FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 7 FEBRUARY 2018 - Part Theory

WITH SOLUTIONS - FOR TEACHING PURPOSES HERE THE SOLUTIONS ARE WIDELY COMMENTED

LAST + FIRST NAME:

(capital letters please)

MATRICOLA: SIGNATURE:

(or PERSON CODE)

INSTRUCTIONS - READ CAREFULLY:

- The exam is in written form and consists of two parts:
  1. Theory (80%): Syntax and Semantics of Languages
     - regular expressions and finite automata
     - free grammars and pushdown automata
     - syntax analysis and parsing methodologies
     - language translation and semantic analysis
  2. Lab (20%): Compiler Design by Flex and Bison
- To pass the exam, the candidate must succeed in both parts (theory and lab), in one call or more calls separately, but within one year (12 months) between the two parts.
- To pass part theory, the candidate must answer the mandatory (not optional) questions; notice that the full grade is achieved by answering the optional questions.
- The exam is open book: textbooks and personal notes are permitted.
- Please write in the free space left and if necessary continue on the back side of the sheet; do not attach new sheets and do not replace the existing ones.
- Time: part lab 60m - part theory 2h.15m

1 Regular Expressions and Finite Automata 20%

1. Consider the following right uni-linear grammar G, over the two-letter terminal alphabet $\Sigma = \{ a, b \}$ and the three-letter nonterminal alphabet $V = \{ S, T, W \}$ (axiom S), which generates a regular language $L = L(G)$:

$$G \begin{cases} S & \rightarrow & a\,T \\ T & \rightarrow & b\,T \mid a\,W \mid a \\ W & \rightarrow & a\,T \mid \varepsilon \end{cases}$$

   Answer the following questions (use the space on the next pages):

   (a) By using a systematic method, transform the grammar G above into a finite-state automaton A_1 (not necessarily deterministic) that accepts language L.

   (b) By using the method of the linear language equations, from grammar G obtain a regular expression R_1 that generates language L.

   (c) By using the *BMC* method (node elimination), obtain a regular expression R_2 from the automaton A_1 of point (a) above. Check that the regular expression R_2 is equivalent to the regular expression R_1 of point (b).

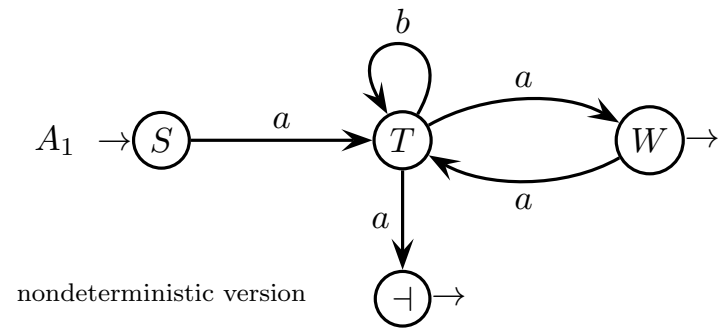
   (d) By using the *BS* method (Berry-Sethi), derive a deterministic finite-state automaton A_2 that accepts language L. Check that the automaton A_2 is equivalent to the automaton A_1 of point (a).

   (e) Design a deterministic automaton A_3 that accepts the complement language $\overline{L}$ of language L.

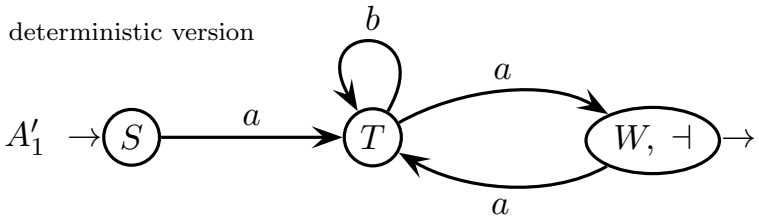
   (f) (optional) Determine whether the language L is local or not, and justify your answer. If the language L is not local, in a systematic way design a finite-state automaton A_4 that accepts the local language that has the same local sets as language L.

Solution

(a) Since grammar G is right-unilinear, though not strictly due to the presence of a terminal non-null rule $T \to a$, an equivalent finite-state automaton A_1 can be immediately obtained by mapping each grammar rule to an automaton transition, as follows:



The correspondence between grammar rules and automaton transitions is one-to-one. The resulting automaton A_1 is not deterministic, anyway (state T has two outgoing a-arcs). A straightforward deterministic version A'_1 is as follows:



The correspondence between rules and transitions is almost one-to-one, with the exception of both rules $T \to a\ W$ and $T \to a$ mapping to the same transition $T \xrightarrow{a} (W,\ \dashv)$, since nonterminal W is nullable, i.e., state W is final. This version A'_1 can be obtained from the previous one A_1 by applying the subset construction, and thus by uniting the states $\dashv$ and W into one group state $(W,\ \dashv)$.

Both versions of automaton A_1 are in the clean form, as grammar G itself is in the reduced form (all the nonterminals are reachable and generating). Furthermore, the deterministic version A'_1 is minimal, since it has two non-final states with outgoing arcs differently labeled, hence these two states are distinguishable from each other, and since it has only one final state, thus obviously distinguishable from the other two.

Notice that grammar G is ambiguous (though limitedly), because for instance the valid string $a\ a \in L(G)$ has two derivations that use different rules:

$$S \xRightarrow{S \to a\ T} a\ T \xRightarrow{T \to a\ W} a\ a\ W \xRightarrow{W \to \varepsilon} a\ a$$

and

$$S \xRightarrow{S \to a\ T} a\ T \xRightarrow{T \to a} a\ a$$

Instead, the deterministic version A'_1 has only one computation that accepts string $a\ a$, namely:

$$S \xrightarrow{a} T \xrightarrow{a} (\, W,\ \dashv\,)$$

which basically unifies the two grammar derivations shown above.

(b) The rules of grammar G can be immediately transformed into a language equation system with three unknown languages L_S, L_T and L_W, as follows:

$$\begin{cases} L_S &= a\ L_T \\ L_T &= b\ L_T\ \cup\ a\ L_W\ \cup\ a \\ L_W &= a\ L_T\ \cup\ \varepsilon \end{cases}$$

Now, by substituting the third equation into the second one:

$$\begin{aligned} L_T &= b\ L_T\ \cup\ a\ \big(\, a\ L_T\ \cup\ \varepsilon\,\big)\ \cup\ a \\ &= b\ L_T\ \cup\ a^2\ L_T\ \cup\ a\ \cup\ a \\ &= \big(\, b\ \cup\ a^2\,\big)\ L_T\ \cup\ a \end{aligned}$$

from where, by the Arden identity:

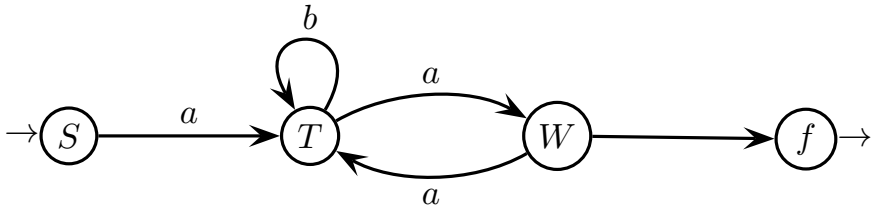
$$L_T = \big(\, b\ \mid\ a^2\,\big)^*\ a$$

and finally, by substituting the above equation into the first one:

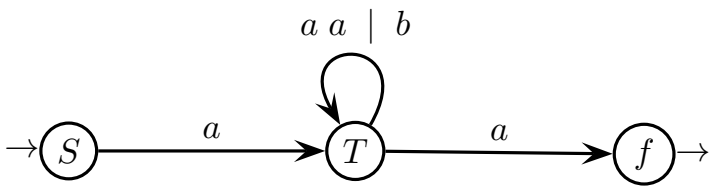
$$L_S = a\ \big(\, b\ \mid\ a^2\,\big)^*\ a = a\ \big(\, a\ a\ \mid\ b\,\big)^*\ a = R_1$$

which provides a regular expression R_1 such that $L\,(R_1) = L\,(S) = L\,(G)$.

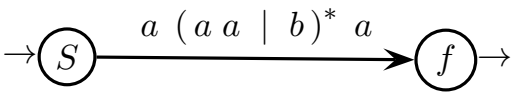
(c) Here is the node elimination method (BMC) applied to the deterministic automaton version A'_1. First, notice that the final state W of A'_1 has outgoing arcs, thus it has to be normalized:



Then, orderly remove the nodes W and T. Eliminate node W:



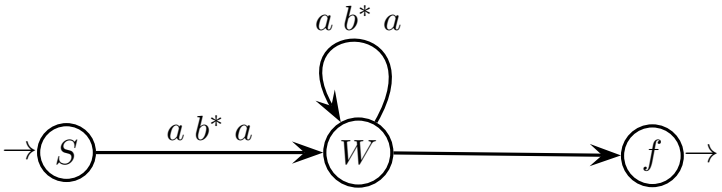
Eliminate node T:



Thus a regular expression R_2 equivalent to automaton A_1 is the following:

$$R_2 = a\ (\ a\ a\ \mid\ b\ )^*\ a$$

It happens that the regular expressions R_1 and R_2 are identical, hence they are surely equivalent, as it is requested to check.
Instead, by first eliminating node T, it would be obtained:



which, by eliminating node W (this step is omitted), immediately leads to the following different formulation R'_2 for the requested regular expression:

$$R'_2 = a\ b^*\ a\ \ (\ a\ b^*\ a\ )^* = (\ a\ b^*\ a\ )^+$$

Of course, these two formulations R_2 and R'_2 are equivalent to each other (and both to R_1), since it is well known that $(\ \alpha\ \mid\ \beta\ )^* = \beta^*\ (\ \alpha\ \beta^*\ )^*$, hence by posing $\alpha = a\,a$ and $\beta = b$ it holds $(\ a\ a\ \mid\ b\ )^* = b^*\ (\ a\ a\ b^*\ )^*$, whence:

$$R_2 = a\ \underbrace{(\ a\ a\ \mid\ b\ )^*}\ a = a\ \underbrace{b^*\ (\ a\ a\ b^*\ )^*}\ a = a\ b^*\ \underbrace{\overbrace{a\ a\ b^*}\ \overbrace{a\ a\ b^*}\ \ldots\ \overbrace{a\ a\ b^*}}_{\text{none, one or more times}}\ a$$

$$= a\ b^*\ a\ \underbrace{\overbrace{a\ b^*\ a}\ \overbrace{a\ b^*\ a}\ \ldots\ \overbrace{a\ b^*\ a}}_{\text{none, one or more times}} = a\ b^*\ a\ \ (\ a\ b^*\ a\ )^* = (\ a\ b^*\ a\ )^+ = R'_2$$

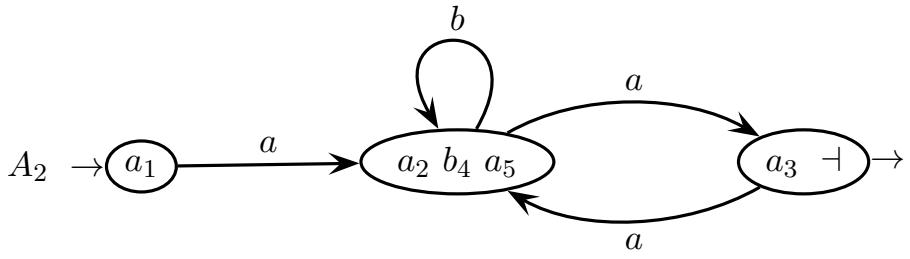
(d) Here is the Berry-Sethi (BS) method applied to the regular expression R_1. First, the numbered version of R_1:

$$R_{1,\#} = a_1\ \left(\ a_2\ a_3\ \mid\ b_4\ \right)^*\ a_5\ \ \dashv$$

Then the table of the initial and follower symbols of expression $R_{1,\#}$:

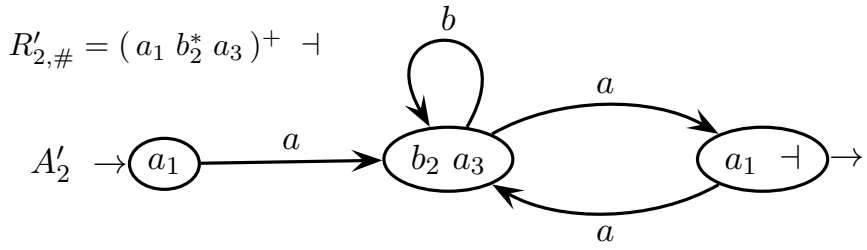
initials	a_1
terminals	**followers**
a_1	$a_2\ b_4\ a_5$
a_2	a_3
a_3	$a_2\ b_4\ a_5$
b_4	$a_2\ b_4\ a_5$
a_5	$\dashv$

Finally, here is the *BS* deterministic finite-state automaton A_2 equivalent to the regular expression R_1 (or to expression R_2):



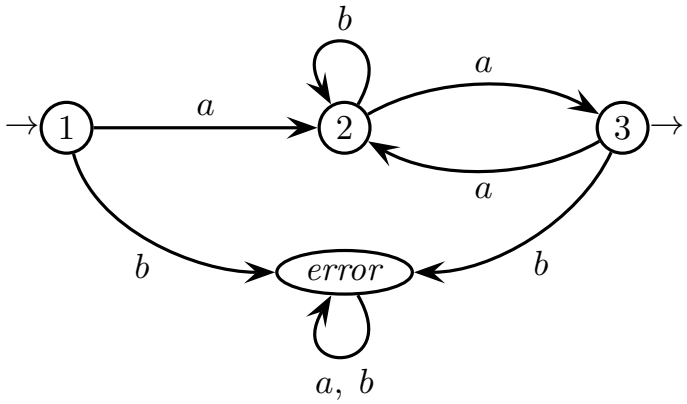
$$R_{1,\#} = R_{2,\#} = a_1 \left(a_2\ a_3 \ \mid\ b_4 \right)^*\ a_5\ \dashv$$

Automaton A_2 is identical to the deterministic version A_1' (up to a change of state names), thus these two automata are equivalent, as it is requested to check. One could apply the *BS* method to the second formulation $R_2' = (\ a\ b^*\ a\ )^+$, which is also equivalent to the regular expression R_1. Here it is applied:

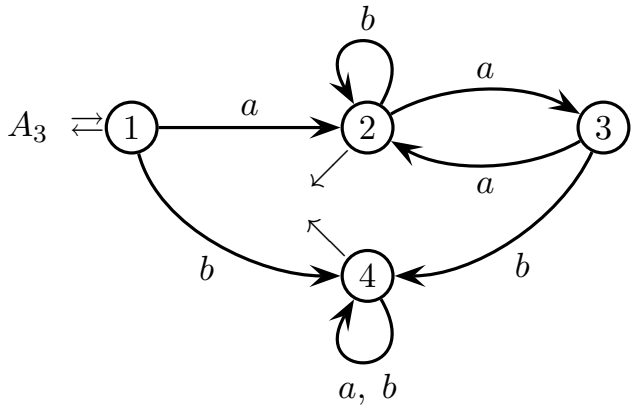


The second version A_2' is equal to the first A_2 (up to a change of state names).

(e) The complement construction applies to a deterministic automaton, which has to be in the naturally complete form, i.e., to have an explicit error state, for the construction to work correctly. Automaton A_1' (or equivalently A_2) recognizes language L and is deterministic, yet it is not in the naturally complete form. Thus automaton A_1' has to be completed (the state names are simplified):

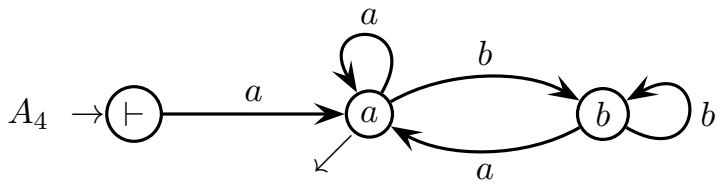


Then the final and non-final nodes are switched (and the error state is renamed), to obtain a complement automaton A_3, which accepts language $\overline{L}$:



Automaton A_3 is deterministic by construction, and it happens to be clean. Furthermore, it happens to be minimal, since the two states 2 and 4 are distinguishable from each other as they have distinguishable a-successors, state 1 is distinguishable from state 2 as they have distinguishable a-successors and from state 4 for the same reason, and the unique final state 3 is obviously standalone.

(f) By inspecting automaton A'_1 (it would be the same with A_1), which recognizes language L, it turns out that the local sets associated to L are $Ini(L) = Fin(L) = \{\ a\ \}$ and $Dig(L) = \{\ a\,a,\ a\,b,\ b\,a,\ b\,b\ \} = \Sigma^2$. Now, string $a\,a\,a$ is compliant with such local sets, yet it holds $a\,a\,a \notin L$, therefore language L is not local.
An automaton A_4 that accepts the local language with such local sets is:



Automaton A_4 is obtained by the known local construction (see the textbook). It is deterministic by construction, and it happens to be clean and even minimal, as it is easy to verify. As expected from what said before, it is structurally different from the deterministic version A'_1, hence surely it is not equivalent to A'_1. In fact, both automata A_4 and A'_1 are minimal, and it is well known that minimal automata, being unique, are equivalent if and only if they are identical.
More generally, the local automaton A_4 accepts the (obviously local) language generated by the following regular expression:

$$L\,(A_4) = a\ (\ a\ \mid\ b^+\ a\ )^*$$

which can be easily obtained by node elimination (for instance eliminate node b and see the result). Through some algebraic manipulation, it is soon obtained:

$$L\,(A_4) = a\ (\ a\ \mid\ b^+\ a\ )^* = a\ (\ (\ \varepsilon\ \mid\ b^+\ )\ a\ )^* = a\ (\ b^*\ a\ )^*$$

$$= a\ \mid\ a\ (\ b^*\ a\ )^+ = a\ \mid\ a\ \overbrace{\underbrace{(\ b^*\ a\ )^*\ b^*}_{\text{equiv. to } \Sigma^*}\ a}^{\text{unroll } (\ b^*\ a\ )^+} = a\ \mid\ a\ \Sigma^*\ a$$

Clearly the relationship between languages L and $L\,(A_4)$ is a strict containment, i.e., $L \subset L\,(A_4)$. Containment is proved by comparing the components $a\ (\ a\ a\ \mid\ b\ )^*\ a$ and $a\ \Sigma^*\ a = a\ (\ a\ \mid\ b\ )^*\ a$, and strictness trivially by string a, since it holds $a \notin L$ and $a \in L\,(A_4)$ (more generally it holds $a^{2n+1} \notin L$ and $a^{2n+1} \in L\,(A_4)$ for any $n \geq 0$). This proves again that language L is not local.

2 Free Grammars and Pushdown Automata 20%

1. Consider the context-free language L below, over the two-letter alphabet $\Sigma = \{\ a,\ b\ \}$:

$$L = \left\{\ a^h\ b^k \mid \quad h > k \geq 1\ \wedge\ h \text{ is even}\ \wedge\ k \text{ is odd}\ \right\}$$

Answer the following questions:

(a) Write the four shortest strings of language L. Write a *BNF* grammar G, no matter if ambiguous, that generates language L. Draw a syntax tree for the sample valid string:

$$a^6\ b^3$$

(b) Argue whether the grammar G written at point (a) is ambiguous or not. In the case that grammar G is ambiguous, write an equivalent non-ambiguous *BNF* grammar G' and justify it.

(c) (optional) Consider the complement language $\overline{L}$ of language L, with respect to the alphabet Σ:

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

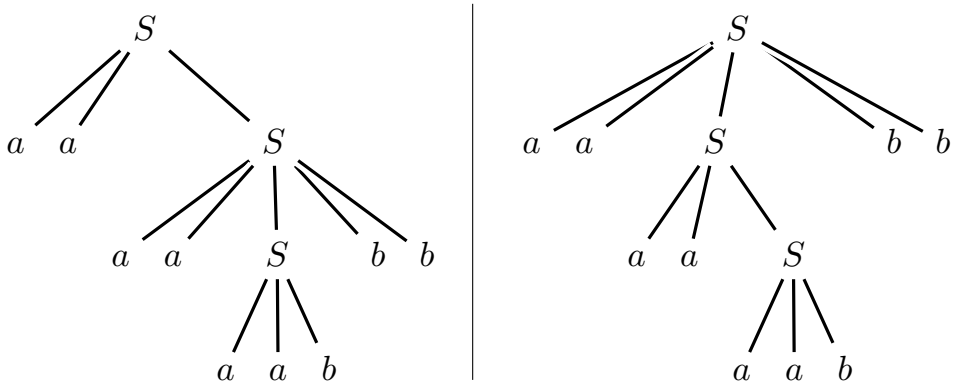
Is the complement language $\overline{L}$ context-free or not ? Explain your answer.

Solution

(a) Here are the four shortest strings of language L: $a^2\ b$, $a^4\ b$, $a^4\ b^3$ and $a^6\ b$. It is clear that language L is not regular, as it has a self-embedded structure.
Here is a possible *BNF* grammar G (axiom S), which is ambiguous:

$$G \begin{cases} 1: & S & \rightarrow & a\ a\ S \\ 2: & S & \rightarrow & a\ a\ S\ b\ b \\ 3: & S & \rightarrow & a\ a\ b \end{cases}$$

The sample string $a^6\ b^3$ has two syntax trees and is ambiguous. Here they are:



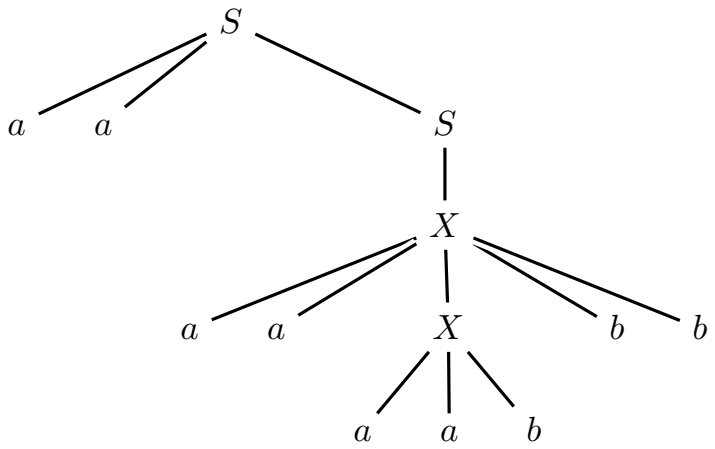
The application order of the rules 1 and 2 in a derivation is not syntactically fixed, yet commuting them does not change the generated string, and consequently grammar G is ambiguous. Clearly, all the valid strings of type $a^h\ b^k$ with $h \geq 6$ and $k \geq 3$ are ambiguous. Furthermore, their ambiguity degree grows with their length, though it remains finite for each of them. Therefore grammar G has an unlimited ambiguity degree. However, grammar G also generates infinitely many non-ambiguous strings, namely all those of type $(\,a\,a\,)^+\,b$, which form a regular subset. In fact, the derivations of such strings never use rule 2, while the rules 1 and 3 alone constitute a right-linear sub-grammar, hence regular. There may be different solutions, of course, more or less ambiguous.

(b) Grammar G is ambiguous, as the sample string has two syntax trees. Here is a possible non-ambiguous *BNF* version G' (axiom S), equivalent to G:

$$G' \begin{cases} 1: & S & \rightarrow & a\ a\ S \\ 2: & S & \rightarrow & X \\ 3: & X & \rightarrow & a\ a\ X\ b\ b \\ 4: & X & \rightarrow & a\ a\ b \end{cases}$$

Now the application order of the rules 1 and 3 in a derivation is fixed: first rule 1 then rule 3. Therefore every valid string has exactly one derivation and tree.

For instance, with grammar G' the syntax tree of the sample string $a^6\ b^3$ is:



and it is the only possible syntax tree for the sample string. In fact, it is structurally similar to the left tree of grammar G drawn above, whereas the right one is here made impossible due to the change of nonterminal names.
An alternative solution is the following grammar G'' (axiom S_E), with six rules:

$$G'' \begin{cases} 1: & S_E & \to & a\ S_O \\ 2: & S_E & \to & X_E \\ 3: & S_O & \to & a\ S_E \\ 4: & X_E & \to & a\ X_O\ b \\ 5: & X_O & \to & a\ X_E\ b \\ 6: & X_O & \to & a \end{cases}$$

Grammar G'' generates language L and is non-ambiguous. With respect to grammar G', it has shorter rules but more nonterminals, which are used to manage the parity counting of terminals. Of course, there may be more solutions.

(c) The complement language $\overline{L}$ of language L can be easily decomposed as follows:

$$\overline{L} = \underbrace{a^*\ b^+\ a\ \Sigma^*}_{\substack{\text{digram } b\,a \\ \text{inadmissible} \\ \text{in language } L}} \cup \underbrace{\left\{\ a^h\ b^k \mid \quad 0 \le h \le k\ \right\}}_{\substack{\text{too few letters } a \text{ with} \\ \text{respect to the letters } b}} \cup \underbrace{L_3\ \cup\ L_4\ \cup\ L_5}_{\substack{\text{invalid parity} \\ \text{combinations}}}$$

where

$$\begin{aligned} L_3 &= \left\{\ a^h\ b^k \mid \quad h > k \ge 0\ \wedge\ \text{both } h \text{ and } k \text{ are even}\ \right\} \\ L_4 &= \left\{\ a^h\ b^k \mid \quad h > k \ge 0\ \wedge\ h \text{ is odd}\ \wedge\ k \text{ is even}\ \right\} \\ L_5 &= \left\{\ a^h\ b^k \mid \quad h > k \ge 1\ \wedge\ \text{both } h \text{ and } k \text{ are odd}\ \right\} \end{aligned}$$

The first two components express clear cases of exclusion from language L. The remaining components L_3, L_4 and L_5 are defined similarly to L, but with the exponents h and k that have invalid parity combinations of type even-even, odd-even and odd-odd. Overall, the five components express all the cases for a string to be excluded from language L. These five subsets are disjoint from each other.

The first component is regular, the second is well-known to be context-free (but non-regular), the remaining three are quite obvious variants of language L and clearly they are context-free as well (it would be easy to provide their grammars). Since the complement language $\overline{L}$ is the union of context-free components, it is context-free as well. A union grammar could be provided for $\overline{L}$, with a few rules.

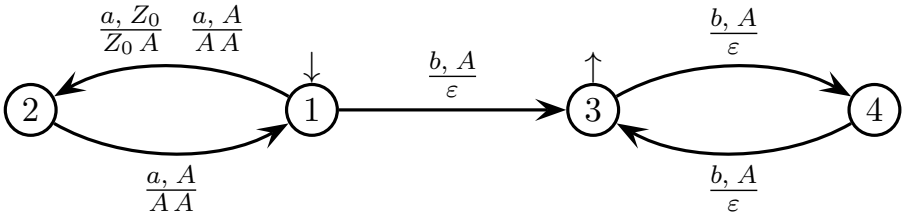
An even simpler though ambiguous decomposition of language $\overline{L}$ is as follows:

$$\overline{L} = \underbrace{\Sigma^* \; b \; a \; \Sigma^*}_{\substack{\text{as before} \\ \text{(ambiguous)}}} \cup \underbrace{\left\{ \; a^h \; b^k \mid \quad 0 \leq h \leq k \; \right\}}_{\text{as before}} \cup \underbrace{\underbrace{a^+}_{\text{any}} \; \underbrace{(\, b \; b \,)^*}_{\text{even}} \cup \underbrace{a \; (\, a \; a \,)^*}_{\text{odd}} \; \underbrace{b^*}_{\text{any}}}_{\text{invalid parity}}$$

where the two last components are regular languages. Notice that the first subset is presented ambiguously (the RE itself is ambiguous), and that the second, third and fourth subset are not disjoint from each other, differently from the previous solution where all the subsets are disjoint. This decomposition shows once again the well-known fact that an ambiguous solution may often be (descriptionally) simpler than a non-ambiguous one. The conclusion however, is the same as before, namely that the complement language $\overline{L}$ of language L is context-free.

An observation: in general the family of the context-free languages is not closed with respect to complement. Some particular context-free languages however, may have a complement language that is context-free as well (for sure all the regular languages do). This is what happens to the context-free yet non-regular language L. Of course, the complement language $\overline{L}$ is not regular either.

A different approach consists of determining whether the language L is deterministic context-free. By definition, a language is deterministic context-free if it is accepted by a deterministic pushdown automaton (*detPDA*). Here is a possible *detPDA* A that accepts language L:



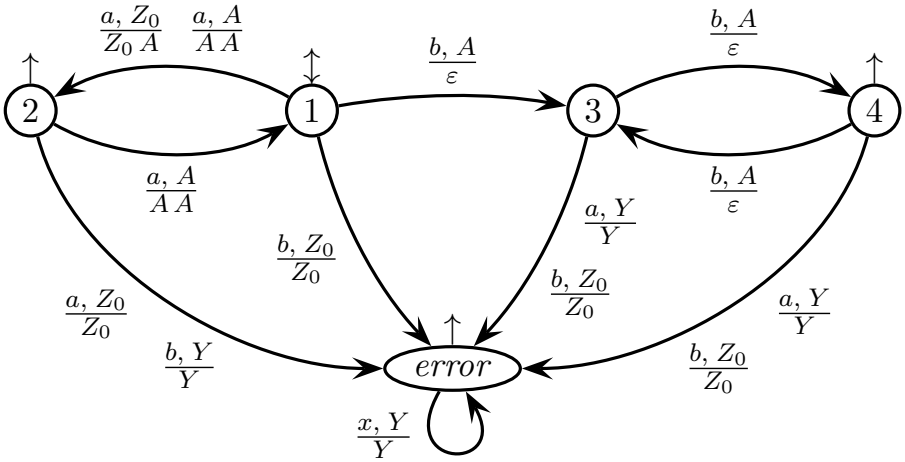
Automaton A recognizes by empty tape and final state (only state 3 is final). At recognition (input finished in the state 3), the stack it not necessarily empty. Automaton A is simple and clearly deterministic. Anyway, notice that it is not a syntax analyzer, as it is unrelated to any specific grammar of language L.

The reader may wish to test automaton A for correctness. For instance, take the previous four shortest strings of L and verify that automaton A accepts them.

Since automaton A is a *detPDA*, language L is deterministic context-free. Now, the family of deterministic context-free languages is closed with respect to set complement. Therefore, the complement language $\overline{L}$ of language L is surely context-free, and actually it is deterministic as well. This answers the question.

How to build the complement *detPDA* that accepts the complement language $\overline{L}$, is a quite obvious generalization of the known complement construction for deterministic finite-state automata. First put the *detPDA* into a complete form by adding an error state, whereto all the missing transitions go, and wherein

the computations stay trapped without changing the stack contents any more. Second switch the final and non-final states. For the interested reader, here is the complement *detPDA* $\overline{A}$ of the *detPDA* A shown above:



Some transition labels of $\overline{A}$ are grouped by assuming $x = a$, b and $Y = Z_0$, A. Thus a label like $\frac{x, Y}{Y}$ (on the self-loop) stands for four labels $\frac{a, Z_0}{Z_0}$, $\frac{a, A}{A}$, $\frac{b, Z_0}{Z_0}$ and $\frac{b, A}{A}$, which correspond to four moves (self-loops) of $\overline{A}$. Notice that the final and non-final states of $\overline{A}$ are switched with respect to those of A, and that now state 1 is both initial and final. Of course, the error state is final for $\overline{A}$.

Some transitions may be useless as they are never enabled by the stack top symbol. For instance, transition $2 \xrightarrow{\frac{a, Z_0}{Z_0}} error$ never fires, as in the state 2 the stack cannot be empty. Such transitions are included for uniformity, that is, for each state to have exactly four outgoing transitions; yet they might be canceled.

The reader may wish to test the *detPDA* $\overline{A}$ and verify that it accepts the complement language $\overline{L}$. For instance, take a sample string for each one of the five components of the previous first decomposition of $\overline{L}$ and verify that each sample is accepted by $\overline{A}$. On the contrary, take the previous four shortest strings of language L and verify that each of them is rejected by $\overline{A}$.

One more (last) approach consists of determining whether the non-ambiguous grammar G' (or G'') of language L is itself deterministic, of type *ELR* or even *ELL*. In fact, if grammar G' were deterministic, then language L would have a deterministic syntax analyzer, i.e., a *detPDA*, thus the complement language $\overline{L}$ would be context-free and even itself deterministic. This is an alternative to directly tackling the construction of the *detPDA* A, as done before instead.

One soon sees that grammar G' (or G'') is not of type *ELL*, or *LL* as it is *BNF*. In fact, the guide sets of the alternative rules 1 and 2 are not disjoint, for any look-ahead window size $k \geq 1$. This would not change with an *ELL* analysis either. The reason is that language L is of type $a^h\, b^k$ with $h \geq k$, plus some parity restrictions on the exponents, which is a language kind known to be non-*ELL*.

However, the language type $a^h\, b^k$ with $h \geq k$ is instead known to be *ELR*, thus chances are that grammar G' (or G'') is of type *ELR* as well. This last verification is left to the reader: first put grammar G' into the form of a machine net, second draw the pilot and third verify the *ELR* condition. A deterministic result is not granted, but it may be worthy trying to do some practice.

2. Consider the fragment of a programming language that includes statements of these four kinds: *assignment*, *iterative*, *conditional* and *branch*. This fragmentary language is informally specified as follows:

   - A phrase of the language is a non-empty list of statements as said above, each of them followed by a semicolon ";".
   - An *assignment* statement consists of a variable name, an assignment operator "=" and an expression.
   - In the entire grammar, all the variables and expressions (or conditions) must be represented by the terminals var and exp, respectively, without expansion.
   - There are three types of *iterative* statement. The first (and simplest) type consists of a non-empty statement list enclosed by the keywords do and end. The second type adds to the first one an initial clause with a keyword while followed by an iteration condition. The third type adds to the first one a final clause with a keyword when followed by a termination condition. The clauses while and when cannot be simultaneously present in the same iterative statement.
   - There are two types of *conditional* statement. The first type is the customary two-way conditional if-then-else-endif, where both ways then and else contain a non-empty statement list, and the else way is optional. The second type is a non-empty linear list of clauses if-do, each of them containing only one statement, and the whole list is enclosed by the two keywords cond and endcond.
   - There are two types of *branch* statement, which just consist of the keywords terminate and exit. The statements terminate and exit may only occur inside an iterative statement and a conditional statement, respectively.
   - All the statement types described above can be freely nested, except those constrained by the previous item.

   For any unspecified minor detail, see the example below, and you may also follow the usual conventions, e.g., those of the C language. Example:

```
var = exp;
do
   cond
      if exp do exit;
      if exp do
         while exp do
            if exp then terminate; else var = exp; endif;
         end;
   endcond;
   do
      var = exp;
      if exp then exit; endif;
   end when exp;
end;
```

   Write a non-ambiguous grammar, in general of type *EBNF*, that models the fragmentary language described above.

Solution

Here is a possible grammar G_1, of type *EBNF* (axiom LANG):

$$
G_1 \left\{
\begin{array}{rcl}
\langle \mathsf{LANG} \rangle & \rightarrow & \left(\langle \mathsf{STRU} \rangle \ \text{`;'} \right)^+ \\
\langle \mathsf{STRU} \rangle & \rightarrow & \langle \mathsf{ASGN} \rangle \mid \langle \mathsf{ITER} \rangle \mid \langle \mathsf{COND} \rangle \\
\hline
\langle \mathsf{ASGN} \rangle & \rightarrow & \mathsf{var} \ \text{`='} \ \mathsf{exp} \\
\hline
\langle \mathsf{ITER} \rangle & \rightarrow & \langle \mathsf{LOOP} \rangle \mid \mathsf{while\ exp}\ \langle \mathsf{LOOP} \rangle \mid \langle \mathsf{LOOP} \rangle\ \mathsf{when\ exp} \\
\langle \mathsf{LOOP} \rangle & \rightarrow & \mathsf{do} \left(\left(\langle \mathsf{STRU} \rangle \mid \mathsf{terminate} \right) \text{`;'} \right)^+ \mathsf{end} \\
\hline
\langle \mathsf{COND} \rangle & \rightarrow & \langle \mathsf{IF\_2W} \rangle \mid \langle \mathsf{IF\_DO} \rangle \\
\langle \mathsf{IF\_2W} \rangle & \rightarrow & \mathsf{if\ exp\ then} \left(\left(\langle \mathsf{STRU} \rangle \mid \mathsf{exit} \right) \text{`;'} \right)^+ \\
 & & \left[\mathsf{else} \left(\left(\langle \mathsf{STRU} \rangle \mid \mathsf{exit} \right) \text{`;'} \right)^+ \right] \\
 & & \mathsf{endif} \\
\langle \mathsf{IF\_DO} \rangle & \rightarrow & \mathsf{cond} \left(\mathsf{if\ exp\ do} \left(\langle \mathsf{STRU} \rangle \mid \mathsf{exit} \right) \text{`;'} \right)^+ \mathsf{endcond}
\end{array}
\right.
$$

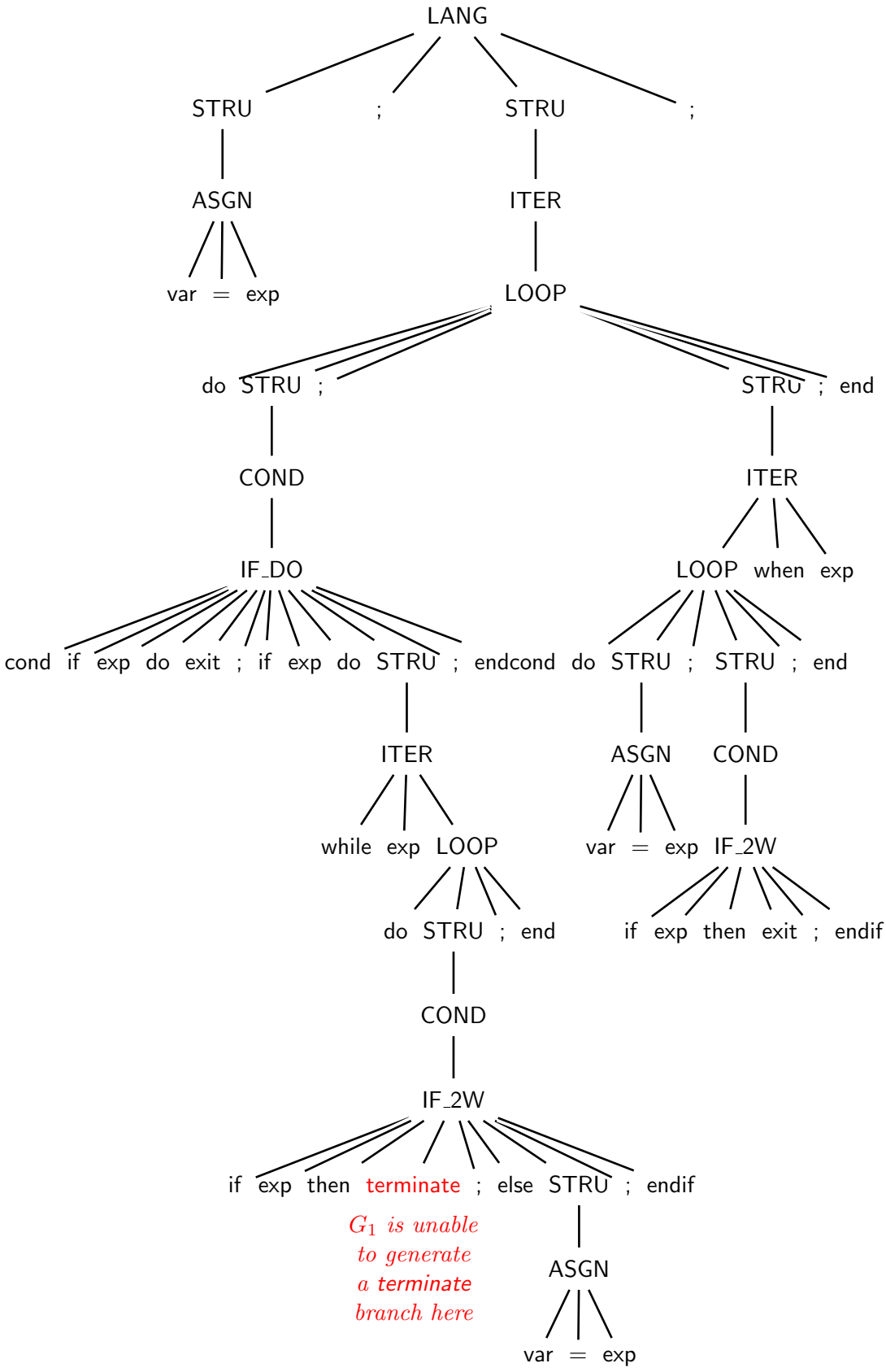
Grammar G_1 is not ambiguous and as a matter of fact it is structurally quite simple. There may be equivalent variants, of course. A simple one consists of introducing a syntactic class ⟨STRU_E⟩, i.e., a nonterminal, to generate the frequently used alternative ⟨STRU⟩ | exit. The last rules of G_1 become even more compact:

$$
\begin{array}{rcl}
\langle \mathsf{IF\_2W} \rangle & \rightarrow & \mathsf{if\ exp\ then} \left(\langle \mathsf{STRU\_E} \rangle \ \text{`;'} \right)^+ \left[ \mathsf{else} \left( \langle \mathsf{STRU\_E} \rangle \ \text{`;'} \right)^+ \right] \mathsf{endif} \\
\langle \mathsf{IF\_DO} \rangle & \rightarrow & \mathsf{cond} \left(\mathsf{if\ exp\ do}\ \langle \mathsf{STRU\_E} \rangle \ \text{`;'} \right)^+ \mathsf{endcond} \\
\langle \mathsf{STRU\_E} \rangle & \rightarrow & \langle \mathsf{STRU} \rangle \mid \mathsf{exit}
\end{array}
$$

Grammar G_1 works properly under the hypothesis that a statement terminate or exit is allowed only if it is immediately contained in a statement of type iterative or conditional, respectively. This is a reasonable though restrictive interpretation.

Anyway, the example is not fully compliant with such an interpretation: see the iterative statement while-do, which contains a conditional statement if-then-else-endif, which contains a terminate statement. Clearly grammar G_1 is unable to generate a terminate in that position, not immediately contained in an iterative statement.

See also the tree on the next page: grammar G_1 generates almost the entire sample, but the branch statement terminate in red, as such a statement immediately occurs inside a conditional one, where only the other branch exit could be generated by G_1.


LANG
STRU
;
STRU
;
ASGN
ITER
var = exp
LOOP
do STRU ;
STRU ; end
COND
ITER
IF_DO
LOOP when exp
cond if exp do exit ; if exp do STRU ; endcond
do STRU ; STRU ; end
ITER
ASGN
COND
while exp LOOP
var = exp IF_2W
do STRU ; end
if exp then exit ; endif
COND
IF_2W
if exp then terminate ; else STRU ; endif
G_1 is unable to generate a terminate branch here
ASGN
var = exp

In a permissive interpretation however, the containment is not necessarily immediate and may instead also occur at an arbitrary nesting depth. In this case, a new grammar G_2 should be designed. A possible grammar G_2 is shown below (axiom LANG) and is designed with the purpose of reusing grammar G_1 as much as possible:

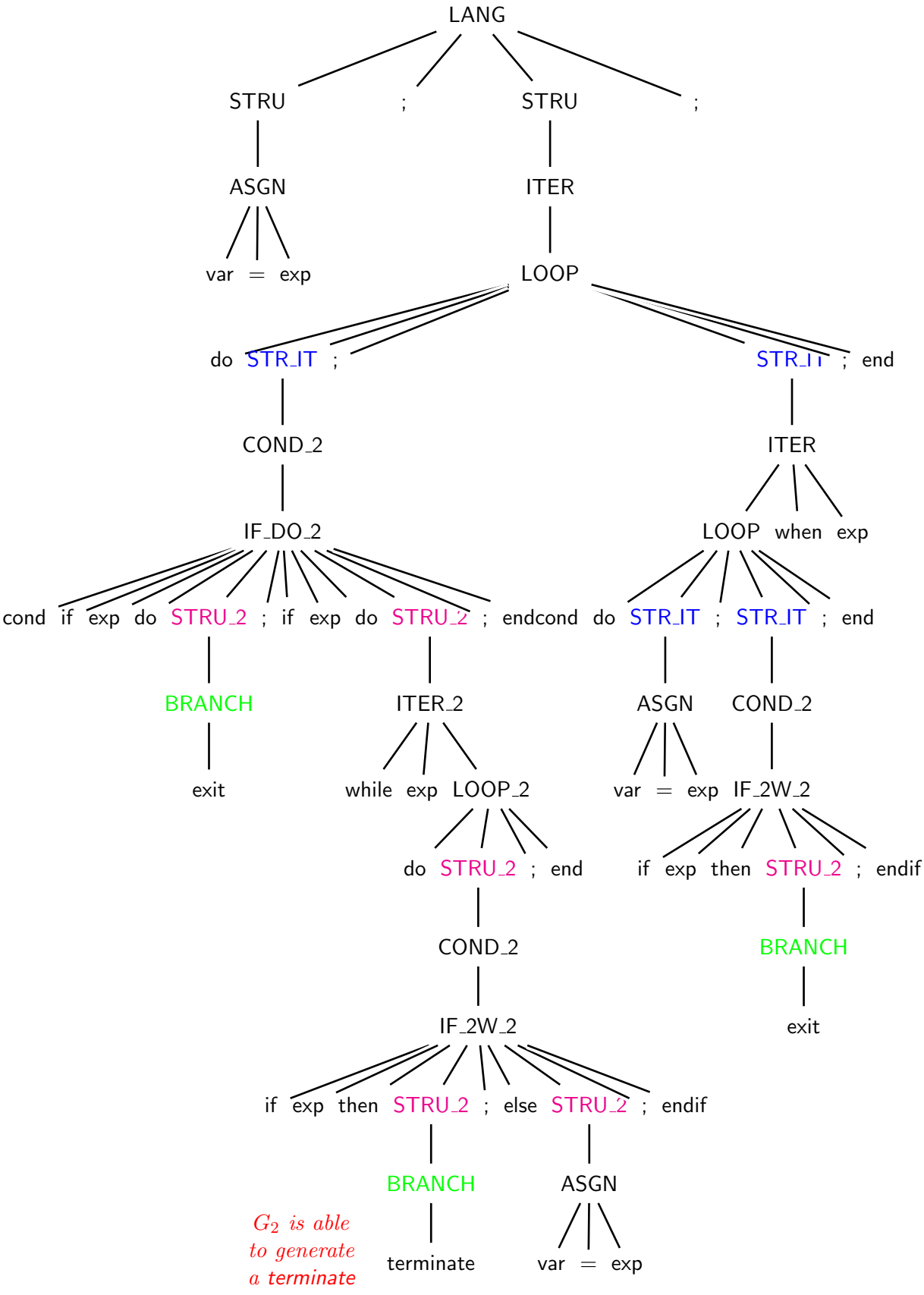
G_2

$$
\begin{array}{rcl}
\langle \mathsf{LANG} \rangle & \rightarrow & \left(\langle \mathsf{STRU} \rangle \text{ ';'} \right)^{+} \\
\langle \mathsf{STRU} \rangle & \rightarrow & \langle \mathsf{ASGN} \rangle \mid \langle \mathsf{ITER} \rangle \mid \langle \mathsf{COND} \rangle \\
\hline
\langle \mathsf{ASGN} \rangle & \rightarrow & \mathsf{var} \text{ '='} \mathsf{exp} \\
\hline
\langle \mathsf{ITER} \rangle & \rightarrow & \langle \mathsf{LOOP} \rangle \mid \mathsf{while\ exp}\ \langle \mathsf{LOOP} \rangle \mid \langle \mathsf{LOOP} \rangle\ \mathsf{when\ exp} \\
\langle \mathsf{LOOP} \rangle & \rightarrow & \mathsf{do} \left(\left(\langle \mathsf{STR\_IT} \rangle \mid \mathsf{terminate} \right) \text{ ';'} \right)^{+} \mathsf{end} \\
\hline
\langle \mathsf{COND} \rangle & \rightarrow & \langle \mathsf{IF\_2W} \rangle \mid \langle \mathsf{IF\_DO} \rangle \\
\langle \mathsf{IF\_2W} \rangle & \rightarrow & \mathsf{if\ exp\ then} \left(\left(\langle \mathsf{STR\_CN} \rangle \mid \mathsf{exit} \right) \text{ ';'} \right)^{+} \\
 & & \left[\mathsf{else} \left(\left(\langle \mathsf{STR\_CN} \rangle \mid \mathsf{exit} \right) \text{ ';'} \right)^{+} \right] \\
 & & \mathsf{endif} \\
\langle \mathsf{IF\_DO} \rangle & \rightarrow & \mathsf{cond} \left(\mathsf{if\ exp\ do} \left(\langle \mathsf{STR\_CN} \rangle \mid \mathsf{exit} \right) \text{ ';'} \right)^{+} \mathsf{endcond}
\end{array}
$$

$$
\begin{array}{rcl}
\langle \mathsf{STR\_IT} \rangle & \rightarrow & \langle \mathsf{ASGN} \rangle \mid \langle \mathsf{ITER} \rangle \mid \langle \mathsf{COND\_2} \rangle \\
\langle \mathsf{STR\_CN} \rangle & \rightarrow & \langle \mathsf{ASGN} \rangle \mid \langle \mathsf{ITER\_2} \rangle \mid \langle \mathsf{COND} \rangle \\
\hline
\langle \mathsf{ITER\_2} \rangle & \rightarrow & \langle \mathsf{LOOP\_2} \rangle \mid \mathsf{while\ exp}\ \langle \mathsf{LOOP\_2} \rangle \mid \langle \mathsf{LOOP\_2} \rangle\ \mathsf{when\ exp} \\
\langle \mathsf{LOOP\_2} \rangle & \rightarrow & \mathsf{do} \left(\langle \mathsf{STRU\_2} \rangle \text{ ';'} \right)^{+} \mathsf{end} \\
\hline
\langle \mathsf{COND\_2} \rangle & \rightarrow & \langle \mathsf{IF\_2W\_2} \rangle \mid \langle \mathsf{IF\_DO\_2} \rangle \\
\langle \mathsf{IF\_2W\_2} \rangle & \rightarrow & \mathsf{if\ exp\ then} \left(\langle \mathsf{STRU\_2} \rangle \text{ ';'} \right)^{+} \\
 & & \left[\mathsf{else} \left(\langle \mathsf{STRU\_2} \rangle \text{ ';'} \right)^{+} \right] \\
 & & \mathsf{endif} \\
\langle \mathsf{IF\_DO\_2} \rangle & \rightarrow & \mathsf{cond} \left(\mathsf{if\ exp\ do}\ \langle \mathsf{STRU\_2} \rangle \text{ ';'} \right)^{+} \mathsf{endcond} \\
\hline
\langle \mathsf{STRU\_2} \rangle & \rightarrow & \langle \mathsf{ASGN} \rangle \mid \langle \mathsf{ITER\_2} \rangle \mid \langle \mathsf{COND\_2} \rangle \mid \langle \mathsf{BRANCH} \rangle \\
\langle \mathsf{BRANCH} \rangle & \rightarrow & \mathsf{terminate} \mid \mathsf{exit}
\end{array}
$$

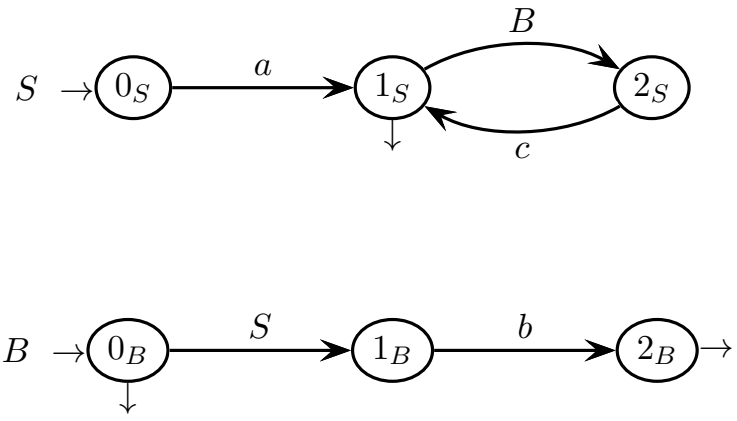
The top sub-grammar structurally coincides with grammar G_1, except that now the syntactic class STRU is not recursive and splits into the two sub-classes STR_IT and STR_CN. Such sub-classes model the occurrence of an iterative or a conditional statement, respectively, and link the top sub-grammar to the bottom additional one. Here the syntactic classes ITER2, COND2 (along with their sub-classes) and STRU2 model the occurrence of both statement types, i.e., nested iterative and conditional. Finally, a class BRANCH models the permission to use both statements terminate and exit. Grammar G_2 is able to generate the sample phrase (see the tree on the next page).

Grammars G_1 and G_2 have a different precision in modeling a solution, and both are acceptable. As before, there may be equivalent variants, of course.


LANG
STRU
;
STRU
;
ASGN
ITER
var = exp
LOOP
do STR_IT ;
STR_IT ; end
COND_2
ITER
IF_DO_2
LOOP when exp
cond if exp do STRU_2 ; if exp do STRU_2 ; endcond
do STR_IT ; STR_IT ; end
BRANCH
ITER_2
ASGN
COND_2
exit
while exp LOOP_2
var = exp
IF_2W_2
do STRU_2 ; end
if exp then STRU_2 ; endif
COND_2
BRANCH
IF_2W_2
exit
if exp then STRU_2 ; else STRU_2 ; endif
BRANCH
ASGN
terminate
var = exp
G_2 is able to generate a terminate branch here

3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the following grammar G, represented as a machine net over the three-letter terminal alphabet $\Sigma = \{ a, b, c \}$ and the two-letter nonterminal alphabet $V = \{ S, B \}$ (axiom S):



Answer the following questions (use the figures / tables / spaces on the next pages):

(a) Draw the complete pilot of grammar G and prove that grammar G is of type $ELR(1)$. Explain your answer.

(b) Write all the guide sets on the shift arcs, call arcs and exit arrows of the net of grammar G, and say if grammar G is of type $ELL(1)$. Explain your answer.

(c) Nonterminal B is nullable. Modify the machines of nonterminals S and B, but without changing the number of states of each machine and the generated language $L(G)$, so that nonterminal B becomes non-nullable. Is the resulting net still of type $ELR(1)$? Explain your answer.

(d) (optional) Simulate the bottom-up analysis of the following valid sample string:

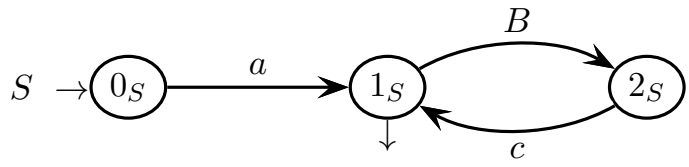
$$a\ c\ a\ b\ c \in L(G)$$

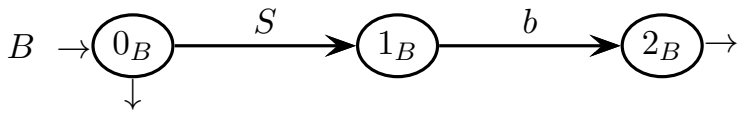
Show the piecewise construction of the syntax tree of the string in the analysis.

---

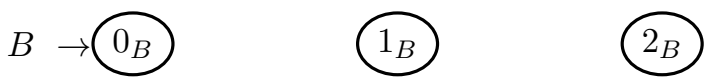
please draw here the complete pilot – question (a)

please add here the call arcs and all the guide sets – question (b)





please answer here – question (c)



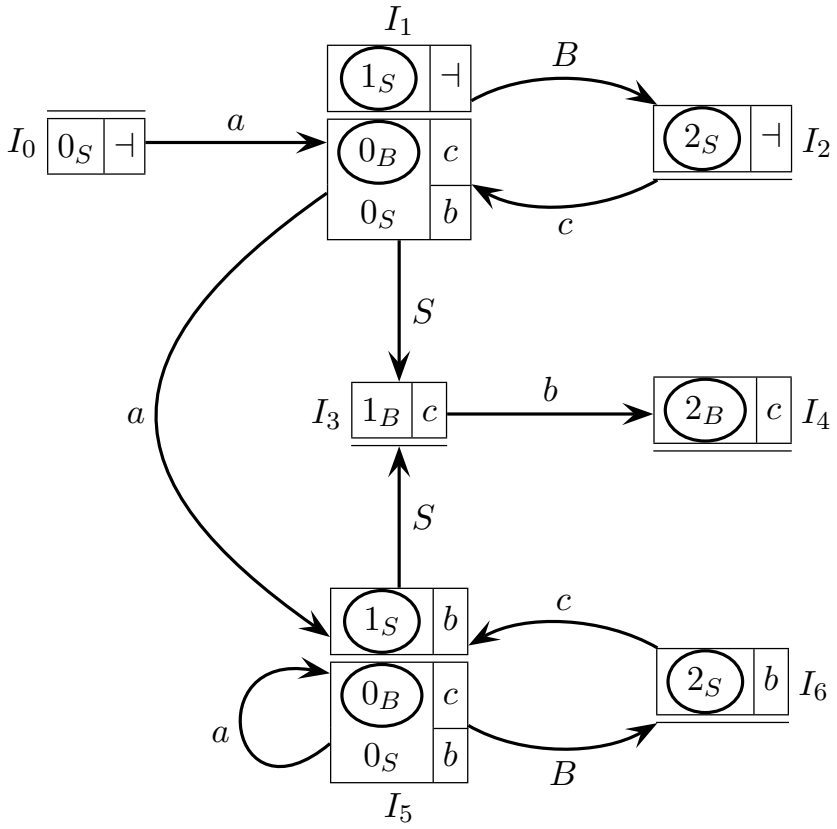
simulation table for the *ELR* analysis to be completed - question (d) - (the number of rows is not significant)

stack of alternated macro-states and grammar symbols	*input left to be scanned*	*move executed*
I_0	$a\ c\ a\ b\ c\ \dashv$	initial configuration
$I_0\ a\ I_{...}$	$c\ a\ b\ c\ \dashv$	terminal shift: $I_0 \xrightarrow{a} I_{...}$

draw here the piecewise tree construction

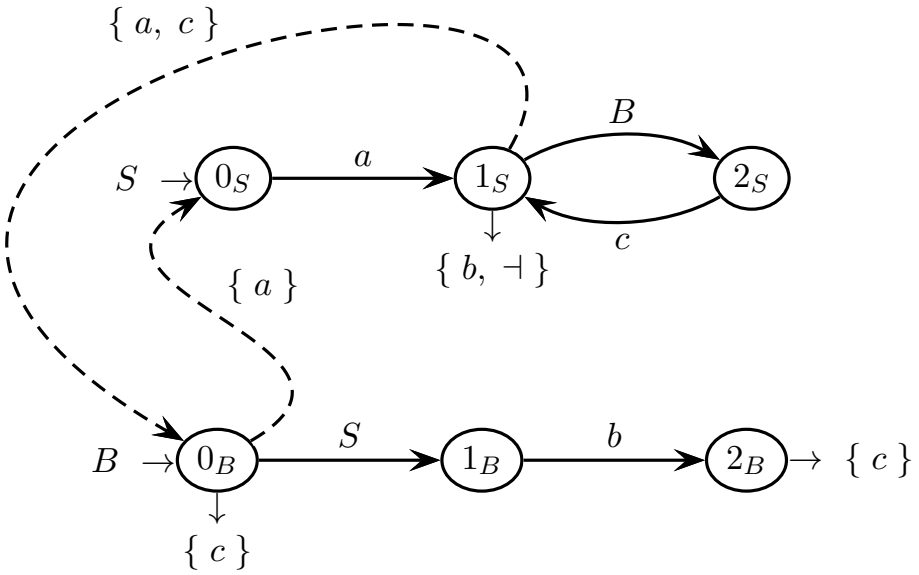
Solution

(a) Here is the complete pilot, with seven m-states:



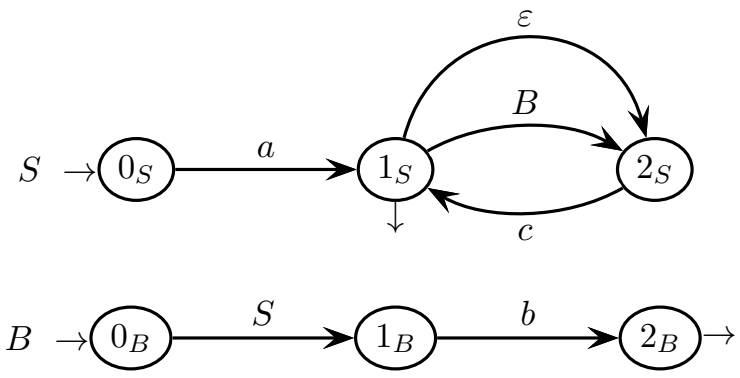
The pilot does not have any conflicts, thus grammar G is of type $ELR(1)$.

(b) Here are the call arcs and all the guide sets:

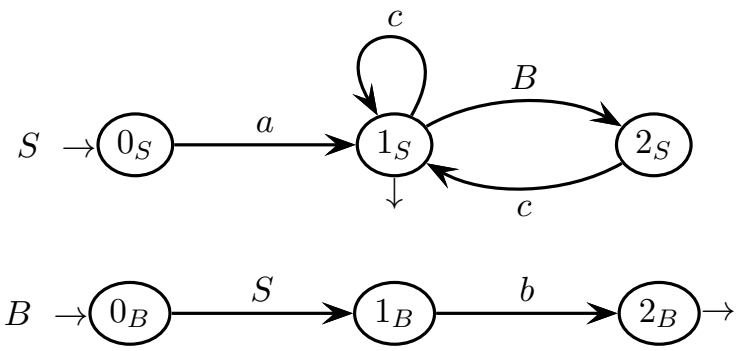


All the guide sets on the bifurcations states (1_S and 0_B) are disjoint, thus grammar G is of type $ELL\,(1)$.

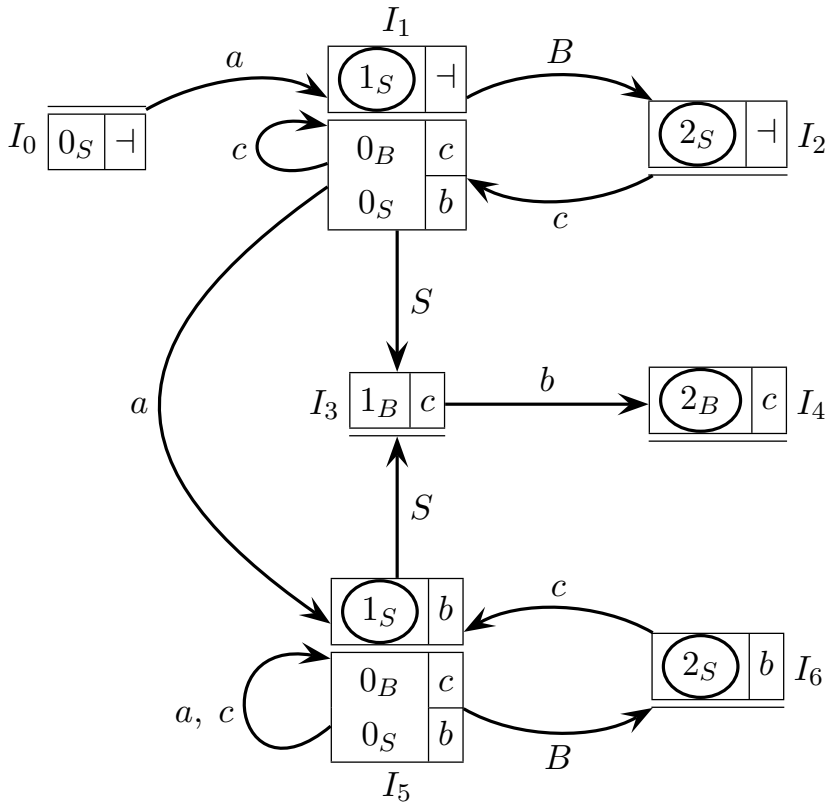
(c) To obtain a new net, equivalent to that of grammar G and with the same net states, but where nonterminal B is non-nullable, one could reason in this way. First make state 0_B non-final and add a spontaneous transition to the machine of nonterminal S, since now nonterminal B is not nullable any longer:



Yet, now the machine net is nondeterministic, and particularly machine M_S. Then, to make machine M_S deterministic again, cut the spontaneous transition $1_S \xrightarrow{\varepsilon} 2_S$, for instance by back-propagation:

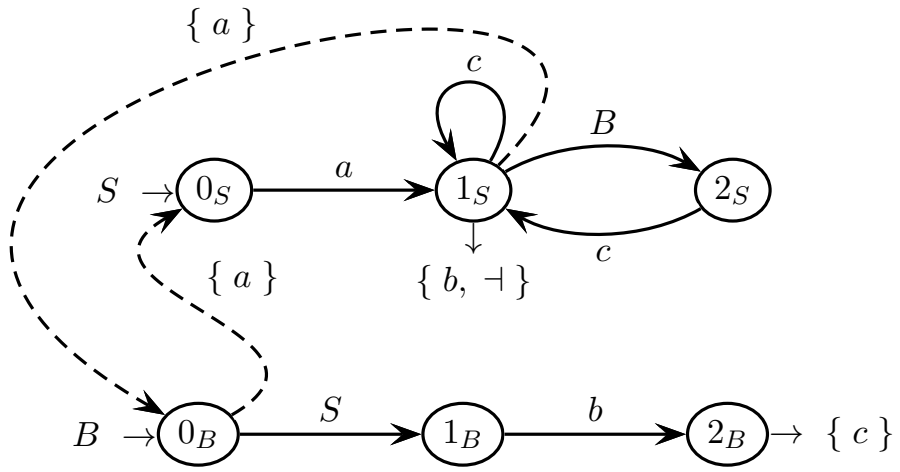


The result is deterministic. The new machine net is still of type $ELR(1)$. In fact, just slightly update the pilot and verify that there are not any conflicts. To update, it suffices to make state 0_B non-final and to put a self-loop labeled c on the m-states with 1_S. Here is the ELR verification:



There are not any conflicts even in the new pilot. Actually, the only topological difference with respect to the old pilot is a c-self-loop being added to the m-states I_1 and I_5. Yet, fortunately the item $\langle\, 0_B,\, c \,\rangle$ that shows up in both m-states is no longer final, as the net state 0_B itself is no longer nullable hence no longer final. Thus no shift-reduce conflict arises in the new pilot, and the rest is unchanged.

One could notice that the new machine net is even of type $ELL\,(1)$. Here is the *ELL* verification:



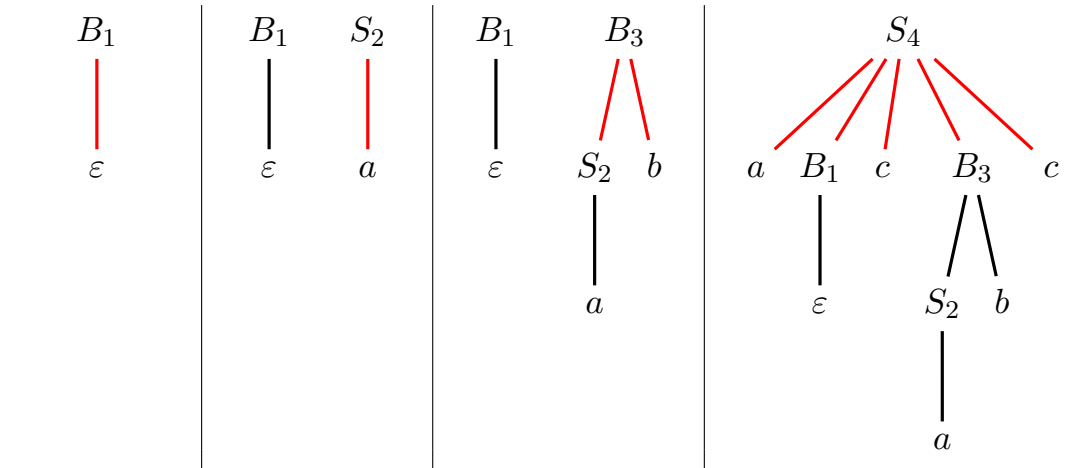
There are not any overlaps between guide sets even in the new machine net. Of course, being deterministic *ELL* implies being deterministic *ELR*, therefore this second verification could replace the previous one, and is even simpler.

A final remark on this point: cutting the spontaneous transition $1_S \xrightarrow{\varepsilon} 2_S$ by forward propagation does not work, as the resulting machine M_S would have two a-arcs outgoing from state 0_S, therefore it would not be deterministic. To circumvent the obstacle, machine M_S should be determinized, for instance by the powerset construction or by the Berry-Sethi algorithm, but this is likely to modify the state set, which is not permitted by the question.

(d) Here are the simulation table for the *ELR* analysis, and the tree construction:

stack of alternated macro-states and grammar symbols	*input left to be scanned*	*move executed*
I_0	$a\ c\ a\ b\ c\ \dashv$	initial configuration
$I_0\ a\ I_1$	$c\ a\ b\ c\ \dashv$	terminal shift: $I_0 \xrightarrow{a} I_1$
$I_0\ a\ I_1$	$c\ a\ b\ c\ \dashv$	(1) null reduction: $\varepsilon \rightsquigarrow B$
$I_0\ a\ I_1\ B\ I_2$	$c\ a\ b\ c\ \dashv$	nonterminal shift: $I_1 \xrightarrow{B} I_2$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1$	$a\ b\ c\ \dashv$	terminal shift: $I_2 \xrightarrow{c} I_1$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1\ a\ I_5$	$b\ c\ \dashv$	terminal shift: $I_1 \xrightarrow{a} I_5$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1$	$b\ c\ \dashv$	(2) reduction: $a \rightsquigarrow S$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1\ S\ I_3$	$b\ c\ \dashv$	nonterminal shift: $I_1 \xrightarrow{S} I_3$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1\ S\ I_3\ b\ I_4$	$c\ \dashv$	terminal shift: $I_3 \xrightarrow{b} I_4$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1$	$c\ \dashv$	(3) reduction: $S\ b \rightsquigarrow B$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1\ B\ I_2$	$c\ \dashv$	nonterminal shift: $I_1 \xrightarrow{B} I_2$
$I_0\ a\ I_1\ B\ I_2\ c\ I_1\ B\ I_2\ c\ I_1$	$\dashv$	terminal shift: $I_2 \xrightarrow{c} I_1$
I_0	$\dashv$	(4) reduction: $a\ B\ c\ B\ c \rightsquigarrow S$
initial stack	empty tape	last move is a reduction to S
acceptance condition verified — string $a\ c\ a\ b\ c$ accepted		

piecewise bottom-up left-to-right tree construction and rightmost derivation



4 Language Translation and Semantic Analysis 20%

1. A (possibly empty) binary string has an *even* parity (respectively an *odd* parity) if it includes an *even* (respectively *odd*) number of bits 1. For instance, strings ε, 0 0 and 0 1 0 1 have an even parity, whereas strings 1, 0 1 and 0 1 0 1 1 0 have an odd one.

   The source language L_s, over the three-letter alphabet $\Sigma = \{\ 0,\ 1,\ s\ \}$, is the set of all the (possibly empty) sequences of (possibly empty) binary strings, where each binary string is followed by a symbol s. Examples:

   $$0\,1\,s\,1\,0\,1\,s \qquad 1\,1\,0\,s\,0\,0\,1\,s\,0\,0\,s \qquad 0\,1\,s\,s\,1\,0\,1\,s$$

   The source language L_s is generated by the following source grammar G_s (axiom S):

   $$G_s \begin{cases} S & \rightarrow & B\ s\ S \mid \varepsilon \\ B & \rightarrow & 0\ B \mid 1\ B \mid \varepsilon \end{cases}$$

   Please answer the following questions and keep in mind that you may change the source grammar (but not the source language), when necessary. The solutions that use simpler grammars, in terms of number of nonterminals and rules, are preferred.

   (a) Design a translation scheme (or grammar) that defines a translation function τ_1, where:

   - each binary string included in the source string is translated into a symbol e or o if it has an even or odd parity, respectively
   - all the symbols s included in the source string are canceled

   Examples:

   $$\begin{aligned} \tau_1\,(0\,1\,s\,1\,0\,1\,s) &= o\,e \\ \tau_1\,(1\,1\,0\,s\,0\,0\,1\,s\,0\,0\,s) &= e\,o\,e \\ \tau_1\,(0\,1\,s\,s\,1\,0\,1\,s) &= o\,e\,e \end{aligned}$$

   (b) Design a translation scheme (or grammar) defining a translation function τ_2 that maps a source string into a destination string $e^h\,o^k$, where h and k ($h,\ k \geq 0$) are the numbers of binary strings with an even and odd parity, respectively, in the source string.

   Examples:

   $$\begin{aligned} \tau_2\,(0\,1\,s\,1\,0\,1s) &= e\,o \\ \tau_2\,(1\,1\,0\,s\,0\,0\,1\,s\,0\,0\,s) &= e\,e\,o \;=\; e^2\,o \\ \tau_2\,(0\,1\,s\,s\,1\,0\,1\,s) &= e\,e\,o \;=\; e^2\,o \end{aligned}$$

   (c) (optional) Can the translation functions τ_1 and τ_2 be defined by means of a regular translation expression or be computed by a deterministic finite-state device? Justify your answer.

Solution

(a) The source language, over the alphabet $\Sigma = \{\ 0,\ 1,\ s\ \}$, can be modeled as a two-level list. The main list (possibly empty) is a sequence of binary strings separated by a letter s, which also terminates the main list. Each sublist is a (possibly empty) binary string. The destination language is defined over the alphabet $\Delta = \{\ e,\ o\ \}$. For instance, take the third sample string $0\,1\,s\,s\,1\,0\,1\,s$:

$$\overbrace{\underbrace{0\,1}_{\text{1-st}}\ s\ \underbrace{\ }_{\text{2-nd}}\ s\ \underbrace{1\,0\,1}_{\text{3-rd}}\ s}^{\text{main list}}$$

Here is a somewhat obvious sub-grammar – often seen in many examples – that generates any binary string with even or odd parity. It has two nonterminals E and O that encode the even and odd parity of the string prefix generated so far, respectively:

$$\begin{cases} E & \to & 0\,E \mid 1\,O \mid \varepsilon \\ O & \to & 0\,O \mid 1\,E \end{cases}$$

The languages $L\,(E)$ and $L\,(O)$ contain all and only the binary strings with even and odd parity, respectively. Therefore a simple right-linear rule set that includes such a sub-grammar and generates the main list is $S \to E\ s\ S \mid O\ s\ S \mid \varepsilon$. Here is a possible translation scheme G_{τ_1} (axiom S) for τ_1, where the divider separates the rules that generate the list from those that generate a sublist:

$$G_{\tau_1} \begin{cases} S & \to & E\ s\ S & \qquad & S & \to & E\ e\ S \\ S & \to & O\ s\ S & & S & \to & O\ o\ S \\ S & \to & \varepsilon & & S & \to & \varepsilon \\ \hline E & \to & 0\,E \mid 1\,O \mid \varepsilon & & E & \to & E \mid O \mid \varepsilon \\ O & \to & 0\,O \mid 1\,E & & O & \to & O \mid E \end{cases}$$

Nonterminal S generates the whole list. It defines *a priori* if a sublist has an even or odd parity, outputs an e or o accordingly and gathers them on the left, in the same order as that of the sublists. Nonterminals E and O generate a sublist with an even or odd parity, respectively, but they do not output anything.

Optimized variant:

$$G'_{\tau_1} \begin{cases} S & \to & E\ s\ S & \qquad & S & \to & E\ S \\ S & \to & \varepsilon & & S & \to & \varepsilon \\ \hline E & \to & 0\,E \mid 1\,O \mid \varepsilon & & E & \to & E \mid O \mid e \\ O & \to & 0\,O \mid 1\,E \mid \varepsilon & & O & \to & O \mid E \mid o \end{cases}$$

Nonterminals E and O generate a sublist and encode the "memory" of an even or odd number of source bits 1, respectively. Upon the termination of the sublist, they output an e or o, respectively. Nonterminal S generates the whole list.

(b) Here is a possible translation scheme G_{τ_2} (axiom S) for τ_2, where the divider separates the rules that generate the list from those that generate a sublist:

$$G_{\tau_2} \left\{ \begin{array}{lcl|lcl} S & \to & E\ s\ S & S & \to & E\ e\ S \\ S & \to & O\ s\ S & S & \to & O\ S\ o \\ S & \to & \varepsilon & S & \to & \varepsilon \\ \hline E & \to & 0\ E \mid 1\ O \mid \varepsilon & E & \to & E \mid O \mid \varepsilon \\ O & \to & 0\ O \mid 1\ E & O & \to & O \mid E \end{array} \right.$$

Nonterminal S generates the whole list. It defines *a priori* if a sublist has an even or odd parity, outputs an e or o accordingly and gathers them on the left or right. Nonterminals E and O generate a sublist with an even or odd parity, respectively, but they do not output anything. Scheme G_{τ_2} is structurally very similar to scheme G_{τ_1}, with the only substantial difference coloured in red.

(c) The translation function τ_1 can be defined by a regular translation expression and can be computed by a deterministic finite-state device. In fact, checking the parity of a sublist only needs to count modulus-2 the number of bits 1 by using a finite memory. Furthermore, the output symbols e and o have to be written in the same position as the sublist they are associated with.

Here is a possible rational expression R_{τ_1} for the translation function τ_1:

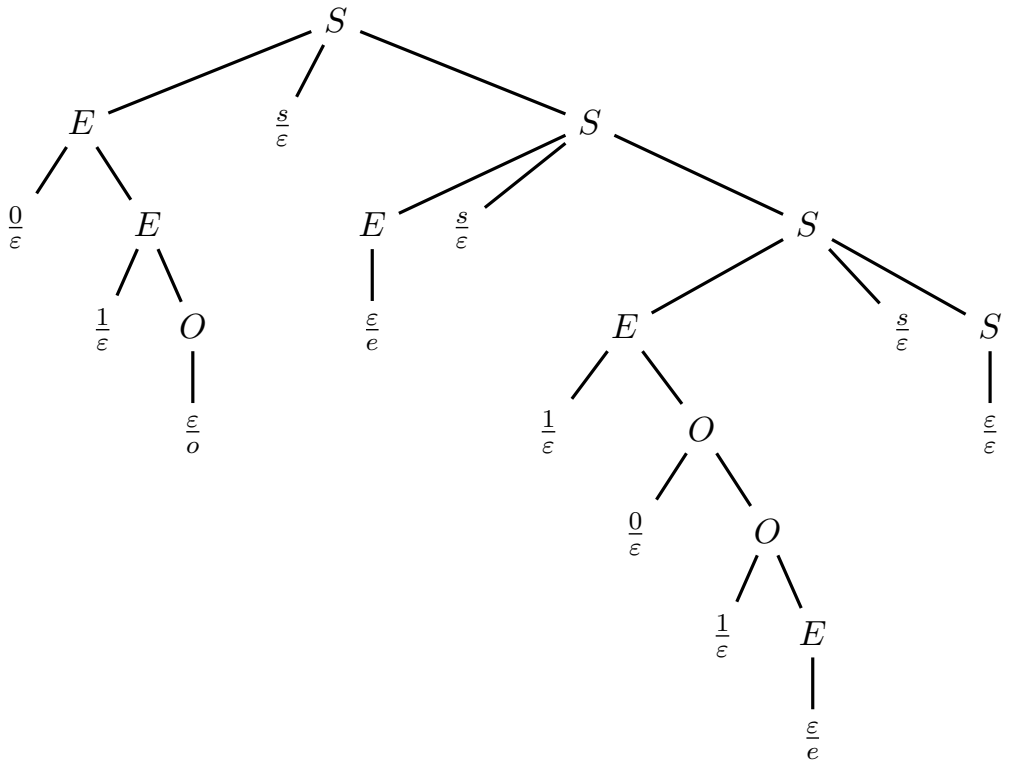
$$R_{\tau_1} = \left(\overbrace{\underbrace{\left(\left(\left(\frac{0}{\varepsilon} \right)^* \frac{1}{\varepsilon} \right)^2 \right)^* \left(\frac{0}{\varepsilon} \right)^* \frac{s}{e}}_{\text{source string with even parity}}}^{\text{even number of bits 1}} \mid \overbrace{\underbrace{\left(\left(\left(\frac{0}{\varepsilon} \right)^* \frac{1}{\varepsilon} \right)^2 \right)^* \left(\frac{0}{\varepsilon} \right)^* \frac{1}{\varepsilon} \left(\frac{0}{\varepsilon} \right)^* \frac{s}{o}}_{\text{source string with odd parity}}}^{\text{even number of bits 1}} \right)^*$$

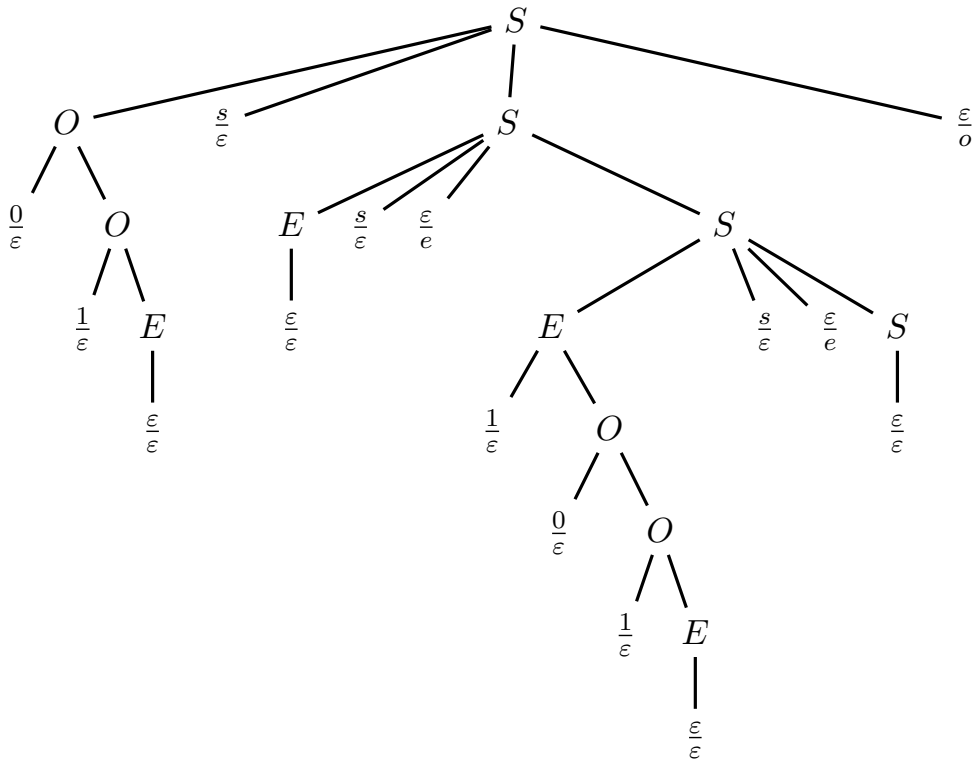
and here is another equivalent formulation, shortened by factorization:

$$R_{\tau_1} = \left(\overbrace{\underbrace{\left(\left(\left(\frac{0}{\varepsilon} \right)^* \frac{1}{\varepsilon} \right)^2 \right)^* \left(\frac{0}{\varepsilon} \right)^*}_{\text{even number of bits 1}}}^{\text{factorized part}} \underbrace{\left(\frac{s}{e} \mid \frac{1}{\varepsilon} \left(\frac{0}{\varepsilon} \right)^* \frac{s}{o} \right)}_{\text{even or odd parity}} \right)^*$$

Instead, the translation function τ_2 involves counting the number of the two sublist types, i.e., with an even or odd parity. Yet notoriously this operation cannot be computed by a finite-state device or be modeled by a regular scheme.

Here are the two syntax trees (source and destination overlapped) of the translation functions τ_1 (with scheme G'_{τ_1}) and τ_2 applied to the third sample string $0\,1\,s\,s\,1\,0\,1\,s$:





These trees reasonably demonstrate the correctness of both translation functions.

2. Consider the grammar below (axiom S), which generates a (possibly empty) two-level list. Each sublist consists of letters a (possibly none) and ends with one letter b.

$$\begin{cases} 1\colon\ S \rightarrow X \\ 2\colon\ X \rightarrow A\ X \\ 3\colon\ X \rightarrow \varepsilon \\ 4\colon\ A \rightarrow a\ A \\ 5\colon\ A \rightarrow b \end{cases}$$

A sample two-level list, with three sublists, is:

$$b\ a\ b\ a^2\ b$$

See also the syntax tree on the next pages.

The *length* ≥ 0 of a sublist is the number of elements, i.e., letters a, in the sublist.

We want to decide whether in a list the length of the sublists increases, from left to right, exactly by one, starting from an empty sublist (length 0). If this property is satisfied, a boolean attribute e assumes value *true*, else *false*. For the sample list above, the attribute e is *true*. If the whole list is empty, the attribute e is conventionally *true*.

Attributes permitted to use (no other attributes are allowed):

name	*type*	*domain*	*symbol*	*meaning*
l	left	integer	A	length ≥ 0 of a sublist
n	right	integer	X	required length of the sublist immediately appended to a node X
e	left	boolean	X, S	*true* if the sublists appended to a node X, or to the root S, have a length that increases by one from left to right (starting from 0), else *false*

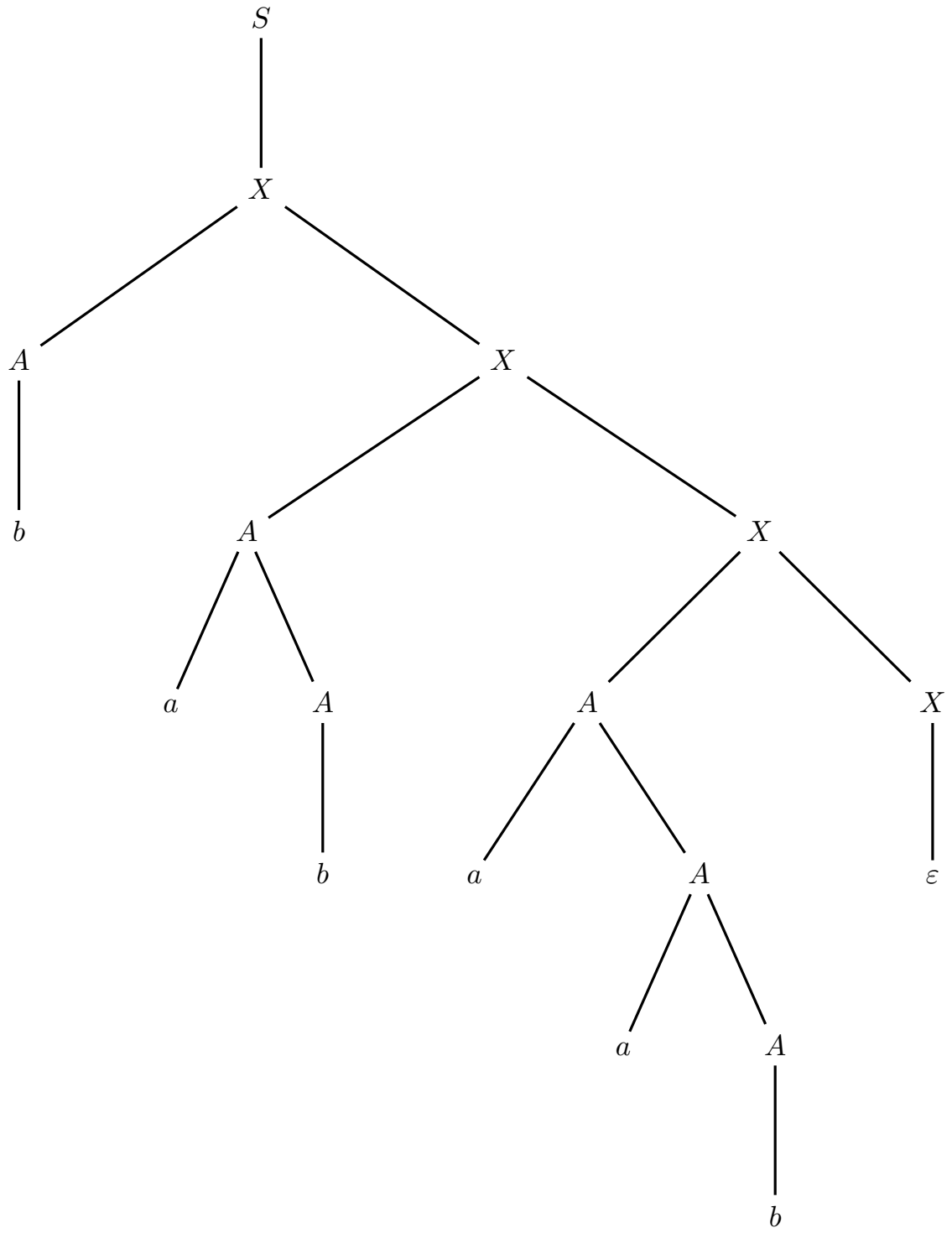
Answer the following questions (use the tables / trees / spaces on the next pages):

(a) Decorate the tree of the sample string $b\ a\ b\ a^2\ b$, prepared on the next page, with the attribute values. Pay attention to the type of each attribute (left or right).

(b) Write an attribute grammar, based on the syntax above and using only the permitted attributes (do not change the attribute types), that computes in the tree root the correctness attribute e for the whole list. The attribute grammar must be of type one-sweep. Please argue that your grammar is so.

(c) (optional) Write the procedure of the semantic analyzer (which exists as the attribute grammar must be of type one-sweep) for nonterminal X. Is it possible to integrate syntactic and semantic analysis ? Please explain your answer.

syntax tree to be decorated – question (a)

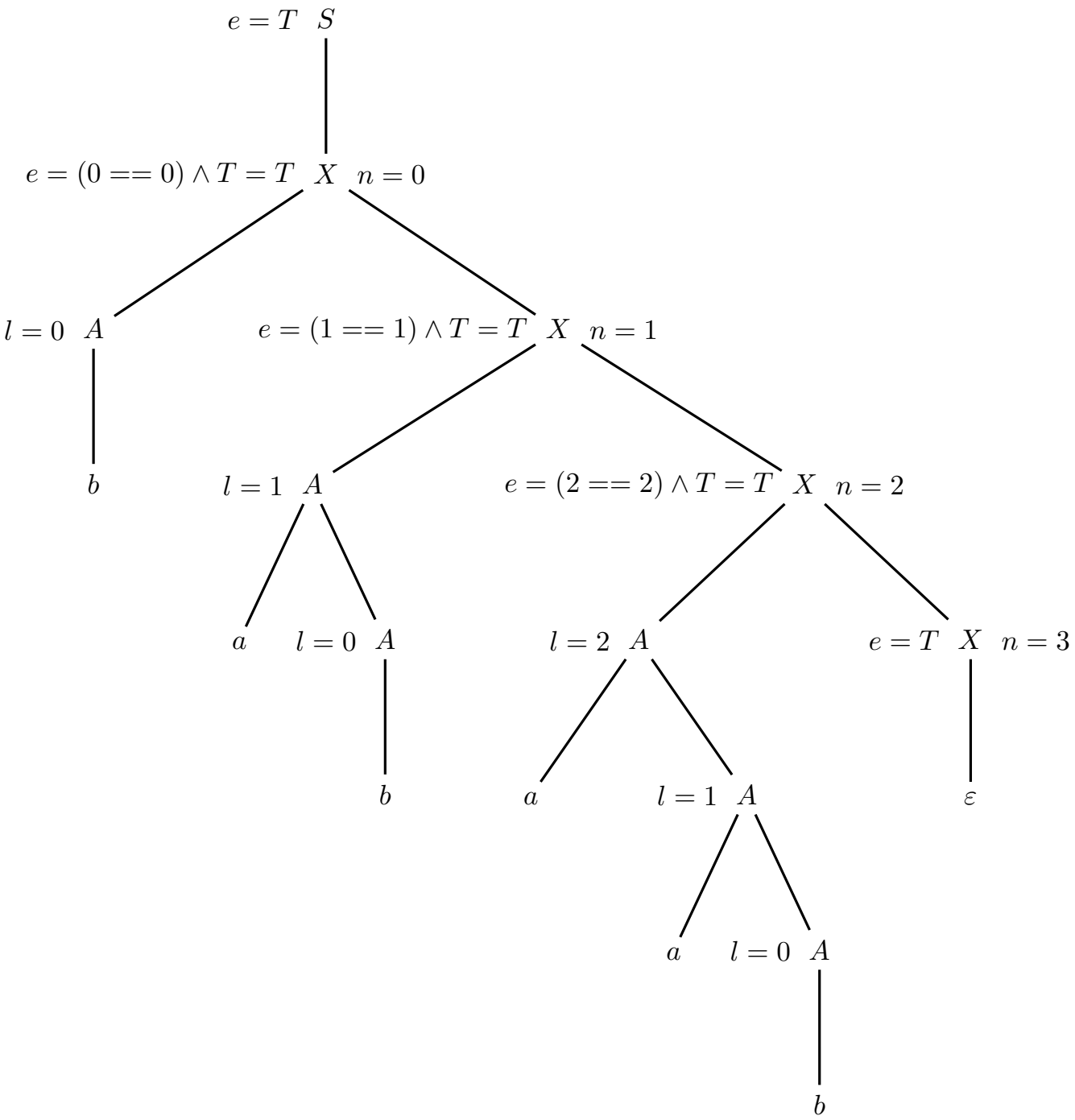


#	*syntax*	*semantics* – question (b)
1:	$S_0 \rightarrow X_1$	
2:	$X_0 \rightarrow A_1\ X_2$	
3:	$X_0 \rightarrow \varepsilon$	
4:	$A_0 \rightarrow a\ A_1$	
5:	$A_0 \rightarrow b$	

semantic procedure of nonterminal X – question (c)

Solution

(a) Here is the decorated syntax tree, computed by means of the second grammar version of point (b):



As customary, the left and the right attributes are written on the left and on the right of the tree node they refer to, respectively.

(b) The idea is that attribute l computes the length of each sublist, that attribute n computes the expected length of each sublist for the whole list to be correct, and that attribute e evaluates the equality between the actual and expected lengths of each sublist, and computes the logical product of all such equality checks.

Attribute l is computed upwards, as $l_0 = l_1 + 1$, starting from letter b with $l_0 = 0$. Attribute n is computed downwards, as $n_2 = n_0 + 1$, starting from $n_1 = 0$ in the root. Attribute e is computed upwards as $e_0 = e_2 \wedge (l_1 == n_0)$ (and as $e_0 = e_1$ in the root), starting form $e_0 = \mathit{true}$ in ε. Here is the complete attribute grammar:

#	*syntax*	*semantics*
1:	$S_0 \rightarrow X_1$	$n_1 = 0$ $e_0 = e_1$

2:	$X_0 \rightarrow A_1\ X_2$	$n_2 = n_0 + 1$
		if $(\ l_1 == n_0\ )$ **then**
		$\quad e_0 = e_2$
		else
		$\quad e_0 = \mathit{false}$
		endif

3:	$X_0 \rightarrow \varepsilon$	$e_0 = \mathit{true}$
4:	$A_0 \rightarrow a\ A_1$	$l_0 = l_1 + 1$
5:	$A_0 \rightarrow b$	$l_0 = 0$

or more compactly:

#	*syntax*	*semantics*
1:	$S_0 \rightarrow X_1$	$n_1 = 0$ $e_0 = e_1$
2:	$X_0 \rightarrow A_1\ X_2$	$n_2 = n_0 + 1$ notation (a) $e_0 = (\ l_1 == n_0\ ) \wedge e_2$ notation (b) $e_0 = (\ l_1 == n_0\ )\ ?\ e_2\ :\ \mathit{false}$
3:	$X_0 \rightarrow \varepsilon$	$e_0 = \mathit{true}$
4:	$A_0 \rightarrow a\ A_1$	$l_0 = l_1 + 1$
5:	$A_0 \rightarrow b$	$l_0 = 0$

The attribute grammar is acyclic, hence it is correct. Furthermore, it is one-sweep: the right attributes depend only on right ones in their parent node, and the left attributes depend only on right ones in their own node and on left ones in their child nodes. Thus the attribute dependencies can be trivially satisfied by the one-sweep tree visit order.

In fact, attribute n (right) depends only on itself in its parent node (see rule 2), attribute l (left) depends only on itself in a child node (see rule 4), and attribute e (left) depends only on a left attribute (namely l) in a child node and on itself in its own node (see rule 2 and, limitedly, rule 1).

(c) Yes, it is possible to integrate semantic and syntactic analysis, as the syntax is evidently of type $LL(1)$ (alternative rules have disjoint guide sets), the semantic is of type one-sweep and the right attribute depends only on itself (so there is not any need to examine the brother graph).
The semantic procedure of nonterminal X is rather simple. Here it is:

```
procedure X ( n0: in; e0: out )
var l1, n2, e2        // varlocs for the attribs of the child nodes
switch rule do
    case 2: X → A X do
        call A ( l1 )
        n2 = n0 + 1                              // inherited update
        call X ( n2, e2 )
        if ( l1 == n0 ) then  e0 = e2           // synthesized update
        else  e0 = false                        // synthesized update
    case 3: X → ε do  e0 = false               // synthesized update
    otherwise do error
```

For completeness, here are the other two procedures, first for nonterminal A:

```
procedure A ( l0: out )
var l1                // varlocs for the attribs of the child nodes
switch rule do
    case 3: A → a A do
        call A ( l1 )
        l0 = l1 + 1                              // synthesized update
    case 4: A → b do  l0 = 0          // synthesized initialization
    otherwise do error
```

and then for the tree root:

```
procedure S ( e0: out )
var n1, e1            // varlocs for the attribs of the child nodes
switch rule do
    case 1: S → X do
        n1 = 0                          // inherited initialization
        call X ( n1, e1 )
        e0 = e1                                  // synthesized update
    otherwise do error
```

The arguments for passing the subtrees are here omitted (see the textbook). The semantic analyzer is invoked starting from the axiomatic procedure S.