitted).

$$G \left\{ \begin{array}{rcl} S & \rightarrow & T\ S \\ S & \rightarrow & \varepsilon \\ T & \rightarrow & \langle \mathsf{NUM} \rangle \text{ ‘\{’ } \langle \mathsf{ADD} \rangle \text{ ‘\}’} \\ \langle \mathsf{ADD} \rangle & \rightarrow & \langle \mathsf{NUM} \rangle\ \langle \mathsf{ADD} \rangle \\ \langle \mathsf{ADD} \rangle & \rightarrow & \langle \mathsf{NUM} \rangle \end{array} \right.$$



The following attributes must be included in the grammar:

- the integer value “ν” of search key NUM, which is computed initially and made available (do not take care of how to compute ν)
- a correctness predicate “φ”, true if and only if an entry passes the check
- a string “τ”, associated with the root S of the syntax tree, which contains the table translated into first normal form and encoded by a string denoted as shown below:

  12, 2 / 12, 10 / 6, 2 / 6, 4 / 30, 2 / 30, 3 / 30, 25 /

It is permitted to extend the existing attributes give before or to define new attributes, depending on necessity and convenience.

Answer the following questions:

1. List all the grammar attributes along with their type and meaning.
2. Write the semantic functions that compute the attributes.
3. For each grammar rule draw the dependence graphs of the attributes.
4. Determine whether attribute grammar G is of type one-sweep.
5. Determine whether attribute grammar G is of type L.

Solution of (1)

Conceptually the transduction process described above can be organised in two phases:

Control Phase Check whether each entry of the source table is correct or not, and mark the correct entries by attribute φ (with $\varphi =$ true if and only if entry is correct).

Write Phase Insert into attribute τ (normalised table) only the correct entries (those previously labeled by $\varphi =$ true), according to the prescribed string format.

Attribute φ is of type right, so that it is possible to propagate it to all the addends to be included in the translated table, and is associated with nonterminal *ADD* (the list of addends of each entry of the source table). Attribute τ has to be extended to nonterminals T and *ADD*. Moreover, define the new left attribute σ to compute the sum of the addends listed in each entry of the source table. Define also the new right attribute χ, to propagate the value of the search key and associate it with each addend in the translated table.

attributes to be used in the grammar

type	name	(non)terminal	domain	meaning and interpretation
		attributes already given in the exercise		
left	ν	*NUM*	integer	value of search key *NUM*
	φ		boolean	correct entry
left	τ	S	string	table in 1^{st} normal form
		attributes to be extended or added		
right	φ	*ADD*	boolean	correct entry (ext.)
left	τ	S, T, *ADD*	string	table in 1^{st} normal form (ext.)
left	σ	*ADD*	integer	sum of addends (new)
right	χ	*ADD*	integer	value of search key *NUM* (new)

Existing attribute ν is left unchanged, existing attributes φ and τ are extended, and attributes σ and χ are defined new. □

Solution of (2)

Here are the semantic functions of attribute grammar G, which map the two phase transduction process outlined before:

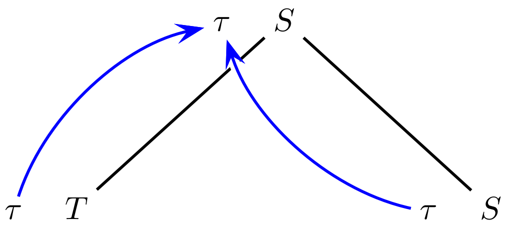
syntax	*semantic function*
$S_0 \to T_1\, S_2$	$\tau_0 = CAT\,(\tau_1, \tau_2)$
$S_0 \to \varepsilon$	$\tau_0 = \varepsilon$
$T_0 \to \langle \mathsf{NUM} \rangle_1$ '{' $\langle \mathsf{ADD} \rangle_2$ '}'	**if** ($\sigma_2 == \nu_1$) **then** $\varphi_2 = \text{true}$ **else** $\varphi_2 = \text{false}$ **endif** $\chi_2 = v_1$ $\tau_0 = \tau_2$
$\langle \mathsf{ADD} \rangle_0 \to \langle \mathsf{NUM} \rangle_1\ \langle \mathsf{ADD} \rangle_2$	$\sigma_0 = \nu_1 + \sigma_2$ $\varphi_2 = \varphi_0$ $\chi_2 = \chi_0$ **if** ($\varphi_0 == \text{true}$) **then** $\tau_0 = CAT\,(\chi_0$, ',', ν_1, '/', $\tau_2)$ **else** $\tau_0 = \varepsilon$ **endif**

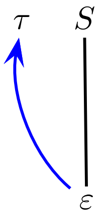
$\langle \mathsf{ADD} \rangle_0 \to \langle \mathsf{NUM} \rangle_1$	$\sigma_0 = \nu_1$
	if ($\varphi_0 == \text{true}$) **then**
	$\tau_0 = CAT\,(\chi_0$, ',', ν_1, '/')
	else
	$\tau_0 = \varepsilon$
	endif

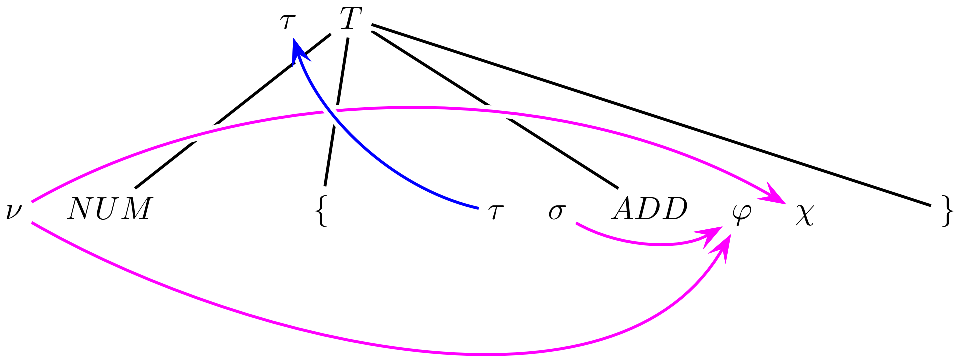
The auxiliary function CAT converts the arguments into strings (unless an argument is already a string) and concatenates them orderly into one string returned as result. □

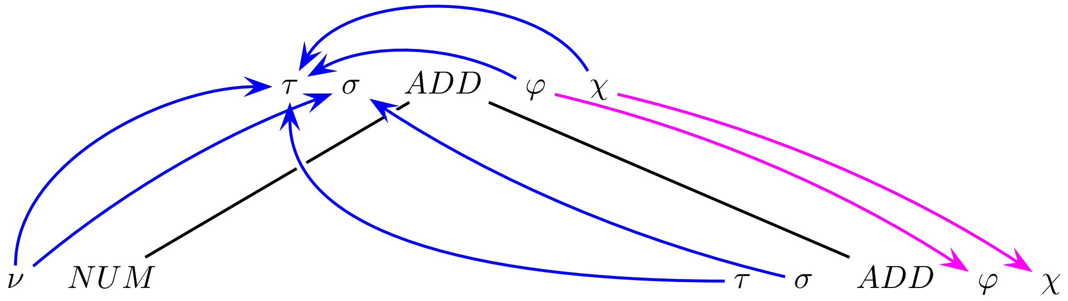
Solution of (3)

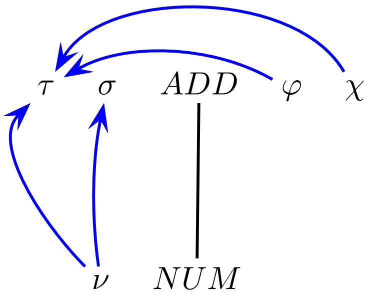
Here are the various dependence graphs of each rule of attribute grammar G:











Blue and magenta arrows indicate the dependences of synthesised and inherited attributes, respectively. A quick look at the dependence graphs should convince the reader of that attribute grammar G is acyclic and that therefore is correctly defined. □

Solution of (4)

The attribute grammar is not of type one-sweep, as to compute the inherited attribute φ and propagate it to the bottom of the syntax tree, it is necessary to have computed before the synthesised attribute σ. It is somewhat evident that to do so more than one sweep of the tree is required. One should see soon that actually two sweeps are needed and suffice: the former to decorate the tree with attribute σ; the latter to compute and propagate downwards attribute φ, and to compute and propagate upwards attribute τ which contains the translated table. □

Solution of (5)

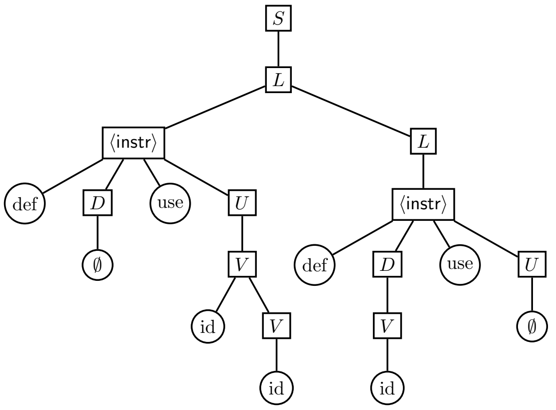
As the attribute grammar is not of type one-sweep, by definition it is not of type L either. □

Exercise 52 Consider a program consisting of a list of statements. Each statement is modeled by means of two sets: the former set contains the names of the variables the instruction defines, the latter one those the instruction uses. For instance, the assignment statement "$c = c + a$" is modeled by the two sets "def c" and "use a, c", while the I/O statement "$read(a, c)$" is modeled by "def a, c" and "use $\emptyset$".

The syntactic support G is the following (axiom S); on the right there is a sample syntax tree:

$$G \left\{ \begin{array}{rcl} S & \to & L \\ L & \to & \langle \mathsf{instr} \rangle \ L \\ L & \to & \langle \mathsf{instr} \rangle \\ \langle \mathsf{instr} \rangle & \to & \text{`def'} \ D \ \text{`use'} \ U \ \text{`;'} \\ D & \to & \text{`}\emptyset\text{'} \\ D & \to & V \\ U & \to & \text{`}\emptyset\text{'} \\ U & \to & V \\ V & \to & \text{`id'} \ \text{`,'} \ V \\ V & \to & \text{`id'} \end{array} \right.$$

sample syntax tree (do not decorate)



Symbol "id" is a variable identifier (name) and here it is modeled syntactically as a pure terminal. The identifier is an available lexical attribute (one need not compute it), called n.

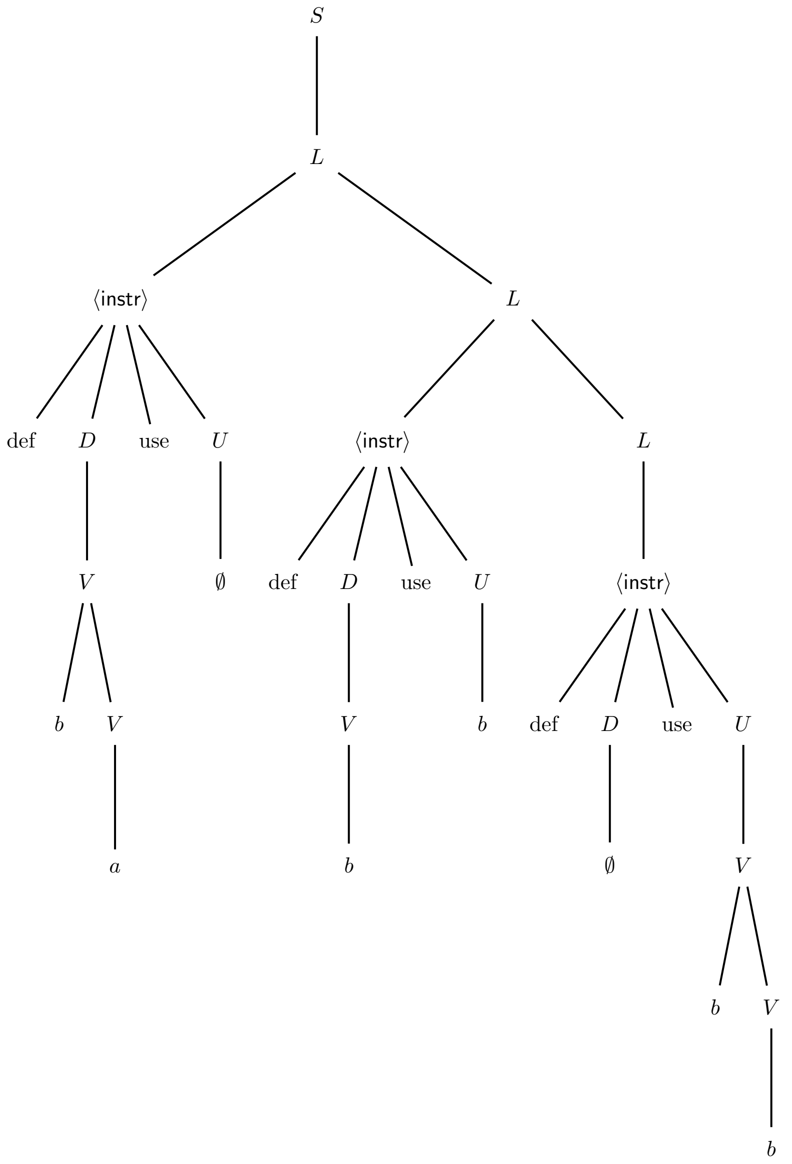
Answer the following questions:

1. Define the appropriate attributes and write the semantic functions of an attribute grammar G, to compute the set of the live variables as an attribute associated with the initial instruction of the program, that is with axiom S.

2. Decorate the syntax tree shown in the next page and write next to each node the values of the associated attributes, according to the semantic functions of attribute grammar G.

3. Draw the dependence graphs of the attributes of the rules of grammar G (at least the most important ones).

4. Write, at least partially, the pseudocode of the semantic evaluator suited for the identified dependence types.



syntax tree to be decorated

First an extended solution is given, which models the computation of live variables as it is commonly presented in the textbooks. Then a second solution is given, as valid as the extended one, but simpler and faster to conceive and write.

Extended solution

Solution of (1)

Here is the list of attributes of grammar G:

attributes to be used in the grammar

type	name	(non)terminal	domain	meaning and interpretation
attributes already given in the exercise				
right	n	id	string	identifier name
attributes to be extended or added				
left	*var*	V	set of strings	list of variable names (new)
left	*def*	D	idem	var.s defined by the current instruction (new)
left	*use*	U	idem	var.s used by the current instruction (new)
left	*live_in*	⟨instr⟩	idem	live var.s at the instruction input (new)
right	*live_out*	⟨instr⟩	idem	live var.s at the instruction output (new)
left	*live*	S, L	idem	live var.s of the program (new)

The only meaningful (inherited) attribute of type right is *live_out* and represents the set of the live variables at the output of the current instruction, because it is inherited by the live variables at the input of the syntax class L that generates the instruction following the current one. Attribute n is associated with a terminal, "id", and therefore by default it is of type right. Here however such an attribute is precomputed (this is why it is of type right), and has a limited importance. All the other attributes are of type left (synthesised).

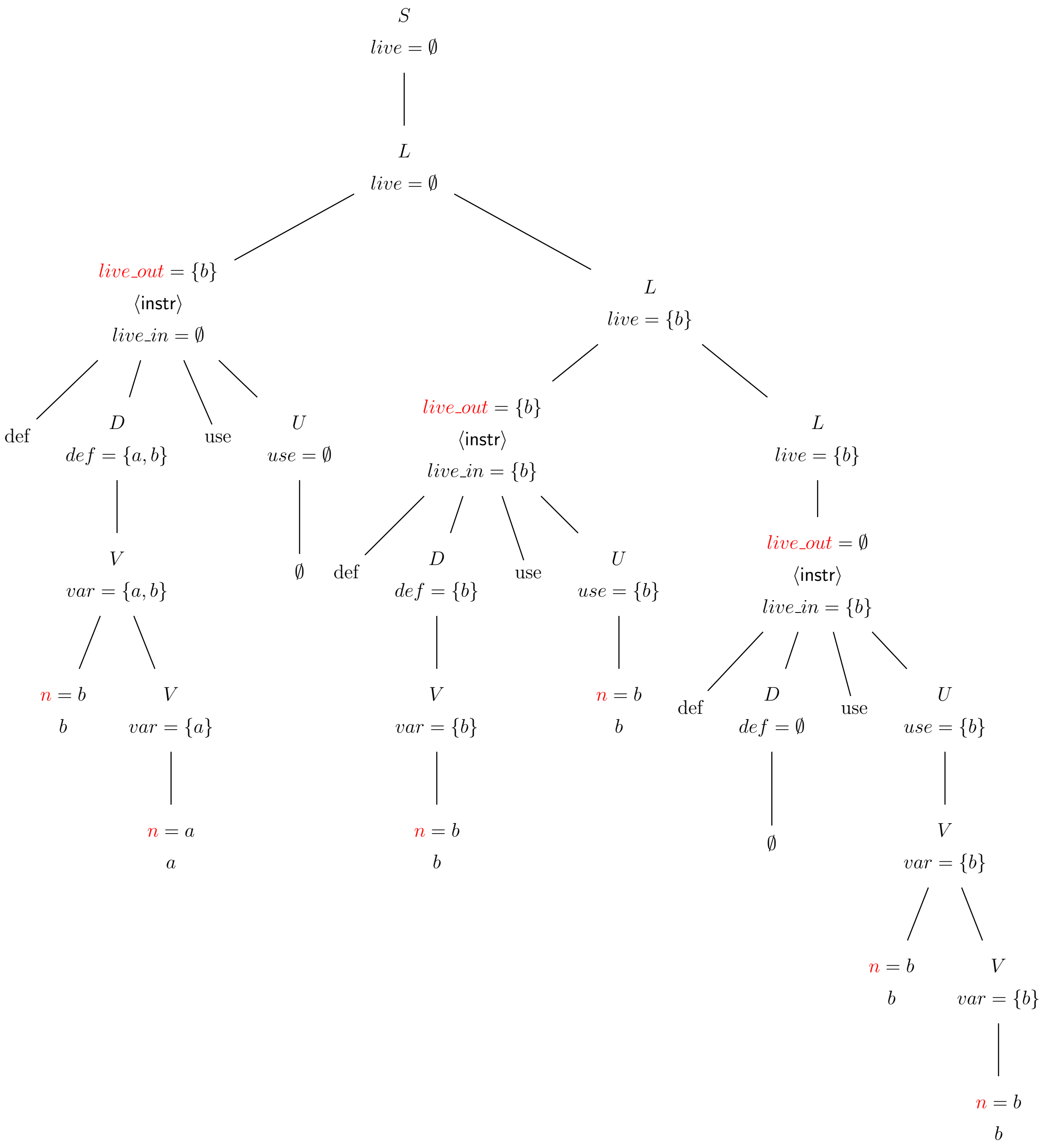
Attributes are defined so as to simulate the standard computation of live variables by means of the method of flow equations, for a purely sequential program (neither loops nor conditional statements are present). In particular, the live variables at the input and output of the current instructions, that is attributes *live_in* and *live_out*, respectively, are associated with node ⟨instr⟩, which represents the current instruction, exactly as it is done in the standard computation. An additional attribute, *live*, collects the live variables at the input of each instruction and propagates them towards the root of the syntax tree, as far as the beginning of the program, by following the chain of nonterminals L. Such an attribute therefore simulates the process of iterative solution of flow equations. The hereditary nature of attribute *live_out* helps to transport live variables to the output of the current instruction, as there such variables are needed to recompute the live variables at the instruction input. Here are the related semantic functions of grammar G:

syntax	*semantic functions*
$S_0 \rightarrow L_1$	$live_0 = live_1$
$L_0 \rightarrow \langle \mathsf{instr} \rangle_1 \; L_2$	$live\_out_1 = live_2$ $live_0 = live\_in_1$
$L_0 \rightarrow \langle \mathsf{instr} \rangle_1$	$live\_out_1 = \emptyset$ $live_0 = live\_in_1$
$\langle \mathsf{instr} \rangle_0 \rightarrow$ ‘def’ D_1 ‘use’ U_2 ‘;’	$live\_in_0 = use_2 \cup (live\_out_0 - def_1)$
$D_0 \rightarrow$ ‘$\emptyset$’	$def_0 = \emptyset$
$D_0 \rightarrow V_1$	$def_0 = var_1$
$U_0 \rightarrow$ ‘$\emptyset$’	$use_0 = \emptyset$
$U_0 \rightarrow V_1$	$use_0 = var_1$
$V_0 \rightarrow$ ‘id’ ‘,’ V_1	$var_0 = \{n\} \cup var_1$
$V_0 \rightarrow$ ‘id’	$var_0 = \{n\}$

Of course, some of the above semantic rules embed the flow equations that are used to define live variables, namely the functions associated with the expansion of nonterminal $\langle \mathsf{instr} \rangle$. □

Solution of (2)

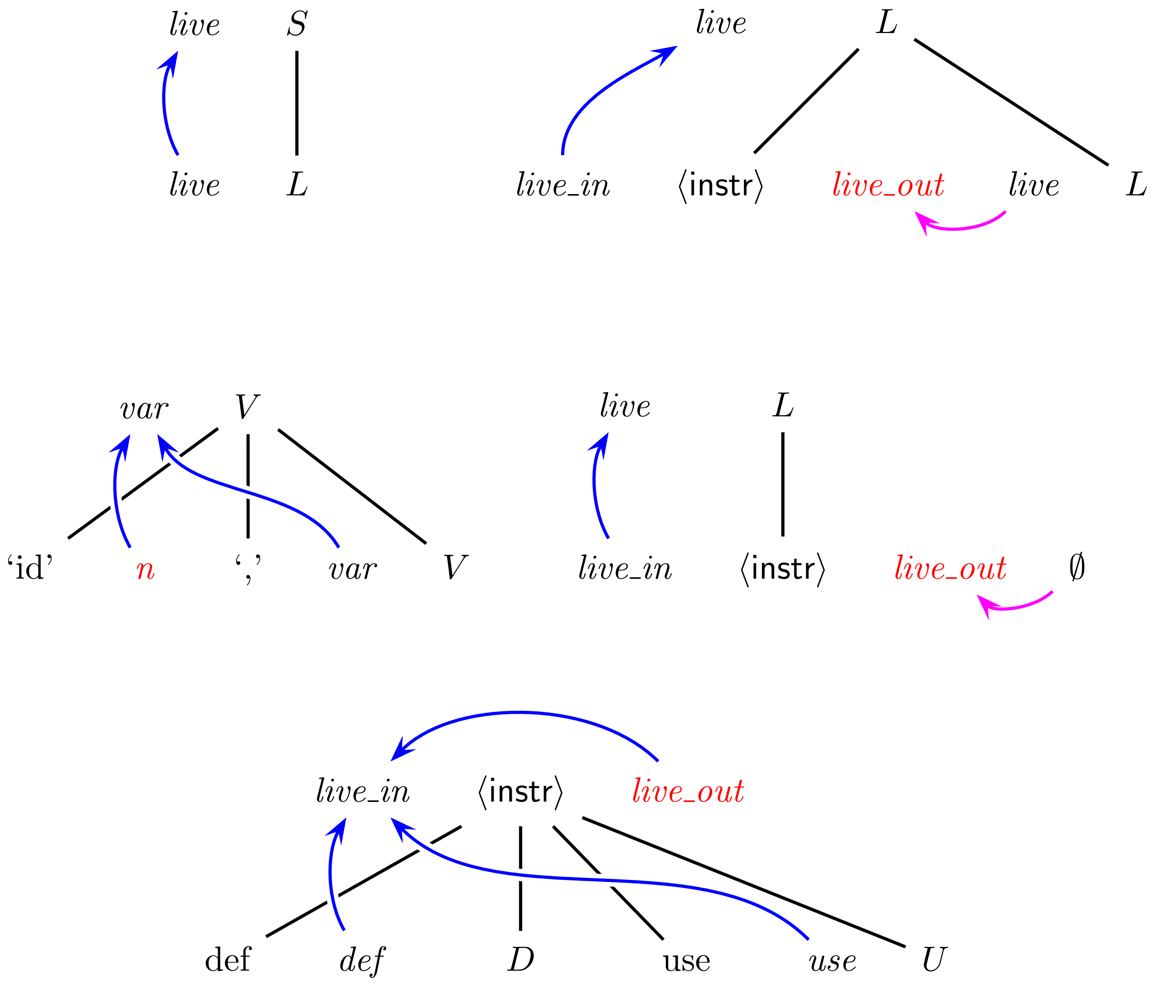
Here is the decorated syntax tree of grammar G:



For reasons of space, the left (synthesised) attributes are written under the reference nonterminal, while the right (inherited) ones are written above; usually left and right attributes are written on the left and right side of the reference nonterminal, respecively. The computation and propagation of attributes follow the semantic functions given before. Inherited attributes *live_out* and n are coloured in red, to put them into evidence at first glance. □

Solution of (3)

All the attributes are synthesised, with the exception of attributes *live_out* and n which are inherited. Here are the function dependences among attributes (only for the most important rules of grammar G):



The remaining rules are trivial and here are not shown. If one examines the dependences among attributes (in blue and magenta for the synthesised and inherited ones respectively), one sees easily that the attribute grammar is well defined because the dependences are clearly loop-free (acyclic). Moreover the attribute grammar fulfills the one-sweep condition, as follows:

- synthesised attributes depend only on attributes of child nodes or on attributes (of any type) of the same node

- the left attribute *live_out*, associated with the subtree ⟨instr⟩ in the rule $L \rightarrow$ ⟨instr⟩ L, depends on a synthesised attribute, *live*, associated with subtree L, which is brother (right) of subtree ⟨instr⟩ (in the same rule of course)

As subtree L is connected on the right of nonterminal ⟨instr⟩, to satisfy the dependence of attribute *live_out* it suffices to evaluate first subtree L and then subtree ⟨instr⟩.

However the attribute grammar is not of type L (left), because as said before it is necessary to swap the two subtrees ⟨instr⟩ and L, with respect to the evaluation order of type L which otherwise would proceed from left to right. □

Solution of (4)

Here are the semantic evaluation procedures (written in pseudocode) of nonterminals L, ⟨instr⟩ and V, which are the three most structured ones (though to answer the question one procedure would suffice):

```
procedure L (live: out; tree: in)
– local variables to pass attributes
    var live_tmp
– alternative rule “$L \rightarrow \langle\mathsf{instr}\rangle\ L$”
    call L (live_tmp, tree → L)
    call INSTR (live, tree → ⟨instr⟩, live_tmp)
– alternative rule “$L \rightarrow \langle\mathsf{instr}\rangle$”
    call INSTR (live, tree → ⟨instr⟩, ∅)
end L
```

Notice that the computation order of nonterminals ⟨instr⟩ and L is swapped with respect to the order they have in the rule; swapping is imposed by the function dependences.

```
procedure INSTR (live_in: out; tree, live_out: in)
– local variables to pass attributes
    var def_tmp, use_tmp
– rule “⟨instr⟩ → ‘def’ $D$ ‘use’ $U$”
    call D (def_tmp, tree → D)
    call U (use_tmp, tree → U)
    $live\_in = use\_tmp \cup (live\_out - def\_tmp)$
end INSTR
```

Here procedures D and U are invoked in the same order as in the rule, because only synthesised attributes are involved (but attribute n which however is precomputed and hence available).

```
procedure V (var: out; tree: in)
– local variables to pass attributes
    var var_tmp
– alternative rule “$V \rightarrow$ ‘id’ ‘,’ $V$”
    – attribute $n$ of ‘id’ precomputed ...
    call V (var_tmp, tree → V)
    $var = \{n\} \cup var\_tmp$
– alternative rule “$V \rightarrow$ ‘id’”
    – attribute $n$ of ‘id’ precomputed ...
    $var = \{n\}$
end V
```

Keywords **in** and **out** indicate the input (inherited attributes) and output (synthesised attributes) parameters of the semantic evaluation procedure, respectively; do not confuse such keywords with attributes *live_in* (synthesised of type **out**) and *live_out* (inherited of type **in**) associated with nonterminal ⟨instr⟩. Local variables (if any) have the same name as the attribute the value of which they store temporarily, with the mnemonic suffix *tmp*; such variables help to transport attributes from one procedure to another one and to compute semantic functions. Input parameter *tree* is the pointer to the (sub)tree the procedure analyses. The list of formal parameters in the procedure heading follows the same order conventionally used to decorate the syntax tree; the (sub)tree pointer (which ideally represents the root nonterminal of the tree) splits the list into a left part (synthesised attributes) and a right part (inherited attributes); therefore the parameter list is easy to read and interpret.

The dependences of alternative '$V \to$ 'id'' are not shown previously, but are at all obvious. As before, the attributes passed to a semantic procedure as input parameters (and hence inherited) are put into evidence and coloured in red (do not forget however that as said before they are all listed on the right side of the subtree pointer).

The remaining semantic procedures are rather trivial and here are omitted. The syntax tree is assumed to have already been entirely constructed and passed to the procedure by means of pointer *tree*. □

Observation

Of course there may exist other solutions and the one given here could probably be simplified a little, at the expense of making it less readable and less conformant to the structure of the standard computation of live variables. For instance, one could try to eliminate inherited attribute *live_out* (which has a somewhat limited use as it helps only to transport live variables from node L the the brother node ⟨instr⟩) and to reduce all to a purely synthesised solution, possibly by renouncing to a perfect simulation of the standard algorithm for computing live variables (this choice is after all not mandatory for the exercise and here has been preferred only to give a structured and general example). See also the next solution, which is drafted soon after.

Simplified solution

Solution of (1)

Only synthesised attributes are used (but attribute n which is trivial and already given in the exercise). With respect to the previous extended solution, here the idea is that of eliminating attributes *live_in* and *live_out*, computing live variables and transporting them by means only of attribute *live* through the chain of nonterminals L (actually this chain is a list). The program is purely sequential (there are neither loop nor conditional statements), hence one set of live variables suffices to carry out the computation, without having the usual sets of live variables at the input and outputs. In fact, having separate sets of live variables at the input and outputs of every node of the program helps essentially when the node has two or more different successors, because the live variables at the outputs might be different depending on which output path is taken, and therefore they should be stored in as many sets of live variables as the output arcs are, while the live variables at the input constitute only one set. But in the present case nodes (instructions) always have a single successor. However it is necessary to extend attributes def and *use* also to nonterminal ⟨instr⟩.

Here is the list of attributes of grammar G:

attributes to be used in the grammar

type	name	(non)terminal	domain	meaning and interpretation
attributes already given in the exercise				
right	n	id	string	identifier name
attributes to be extended or added				
left	*var*	V	set of strings	list of variable names (new)
left	*def*	D, $\langle \mathsf{instr} \rangle$	idem	var.s defined by the current instruction (new)
left	*use*	U, $\langle \mathsf{instr} \rangle$	idem	var.s used by the current instruction (new)
left	*live*	S, L	idem	live variables of the program (new)

As one sees soon all the meaningful attributes are synthesised, with the exception of attribute n which is precomputed and for this reason is not of any disturb.

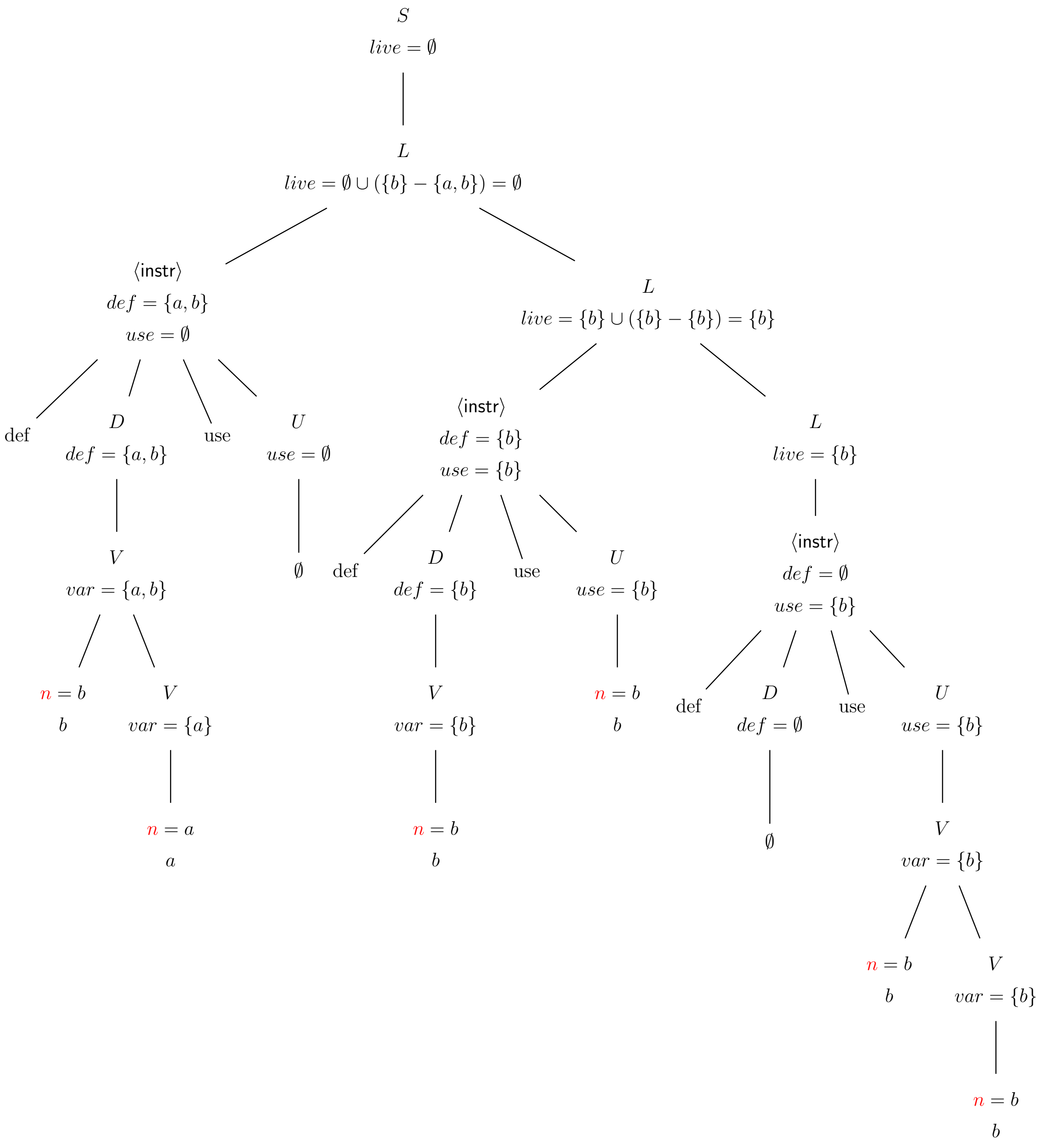
Here are the semantic functions of grammar G:

syntax	*semantic function*
$S_0 \rightarrow L_1$	$live_0 = live_1$
$L_0 \rightarrow \langle \mathsf{instr} \rangle_1 \; L_2$	$live_0 = use_1 \cup (live_2 - def_1)$
$L_0 \rightarrow \langle \mathsf{instr} \rangle_1$	$live_0 = use_1$
$\langle \mathsf{instr} \rangle_0 \rightarrow$ ‘def’ D_1 ‘use’ U_2 ‘;’	$def_0 = def_1$ $use_0 = use_2$
$D_0 \rightarrow$ ‘$\emptyset$’	$def_0 = \emptyset$
$D_0 \rightarrow V_1$	$def_0 = var_1$
$U_0 \rightarrow$ ‘$\emptyset$’	$use_0 = \emptyset$
$U_0 \rightarrow V_1$	$use_0 = var_1$
$V_0 \rightarrow$ ‘id’ ‘,’ V_1	$var_0 = \{n\} \cup var_1$
$V_0 \rightarrow$ ‘id’	$var_0 = \{n\}$

The essential point is that the computation of live variables is concentrated at the level of nonterminal L, which generates the list of instructions, not at the level of individual instruction, that is nonterminal $\langle \mathsf{instr} \rangle$. □

Solution of (2)

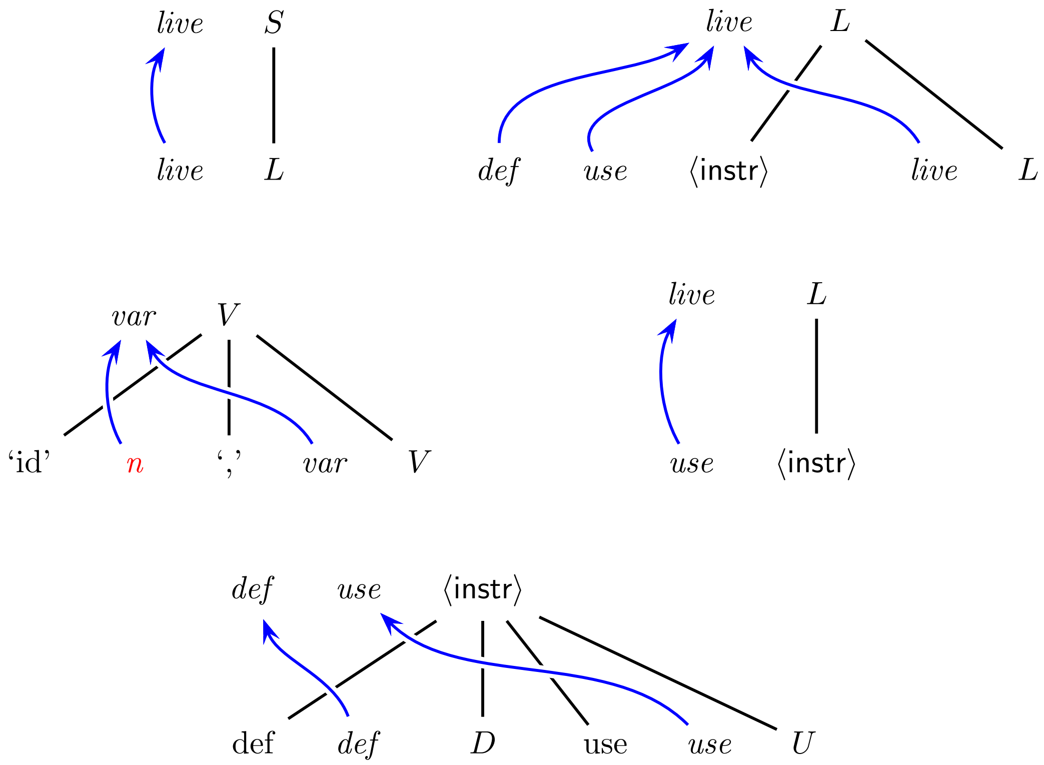
Here is the decorated syntax tree (it also shows the computation of live variables), where for reasons of space attributes (all synthesised) are listed below nodes, instead of listing them on the left side of the node, as usual:



The only inherited attribute (n precomputed) is coloured in red.

Solution of (3)

Here are the elementary dependence graphs of the rules of grammar G:



The remaining rules are trivial and here are not shown. Of course all the arrows are directed upwards, as all the attributes are synthesised (but attribute n).

Since here only synthesised attributes are used, grammar G is certainly of type L (of course this implies that it is of type one-sweep as well). □

Solution of (4)

Here is a meaningful semantic evaluation procedure (that of nonterminal L), which contains the actual computation of live variables:

```
procedure L (live: out; tree: in)
– local variables to pass attributes
        var def_tmp, use_tmp, live_tmp
– alternative rule "$L \rightarrow \langle\text{instr}\rangle\ L$"
        call INSTR (def_tmp, use_tmp, tree $\rightarrow \langle\text{instr}\rangle$)
        call L (live_tmp, tree $\rightarrow L$)
        $live = use\_tmp \cup (live\_tmp - def\_tmp)$
– alternative rule "$L \rightarrow \langle\text{instr}\rangle$"
        call INSTR (def_tmp, use_tmp, tree $\rightarrow \langle\text{instr}\rangle$)
        $live = use\_tmp$
end L
```

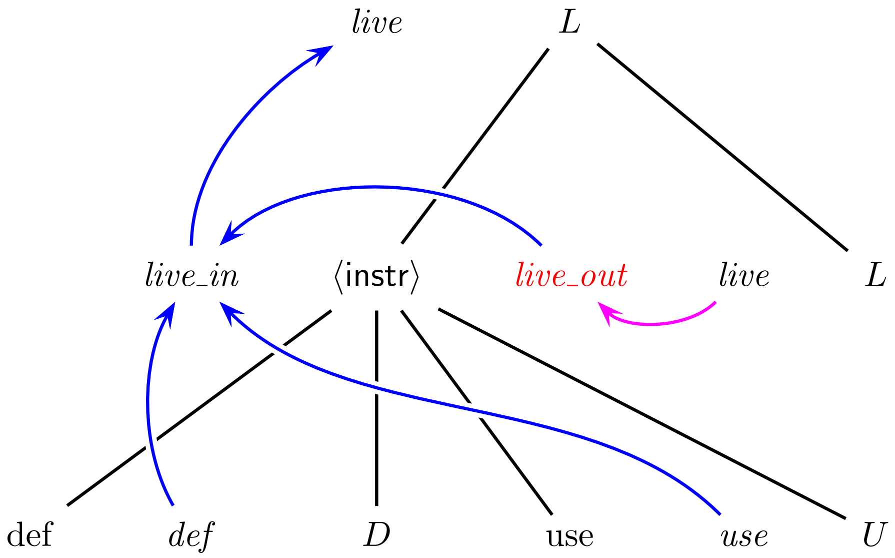
Of course the invocation order of the procedures of the child nodes associated with nonterminal L is from left to right. It is useful to compare this version of the procedure with the same one in the previous extended solution.

The reader is left the task of writing the remaining procedures (one suffices for the exercise anyway). Clearly the procedures have only output parameters, corresponding to the synthesised attributes, but parameter *tree*, the pointer to the subtree to analyse, which is always present and obviously is an input parameter. □

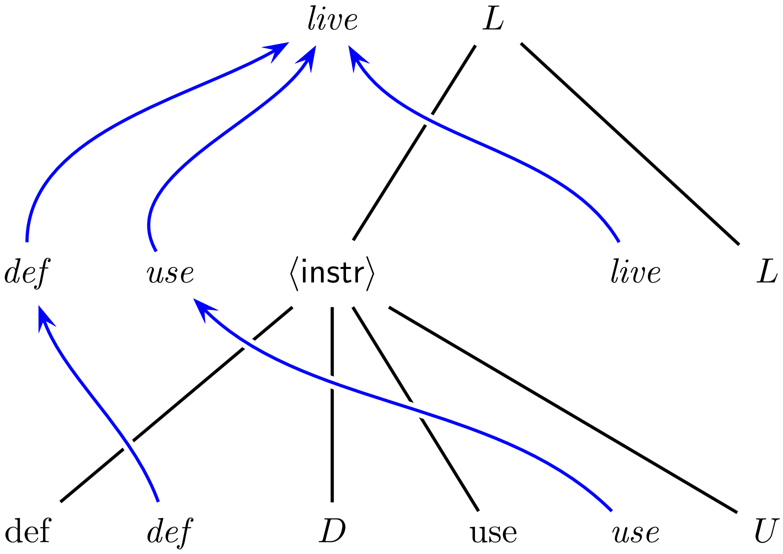
Observation

It is evident that the solution is extremely simple though at all valid. In order to find it, it is necessary first to have well assimilated the essence of the computation of live variables and then model it by means of an attribute scheme which is as linear as possible. To this purpose, compare the attribute value propagation that takes place when combining the rules expanding nonterminal L and ⟨instr⟩ in the extended solution with the same situation in the simplified solution. Here is the comparison:

in the extended solution



in the simplified solution



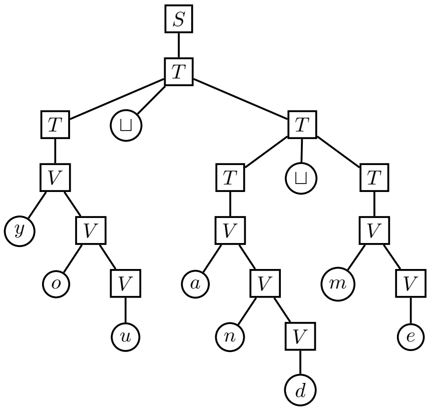
Such a graphic representation shows clearly that the simplification consists of avoiding the intermediate passage of live variables mediated by attributes *live_out* and *live_in*, inherited and synthesised respectively, both of which are eliminated (while attributes *def* and *use* are extended also to ⟨instr⟩).

On the other side, the extended solution is better suited for generalisation, in particular if the instruction has two or more outputs, as in such cases it would be necessary to have two or more separate sets of live variables at the outputs (one set for each output), as said before.

Exercise 53 A (simplified) text in natural language, consisting of a sequence of words separated by one blank "⟨blank⟩", must be centred in a rectangular terminal window of width equal to $W \geq 1$ columns, so that each text row is centred in a window line. The word is modeled roughly as a string of undifferentiated characters c.

Take as a reference the attribute grammar presented in the course textbook (at pp. 300 of the italian edition). Such grammar alignes the text rows on the left side of the terminal window (so it does not centre them). The syntax support G of that grammar is shown below (axiom S), with a sample syntax tree ("␣" replaces "⟨blank⟩"):

$$G \begin{cases} S \rightarrow T \\ T \rightarrow T \text{ ‘}\langle\mathsf{blank}\rangle\text{’ } T \\ T \rightarrow V \\ V \rightarrow c\, V \\ V \rightarrow c \end{cases}$$



you ⊔ and ⊔ me

One wishes one modified such an attribute grammar by adding new semantic attributes, if necessary, and / or by modifying the already known ones, in order to compute the column of the end character of the rightmost word of each centred row. Here follows an example, the same as in the textbook but with centring rather than left alignment: the phrase “la torta ha gusto ma la grappa ha calore” is centred in a terminal window of width $W = 13$ columns:

1	2	3	4	5	6	7	8	9	10	11	12	13
	l	a		t	o	r	t	a		h	a	
	g	u	s	t	o		m	a		l	a	
		g	r	a	p	p	a		h	a		
			c	a	l	o	r	e				

Attribute *ultimo* (*last*) has value 3 for the word “la”, 9 for “torta”, 12 for “ha”, . . . , and 9 for “calore”. Odd blanks, if any, are left on the right side of text rows (see the bottom row).

Answer the following questions:

1. Write the above described attribute grammar G, and give attributes and semantic rules.
2. Check whether the designed grammar G is of type one-sweep or even of type L.

Solution of (1)

Here preference is given to a solution inspired to the grammar presented in the textbook of the course (though it is not mandatory to use that grammar). The necessary attributes are adapted from those in the textbook, which are illustrated at page 300. Both the leftmost and rightmost columns of the row of text are considered (instead of only the rightmost one), as the row of text must be centred in the window line and not simply aligned on the left side.

Therefore the attributes of the new grammar specialise as follows:

- the left (synthesised) attribute *ult* splits into two left attributes cur^l and cur^r (current left and right columns)

- the right (inherited) attribute *prec* splits into two right attributes pre^l e pre^r (precedent left and right columns)

- the left (synthesised) attribute *lun* is left unchanged, but renamed *len* (length)

Here is the detailed list of the new attributes of grammar G, with explanation:

attributes to be used for the grammar

type	name	(non)terminal	domain	meaning and interpretation
attributes already defined in the exercise				
left	*len*	V	integer	length of a word expressed as number of characters
attributes to be extended or added				
left	cur^l	T	integer	leftmost column of the current centred row of text (new)
left	cur^r	T	integer	rightmost column of the current centred row of text (new)
right	pre^l	T	integer	leftmost column of the centred row of text that precedes the current one (new)
right	pre^r	T	integer	rightmost column of the centred row of text that precedes the previous one (new)

The new attribute grammar G centres again the text row and recomputes the leftmost and righmost text columns, whenever a new word is generated and concatenated to the part of already generated text. If it is necessary the grammar starts filling with text the new window line.

Semantic function are a consequence of the above defined attributes and can be easily justified by means of the same reasoning illustrated in the textbook at page 300. Notice the treatment of the two inherited attributes pre^l and pre^r. The new solution is displayed in parallel with the old one, to put into evidence the structural analogy.

syntax	*new semantic function*	*old function*
$S_0 \rightarrow T_1$	$pre^l_1 = 1$	
	$pre^r_1 = 0$	$prec_1 = -1$
$T_0 \rightarrow T_1$ ‘⟨blank⟩’ T_2	$pre^l_1 = pre^l_0$	
	$pre^r_1 = pre^r_0$	$prec_1 = prec_0$
	$pre^l_2 = cur^l_1$	
	$pre^r_2 = cur^r_1$	$prec_2 = ult_1$
	$cur^l_0 = cur^l_2$	
	$cur^r_0 = cur^r_2$	$ult_0 = ult_2$
$T_0 \rightarrow V_1$	**if** $\big(pre^r_0 - pre^l_0 + 1 + len_1 \leq W\big)$ **then**	$prec_0 + 1 + lun_1 \leq W$
	$cur^l_0 = \left\lfloor \frac{\left(W - \left(pre^r_0 - pre^l_0 + 1 + lrn_1\right)\right)}{2} \right\rfloor$	$ult_0 =$
	$cur^r_0 = \left\lceil \frac{\left(W + \left(pre^r_0 - pre^l_0 + 1 + len_1\right)\right)}{2} \right\rceil$	$prec_0 + 1 + lun_1$
	else	
	$cur^l_0 = \left\lfloor \frac{(W - len_1)}{2} \right\rfloor$	
	$cur^r_0 = \left\lceil \frac{(W + len_1)}{2} \right\rceil$	$ult_0 = lun_1$
	endif	
$V_0 \rightarrow c\, V_1$	$len_0 = 1 + len_1$	
$V_0 \rightarrow c$	$len_0 = 1$	

Symbols “$\lceil\ \rceil$” and “$\lfloor\ \rfloor$” (which are named “Gauss’ parentheses”), applied to a fractional number, return the upper and lower approximating integer, respectively: for instance it holds $\lceil 1,5 \rceil = 2$ and $\lfloor 1,5 \rfloor = 1$). Here Gauss parentheses are used to place the possible odd blank on the right end of the window line. □

Observation

The behaviour of the described attribute grammar G reminds that of the commonest text processing utilities (like for instance Word and others) when a text document is edited in automatic centring mode: for every new input word, or more precisely for every new character input by the keyboard, the text row is soon centred again on screen in the window line, and the leftmost and rightmost text columns are recomputed. One can thus imagine a page formatting algorithm structured as an attribute grammar, which is run whenever a new character or word is added to the text row, rebuilds (or updates) the syntax tree of the input text and recomputes the attributes. When this happens the text row is immediately redisplayed on screen. Such an algorithm is implemented as a part of the graphic system of the computer and is activated automatically as an interrupt routine associated with every key pressure.

Solution of (2)

The designed attribute grammar G is of type one-sweep as well as the original grammar at page 300 of the textbook of the course, which is the model the new grammar is inspired to. This can be easily verified by checking the one-sweep condition and the reader is left the task of doing so by himself. Grammar G is of type L as well, because the computation flow of attributes goes from left to right. Notice however that the syntactic support of G is ambiguous (due to the recursive two-sided rule $T \to T \perp T$) and that therefore it is necessary to build in some way one syntax tree (of the many possible ones) and to give the semantic evaluator such an already built tree.

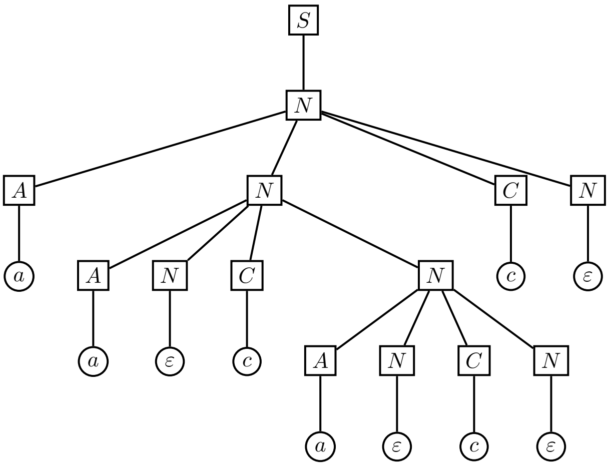
To convince oneself of the correctness of the new attribute grammar G, one can reconsider the decorated syntax tree drawn at page 300 of the textbook and adapt it to the new attributes, by using the sample text fragment shown above. The reader is left such a task, too.

Of course there may exist alternative solutions, structurally different from the grammar presented in the textbook and exploited here, though as valid as that. □

Exercise 54 Consider the Dyck language over alphabet $\{a, c\}$. Given a language phrase, one must enumerate progressively the "parentheses nests" from 1 on. As one goes down into the nested structure, at each new level the enumeration starts from 1 again. The outer nests (those not contained in any other nest) are all numbered with 1.

The source language is defined by the following syntax support G (axiom S), with a sample syntax tree aside:

$$G \begin{cases} S & \to & N \\ N & \to & A\,N\,C\,N \\ N & \to & \varepsilon \\ A & \to & a \\ C & \to & c \end{cases}$$



$a\,a\,c\,a\,c\,c$

The syntax support G may not be modified.

Here follows an example of numbering (the parentheses nests are outlined):

$$\frac{\text{source:}}{\text{dest.:}}\quad \underbrace{\frac{a}{1}\ \underbrace{\frac{a}{1}\frac{c}{1}}_{\substack{\text{inner nest}\\ 1}}\ \underbrace{\frac{a}{2}\frac{c}{2}}_{\substack{\text{inner nest}\\ 2}}\ \frac{c}{1}}_{\text{outer nest 1}}\ \underbrace{\frac{a}{1}\ \underbrace{\frac{a}{1}\frac{c}{1}}_{\substack{\text{inner nest}\\ 1}}\ \underbrace{\frac{a}{2}\ \underbrace{\frac{a}{1}\frac{c}{1}}_{\substack{\text{inner nest}\\ 1}}\ \frac{c}{2}}_{\text{inner nest 2}}\ \underbrace{\frac{a}{3}\frac{c}{3}}_{\substack{\text{inner nest}\\ 3}}\ \frac{c}{1}}_{\text{outer nest 1}}\ \underbrace{\frac{a}{1}\frac{c}{1}}_{\substack{\text{outer nest}\\ 1}}$$

The number n to be output in correspondence with every input letter, must be computed by means of an attribute grammar that uses G as its syntax support. Such a number must be modeled by a right attribute (that is an inherited attribute) of nonterminals A and C.

Answer the following questions:

1. List all the attributes used in the design of attribute grammar G, and write the semantic functions that are necessary to compute attribute n (for both purposes fill the tables prepared in the next two pages).

2. Examine the translation of the Dyck language into the string of numbers n, for instance from the source Dyck string $a\,a\,c\,a\,c\,c\,a\,a\,c\,a\,a\,c\,c\,a\,c\,c\,a\,c$ to the destination numerical string $1\,1\,1\,2\,2\,1\,1\,1\,1\,2\,1\,1\,2\,3\,3\,1\,1\,1$ (it is the same example shown above), and check whether such a translation is invertible.

Solution of (1)

The idea is to have a numerical attribute n to denumerate all the parantheses nests and a boolean attribute *ext* to distinguish between outer nests, all numbered with 1, and inner nests (at depth level 2, 3, etc), progressively numbered from 1 up starting from the first nest (the leftmost one). Here are the attributes:

attributes to be used in the grammar

<table>
<tr><th>type</th><th>name</th><th>(non)terminal</th><th>domain</th><th>meaning and interpretation</th></tr>
<tr><td colspan="5">attributes already defined in the exercise</td></tr>
<tr><td>right</td><td>n</td><td>A, C</td><td>integer</td><td>numbering of parentheses nest</td></tr>
<tr><td colspan="5">attributes to be extended or added</td></tr>
<tr><td>right</td><td>n</td><td>N</td><td>integer</td><td>numbering of parentheses nest (ext.)</td></tr>
<tr><td>right</td><td>ext</td><td>N</td><td>boolean</td><td>holds true and false for outer and inner parentheses nests, respectively (new)</td></tr>
</table>

Attribute n, already defined for nonterminals A and C, is extended and associated also with nonterminal N. Attribute ext is added and associated with nonterminal N. All the attributes are inherited, as here in fact synthesised attributes do not play any role.

And here are the semantic functions to be associated with the rules of grammar G:

syntax	*semantic function*	
$S_0 \rightarrow N_1$	$n_1 = 1$	$ext_1 = \text{true}$
$N_0 \rightarrow A_1\, N_2\, C_3\, N_4$	$n_1 = n_0$ $n_2 = 1$ $n_3 = n_0$ $n_4 = \textbf{if } ext_4 \textbf{ then } 1 \textbf{ else } n_0 + 1$	$ext_2 = \text{false}$ $ext_4 = ext_0$
$N_0 \rightarrow \varepsilon$	nothing	
$A_0 \rightarrow a$	**print** (n_0)	
$C_0 \rightarrow c$	**print** (n_0)	

The semantic functions process the attributes as follows:

- Attribute *ext* is true only for the outer parentheses nests, while for the inner ones, at any depth level, it is always false.

- Attribute n is initially set to 1 for the outer nests, while for the inner ones it starts from 1 and is progressively incremented at every new nest.

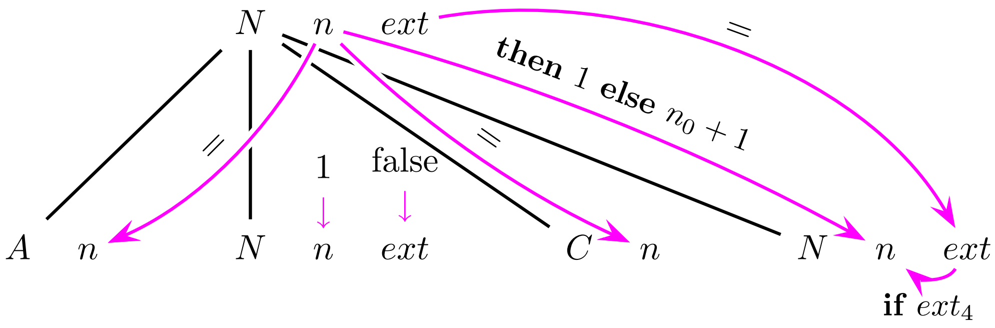
- The print function outputs the translation at the level of nonterminals A and C (or if one prefers at the level of the terminals a and c that expand A and C).

One could associate the complete translation string with the tree root S, by concatenating the partial translation strings associated with nonterminals A and C, and by moving the resulting string upwards to the nonterminal S by means of a synthesised attribute.

It is easy to verify that attribute grammar G is of type one-sweep (check the one-sweep condition). Moreover, as the the computation flow of inherited attributes goes downwards (from root to leaves) and as the syntactic support is of type $LL(1)$ (it is nothing else but the usual Dyck grammar), attribute grammar G turns out to be of type L as well. □

Observation

As a counterproof, here the dependence graph of rule $N \to A\, N\, C\, N$ is given (this is the most complex and important rule of the whole attribute grammar). Here it is:



The computation flow goes only from parent to child, there does not exist inheritance between brother nodes or upward flow of values from child to parent (because there are not any synthesised attributes). The dependence arcs are labeled with the corresponding semantic functions.

Solution of (2)

It suffices to notice that there exist two source strings with the same translation:

$$\tau(a\,a\,c\,c) = 1\,1\,1\,1 \qquad\qquad \tau(a\,c\,a\,c) = 1\,1\,1\,1$$

to conclude that transduction τ is not invertible. The example above generalises easily to strings of any length. □

Observation

If one wishes to examine transduction τ in a complete way, one can start noticing that the domain of τ is the Dyck language and that the image is a set of strings of integers. For every nonterminal A and C a well defined integer is output, hence every Dyck string causes the emission of only one numerical string and therefore transduction τ is a function (not a generic binary relation). However one should also notice that not every numerical string is the valid image of a Dyck string. Thus such a function can be modeled as follows:

$$\tau\colon \text{Dyck language over } \{a, c\} \longrightarrow \text{Valid numerical strings}$$

The following considerations apply:

- not all pairs of Dyck strings have different denumeration; for instance, given $h, k \geq 1$ with $h \neq k$, the two following strings are different:

  $$\left(a^h\,c^h\right)^k \neq \left(a^k\,c^k\right)^h$$

  but their translations are identical, because:

  $$\tau\left(\left(a^h\,c^h\right)^k\right) = \left(1^h\,1^h\right)^k = 1^{2hk} \qquad \tau\left(\left(a^k\,c^k\right)^h\right) = \left(1^k\,1^k\right)^h = 1^{2hk}$$

  therefore function τ is not injective

- as the image set is restricted to the strings that represent valid denumerations of Dyck strings, function τ is surjective

In conclusion, as transduction (or function) τ is not injective, it is not one-to-one either[1] and therefore it is not invertible.

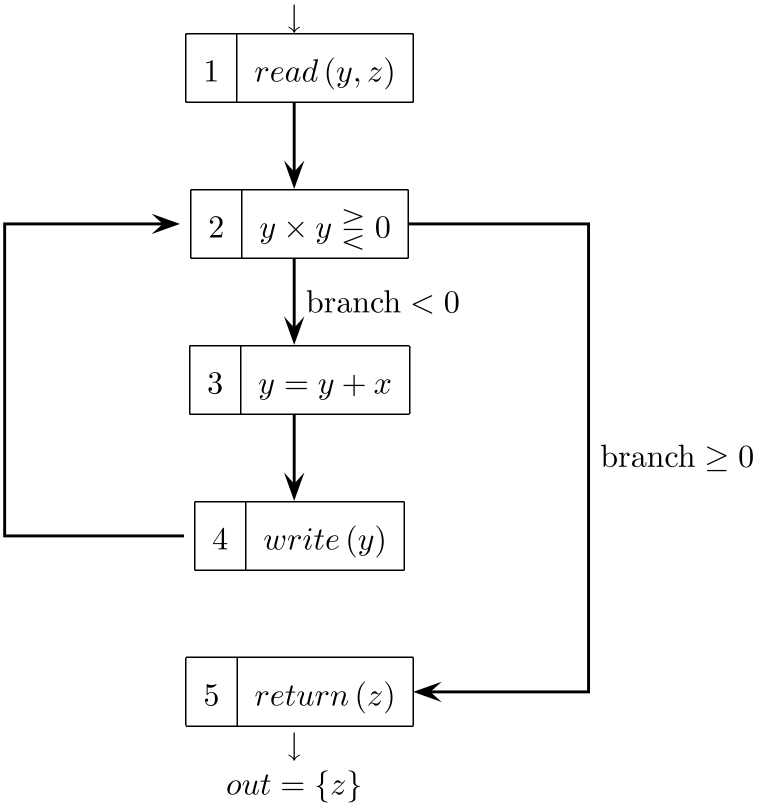
[1] A function is one-to-one if and only if it is both injective and surjective.

Chapter 6

Static Flow Analysis

6.1 Live Variables

Exercise 55 A subprogram (routine) with an input and an output parameter x and z, respectively, both of integer type, is modeled by means of the following control graph:



Node 2 is a two-way conditional instruction and parameter z is live at the output of node 5. Answer the following questions:

1. Write the control flow equations to compute the *live variables* at each node in the program.

2. Compute the solution of the flow equations and list live variables.

3. Suppose that conditional node 2 executes the following test:

   if $(y \times y \geq 0)$ go to node 5 else go to node 3

   Indicate which sets of live variables change as a consequence, and how.

Solution of (1)

The exercise does not have any difficulty and the solution technique is fully standard, divided into a series of phases. Here it is:

Here is the computation of the constant terms (definition and use of variables):

#	**def**	**use**
1	y, z	–
2	–	y
3	y	x, y
4	–	y
5	–	z

table of constant terms

- y and z appear in *read* in 1, hence there they are defined
- y appears in the expression in 2, hence there it is used
- y is assigned in 3, hence there it is defined
- x and y appear in the expression in 3, hence there they are used
- y appears in *write* in 4, hence there it is used
- z is used by the caller program

And here are the flow equations of the live variables at the nodes:

$$
\begin{array}{rcll}
in(1) & = & use(1) \cup (out(1) - def(1)) = & \text{– definition of input liveness} \\
 & = & \emptyset \cup (out(1) - \{y, z\}) = & \\
 & = & out(1) - \{y, z\} & \\
out(1) & = & in(2) & \text{– node 1 has only one output} \\
\hline
in(2) & = & use(2) \cup (out(2) - def(2)) = & \text{– definition of input liveness} \\
 & = & \{y\} \cup (out(2) - \emptyset) = & \\
 & = & \{y\} \cup out(2) & \\
out(2) & = & in(3) \cup in(5) & \text{– node 2 has two outputs} \\
\hline
in(3) & = & use(3) \cup (out(3) - def(3)) = & \text{– definition of input liveness} \\
 & = & \{x, y\} \cup (out(3) - \{y\}) = & \text{– read observation below} \\
 & = & \{x, y\} \cup out(3) & \\
out(3) & = & in(4) & \text{– node 3 has only one output} \\
\hline
in(4) & = & use(4) \cup (out(4) - def(4)) = & \text{– definition of input liveness} \\
 & = & \{y\} \cup (out(4) - \emptyset) = & \\
 & = & \{y\} \cup out(4) & \\
out(4) & = & in(2) & \text{– node 4 has only one output} \\
\hline
in(5) & = & use(5) \cup (out(5) - def(5)) = & \text{– definition of input liveness} \\
 & = & \{z\} \cup (out(5) - \emptyset) = & \\
 & = & \{z\} \cup out(5) & \\
out(5) & = & \{z\} & \text{– use is specified in the exercise}
\end{array}
$$

The above equations do not admit any special simplification. □

Observation

Pay attention not to make any mistakes and not to perform the following wrong simplification:

$$\{x, y\} \cup (out(3) - \{y\}) = \{x\} \cup out(3) \qquad \text{IS FALSE !!!}$$

Think well about how the set difference operator is defined and works!

Solution of (2)

Here is the iterative computation of the solution to the flow equations:

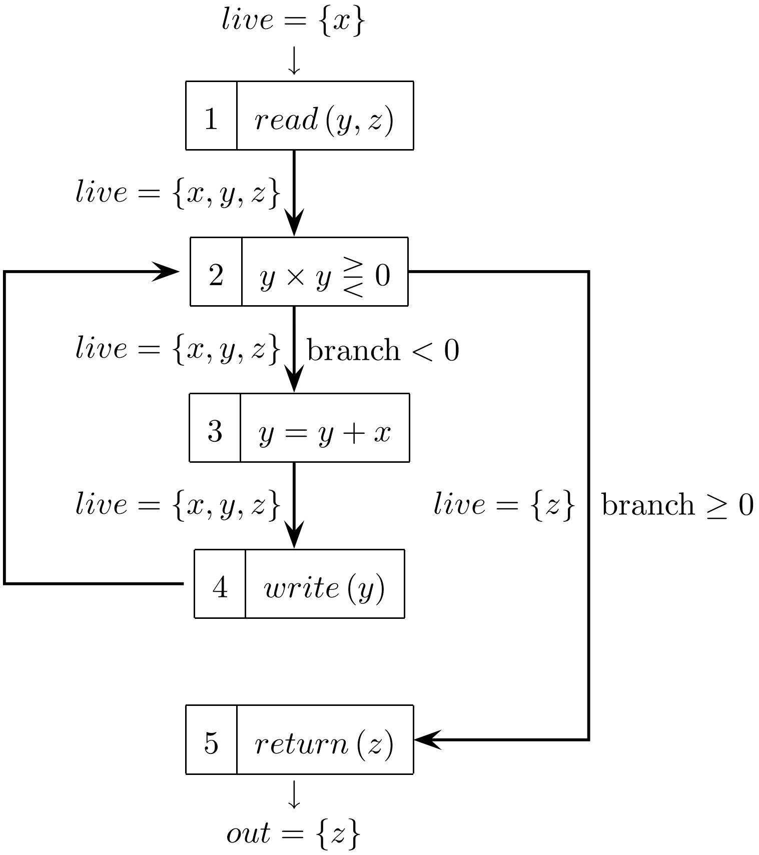
		step 1		**step** 2		**step** 3		**step** 4		
	state 0		**state** 1		**state** 2		**state** 3		**state** 4	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	y	$\emptyset$	$x\,y\,z$	x	$x\,y\,z$
2	$\emptyset$	$\emptyset$	y	$\emptyset$	y	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$
3	$\emptyset$	$\emptyset$	$x\,y$	$\emptyset$	$x\,y$	y	$x\,y$	y	$x\,y$	y
4	$\emptyset$	$\emptyset$	y	$\emptyset$	y	y	y	y	y	$x\,y\,z$
5	$\emptyset$	$\emptyset$	z	z	z	z	z	z	z	z

(continues and finishes on the next table)

		step 5		**step** 6		$6 = 7$		
	state 4		**state** 5		**state** 6		**state** 7	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	x	$x\,y\,z$	x	$x\,y\,z$	x	$x\,y\,z$	x	$x\,y\,z$
2	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$
3	$x\,y$	y	$x\,y$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$
4	y	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$	$x\,y\,z$
5	z	z	z	z	z	z	z	z

At state number 6 convergence is reached, as states 6 and 7 do not differ. Therefore in total six steps are needed to compute the solution. The computation may be sped up, if one succeeds in deducing from the equations some set identities that can be exploited to merge two or more steps into one. The reader could exercise by himself to speed up the computation in this way, if possible.

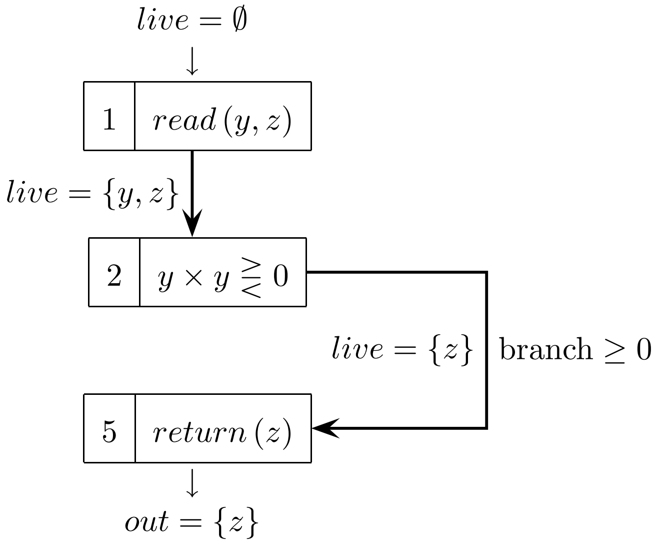
For better clarity, here is the control flow graph of the program labeled with the sets of live variables at the nodes:



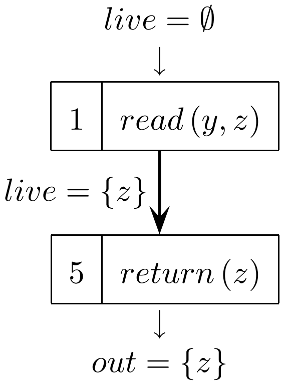
At the internal nodes of the program (nodes 2, 3 and 4) all the variables are live. □

Solution of (3)

If one examines more closely the semantic of the routine, one sees soon that the loop body 3 4 is never executed, because the square $y \times y = y^2$ may not be < 0, and hence nodes 3 and 4 may be eliminated. Thus at the input of node 2 parameter x is not live, actually. Then x is not live at the input of node 1 either (in conclusion x is unuseful and nodes 3 4 are unreachable). Here is (figure (a)) the simplified control graph of the program, where live variables are reduced:



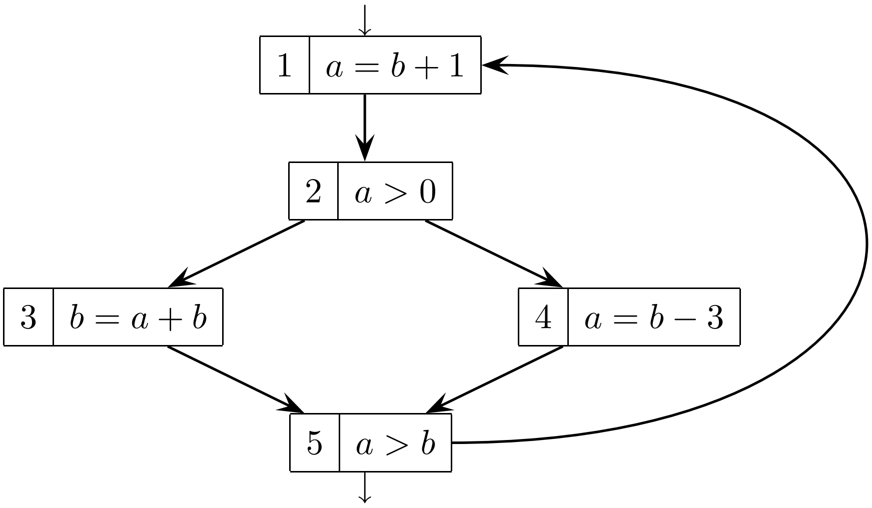
(a)



(b)

Then one can eliminate the test node 2 as well, because predicate $y \times y \gtreqless 0$ contained there is tautological, and as a consequence variable y is not live at the node input any longer. In conclusion the routine reads only variable z and returns it as is. Above is (figure (b)) the simplified control flow graph where live variables are further reduced. Of course, at this point one could also eliminate variable y form instruction *read* (such a removal however has a side effect on the I/O behaviour of the routine). □

Exercise 56 Consider the following control flow graph of a program:



At the end of the program, no variable is live any longer.

Answer the following questions:

1. Write the flow equations to compute the liveness intervals of the variables.

2. Compute and write the sets of live variables at every point of the program.

Solution of (1)

Here is the computation of the constant terms (definition and use of variables):

#	**def**	**use**
1	a	b
2	-	a
3	b	a, b
4	a	b
5	-	a, b

table of constant terms

- a is assigned in 1, hence there it is defined, b appears in the expression in 1, hence there it is used
- a appears in the expression in 2, hence there it is used
- b is assigned in 3, hence there it is defined, a e b appear in the expression in 3, hence there they are used
- a is assigned in 4, hence there it is defined, b appears in the expression in 4, hence there it is used
- a e b appear in the expression in 5, hence there they are used

And here are the flow equations of the live variables at the nodes:

$$
\begin{array}{rcll}
in(1) & = & use(1) \cup (out(1) - def(1)) = & \text{– definition of input liveness} \\
 & = & \{b\} \cup (out(1) - \{a\}) = & \\
 & = & \{b\} & \\
out(1) & = & in(2) & \text{– node 1 has only one output} \\
\hline
in(2) & = & use(2) \cup (out(2) - def(2)) = & \text{– definition of input liveness} \\
 & = & \{a\} \cup (out(2) - \emptyset) = & \\
 & = & \{a\} \cup out(2) & \\
out(2) & = & in(3) \cup in(4) & \text{– node 2 has two outputs} \\
\hline
in(3) & = & use(3) \cup (out(3) - def(3)) = & \text{– definition of input liveness} \\
 & = & \{a, b\} \cup (out(3) - \{b\}) = & \\
 & = & \{a, b\} & \\
out(3) & = & in(5) & \text{– node 3 has only one output} \\
\hline
in(4) & = & use(4) \cup (out(4) - def(4)) = & \text{– definition of input liveness} \\
 & = & \{b\} \cup (out(4) - \{a\}) = & \\
 & = & \{b\} & \\
out(4) & = & in(5) & \text{– node 4 has only one output} \\
\hline
in(5) & = & use(5) \cup (out(5) - def(5)) = & \text{– definition of input liveness} \\
 & = & \{a, b\} \cup (out(5) - \emptyset) = & \\
 & = & \{a, b\} & \\
out(5) & = & \emptyset \cup in(1) = & \text{– node 5 has two outputs} \\
 & = & in(1) &
\end{array}
$$

Several equations can be solved soon and turn out to be simply assignments to a constant. □

Solution of (2)

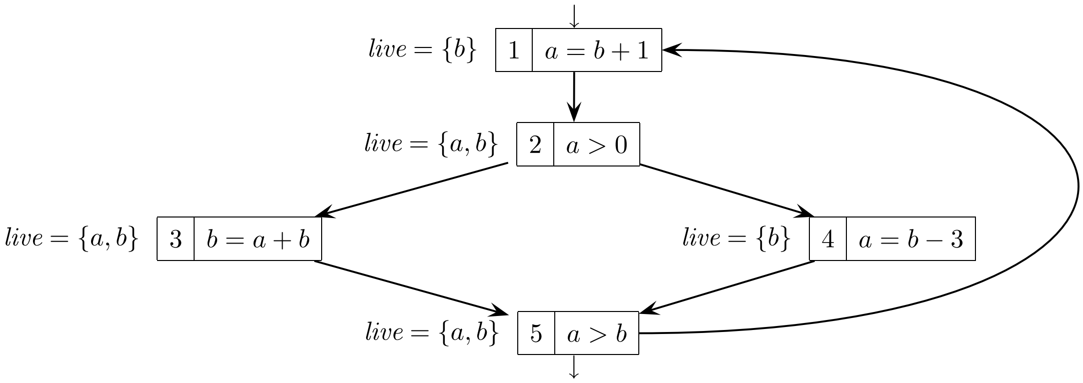
Here is the iterative computation of the solution to the flow equations:

		step 1		**step** 2		**step** 3		**step** 4		$4 = 5$	
	state 0		**state** 1		**state** 2		**state** 3		**state** 4		**state** 5
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$\emptyset$	$\emptyset$	b	$\emptyset$	b	a	b	a	b	$a\,b$	b	$a\,b$
2	$\emptyset$	$\emptyset$	a	$\emptyset$	a	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$
3	$\emptyset$	$\emptyset$	$a\,b$	$\emptyset$	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$	$a\,b$
4	$\emptyset$	$\emptyset$	b	$\emptyset$	b	$a\,b$	b	$a\,b$	b	$a\,b$	b	$a\,b$
5	$\emptyset$	$\emptyset$	$a\,b$	$\emptyset$	$a\,b$	b	$a\,b$	b	$a\,b$	b	$a\,b$	b

The iterative solution algorithm converges in four steps, as states 4 and 5 are identical. Both variables are live at all the nodes (that it at the respective inputs), but at nodes 1 and 4 where only variable b is live. □

Observation

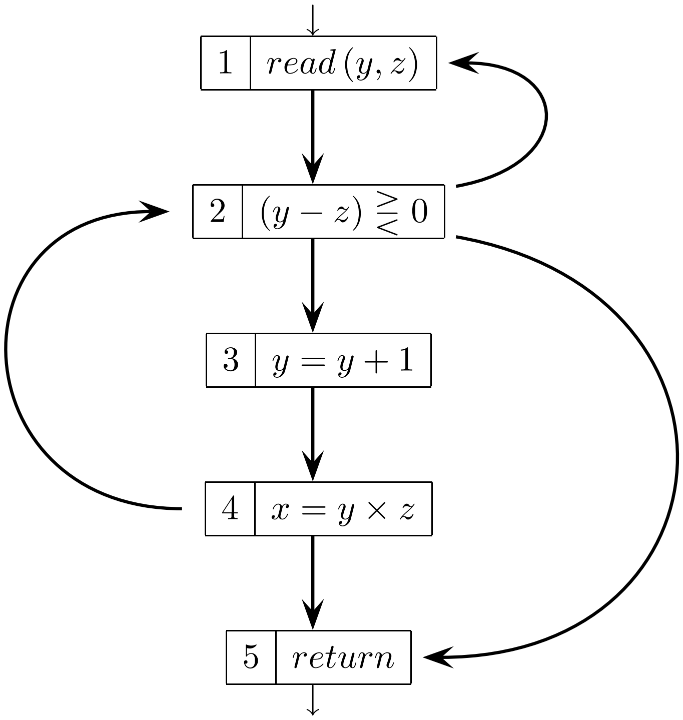
For better clarity, here is the control flow graph of the program labeled with the sets of live variables at the nodes:



This is simply a different way of representing the solution. □

6.2 Reaching Definitions

Exercise 57 A routine with input parameter x is modeled as it is shown in the following control graph:



Statements 2 and 4 are a three-way and a two-way conditional structure, respectively. Answer the following questions:

1. Write the flow equations to compute the reaching definitions of every point in the routine.
2. Solve the flow equations written at point (1) and label the control flow graph with the reaching definitions of every point in the routine.
3. Suppose that statement 2 is the following three-way conditional:

   if $(y - z > 0)$ go to 3 else if $(y - z = 0)$ go to 5 else if $(y - z < 0)$ go to 1

   Moreover, suppose that statement 1 reads variables y and z, and assigns them two values which are always positive (non-null) integers. Compute in an exact way the reaching definitions at the output of node 5.

Solution of (1)

The definition of variable x passed to the routine as an input parameter, comes from outside (i.e. from the caller program) and enters node 1. Such a definition is denoted as $x_?$ (where ? is the label of an unknown node located somewhere outside of the routine under analysis).

Before writing flow equations, list constant terms. Here they are, with explanation:

#	**statement**	**def**	**sup**
1	*read* (y, z)	y_1, z_1	y_3
2	$(y - z) \gtreqless 0$	$\emptyset$	$\emptyset$
3	$y = y + x$	y_3	y_1
4	$x = y \times z$	x_4	$x_?$
5	*return*	$\emptyset$	$\emptyset$

table of constant terms

- both y and z are assigned in 1, hence there they are defined; consequently y_3 is suppressed
- in 2 nothing is defined or suppressed
- y is assigned in 3, hence there it is defined; consequently y_1 is suppressed
- x is assigned in 4, hence there it is defined; consequently $x_?$ is suppressed
- in 5 nothing is defined or suppressed

And here follow the required flow equations (symbol $-$ is set difference):

$$
\begin{array}{rcll}
in(1) & = & in\,(caller\ program) \cup out(2) = & \text{- node 1 has two inputs} \\
 & = & \{x_?\} \cup out(2) & \\
out(1) & = & def(1) \cup (in(1) - sup(1)) = & \text{- definition of output reachability} \\
 & = & \{y_1, z_1\} \cup (in(1) - \{y_3\}) & \\
\hline
in(2) & = & out(1) \cup out(4) & \text{- node 2 has two inputs} \\
out(2) & = & def(2) \cup (in(2) - sup(2)) = & \text{- definition of output reachability} \\
 & = & \emptyset \cup (in(2) - \emptyset) = & \\
 & = & in(2) & \\
\hline
in(3) & = & out(2) & \text{- node 3 has only one input} \\
out(3) & = & def(3) \cup (in(3) - sup(3)) = & \text{- definition of output reachability} \\
 & = & \{y_3\} \cup (in(3) - \{y_1\}) & \\
\hline
in(4) & = & out(3) & \text{- node 4 has only one input} \\
out(4) & = & def(4) \cup (in(4) - sup(4)) = & \text{- definition of output reachability} \\
 & = & \{x_4\} \cup (in(4) - \{x_?\}) & \\
\hline
in(5) & = & out(2) \cup out(4) & \text{- node 5 has two inputs} \\
out(5) & = & def(5) \cup (in(5) - sup(5)) = & \text{- definition of output reachability} \\
 & = & \emptyset \cup (in(5) - \emptyset) = & \\
 & = & in(5) &
\end{array}
$$

Some equations can be soon simplified, as already done above, and reduce to an identity. None of them however reduces immediately to a constant term. □

Solution of (2)

Start from the initial approximations $\forall i \; in(i) = out(i) = \emptyset$ (i.e. all the sets are empty), and iterate the equations. One obtains the sets shown below:

		step 1		**step** 2		**step** 3		**step** 4		
	state 0		**state** 1		**state** 2		**state** 3		**state** 4	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$\emptyset$	$\emptyset$	$x_?$	$y_1\, z_1$	$x_?$	$x_?\, y_1\, z_1$	$x_?$	$x_?\, y_1\, z_1$	$x_?\, x_4\, y_1\, z_1$	$x_?\, y_1\, z_1$
2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$x_4\, y_1\, z_1$	$\emptyset$	$x_?\, x_4$	$x_4\, y_1\, z_1$	$x_?\, x_4\, y_1\, y_3\, z_1$	$x_?\, x_4$
3	$\emptyset$	$\emptyset$	$\emptyset$	y_3	$\emptyset$	y_3	$\emptyset$	y_3	$x_4\, y_1\, z_1$	y_3
4	$\emptyset$	$\emptyset$	$\emptyset$	x_4	y_3	x_4	y_3	$x_4\, y_3$	y_3	$x_4\, y_3$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	x_4	$\emptyset$	x_4	x_4	$x_4\, y_1\, y_3\, z_1$	x_4

(continues on the next table)

		step 5		**step** 6		
	state 4		**state** 5		**state** 6	
#	*in*	*out*	*in*	*out*	*in*	*out*
1	$x_? x_4 y_1 z_1$	$x_? y_1 z_1$	$x_? x_4$	$x_? x_4 y_1 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$
2	$x_? x_4 y_1 y_3 z_1$	$x_? x_4$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$
3	$x_4 y_1 z_1$	y_3	$x_? x_4$	$x_4 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3$
4	y_3	$x_4 y_3$	y_3	$x_4 y_3$	$x_4 y_3 z_1$	$x_4 y_3$
5	$x_4 y_1 y_3 z_1$	x_4	$x_? x_4 y_3$	$x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3$

(continues on the next table)

		step 7		
	state 6		**state** 7	
#	*in*	*out*	*in*	*out*
1	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$
2	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$
3	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3 z_1$
4	$x_4 y_3 z_1$	$x_4 y_3$	$x_? x_4 y_3$	$x_4 y_3 z_1$
5	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$

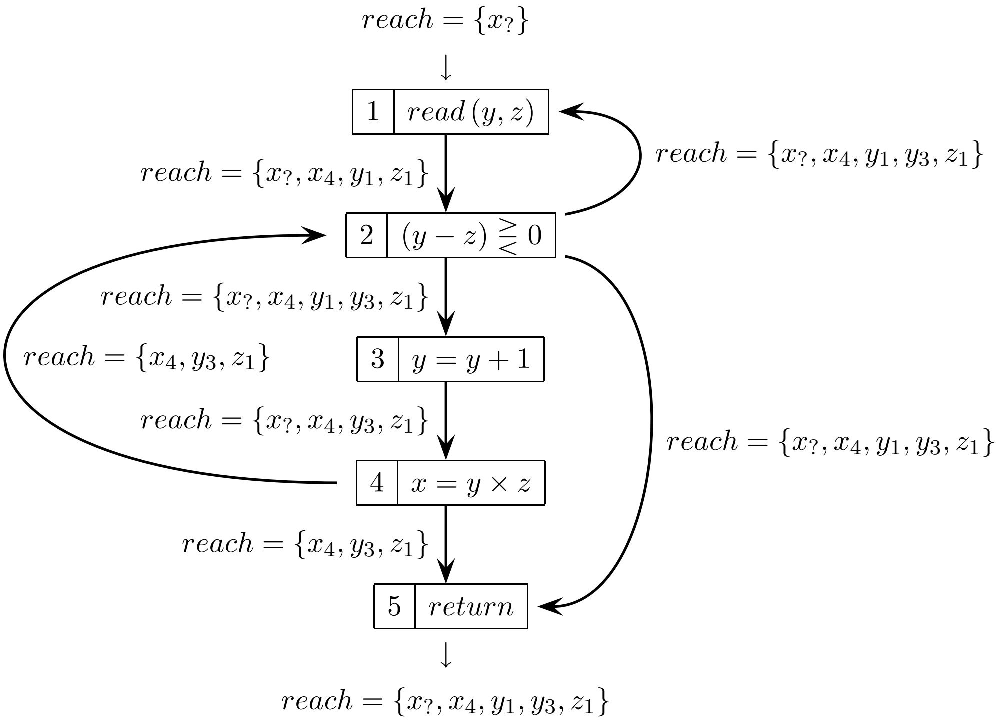
(continues on the next table)

		step 8		
	state 7		**state** 8	
#	*in*	*out*	*in*	*out*
1	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$
2	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$
3	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3 z_1$
4	$x_? x_4 y_3$	$x_4 y_3 z_1$	$x_? x_4 y_3 z_1$	$x_4 y_3 z_1$
5	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$

(continues and finishes on the next table)

		$8 = 9$		
	state 8		**state** 9	
#	*in*	*out*	*in*	*out*
1	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 z_1$
2	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$
3	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_3 z_1$
4	$x_? x_4 y_3 z_1$	$x_4 y_3 z_1$	$x_? x_4 y_3 z_1$	$x_4 y_3 z_1$
5	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$	$x_? x_4 y_1 y_3 z_1$

States 8 and 9 coincide, hence eight steps are enough for the iteration process of the flow equations to converge. The *out* sets in the last step 8 represent the reaching definitions at the outputs of every node of the routine. The following figure shows the control graph of the routine labeled with the sets (renamed *reach* for clarity) of reaching definitions:


$reach = \{x_?\}$
1 $read\,(y, z)$
$reach = \{x_?, x_4, y_1, z_1\}$
$reach = \{x_?, x_4, y_1, y_3, z_1\}$
2 $(y - z) \gtreqless 0$
$reach = \{x_?, x_4, y_1, y_3, z_1\}$
$reach = \{x_4, y_3, z_1\}$
3 $y = y + 1$
$reach = \{x_?, x_4, y_3, z_1\}$
$reach = \{x_?, x_4, y_1, y_3, z_1\}$
4 $x = y \times z$
$reach = \{x_4, y_3, z_1\}$
5 $return$
$reach = \{x_?, x_4, y_1, y_3, z_1\}$

Of course, the definition reaching node 1 from outside is simply $x_?$, as stated in the text of the exercise. The definitions reaching the input of the nodes are simply the union of the definitions on the input arcs of the nodes, and here they are not shown explicitly (read the table before in the state 8). □

Observation

The iterative solution algorithm shown before needs eight steps to converge. It is sometimes possible to speed up the algorithm by observing that some flow equations reduce to identities, which can be exploited to merge two or more steps into one. Moreover, the initial state can be initialised differently. This happens in the present case: the set $in(1)$ can be obviously initialised at $\{x_?\}$ and some identities can be exploited (namely $out(2) = in(2)$, $in(3) = out(2)$, $in(4) = out(3)$ and $in(5) = out(5)$). Here is the accelerated solution:

			step 1		**step** 2	
	state 0		**state** 1		**state** 2	
#	*in*	*out*	*in*	*out*	*in*	*out*
1	$x_?$	$\emptyset$	$x_?$	$x_?\, y_1\, z_1$	$x_?\, y_1\, z_1$	$x_?\, y_1\, z_1$
2	$\emptyset$	$\emptyset$	$x_?\, y_1\, z_1$	$\leftarrow\ x_?\, y_1\, z_1$	$x_?\, x_4\, y_1\, y_3\, z_1$	$\leftarrow\ x_?\, x_4\, y_1\, y_3\, z_1$
3	$\emptyset$	$\emptyset$	$x_?\, y_1\, z_1 \nearrow$	$x_?\, y_1\, z_1$	$x_?\, x_4\, y_1\, y_3\, z_1 \nearrow$	$x_?\, x_4\, y_3\, z_1$
4	$\emptyset$	$\emptyset$	$x_?\, y_3\, z_1 \nearrow$	$x_4\, y_3\, z_1$	$x_?\, x_4\, y_3\, z_1 \nearrow$	$x_4\, y_3\, z_1$
5	$\emptyset$	$\emptyset$	$x_?\, y_1\, y_3\, z_1$	$\leftarrow\ x_?\, y_1\, y_3\, z_1$	$x_?\, x_4\, y_1\, y_3\, z_1$	$\leftarrow\ x_?\, x_4\, y_1\, y_3\, z_1$

The arrows point to the sets that can be simply copied, because they are related by an identity. The number of steps is only three, hence much lower than before, but the algorithm is somewhat more complex to execute. Of course the solution is the same.

Solution of (3)

Since node 3 is entered if and only if condition $y > z$ holds and since statement 3 increments variable y, the program loop 2 3 4 2 ... is endless. As a consequence, passing from node 4 to 5 never happens. Therefore the only really executable computations are the following ones:

$$1\,2\,5 \qquad 1\,2\,1\,2\,5 \qquad (1\,2)^+\,5$$

In conclusion the set of the reaching definitions at the output of node 5 is $\{x_?, y_1, z_1\}$. This means that if the semantic of the routine is taken into consideration, definitions x_4 and y_3 are actually not reaching. □

Bibliography

[1] J. Berstel, *Transductions and Context-Free Languages*, Teubner Studienbücher, 1979 - *textbook on the algebraic approach to the theory of formal languages, mainly dedicated to rational (regular) and algebraic (context-free) languages and relations (transductions)*

[2] S. Crespi Reghizzi, *Linguaggi Formali e Compilazione*, Pitagora Editrice Bologna, Italy, 2006 - *textbook containing the theory material necessary for solving the exercises herewith enclosed, mainly dedicated to regular expressions, grammars, syntax analysis and basics of syntactic transduction, semantic and static flow analysis of programs*

[3] C. Ghezzi, D. Mandrioli, *Theoretical Foundations of Computer Science*, John Wiley & Son, USA, 1987 - *textbook on the theoretical foundations of computer science, dedicated to computation theory and to the basics of complexity theory of algorithms*

[4] J. Hopcroft, J. Ullman, *Formal Languages and their Relation to Automata*, Addison Wesley, USA, 1969 - *classical good textbook on the theory of formal languages, mainly dedicated to regular and context-free languages*

[5] A. Salomaa, *Formal Languages*, Academic Press, USA, 1973 - *classical excellent and very complete textbook on the theory of formal languages, dedicated to a complete presentation of the main properties of type* 0 *(recursively enumerable),* 1 *(linear-bounded space),* 2 *(context-free) and* 3 *(regular) grammars and languages of the Chmosky hierarchy*