FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 5 July 2017 - Part Theory

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

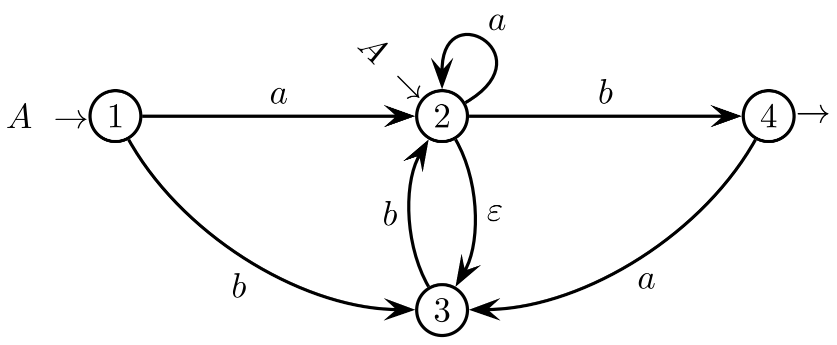
LAST + FIRST NAME:	
(capital letters please)	
MATRICOLA:	SIGNATURE:
(or PERSON CODE)	

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 nondeterministic finite-state automaton A below, over the two-letter alphabet $\Sigma = \{ a, b \}$, with two initial states 1 and 2, and with one final state 4:



Answer the following questions:

(a) Find the shortest string of the regular language $L(A)$ that is recognized by automaton A with at least two different computations and that therefore is ambiguous, and suitably justify your answer. Provide evidence that the string is the shortest one with such a property.

(b) Transform automaton A and obtain an automaton A', possibly still nondeterministic, that has only one initial state and does not have any spontaneous transitions (ε-transitions), by properly cutting such transitions.

(c) By using the Berry-Sethi method, obtain a deterministic automaton A'' equivalent to the original nondeterministic automaton A.

(d) (optional) Minimize the deterministic automaton A'' found at point (c). Then write a regular expression R equivalent to the automaton A'' found. Notice: for this whole question, you may proceed systematically or informally, but in the latter case provide some evidence of correctness.

Solution

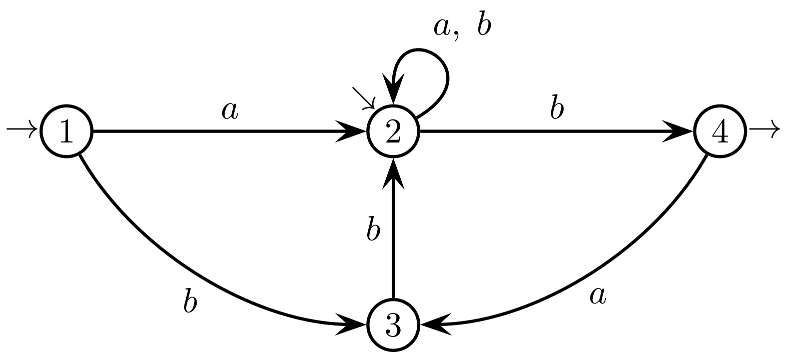
(a) The valid string $a\,b \in L\,(A)$ is ambiguous. In fact, automaton A has exactly two accepting computations for this string:

$$1 \xrightarrow{a} 2 \xrightarrow{b} 4$$

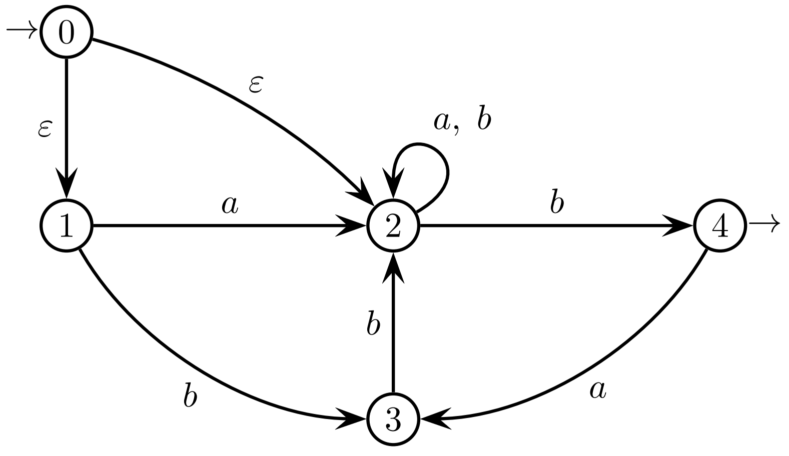
$$2 \xrightarrow{a} 2 \xrightarrow{b} 4$$

About the minimality of length of this string. Language $L\,(A)$ does not contain the empty string ε. In the automaton A, the only accepting computation of length one is $2 \xrightarrow{b} 4$, so the valid string b of length one is not ambiguous. There is only one more computation (of length three) in A that accepts a string of length two: $2 \xrightarrow{\varepsilon} 3 \xrightarrow{b} 2 \xrightarrow{b} 4$; so the valid string $b\,b$ of length two is not ambiguous. Therefore, the string $a\,b$ is actually the shortest ambiguous one.

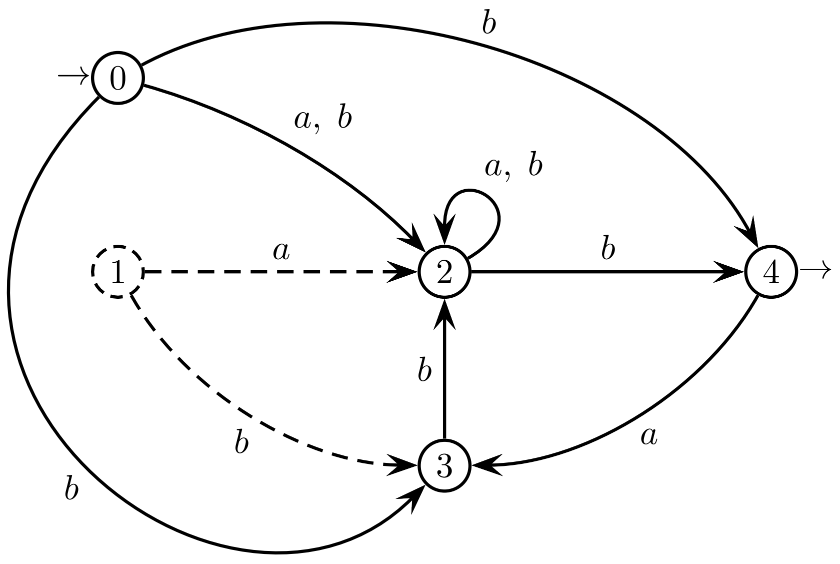
(b) Cutting arc $3 \xrightarrow{\varepsilon} 2$ by back-propagating arc $3 \xrightarrow{b} 2$ onto state 2:



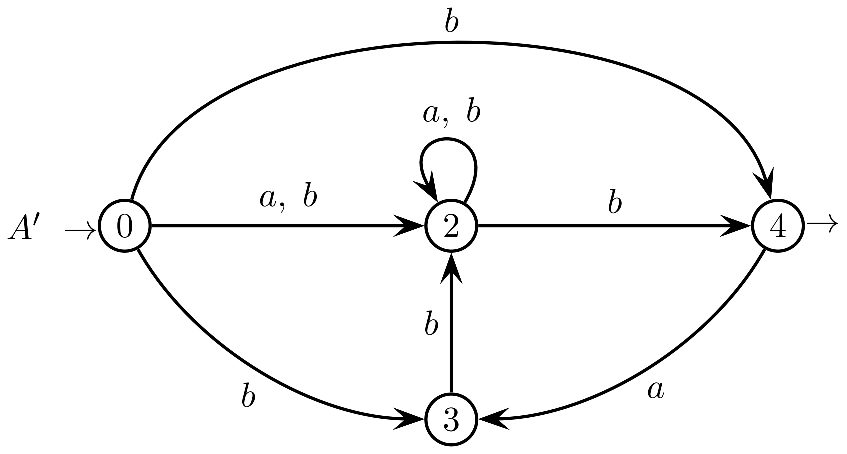
Introducing only one initial state 0 (by means of ε-transitions):



Cutting the two spontaneous transitions by back-propagation:

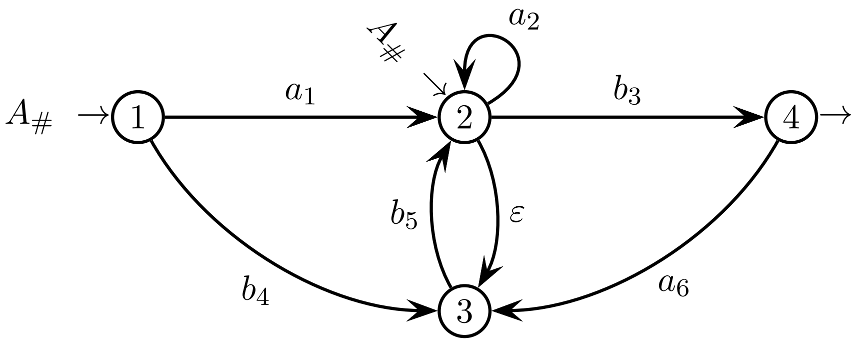


Finally, cleaning (dashed nodes and arcs) and rearranging the layout:



Automaton A' has only one initial state and does not have any spontaneous transitions. Anyway, it is still nondeterministic, since it has quite a few arcs with an equal label that exit from the same source states.

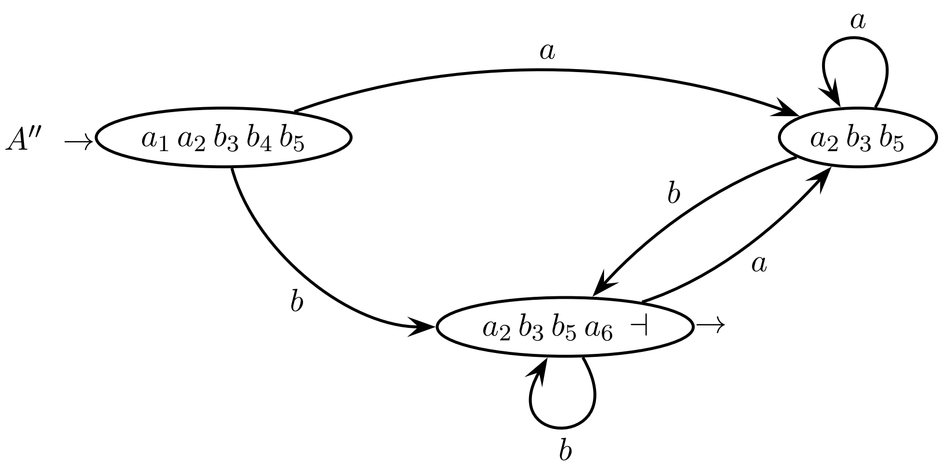
(c) Here is the procedure to obtain the *BS* automaton A'' from the original automaton A. Remember that the original automaton A also includes spontaneous transitions and has two initial states. Numbering of A:



Here are the initials and the followers of each terminal of $A_\#$ (some terminals are listed together if they have the same followers):

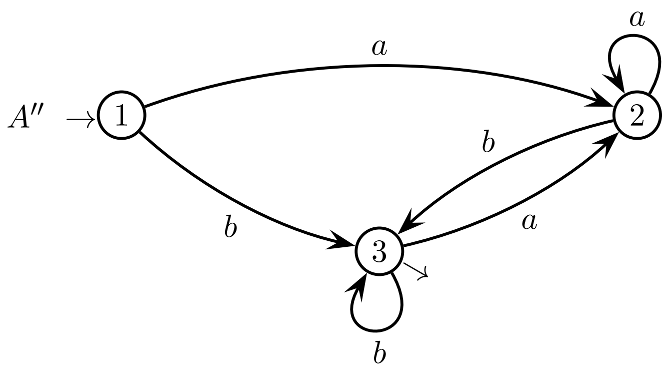
initials	$a_1,\ a_2,\ b_3,\ b_4,\ b_5$
terminals	followers
$a_1,\ a_2,\ b_5$	$a_2,\ b_3,\ b_5$
b_3	$a_6,\ \dashv$
$b_4,\ a_6$	b_5

Then, here is the state-transition graph of the deterministic automaton A'' produced by the *BS* algorithm, with three states:

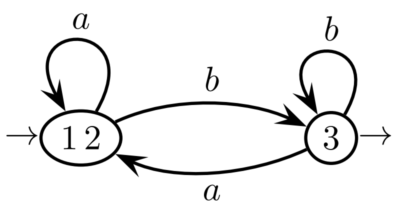


Automaton A'' is clean by construction. Of course, it may be not minimal.

(d) Concerning the minimality of automaton A'' (after renaming the states):



Informally: the non-final states 1 and 2 are undistinguishable, as their a-arcs go to the same state 2 and their b-arcs go to the same state 3. So there are two state equivalence classes: $[1, 2]$ and $[3]$. The minimal automaton has two states:



An equivalent regular expression R is:

$$R = a^* \; b \; b^* \; (\, a \; a^* \; b \; b^* \,)^*$$

obtained by scanning all the accepting paths in the minimal automaton. It can be immediately simplified into:

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

With a little more insight, one can find an even simpler form:

$$R = \Sigma^* \; b$$

that is, all the strings that end by letter b.

If one wished to proceed systematically, one could for instance minimize the automaton by the standard Nerode method (based on the triangular table of state undistinguishability) and then obtain a regular expression by the Brzozowsky (node elimination) method. Given the simplicity of the problem here, proceeding systematically may be uselessly tedious, though perfectly correct.

2 Free Grammars and Pushdown Automata 20%

1. Consider the following language L over the three-letter alphabet $\Sigma = \{ a, \ b, \ c \}$:

$$L = \left\{ w \ c^n \ \middle| \ \begin{array}{l} w \in \{ a, \ b \}^* \ \text{ and } \ n > 0 \\ \text{and } \ \big(\ \#_a(w) = n \ \text{ or } \ \#_b(w) = n \ \big) \end{array} \right\}$$

For instance: $b\,c, \ b\,a\,a\,c\,c, \ a\,b\,a\,a\,b\,c\,c \in L$; while $a\,a\,c, \ b\,c\,c, \ a\,b\,a\,a\,c\,c \notin L$.

Answer the following questions:

(a) Write a *BNF* grammar G (no matter if ambiguous) that generates language L.

(b) Sketch the syntax trees of the strings $b\,c, \ b\,a\,a\,c\,c \in L$.

(c) (optional) Determine if grammar G is ambiguous and justify your answer: if it is ambiguous then provide a string with two or more syntax trees, else argue that it does not generate any ambiguous string.

Solution

(a) Here is a possible *BNF* grammar G for language L:

$$G \begin{cases} S & \to & A \mid B \\ A & \to & b\ A \mid a\ A_1\ c \\ A_1 & \to & b\ A_1 \mid a\ A_1\ c \mid \varepsilon \\ B & \to & a\ B \mid b\ B_1\ c \\ B_1 & \to & a\ B_1 \mid b\ B_1\ c \mid \varepsilon \end{cases}$$

It is a union grammar: nonterminal A generates the strings $w \in L$ that satisfy $\#_a(w) = n$, and nonterminal B those that satisfy $\#_b(w) = n$, both for $n > 0$. Ambiguity is not an issue here, however we can notice that the strings $w \in L$ with $\#_a(w) = \#_b(w) = n$ are generated by both nonterminals A and B, so that grammar G is ambiguous. Whether the language L is inherently ambiguous, would require to make a closer analysis of the language structure.

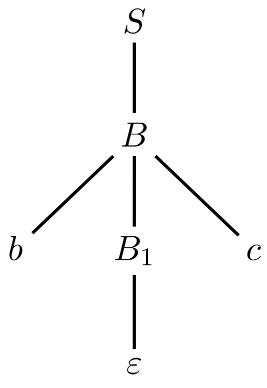
(b) We report a leftmost derivation for string $b\ c \in L$:

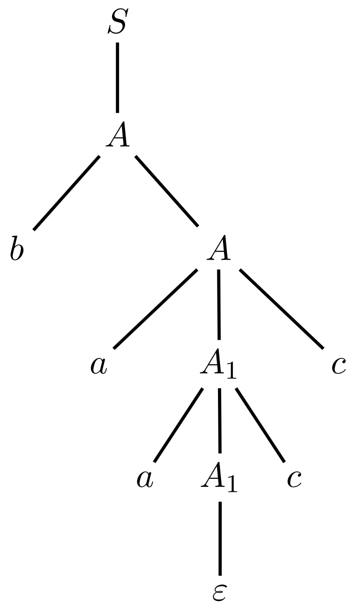
$$S \Rightarrow B \Rightarrow b\ B_1\ c \Rightarrow b\ c$$

and a leftmost one for string $b\ a\ a\ c\ c \in L$:

$$S \Rightarrow A \Rightarrow b\ A \Rightarrow b\ a\ A_1\ c \Rightarrow b\ a\ a\ A_1\ c\ c \Rightarrow b\ a\ a\ c\ c$$

Here are the corresponding syntax trees:





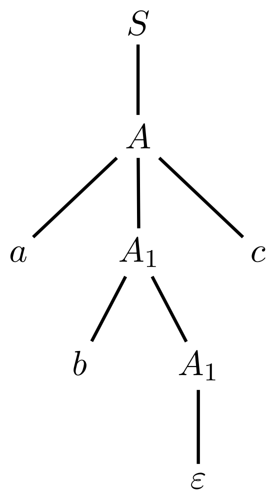
None of these two strings is ambiguous, so they do not have other syntax trees.

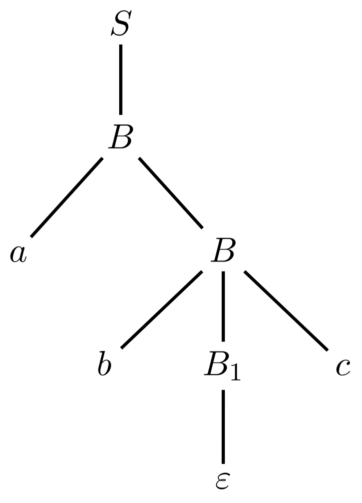
(c) The string $a\, b\, c \in L$ is ambiguous, and here are two leftmost derivations for it:

$$S \Rightarrow A \Rightarrow a\, A_1\, c \Rightarrow a\, b\, A_1\, c \Rightarrow a\, b\, c$$

$$S \Rightarrow B \Rightarrow a\, B_1\, c \Rightarrow a\, b\, B_1\, c \Rightarrow a\, b\, c$$

Here are the corresponding syntax trees:





The two trees are different, so string $a\,b\,c$ is ambiguous. Grammar G generates infinitely many ambiguous strings, namely all those with $\#_a(w) = \#_b(w)$. Likewise, each such string has exactly two syntax trees, whereas every other valid string generated by grammar G is not ambiguous.

Remark Here is a different version G' of the grammar (axiom S), for completeness:

$$G' \begin{cases} S & \to & A \mid B \\ A & \to & X\, a\, A\, c \mid X\, a\, X\, c \\ B & \to & Y\, b\, B\, c \mid Y\, b\, Y\, c \\ X & \to & b\, X \mid \varepsilon \\ Y & \to & a\, Y \mid \varepsilon \end{cases}$$

The reader can verify grammar G' by drawing the same syntax trees as above. Grammar G' is as ambiguous as grammar G. The difference between them is structural: grammar G generates trees with only one stem of arbitrary length, whereas grammar G' can generate trees with two or more stems of arbitrary length.

2. Consider a fragment of a programming language that is structured according to the following specifications:

   - The program consists of a declaration section followed by an execution section.
   - The declaration section is a possibly empty sequence of typed variable lists; commas " , " separate the variables in a list and a semicolon " ; " terminates each list in the sequence.
   - There are two variable types: scalars (i.e., integers) and fixed-size one-dimensional arrays of scalars (i.e., integers).
   - The execution section consists of a non-empty list of statements, each one terminated by a semicolon " ; ".
   - There are two statement types:
     - assignment: a scalar or an array element = an arithmetic expression
     - procedure call: name (possibly empty list of args separated by comma " , ")
   - An arithmetic expression may consist of numbers, scalars, array elements, function calls (structured like procedure calls), operators of addition " + ", subtraction " − " (possibly unary), multiplication " ∗ " and division " / ". The operator precedences are as usual. Parenthesized subexpressions are not allowed, but an array element may be indicized by an arithmetic expression.
   - A procedure or function parameter may be a number, a scalar, an array element or an arithmetic expression.
   - Identifiers and numbers are represented by the terminals id and num, respectively.

   Further detailed information about the lexical and syntactic structure of the programming language can be inferred from the example below:

```
int a, b, c;

int v[100], d, z[50];

a = a + b * c;

v[7] = funct (a, b + 5);

proc (a, v[a + b]);

b = myfun ( );

z[c] = -d;
```

   Write a grammar, *EBNF* and unambiguous, that models the sketched programming language.

Solution

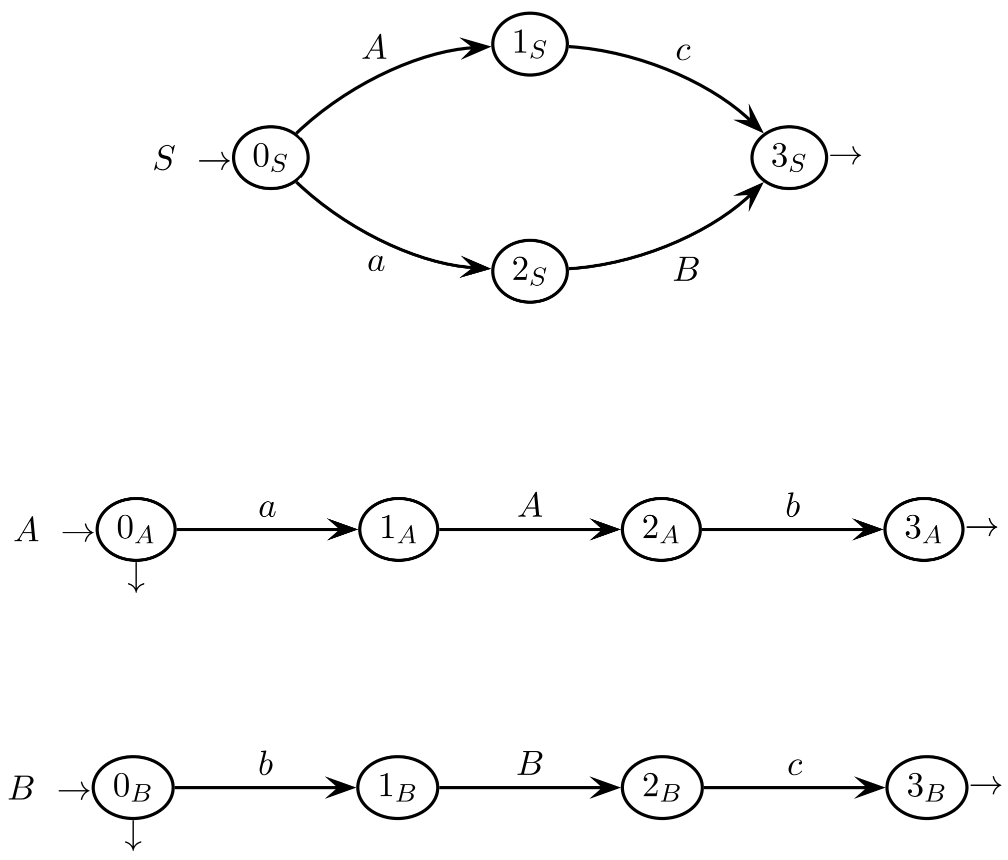
Here is a possible grammar G (axiom PROG) for the sketched programming language:

$$\left\{\begin{array}{rcl}
\langle\mathsf{PROG}\rangle & \rightarrow & \langle\mathsf{DECL}\rangle\ \langle\mathsf{EXEC}\rangle \\
\hline
\langle\mathsf{DECL}\rangle & \rightarrow & \big(\ \langle\mathsf{VARS}\rangle\ \text{‘};\text{’}\big)^{*} \\
\langle\mathsf{VARS}\rangle & \rightarrow & \mathsf{int}\ \langle\mathsf{VAR}\rangle\ \big(\text{‘},\text{’}\ \langle\mathsf{VAR}\rangle\ \big)^{*} \\
\langle\mathsf{VAR}\rangle & \rightarrow & \mathsf{id}\ \mid\ \mathsf{id}\,\text{‘[’}\,\mathsf{num}\,\text{‘]’} \\
\hline
\langle\mathsf{EXEC}\rangle & \rightarrow & \big(\ \langle\mathsf{STAT}\rangle\ \text{‘};\text{’}\big)^{+} \\
\langle\mathsf{STAT}\rangle & \rightarrow & \langle\mathsf{ASGN}\rangle\ \mid\ \langle\mathsf{CALL}\rangle \\
\langle\mathsf{ASGN}\rangle & \rightarrow & \big(\mathsf{id}\ \mid\ \langle\mathsf{AELM}\rangle\ \big)\,\text{‘}=\text{’}\ \langle\mathsf{EXPR}\rangle \\
\langle\mathsf{CALL}\rangle & \rightarrow & \mathsf{id}\,\text{‘(’}\big[\ \langle\mathsf{PARS}\rangle\ \big]\text{‘)’} \\
\langle\mathsf{AELM}\rangle & \rightarrow & \mathsf{id}\,\text{‘[’}\ \langle\mathsf{EXPR}\rangle\ \text{‘]’} \\
\langle\mathsf{PARS}\rangle & \rightarrow & \langle\mathsf{EXPR}\rangle\ \big(\text{‘},\text{’}\ \langle\mathsf{EXPR}\rangle\ \big)^{*} \\
\hline
\langle\mathsf{EXPR}\rangle & \rightarrow & \big[\text{‘}-\text{’}\big]\ \langle\mathsf{TERM}\rangle\ \Big(\big(\text{‘}+\text{’}\ \mid\ \text{‘}-\text{’}\big)\ \langle\mathsf{TERM}\rangle\ \Big)^{*} \\
\langle\mathsf{TERM}\rangle & \rightarrow & \langle\mathsf{FACT}\rangle\ \Big(\big(\text{‘}*\text{’}\ \mid\ \text{‘}/\text{’}\big)\ \langle\mathsf{FACT}\rangle\ \Big)^{*} \\
\langle\mathsf{FACT}\rangle & \rightarrow & \mathsf{num}\ \mid\ \mathsf{id}\ \mid\ \langle\mathsf{AELM}\rangle\ \mid\ \langle\mathsf{CALL}\rangle
\end{array}\right.$$

Grammar G is *EBNF* and non-ambiguous. Notice that an array must be declared with a fixed size, but that it can be indexed dynamically according to the specifications. Arithmetic expressions may have only two levels (no subexpressions), and the operator precedence is as usual: addition/subtraction lower, multiplication/division higher.

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 three-letter nonterminal alphabet $V = \{ S, A, B \}$ (axiom S):



Answer the following questions:

(a) Draw the complete pilot of grammar G, say if grammar G is of type $ELR(1)$, and shortly justify your answer. If grammar G is not $ELR(1)$ then highlight all the conflicts in the pilot.

(b) Write the necessary guide sets on the arcs of the machine net, show that grammar G is not of type $ELL(1)$, based on the guide sets, and shortly justify your answer (for the guide sets please use the figure prepared on the next page).

(c) (optional) Say if grammar G is of type $ELL(k)$ for some $k > 1$, and provide an adequate justification to your answer.

(d) Analyze string $a\,b\,c$ by using the Earley method, and determine if the string belongs to language $L(G)$, justifying your answer based on the contents of the Earley vector.

please here draw the call arcs and write the guide sets with $k = 1$

$$\begin{tikzcd}
A \arrow[r] & 0_A \arrow[r, "a"] \arrow[d] & 1_A \arrow[r, "A"] & 2_A \arrow[r, "b"] & 3_A \arrow[r] & {} \\
 & {} & & & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & 1_S \arrow[dr, bend left, "c"] & & \\
S \arrow[r] & 0_S \arrow[ur, bend left, "A"] \arrow[dr, bend right, "a"'] & & 3_S \arrow[r] & {} \\
 & & 2_S \arrow[ur, bend right, "B"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
B \arrow[r] & 0_B \arrow[r, "b"] \arrow[d] & 1_B \arrow[r, "B"] & 2_B \arrow[r, "c"] & 3_B \arrow[r] & {} \\
 & {} & & & &
\end{tikzcd}$$

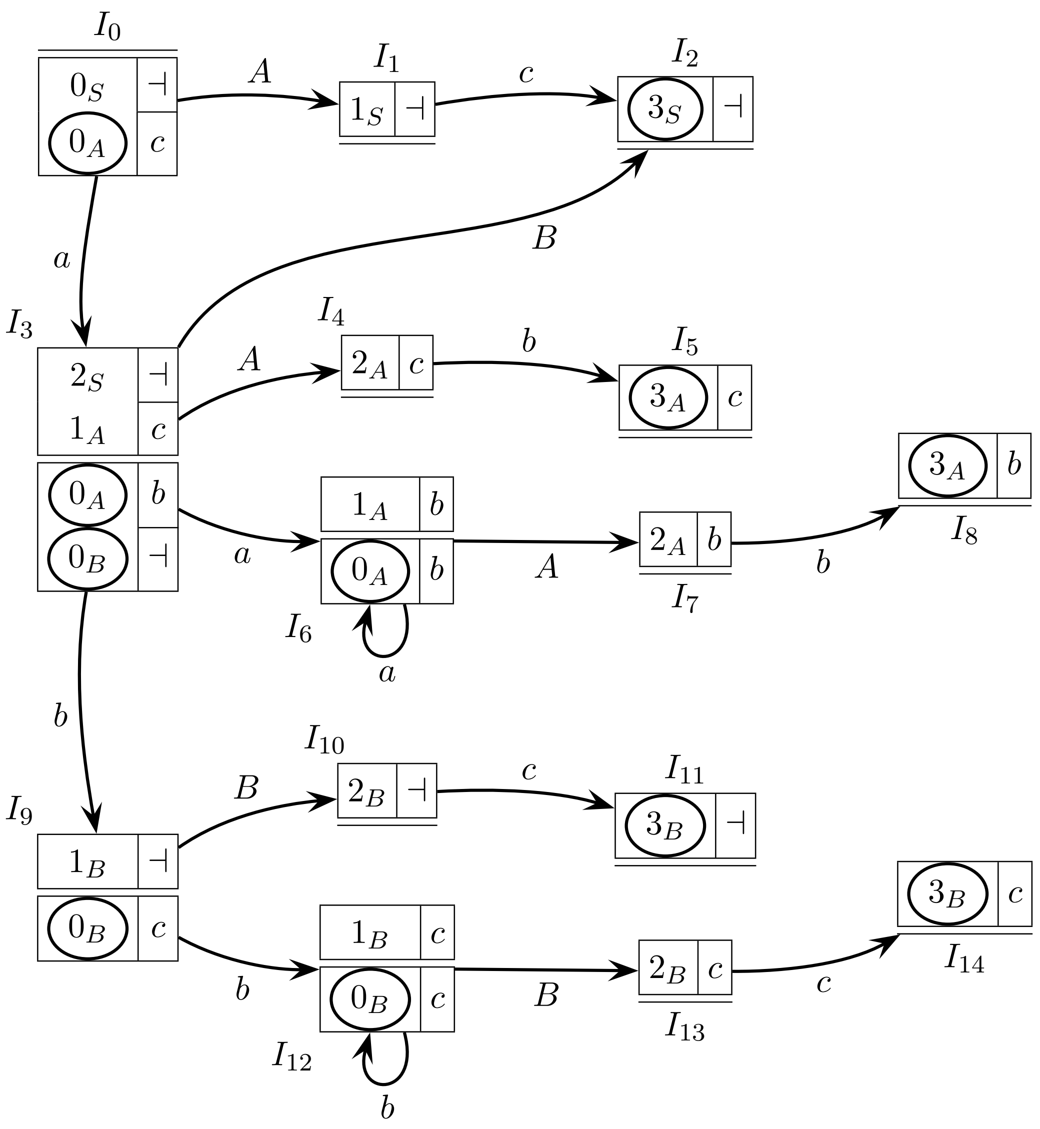
Earley vector of string $a\ b\ c$ (to be filled in)

(the number of rows is not significant)

0	a	1	b	2	c	3

Solution

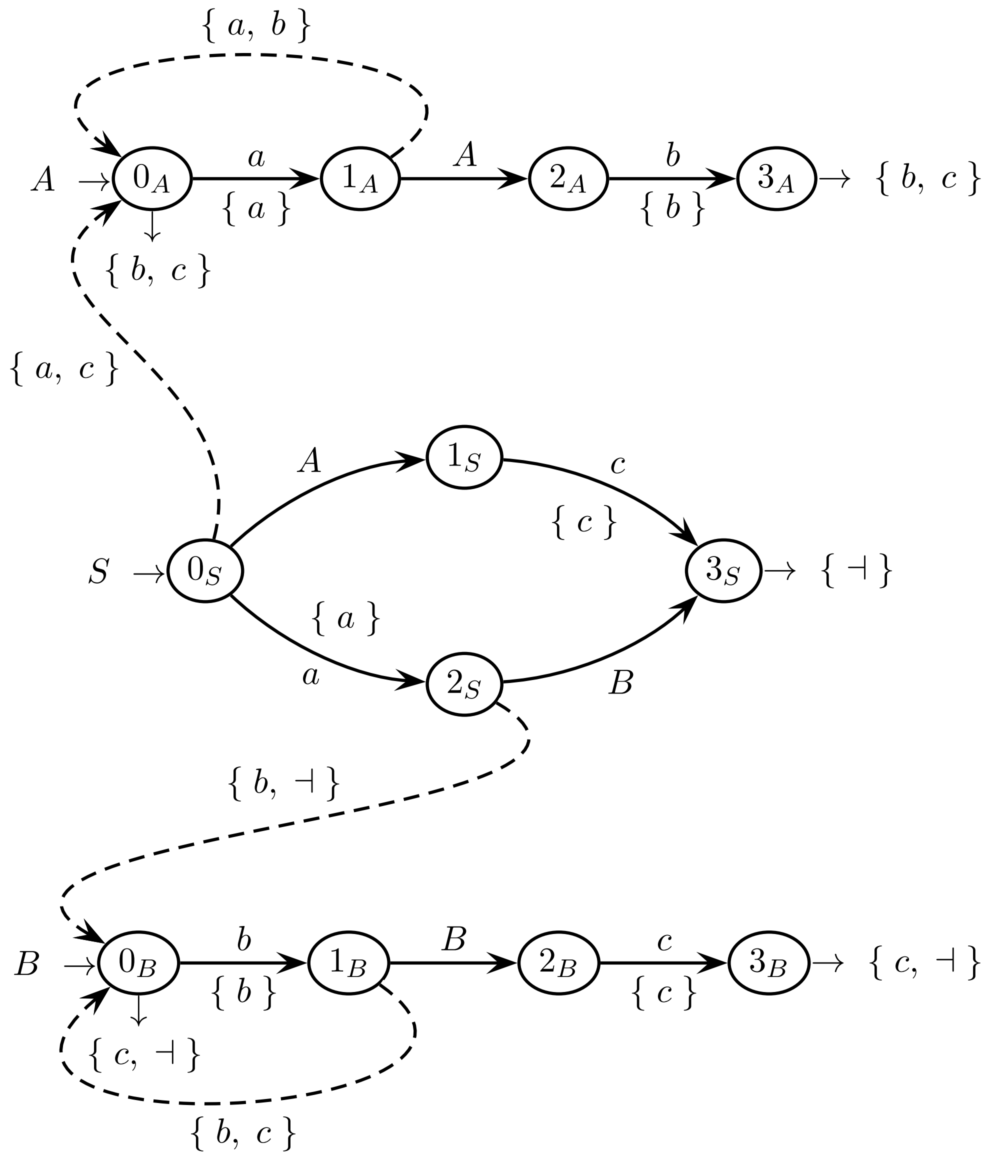
(a) Here is the complete pilot of grammar G with 15 m-states (initial m-state I_0):



The m-state I_3 contains a shift-reduce conflict on terminal b. The items of I_3 involved in the conflict are $\langle\, 0_A,\, b\,\rangle$ and $\langle\, 0_B,\, \dashv\rangle$: the former is a reduction item (since the net state 0_A is final) with look-ahead b, and the latter is a shift item on terminal b (since the net state 0_B has an outgoing b-arc). Thus grammar G is not of type $ELR\,(1)$. The pilot of G does not have any other conflicts, anyway.

The reader may notice that actually the valid string $a\,b\,c$ is ambiguous, as it has two syntax trees (we leave the reader to find them), so that for sure grammar G cannot be deterministic of any type. Anyway, all the other language strings are not ambiguous, so one might reasonably conjecture that the language of G is not inherently ambiguous, and that it may even have a deterministic grammar G' (which however should be found and proved correct).

(b) Here is the complete *PCFG* (Parser Control Flow-Graph) of grammar G (with $k = 1$), with all the call arcs and exit arrows, and all their guide sets:



The *PCFG* (with $k = 1$) of grammar G has three bifurcation states: 0_S, 0_A and 0_B, with the last two being final. However only state 0_S is conflicting, because its guide sets share terminal a. Thus grammar G is not of type $ELL\,(1)$. Notice that, for the valid string $a\,b\,c \in L\,(G)$, the *PCFG* gets early into conflict, on reading the first input character (namely a), whereas the pilot reads it deterministically and gets into conflict a little later (see point (a)), on reading the second input character (namely b). This different behaviour could be expected, since in general a *PCFG* is less powerful than a pilot in achieving determinism.

(c) Grammar G cannot be of type $ELL\,(k)$ for any $k > 1$, because it is ambiguous, as already observed at point (a). Ambiguity concerns only the valid string $a\,b\,c \in L\,(G)$, yet this suffices to make grammar G non-$ELL\,(k)$ for any $k > 1$.

(d) TO BE DONE (somewhat easy)

4 Language Translation and Semantic Analysis 20%

1. The *BNF* source grammar G below generates a non-empty list of elements a and b, separated by element z (axiom S):

$$G \begin{cases} S \rightarrow a \mid b \\ S \rightarrow a\,T \mid b\,T \\ T \rightarrow z\,S \end{cases}$$

Please answer the following questions:

(a) Write a destination grammar associated to the source grammar G, without changing G, that mirrors the source string. For instance:

$$a\ z\ a\ z\ b \ \mapsto\ b\ z\ a\ z\ a$$

To verify it, draw the source and destination syntax trees (if you wish, overlap the two trees) of the translation example.

(b) Could the translation of point (a) be defined by a formalism simpler than a translation grammar (or scheme) ? Please adequately justify your answer.

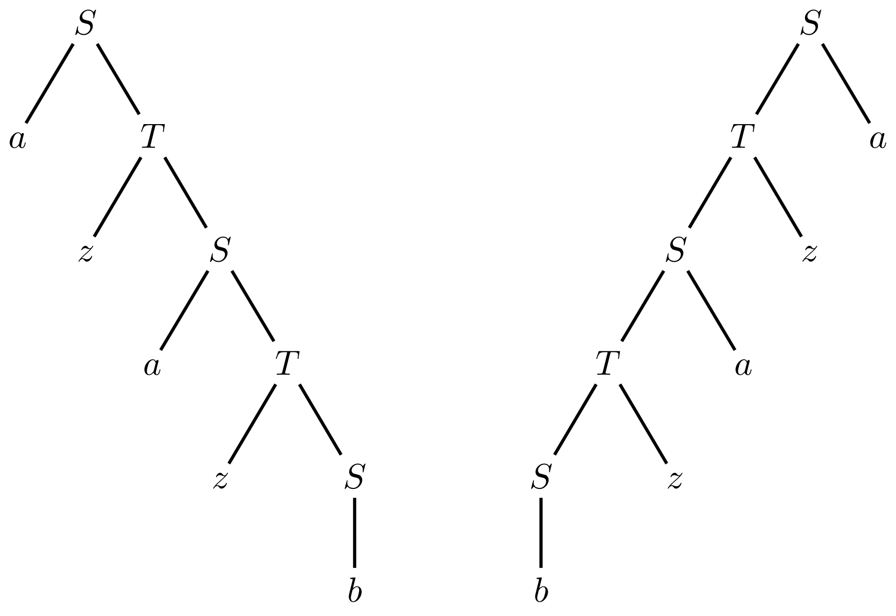
(c) (optional) The source grammar G is of type *ELL* (1). For nonterminal S, write a recursive descent syntactic procedure that also outputs the translation.

Solution

(a) Here is a working translation scheme G (axiom S), modeled on the proposed source grammar:

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

Here are the separated source (left) and destination (right) syntax trees of the translation sample:



The source and destination trees are right- and left-linear, respectively. In fact, the source and destination languages, independently of each other, are regular.

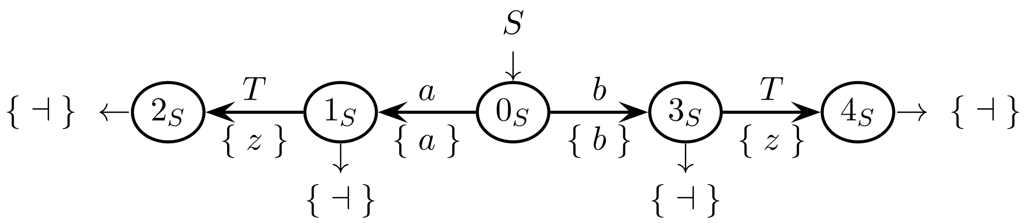
(b) It is impossible to significantly simplify the formalism used for the proposed translation. In fact, though the source and destination grammars — independently of each other — are right- and left-linear, respectively, and therefore are regular, the translation grammar (or scheme) mirrors a string and, as it is well known, such an operation is too powerful for a finite-state device.

(c) The translation grammar G is very simple. Just write the source and destination rules of nonterminal S into the "fractional" form: $S \to \frac{a}{\varepsilon}\ \frac{\varepsilon}{a}$ and $S \to \frac{a}{\varepsilon}\ T\ \frac{\varepsilon}{a}$; similarly for the two rules with terminal b. Both the rules with terminal a read and write the same things, and their only difference is in the presence of nonterminal T, the presence of which is in correspondence with the occurrence of terminal z in the input; same for the two rules with b. Then a syntactic (recursive descent) translation procedure for S is immediate. Here it is:

```
procedure $S$
if $cc == a$ then                          // cases of rule $S \to a$ or $S \to a\,T$
    $cc = next$                                                    // input $a$
    if $cc == z$ then call $T$                           // case of rule $S \to a\,T$
    if $cc == \dashv$ then                    // both rules are followed by $\dashv$
        write ($a$)                                               // output $a$
        return
else if $cc == b$ then                     // cases of rule $S \to b$ or $S \to b\,T$
    $cc = next$                                                    // input $b$
    if $cc == z$ then call $T$                           // case of rule $S \to b\,T$
    if $cc == \dashv$ then                    // both rules are followed by $\dashv$
        write ($b$)                                               // output $b$
        return
$error$                                            // all error cases grouped
```

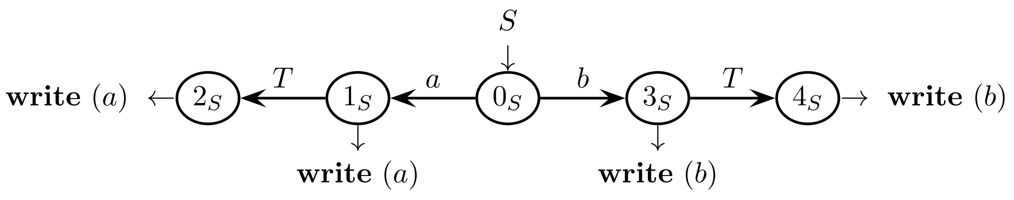
Algorithm: Recursive descent translator (partial).

Systematically: obviously grammar G is $ELL(1)$, as it immediately turns out by factoring the terminals a and b in their respective source rules. So here is a straightforward deterministic source machine M_S, with all the guide sets:



Being the source grammar right-linear, all the guide sets on the exit arrows of machine M_S coincide with the terminator $\dashv$. This machine does not have any conflicts between guide sets, of course.

Now one can notice that the four rules that expand nonterminal S, basically output something at the rule end, which corresponds to having a translation grammar written in the postfix form. Thus, here is the same machine (all guide sets are omitted) with the output actions on the exit arrows (postfix from):



The pseudo-code that encodes this machine, obtained by examining each state and faithfully translating each arc, exit arrow and error case, is the following:

```
procedure $S$
if $cc == a$ then                                        // state $0_S$
    $cc = next$                                          // goto state $1_S$
    if $cc == \dashv$ then                               // state $1_S$
        write $(a)$                         // on exiting from state $1_S$
        return                                 // exit from state $1_S$
    else if $cc == z$ then                               // state $1_S$
        call $T$                                         // goto state $2_S$
        if $cc == \dashv$ then                           // state $2_S$
            write $(a)$                     // on exiting from state $2_S$
            return                             // exit from state $2_S$
        else $error$                                     // state $2_S$
    else $error$                                         // state $1_S$
else if $cc == b$ then                                   // state $0_S$
    $cc = next$                                          // goto state $3_S$
    if $cc == \dashv$ then                               // state $3_S$
        write $(b)$                         // on exiting from state $3_S$
        return                                 // exit from state $3_S$
    else if $cc == z$ then                               // state $3_S$
        call $T$                                         // goto state $4_S$
        if $cc == \dashv$ then                           // state $4_S$
            write $(b)$                     // on exiting from state $4_S$
            return                             // exit from state $4_S$
        else $error$                                     // state $4_S$
    else $error$                                         // state $3_S$
else $error$                                             // state $0_S$
```

Algorithm: Recursive descent translator (partial), systematically encoded from the transducer machine.

Of course, this more detailed machine version is equivalent to the straightforward previous one. Just try to apply known code transformation techniques, to factorize identical code pieces and to group error cases.

The reader may notice that, if the machine is not (completely) written in the postfix form (which is unnecessary for recursive descent), but has some write actions associated with the arcs, then the final states 2_S and 4_S can be merged, since in the source machine they are undistinguishable. Here is this form:

$$\begin{tikzcd}
 & & S \arrow[d] & & \\
\textbf{write}\ (a) & 1_S \arrow[l] \arrow[dr, bend right, "T"', "\textbf{write}\ (a)"] & 0_S \arrow[l, "a"'] \arrow[r, "b"] & 3_S \arrow[r] \arrow[dl, bend left, "T", "\textbf{write}\ (b)"'] & \textbf{write}\ (b) \\
 & & 2_S\ 4_S \arrow[d] & & \\
 & & {} & &
\end{tikzcd}$$

Encoding this transducer machine version would produce an equivalent syntactic translation procedure, of course.

A final remark: in the source machine, also the two final states 1_S and 3_S are undistinguishable (but they are still distinguishable from states 2_S and 4_S), since both have a T-arc directed to undistinguishable states (namely 2_S and 4_S), and therefore, *though only in the source machine*, they could be merged. Yet they *may not be merged in the translator machine*, since then it would be unclear what or when to write, which indeed would be a form of nondeterminism !

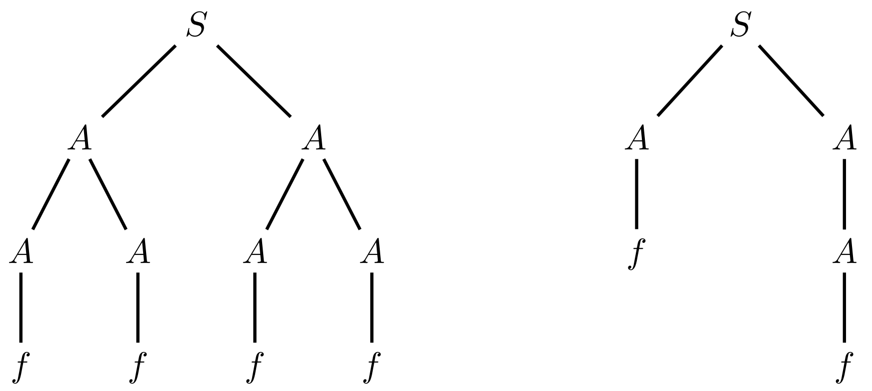
2. A binary tree is said to be *triangular* if and only if both these conditions are satisfied:

   - all its leaves are at the same *depth*, where the depth of a node is the length of the path from the root to that node
   - the number of leaves is equal to 2^{height}, where the height of the tree is the length of the path from the root to the deepest non-leaf node

   We encode binary trees through syntax trees generated by this grammar (axiom S):

$$\begin{cases} 1\colon\ S \rightarrow A\ A \\ 2\colon\ A \rightarrow A\ A \\ 3\colon\ A \rightarrow A \\ 4\colon\ A \rightarrow f \end{cases}$$

   The figure below shows a triangular tree (left) and a non-triangular tree (right).



   Notice that, according to the above definitions, the tree on the right side of the figure has one leaf at depth 2 and one leaf at depth 3, and that the height of both trees is 2.

   We intend to compute, in a boolean attribute *tri* of the root node of the syntax tree, the property of the tree being triangular. The attributes to use are already assigned, see the table on the next page. Other attributes are unnecessary and should not be added.

   Answer the following questions:

   (a) Write an attribute grammar (in the table prepared on the next page), based on the above syntax, that defines the attribute values. It is required to use only the attributes illustrated in the table on the next page.

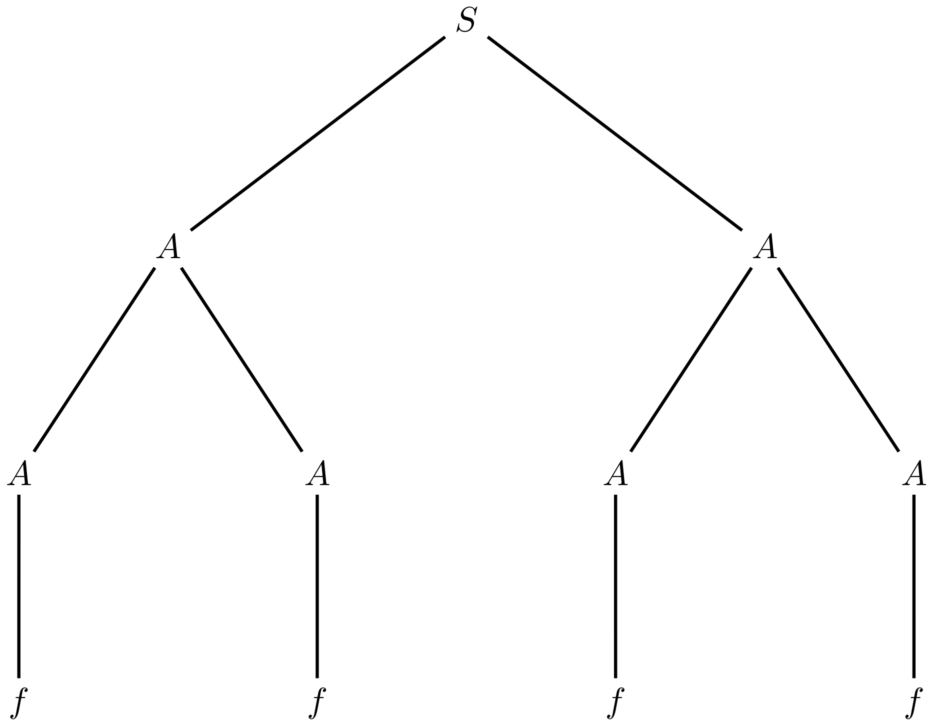
   (b) Decorate the two sample syntax trees with the appropriate attribute values. Use the two syntax trees prepared on the next page.

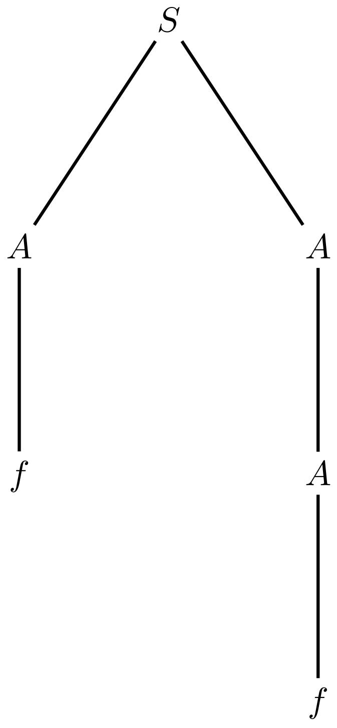
   (c) (optional) Determine if the attribute grammar is of type one-sweep and if it satisfies the L condition, and reasonably justify your answers (do not use generic or tautological sentences).

attributes assigned to be used

type	*name*	*domain*	*(non)term.*	*meaning*
right	d	integer	A	depth of the node from root
left	*nol*	integer	S, A	number of leaves of the subtree rooted at the node
left	h	integer	S, A	height of the subtree rooted at the node
left	*mind, maxd*	integer	S, A	minimal (respectively, maximal) depth of a leaf of the subtree rooted at the node; notice that for the left sample tree it holds $mind = maxd = 3$, while for the right one it holds $mind = 2$ and $maxd = 3$
left	*tri*	boolean	S	true if the tree is triangular, otherwise false

syntax trees to be decorated - question (b)





attribute grammar to write - question (a)

#	*syntax*	*semantics*
1:	$S_0 \rightarrow A_1\ A_2$	
2:	$A_0 \rightarrow A_1\ A_2$	
3:	$A_0 \rightarrow A_1$	
4:	$A_0 \rightarrow f$	

Solution

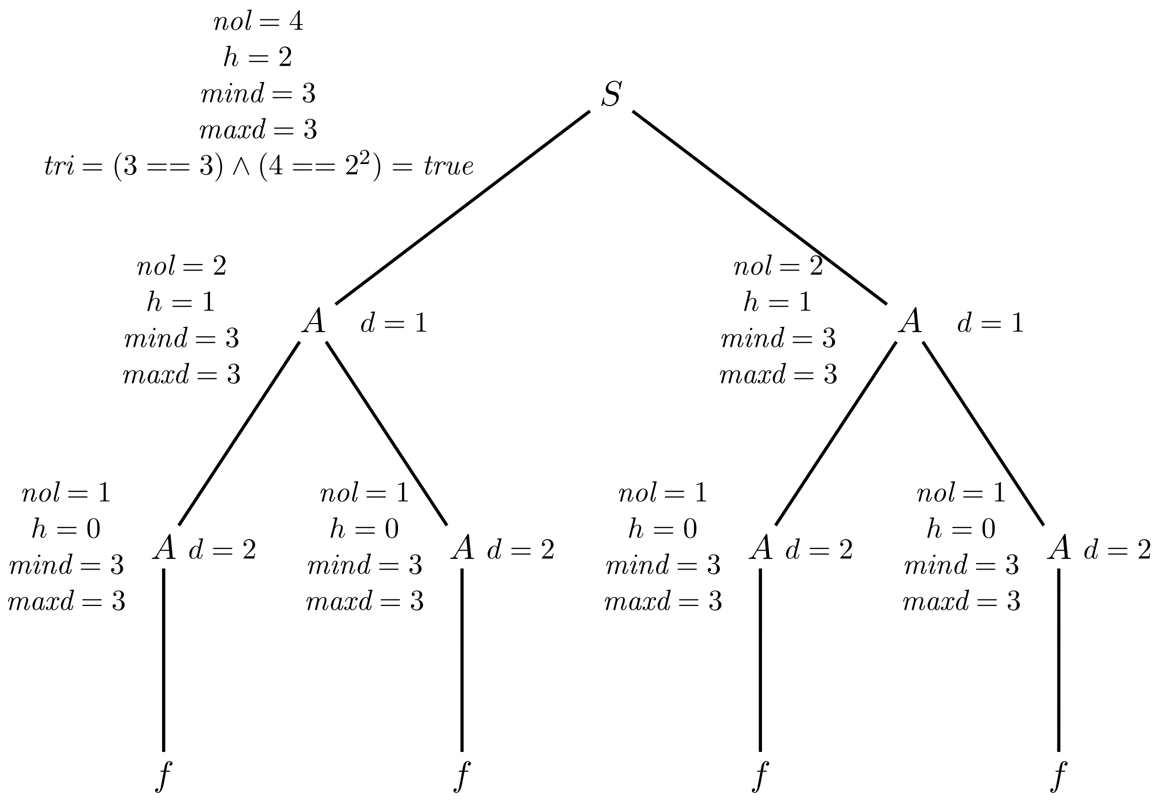
(a) Here is a working attribute grammar, which uses only the proposed attributes:

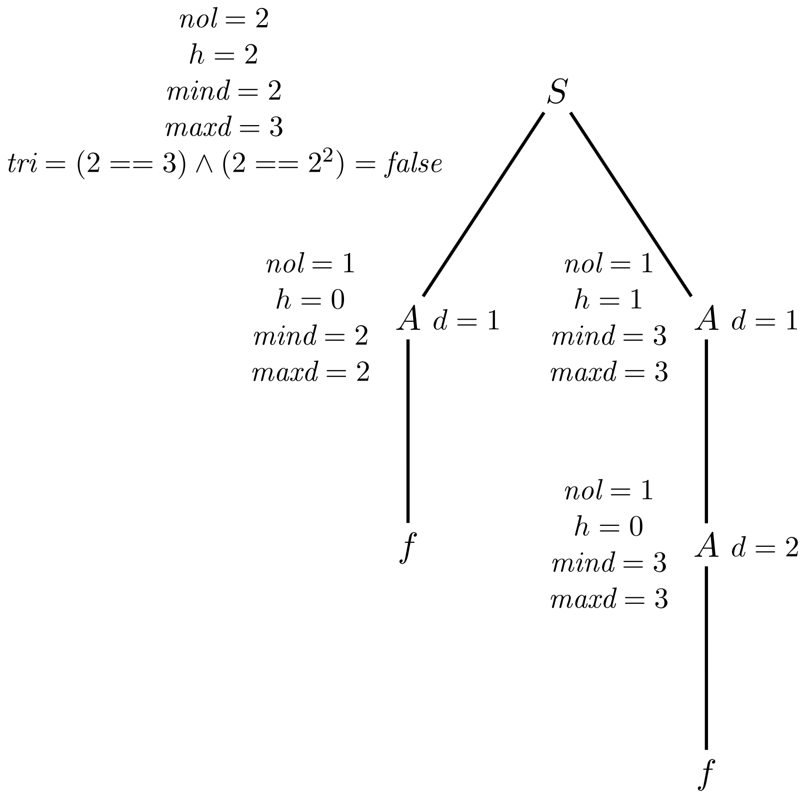
#	*syntax*	*semantics*
1:	$S_0 \rightarrow A_1\ A_2$	$d_1,\ d_2 = 1$ $nol_0 = nol_1 + nol_2$ $h_0 = \max(h_1,\ h_2) + 1$ $mind_0 = \min(mind_1,\ mind_2)$ $maxd_0 = \max(maxd_1,\ maxd_2)$ $tri_0 = (mind_0 == maxd_0) \wedge \left(nol_0 == 2^{h_0}\right)$
2:	$A_0 \rightarrow A_1\ A_2$	$d_1,\ d_2 = d_0 + 1$ $nol_0 = nol_1 + nol_2$ $h_0 = \max(h_1,\ h_2) + 1$ $mind_0 = \min(mind_1,\ mind_2)$ $maxd_0 = \max(maxd_1,\ maxd_2)$
3:	$A_0 \rightarrow A_1$	$d_1 = d_0 + 1$ $nol_0 = nol_1$ $h_0 = h_1 + 1$ $mind_0 = mind_1$ $maxd_0 = maxd_1$
4:	$A_0 \rightarrow f$	$nol_0 = 1$ $h_0 = 0$ $mind_0,\ maxd_0 = d_0 + 1$

Attributes: *d* (length from root, integer, inherited) is initialized at the top and is incremented top-down; *nol* (number of subtree leaves, integer, synthesized) is initialized at the bottom and is accumulated bottom-up; *h* (max length from leaf, integer, synthesized) is initialized at the bottom and is maximized and incremented bottom-up; *mind* and *maxd* (min and max lengths from root to leaf, both integer and synthesized) are initialized at the bottom and are minimized and maximized bottom-up, respectively; finally, the boolean attribute *tri* (triangular tree, synthesized) is computed only once directly at the root node, since it is associated exclusively with the grammar axiom.

The attribute grammar G is acyclic by construction, and it is reasonably correct and equivalent to the specifications. The computation of each attribute immediately reflects its definition. Notice that a dependence between left and right attributes occurs only at the tree bottom, and that the triangularity property is computed as logical conjunction of the two conditions specified in the text.

(b) Here are the syntax trees decorated according to the semantic functions:





(c) As already observed at point (a), the attribute grammar is acyclic and reasonably correct according to the specifications. Therefore it is computable, as already shown by simulation in a couple of different cases at point (b).

Furthermore this attribute grammar is of type one-sweep: in fact, all the attributes are synthesized, except attribute d, which is inherited and depends only on itself from top to bottom, and which contributes to the other attributes — particularly to *mind* and *maxd* — only at the tree bottom (i.e., in the rule 4) and nowhere else. Therefore all the attributes can be computed first top-down (attribute d) and then bottom-up (all the others).

Concerning the L condition, notice that there are not any attributes of a child node that depend on attributes of its brother nodes in the same rule. Therefore, this attribute grammar immediately satisfies the L condition, since all the sibling graphs of its rules are empty. Thus, the brother nodes in a rule can be semantically evaluated in any order, not necessarily from left to right.

Anyway, the semantic evaluation of this attribute grammar cannot be integrated with deterministic parsing, as the syntactic support is ambiguous (see the two-sided recursive rule $A \to A\ A$) and thus deterministic parsing is impossible.