FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 23 September 2015 - Part Theory

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

NAME:

MATRICOLA: SIGNATURE:

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 regular expression R, over the three-letter alphabet $\Sigma = \{ a, b, c \}$:

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

Answer the following questions:

(a) Find a deterministic automaton A equivalent to the regular expression R, and if necessary minimize it.

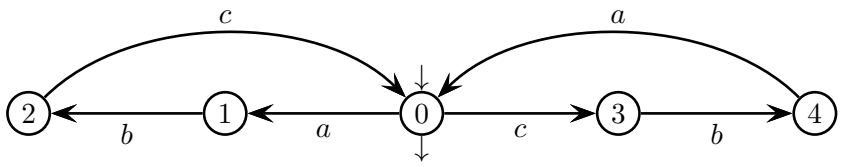
(b) In the minimal automaton A costructed before, project letter c onto the empty string ε, then eliminate the spontaneous transitions, and obtain an automaton B (possibly nondeterministic) that recognizes the projected language.

(c) By using some systematic method (not just your intuition), find a regular expression R' for the language $\neg L\,(B)$, that is, for the complement of the language accepted by the automaton B constructed before.

(d) (optional) Say if the original regular expression R is ambiguous or not, and justify your answer.

Solution

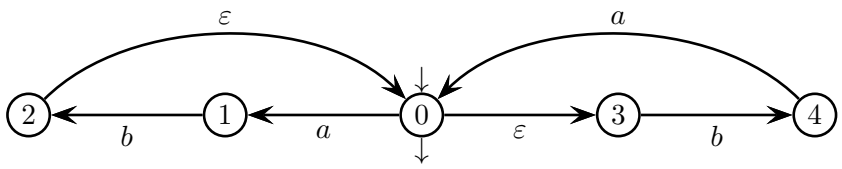
(a) Notice that the regular expression R is actually equivalent to the simpler regular expression $(\, a\, b\, c \mid c\, b\, a\,)^*$, whose language consists of all and only the free repetitions of substrings $a\,b\,c$ and $c\,b\,a$. It is not difficult to find a deterministic automaton A by intuition, using such an equivalent form. Here is a solution:



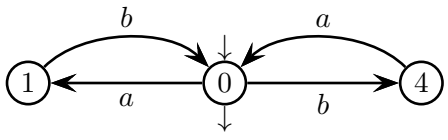
To proceed systematically, we can use the Berry-Sethi algorithm, which yields a deterministic automaton identical to the one above (so it is not shown here).

Automaton A is minimal. The final state 0 is distinguishable from all the non-final states. About the non-final states: 2 is distinguishable from both 3 and 4 (different outgoing labels); 4 is distinguishable from 1 (same reason as before); finally 1 and 3 are distinguishable as, though they have the same outgoing label b, their next states are 2 and 4, which have already been found to be distinguishable. Since the states are all distinguishable, automaton A is minimal.

(b) Automaton B after projecting letter c onto the empty string ε:



To obtain automaton B, we must cut the spontaneous transitions. This is quite easy, if we replicate the b-arc entering state 2 and the b-arc leaving state 3, and then we cancel the useless states (2 becomes not post-accessible and 3 becomes not accessible). Here is automaton B:

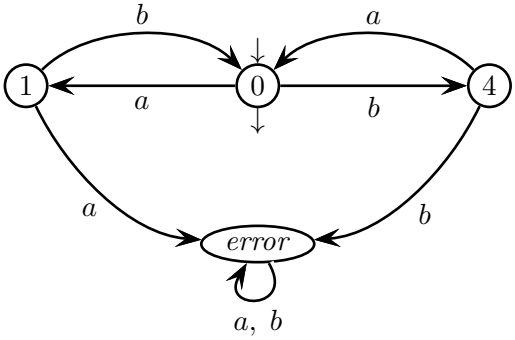


Notice that automaton B is deterministic. It is also evidently minimal.

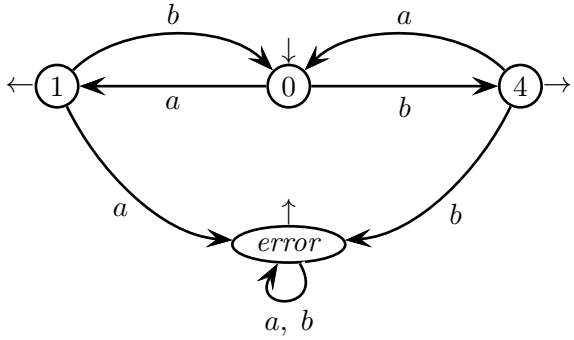
(c) Since the automaton B found before is already deterministic, we can apply the complement construction to it. Of course, the alphabet is $\Sigma = \{\, a,\, b,\, c\,\}$. In order to speed up the construction, we can however notice that automaton B only has transitions with letters a and b. Therefore its complement automaton will surely recognize any string that contains at least one letter c, i.e., all the strings generated by the r.e. $\Sigma^*\, c\, \Sigma^*$, besides recognizing also the strings that do not contain any letter c but that are not accepted by B. Hence we can assume that

the alphabet is restricted to two letters, i.e., $\{ a, b \}$, apply to B the complement construction and then, after finding a r.e. equivalent to automaton $\neg B$, unite the term $\Sigma^* c \Sigma^*$ to such a r.e., eventually obtaining the final r.e. R'.

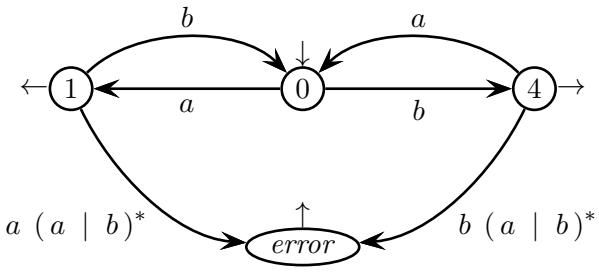
Automaton B completed with the error state (with respect to the restricted alphabet $\{ a, b \}$):



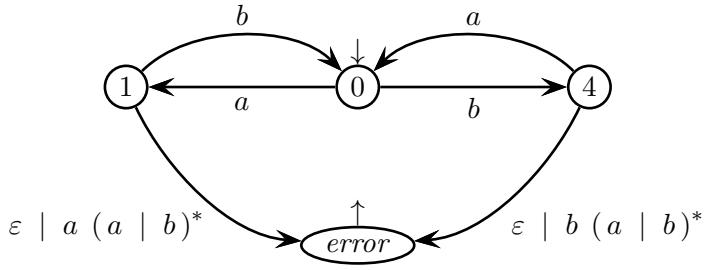
After complementing (switch final and non-final states):



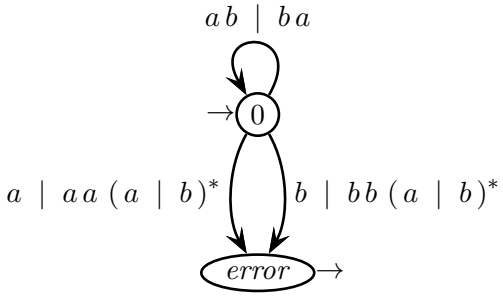
Now we can apply the *BMC* method (node elimination), with some intuitive shortcut to speed it up. First remove the self-loop on the (former) error state:



Then make states 1 and 4 non-final:



Now eliminate states 1 and 4:



Conclusion, after uniting the term $\Sigma^*\, c\, \Sigma^*$:

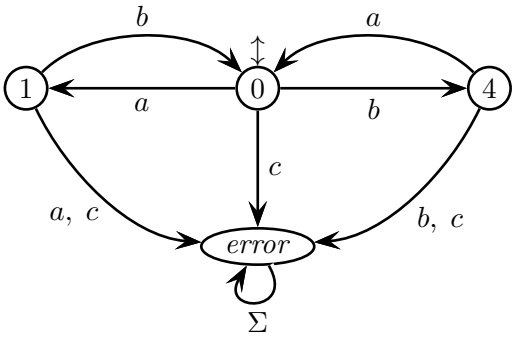
$$R' = (a\,b \mid b\,a)^* \left(\left(a \mid a\,a\,(a \mid b)^* \right) \mid \left(b \mid b\,b\,(a \mid b)^* \right) \right) \mid \Sigma^*\, c\, \Sigma^*$$

and with some simplification:

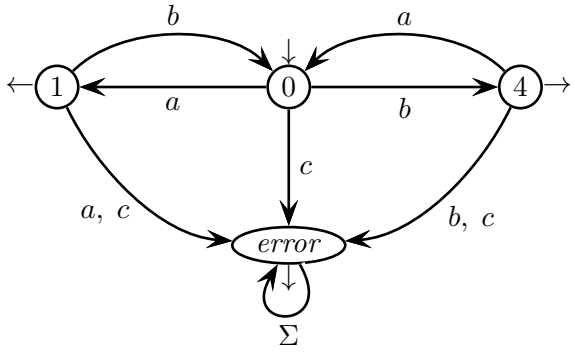
$$R' = (a\,b \mid b\,a)^* \left(a \mid a\,a\,(a \mid b)^* \mid b \mid b\,b\,(a \mid b)^* \right) \mid \Sigma^*\, c\, \Sigma^*$$

Of course, alternatively we can also apply the same procedure using the original three-letter alphabet.

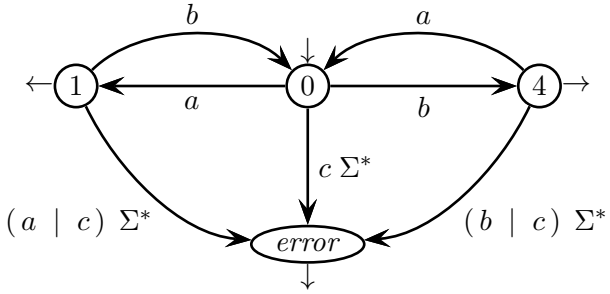
Automaton B completed with the error state (with respect to the original three-letter alphabet Σ):

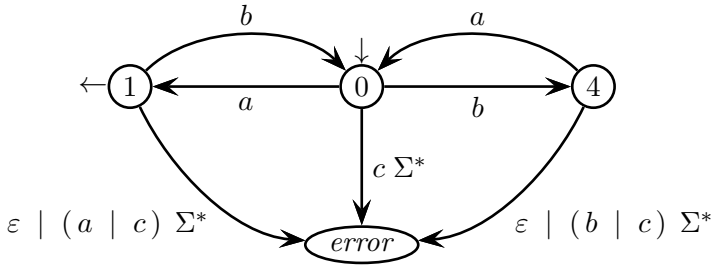


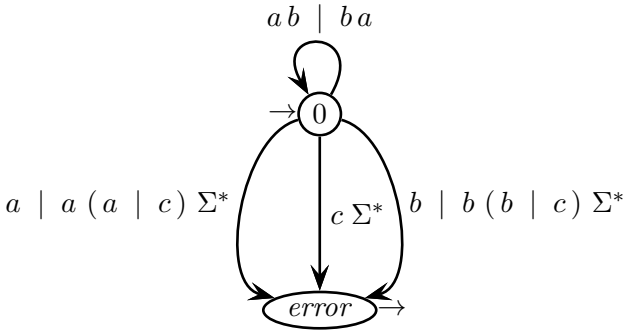
After complementing (switch final and non-final states):



Then basically the same passages as before:







And eventually:

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

Of course, this second version of R' is equivalent to the previous one.

(d) The regular expression R is ambiguous. The empty string $\varepsilon \in L(R)$ can be derived in at least two different ways:

$$R = \big(\ (a\ b\ c)^{*=1}\ \ (c\ b\ a)^{*=1}\ \big)^{*=0} \Rightarrow \varepsilon$$

or

$$R = \big(\ (a\ b\ c)^{*=0}\ \ (c\ b\ a)^{*=0}\ \big)^{*=1} \Rightarrow \varepsilon\ \varepsilon = \varepsilon$$

Notice that the number of iterations of the three stars is different in the two derivations, which therefore correspond to different choices.

Actually the empty string has an infinite degree of ambiguity: for instance the outer star is iterated 0 times and the inner stars are iterated any number of times, yet the result will always be the empty string ε.

Notice that the ambiguity of R is not immediately recognizable by numbering the letters and then examining whether a string is generated with two different numberings (this is a sufficient condition for ambiguity, not a necessary one), but by observing that the same string (e.g., $a\,b\,c$) can be obtained by iterating the star operators in different ways.

2 Free Grammars and Pushdown Automata 20%

1. Define a free grammar (in *BNF* form) for each of the following three subsets of the Dyck language (with one parenthesis type). Simple solutions are preferred especially for what concerns the number of nonterminal symbols and rules.

   (a) At every nesting level there are exactly two nested parenthesis structures.
   Samples:

   ε $\quad$ ( ) (( ) ( )) $\quad$ (( ) (( ) ( ))) ( )

   Draw the syntax tree for string: ( ) (( ) ( ))

   (b) At every nesting level there is an even number of nested parenthesis structures.
   Samples:

   ε $\quad$ ( ) ( ) ( ) ( )

   Draw the syntax tree for string: ( ) (( ) ( ))

   (c) (optional) At every nesting level there is an odd number of nested parenthesis structures.
   Samples:

   ( ) $\quad$ ( ) ( ) ( ) $\quad$ ( ) ( ) (( ) (( )) ( ))

   Counterexamples:

   ε $\quad$ ( ) ( ) $\quad$ (( ) ( ))

   Draw the syntax tree for string: ( ) (( ) ( ) ( )) ( )

Solution

(a) Here it is (non-ambiguous):

$$\begin{cases} S & \to & (\ S\ )\ (\ S\ ) \\ S & \to & \varepsilon \end{cases}$$

(b) Here it is (non-ambiguous):

$$\begin{cases} S & \to & (\ S\ )\ (\ S\ )\ S \\ S & \to & \varepsilon \end{cases}$$

(c) Here it is (non-ambiguous, axiom S):

$$\begin{cases} S & \to & (\ Y\ )\ X \\ X & \to & (\ Y\ )\ (\ Y\ )\ X \\ X & \to & \varepsilon \\ Y & \to & (\ Y\ )\ X \\ Y & \to & \varepsilon \end{cases}$$

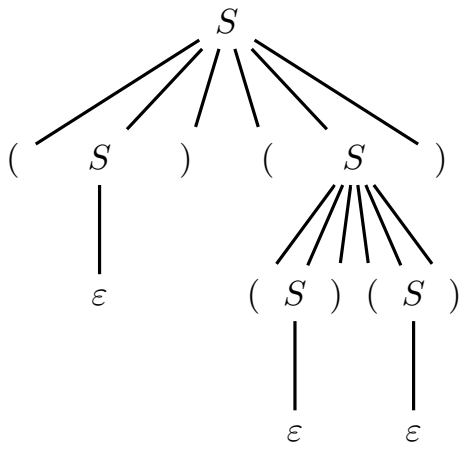
Ratio: S odd (non null) number, X = even number (including null), Y = odd or null number.

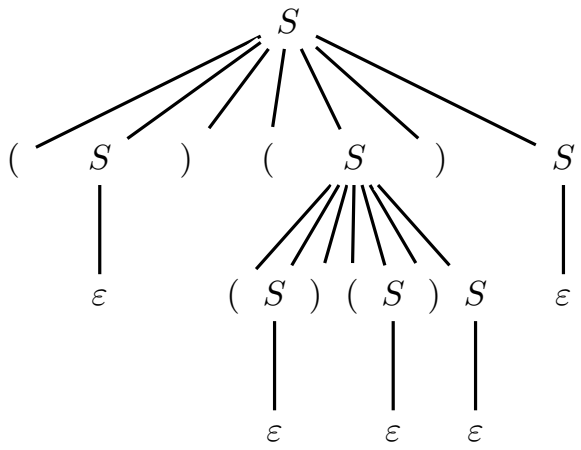
Nonterminal Y can be identified to the axiom S, thanks to the fact that the fourth rule is structurally identical to the axiomatic one, though by adding many more alternative rules because Y is nullable while S is not:

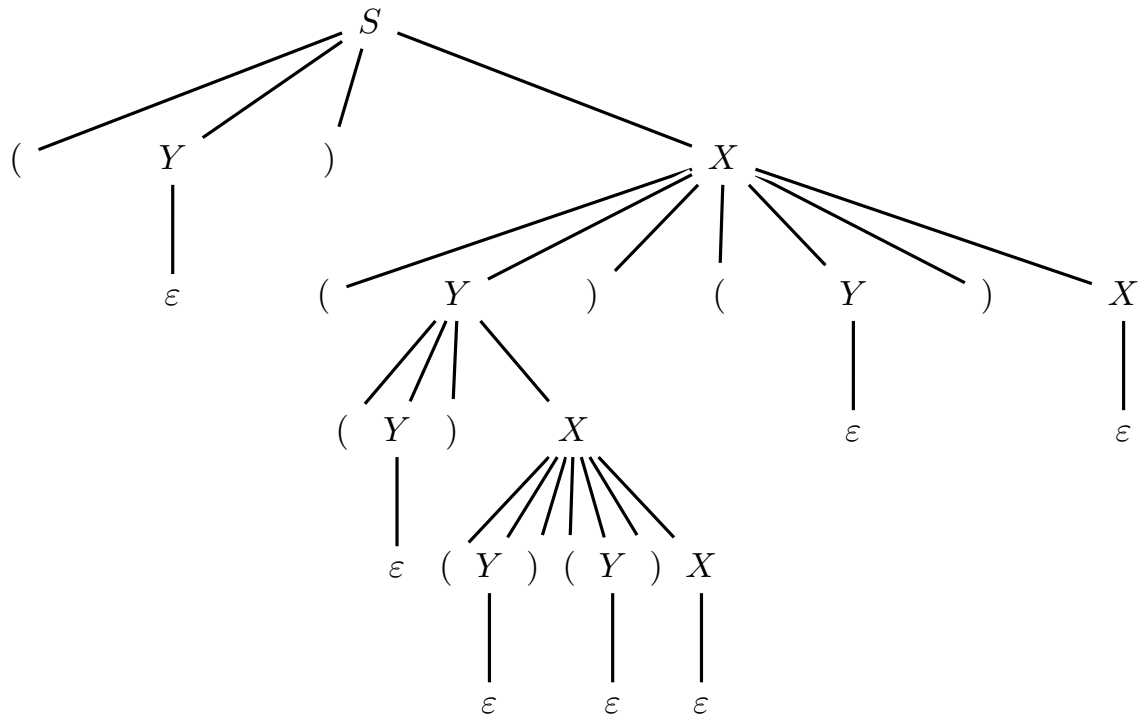
$$\begin{cases} S & \to & (\ S\ )\ X \\ S & \to & (\ )\ X \\ X & \to & (\ S\ )\ (\ S\ )\ X \\ X & \to & (\ )\ (\ S\ )\ X \\ X & \to & (\ S\ )\ (\ )\ X \\ X & \to & (\ )\ (\ )\ X \\ X & \to & \varepsilon \end{cases}$$

Still non-ambiguous.

Here are all the sample trees:







Similarly, we could draw a tree for the second version of grammar (c), with only two nonterminals instead of three.

2. Consider a language of integer arithmetic expressions that features the following structures and operations:

   - The integer variables are symbolized by the one terminal v.
   - The integer constants are sequences of one or more decimal digits 0, ..., 9, not starting with one or more digits 0.
   - There are the operations of addition + and multiplication $\times$, and their precedence is as usual (multiplication has a priority higher than addition does).
   - There are the operations of factorial $(a)\,!$ and of power $(b) * (e)$, e.g., $(v) * (2)$ means v^2 in traditional notation. The elements a, b and e may be an atom or a whole expression, and they are always enclosed in parentheses. Valid examples:

     $$(\,v\,)\,! \qquad (\,2 \times v + 1\,)\,! \qquad (\,v\,) * (\,v\,) \qquad (\,v\,) * (\,v + 1\,) \qquad (\,2 \times v\,) * (\,v\,)$$

   - Since the numerical value of the factorial and power expressions grows very quickly, it is forbidden that the argument a of a factorial and the exponent e of a power contain more factorials or powers. For instance, these expressions are forbidden:

     $$\big(\,(\,v\,)\,!\,\big)\,! \qquad \big(1 + (\,v\,) * (\,v\,)\,\big)\,! \qquad (\,2\,) * \big(\,(\,v\,)\,!\,\big) \qquad (\,2\,) * \big(\,(\,v\,) * (\,v\,)\,\big)$$

     but these are permitted (because factorials and powers may occur in the base of any power):

     $$\big(\,(\,v\,) * (\,v\,)\,\big) * (\,2\,) \qquad \big(\,(\,v\,)\,!\,\big) * (\,2\,)$$

   - The factorial and power operations have a priority higher than addition and multiplication do.
   - In addition to those mandatory for the factorial and power operations, there are parentheses (round open "(" and closed ")"), which can be used to change the conventional precedence of the various operations.

   Here is a longer valid expression:

   $$v + 2 \times (\,v + 1\,) * (\,v + v\,) + (\,2 + v \times v\,)\,!$$

   Write an extended (*EBNF*) grammar, not ambiguous, that generates the language sketched above, and draw the syntax tree of the sample long expression above.

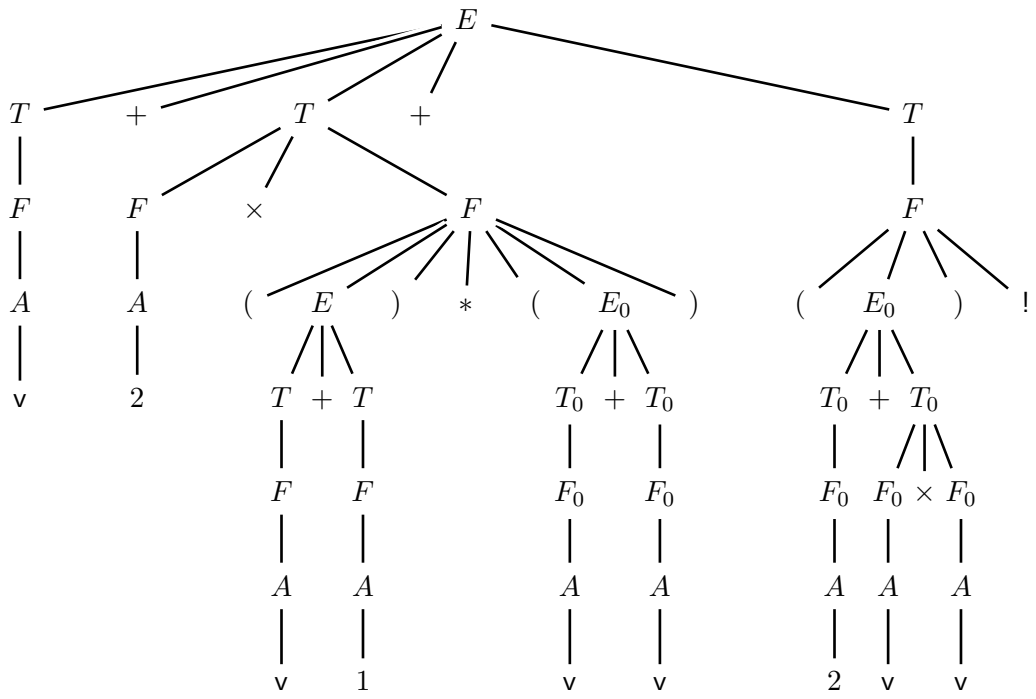
Solution

Here is the requested grammar (axiom E):

$$\begin{cases} E \;\to\; T\ (+\,T)^* \\ T \;\to\; F\ (\times\,F)^* \\ F \;\to\; \text{`(' } E_0 \text{ `)'!} \mid \text{`(' } E \text{ `)'} * \text{`(' } E_0 \text{ `)'} \mid \text{`(' } E \text{ `)'} \mid A \\ \hline E_0 \;\to\; T_0\ (+\,T_0)^* \\ T_0 \;\to\; F_0\ (\times\,F_0)^* \\ F_0 \;\to\; \text{`(' } E_0 \text{ `)'} \mid A \\ \hline A \;\to\; \mathsf{v} \mid \{\,1,\ldots,9\,\}\ \{\,0,\ldots,9\,\}^* \end{cases}$$

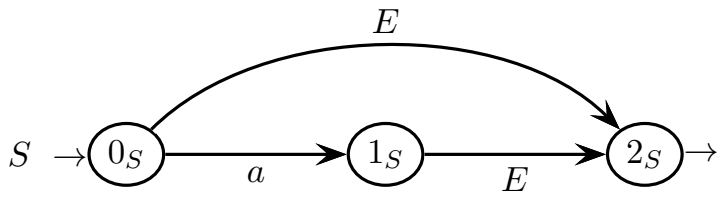
The grammar is non-ambiguous, as it is a sort of "duplication" of the well known standard *EBNF* grammar of the arithmetic expressions. The operations of factorial and power are introduced at the highest priority. The syntactic class E_0 models the (sub)expressions without factorial and power.

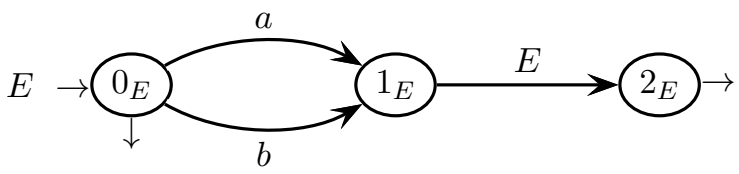
Syntax tree (since the grammar is *EBNF*, some nodes may have a variable arity):



3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the grammar below, over the two-letter alphabet $\Sigma = \{\ a,\ b\ \}$, represented as a machine net (axiom S):





Answer the following questions:

(a) Draw the complete *PCFG* of the grammar, say if the grammar is of type $ELL\,(1)$, or at least of type $ELL\,(k)$ for some $k \geq 2$, and justify your answer.

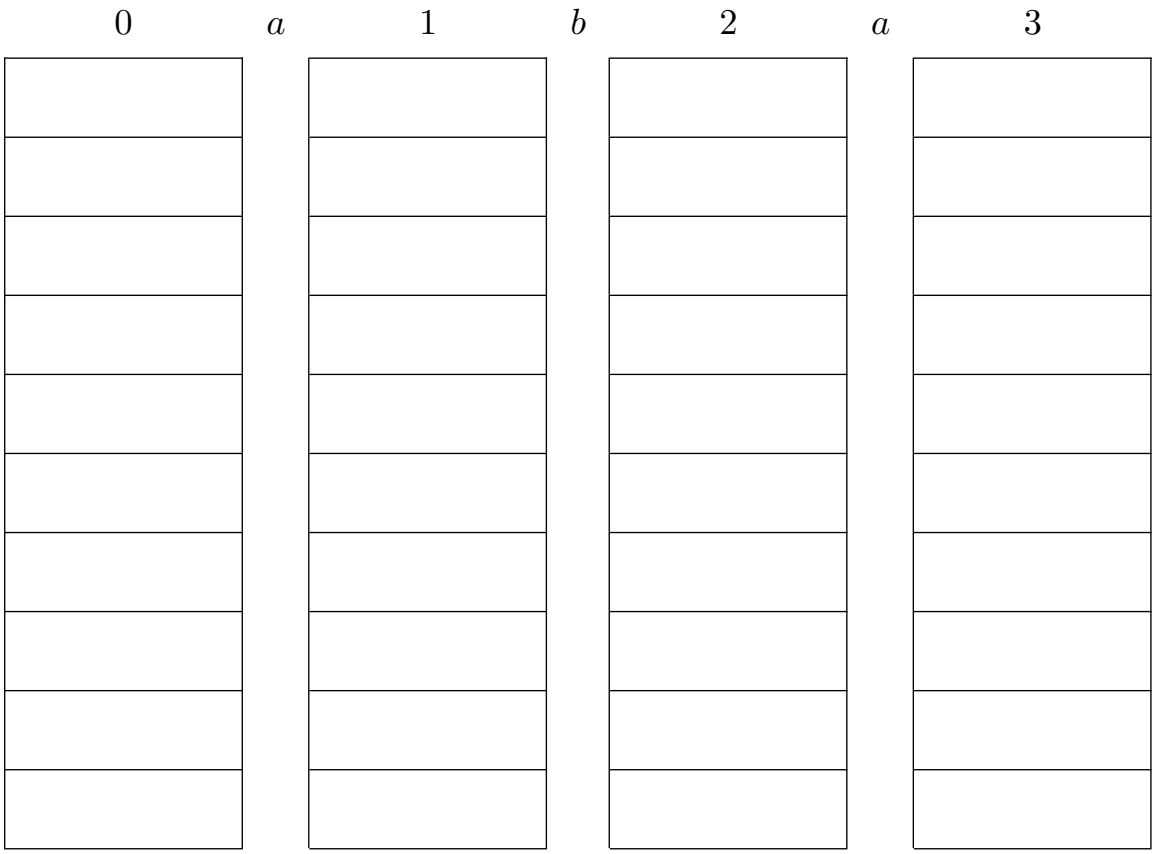
(b) Draw the complete pilot of the grammar, say if the grammar is of type $ELR\,(1)$ and justify your answer (list all the conflicts if any).

(c) Analyze the valid sample string $a\,b\,a$ by means of the Earley method, and show on the Earley vector why such a string is accepted. Furthermore, show on the vector each prefix of the sample string that would be accepted as well.

(d) (optional) Say if the language can be represented by a deterministic free grammar, and justify your answer.

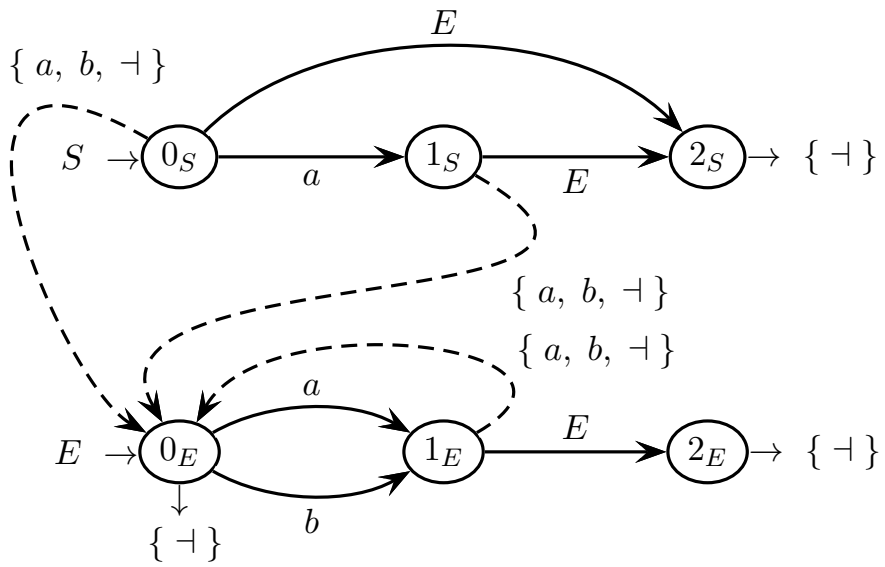
Earley vector of string $a\ b\ a$ (to be filled)

(the number of rows is not significant)



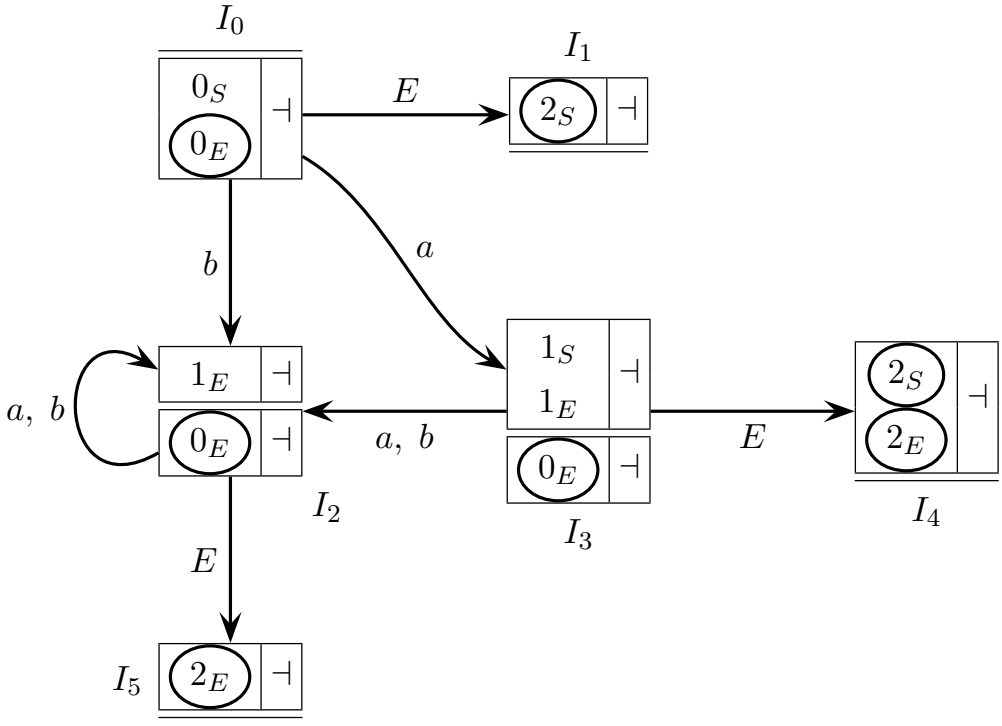
Solution

(a) For sure it will not be $ELL(k)$, for any $k \geq 1$ (ambiguous). Here is the *PCFG*:



There are overlapping guide sets on state 0_S. So the *PCFG* is not $ELL(1)$.

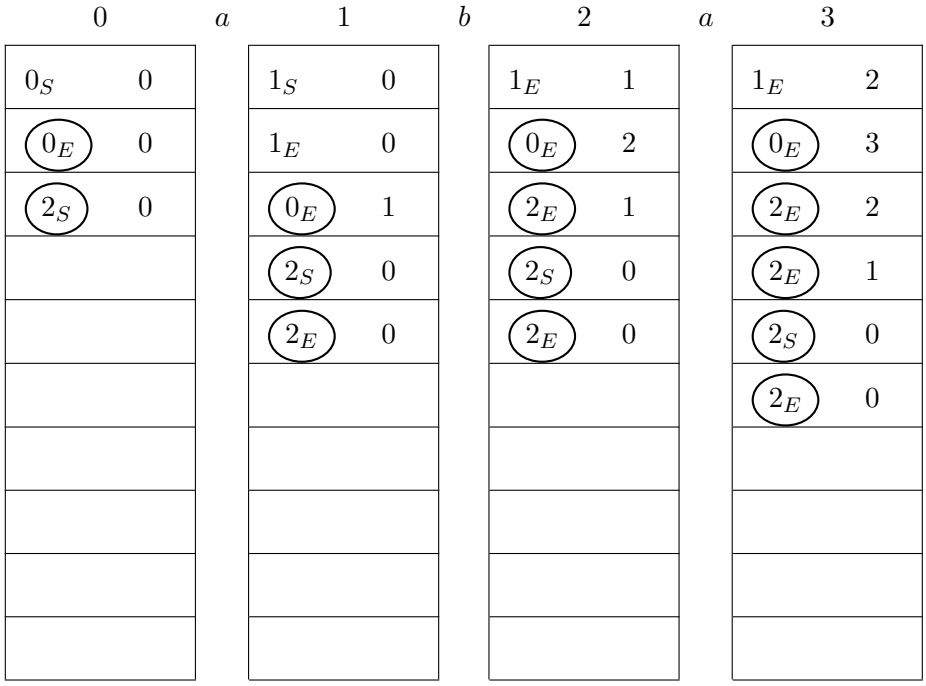
(b) For sure it will not be $ELR(1)$ (ambiguous). Here is the pilot:



There is a reduce-reduce conflict in the m-state I_4. By the way, notice the pilot does not have the *STP* (m-states I_3 and I_4 have a base with two candidates).

(c) Here is the filled Earley vector:

Earley vector of string $a\ b\ a$



The string $a\,b\,a$ is accepted. Also all the prefixes are accepted, including the empty string: ε, a and $a\,b$. In fact, the language is the universal one Σ^*, since the *EBNF* grammar has a rule $E \to a\ E \mid b\ E \mid \varepsilon$, and clearly $L(E) = \Sigma^*$.

(d) The grammar is ambiguous, so it may not be deterministic, irrespectively of the analysis method used. However, notice the grammar is basically right-linear: $S \to E \mid a\ E$ and $E \to a\ E \mid b\ E \mid \varepsilon$. Thus the language is regular. Therefore there is a deterministic grammar for this language: for instance, the right-linear grammar derived from the minimal deterministic finite automaton that recognizes the language, is surely of type $LL(1)$.

4 Language Translation and Semantic Analysis 20%

1. Consider the following translation τ, with identical source and destination alphabets $\Sigma = \Delta = \{\ a,\ b\ \}$ of two letters:

$$\tau\left(\ a^h\ b^k\ \right) = b^k\ a^h \qquad\qquad \text{with } h \geq k \geq 0$$

   For instance $\tau\left(\ a^3\ b^2\ \right) = b^2\ a^3$. Translation τ acts as a sort of commutator.

   Answer the following questions:

   (a) Write a *BNF* syntax translation scheme (or a *BNF* translation grammar) that computes translation τ, and briefly justify its correctness.

   (b) Draw the source and destination syntax trees of the sample translation shown above.

   (c) (optional) Say if the syntax transduction scheme found before is deterministic or not, and justify your answer. In the case it is not, argue if the translation τ is deterministic or not.

Solution

(a) Here is a *BNF* syntax translation scheme for translation τ (axiom S):

$$\begin{array}{lcl} S & \to & a\ S \\ S & \to & X \\ X & \to & a\ X\ b \\ X & \to & \varepsilon \end{array} \qquad \left| \qquad \begin{array}{lcl} S & \to & S\ a \\ S & \to & X \\ X & \to & b\ X\ a \\ X & \to & \varepsilon \end{array} \right.$$

As a *BNF* translation grammar, the relationship between source and destination is even more perspicuous:

$$\begin{array}{lcl} S & \to & \frac{a}{\varepsilon}\ S\ \frac{\varepsilon}{a} \\ S & \to & X \\ X & \to & \frac{a}{b}\ X\ \frac{b}{a} \\ X & \to & \frac{\varepsilon}{\varepsilon} \end{array}$$

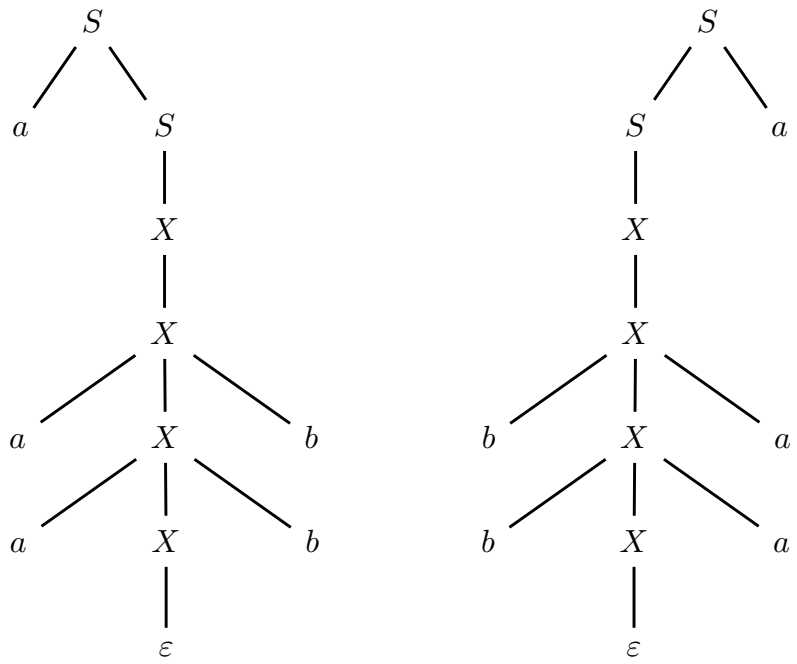
Clearly rule $X \to \frac{a}{b}\ X\ \frac{b}{a} \mid \frac{\varepsilon}{\varepsilon}$ translates $a^k\ b^k$ into $b^k\ a^k$. The rest of the grammar comes easy with a little intuition: just think that to obtain the source string $a^h\ b^k = a^{h-k}\ a^k\ b^k$, we need to append $h-k$ more letters a onto the left of $a^k\ b^k$, which letters will appear appended onto the right of the destination string $b^k\ a^h = b^k\ a^k\ a^{h-k}$. This is precisely what the rest of the grammar does.

A similar approach consists of pushing the additional $h-k$ letters a into the centre of the structure $a^k\ b^k$. In practice the roles of the nonterminals S and X are switched (but the axiom is still S). Here is this second version of the translation grammar (we omit the scheme form):

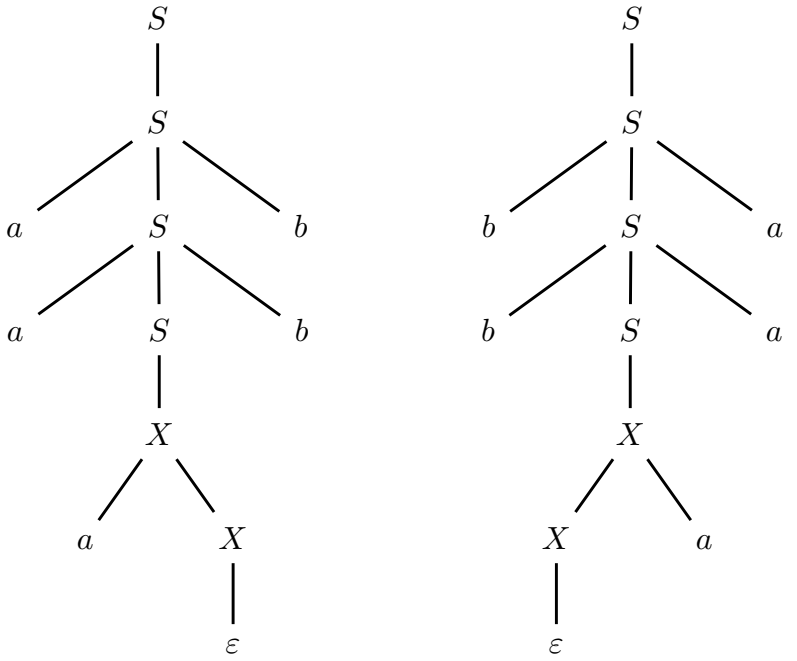
$$\begin{array}{lcl} S & \to & \frac{a}{b}\ S\ \frac{b}{a} \\ S & \to & X \\ X & \to & \frac{a}{\varepsilon}\ X\ \frac{\varepsilon}{a} \\ X & \to & \frac{\varepsilon}{\varepsilon} \end{array} \qquad \text{or} \quad X \to \frac{a}{a}\ X \quad \text{or} \quad X \to X\ \frac{a}{a}$$

Notice that the third rule can be written in a few different alternative ways, all equivalent to one another, since in practice this rule is right- (or left-) linear and it generates just one letter type (so it is insensible to commutations).

(b) The sample trees will help us understand how the (first) proposed syntax translation scheme works:



Here are instead the (separate) trees of the second translation grammar version:



The small changes to the destination tree if we use the other formulations of the third rule are quite obvious.

(c) The source language is well known not to be of type $LL(k)$ for any $k \geq 1$, so the proposed source grammar itself is surely not of type $LL(k)$ either, as well as any other possible equivalent source grammar.

To check if the proposed scheme is of type $LR(1)$ or not, one should put it into the postfix form (all the destination terminals should occur on the right end of the destination rules), as below:

$S \rightarrow a\ S$	$S \rightarrow S\ a$
$S \rightarrow X$	$S \rightarrow X$
$X \rightarrow Y\ X\ b$	$X \rightarrow Y\ X\ a$
$X \rightarrow \varepsilon$	$X \rightarrow \varepsilon$
$Y \rightarrow a$	$Y \rightarrow b$

and then one should draw the pilot of the source grammar and check if it is of type $LR(1)$. We leave this exercise to the reader.

2. Consider the following abstract syntactic support for a language of arithmetic expressions with addition and parentheses, where the atomic addends are symbolized by a terminal a (axiom S).

$$\begin{array}{lcl} S & \rightarrow & E \\ E & \rightarrow & E \ + \ E \\ E & \rightarrow & (\ E\ ) \\ E & \rightarrow & a \end{array}$$

For instance, a possible expression is $a + (a + a) + a$.

Starting from this syntactic support, define an attribute grammar that answers the following questions (use the various tables and trees prepared on the next pages):

(a) The *nesting level* of an addend is the number of parenthesis pairs around the addend. By using a synthesized attribute h, compute the *height* of the whole expression, i.e., 1 plus the maximal nesting level of all the atomic addends. In the sample expression $a + (a + a) + a$, the value of h is 2. Decorate the syntax tree of the sample expression above with the values of h.

(b) For each atomic addend a, compute the attribute nl that expresses the nesting level of a (see the definition above). The minimal value of nl is 0.

For instance, in the above sample expression, if we represent nl as a subscript of each atomic addend, i.e., a_{nl}, then its values can be visualized as follows:

$$a_0 + (a_1 + a_1) + a_0$$

Decorate the syntax tree of the sample expression above with the values of nl.

(c) (optional) For each atomic addend a, compute the attribute p that expresses the position (from the left) of a in the list of addends (atomic or not) at the same nesting level as a in a (sub)expression. The minimal value of p is 0.

For instance, in the above sample expression, if we represent p as a subscript of each atomic addend, i.e., a_p, then its values can be visualized as follows:

$$a_0 + (a_0 + a_1) + a_2$$

Decorate the syntax tree of the sample expression above with the values of p and of any other auxiliary attribute possibly used to compute p.

Hint: we suggest to use an auxiliary attribute *na*, of synthesized type, to count the number of addends at the same nesting level in a subexpression; the examinee may however choose any other solution (s)he prefers.

---

attributes assigned to be used (complete the specification)

type	*name*	*domain*	*(non)term.*	*meaning*
left	h	integer ≥ 1		height of the whole expression
	nl	integer ≥ 0		number of parenthesis pairs around an addend
	p	integer ≥ 0		position of an addend in the list of addends at the same nesting level in a (sub)expression

attributes to be added, if any (complete the specification)

type	*name*	*domain*	*(non)term.*	*meaning*
left	na	integer ≥ 1		number of addends at the same nesting level in a (sub)expression

Question (a):

#	*syntax*	*semantics* - question (a)
1:	$S_0 \rightarrow E_1$	
2:	$E_0 \rightarrow E_1 + E_2$	
3:	$E_0 \rightarrow (E_1)$	
4:	$E_0 \rightarrow a$	

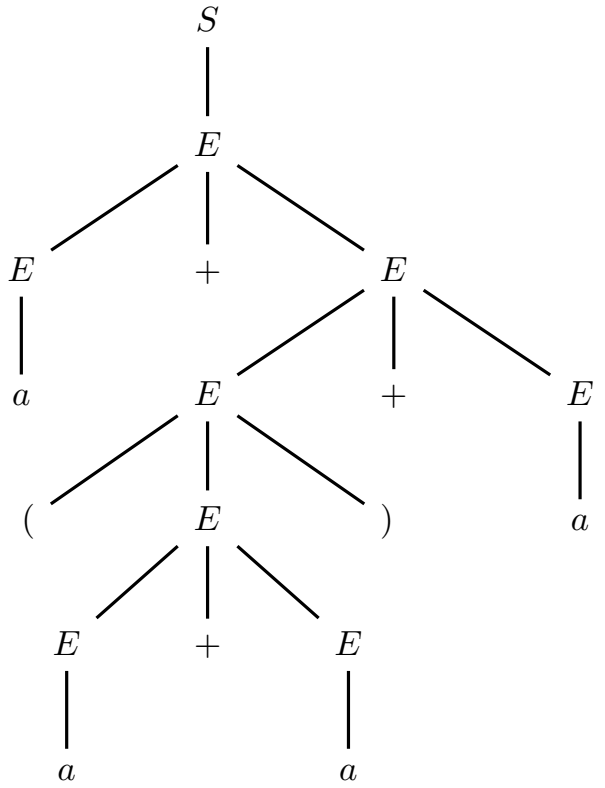
Question (b):

#	*syntax*	*semantics* - question (b)
1:	$S_0 \rightarrow E_1$	
2:	$E_0 \rightarrow E_1 + E_2$	
3:	$E_0 \rightarrow (E_1)$	
4:	$E_0 \rightarrow a$	

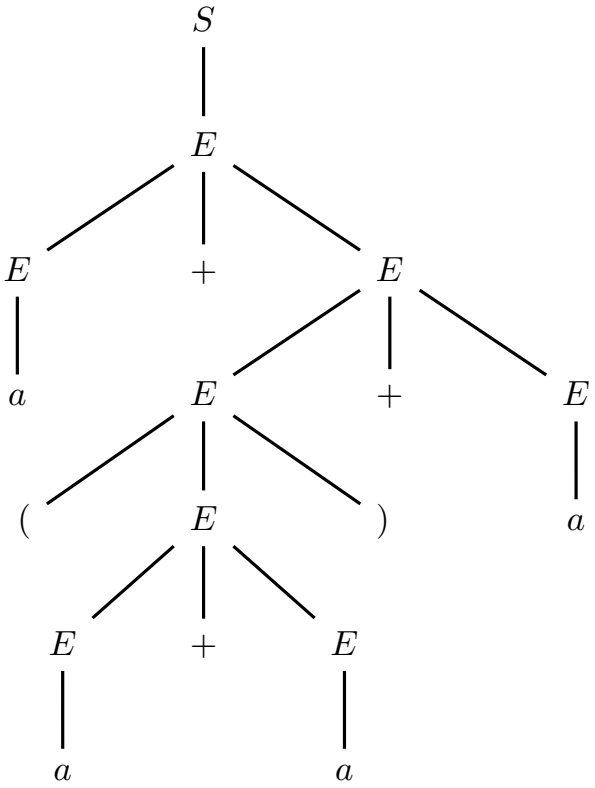
Question (c):

#	*syntax*	*semantics* - question (c)
1:	$S_0 \rightarrow E_1$	
2:	$E_0 \rightarrow E_1 + E_2$	
3:	$E_0 \rightarrow (E_1)$	
4:	$E_0 \rightarrow a$	

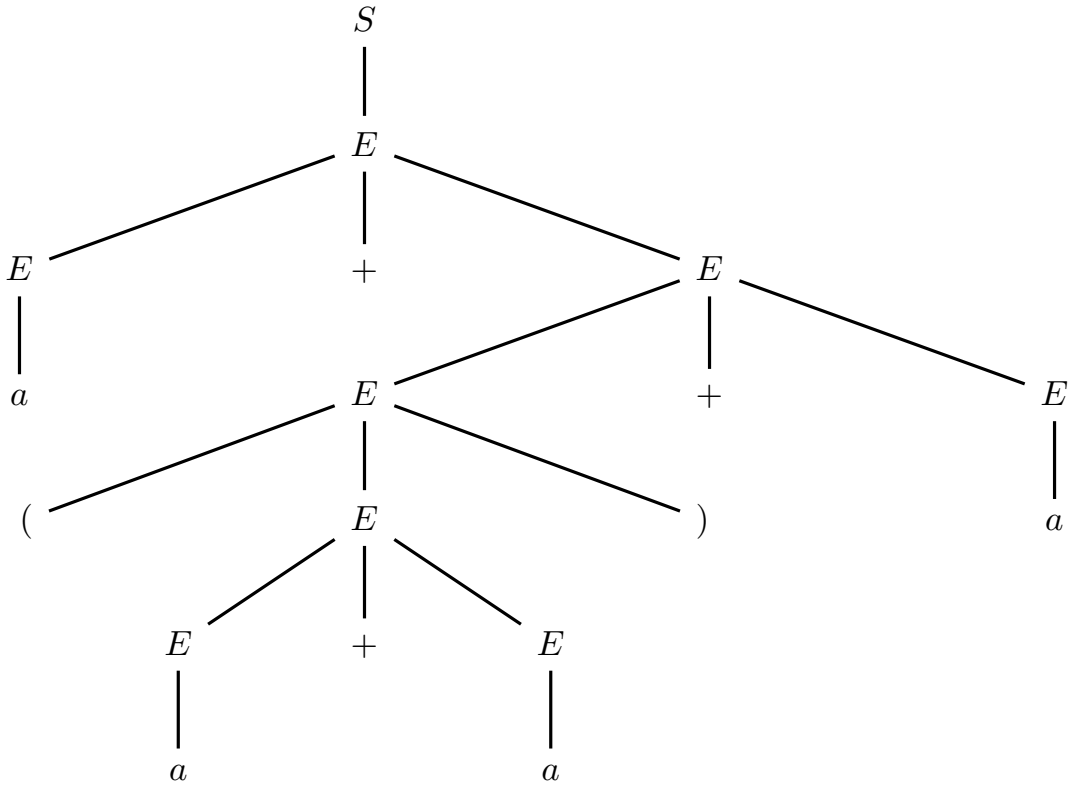
question (a)



question (b)



question (c)



Solution

(a) Attributes and semantics:

attributes assigned to be used

type	*name*	*domain*	*(non)term.*	*meaning*
left	h	integer ≥ 1	S, E	height of the whole expression

#	*syntax*	*semantics* - question (a)
1:	$S_0 \rightarrow E_1$	$h_0 := 1 + h_1$
2:	$E_0 \rightarrow E_1 + E_2$	$h_0 := \max(h_1, h_2)$
3:	$E_0 \rightarrow (E_1)$	$h_0 := 1 + h_1$
4:	$E_0 \rightarrow a$	$h_0 := 0$

The rationale of this attribute grammar is at all obvious.

(b) Attributes and semantics:

attributes assigned to be used

type	*name*	*domain*	*(non)term.*	*meaning*
right	nl	integer ≥ 0	E, a	number of parenthesis pairs around an addend

#	*syntax*	*semantics* - question (b)
1:	$S_0 \rightarrow E_1$	$nl_1 := 0$
2:	$E_0 \rightarrow E_1 + E_2$	$nl_1 := nl_0$ $nl_2 := nl_0$
3:	$E_0 \rightarrow (E_1)$	$nl_1 := 1 + nl_0$
4:	$E_0 \rightarrow a$	$nl_a := nl_0$

The rationale of this attribute grammar is also at all obvious.

Notice: we might simply avoid to propagate the attribute nl down to terminal a and stop the propagation at the immediate parent node E thereof.

(c) Attributes and semantics:

attributes assigned to be used

type	*name*	*domain*	*(non)term.*	*meaning*
right	p	integer ≥ 0	E, a	position of an addend in the list of addends at the same nesting level in a (sub)expression

attributes to be added, if any

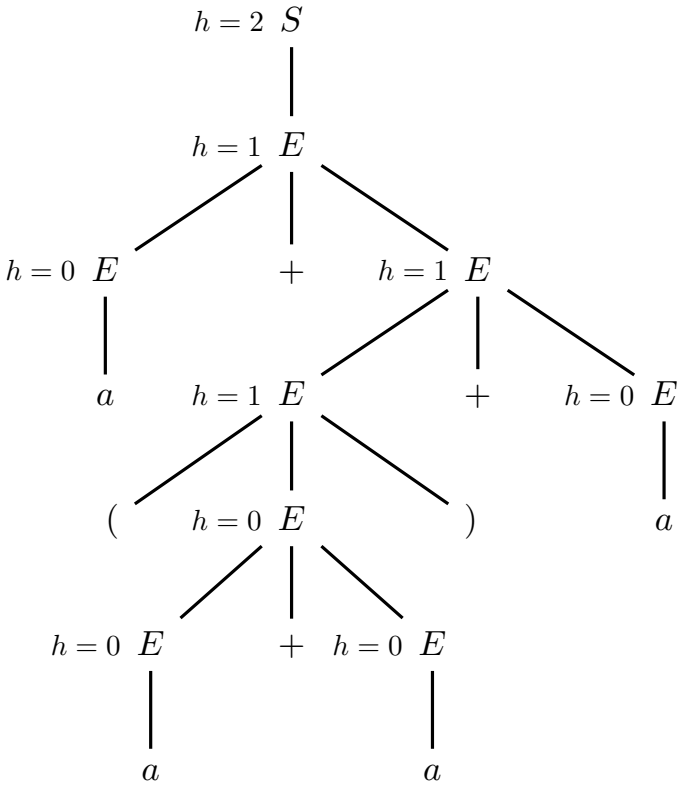
type	*name*	*domain*	*(non)term.*	*meaning*
left	na	integer ≥ 1	E	number of addends at the same nesting level in a (sub)expression

#	*syntax*	*semantics* - question (*c*)
1:	$S_0 \rightarrow E_1$	$p_1 := 0$
2:	$E_0 \rightarrow E_1 + E_2$	$p_1 := p_0$ $p_2 := p_1 + na_1$ $na_0 := na_1 + na_2$
3:	$E_0 \rightarrow (E_1)$	$p_1 := 0$ $na_0 := 1$
4:	$E_0 \rightarrow a$	$p_a := p_0$ $na_0 := 1$

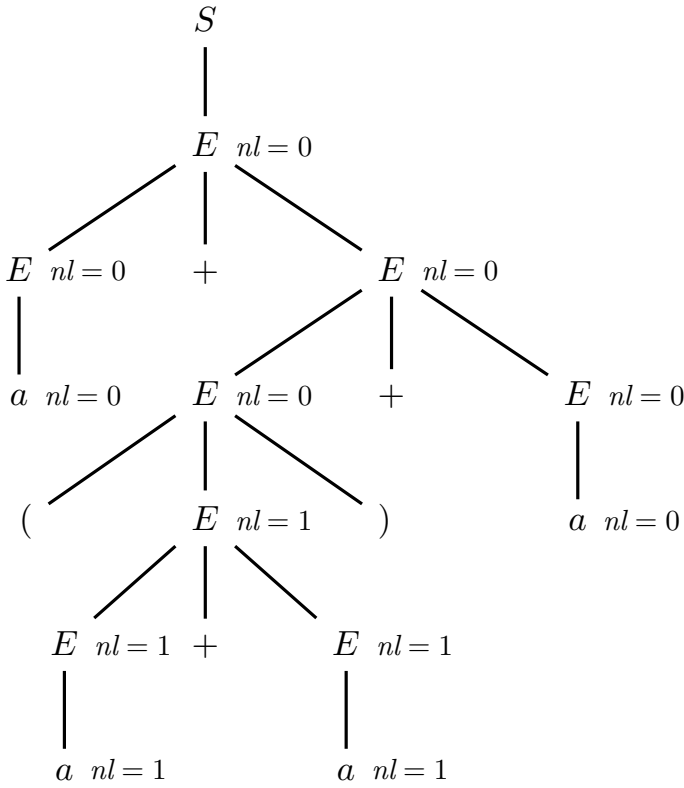
The rationale of this attribute grammar is a little less obvious: attribute na counts the total number of addends (atomic or not) at the same level inside of each parenthesized (sub)expression; attribute p expresses the positions (starting from 0 on the left) of the addends (atomic or not) inside of each parenthesized (sub)expression. Consider rule 3: whenever a new subexpression is met, attribute p is reinitialized as in the root node, so in practice a new subexpression resets the position numbering like the axiom S does for the whole expression; whenever a new subexpression is met, attribute nl is reinitialized as in a leaf node, so in practice a new subexpression is counted like an atom a.

Notice: as before, we might simply avoid to propagate the attribute p down to terminal a and stop the propagation at the immediate parent node E thereof.

question (a)



question (b)



question (c)