FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Thu 2 February 2017 - Part Theory

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

NAME (capital letters pls.):

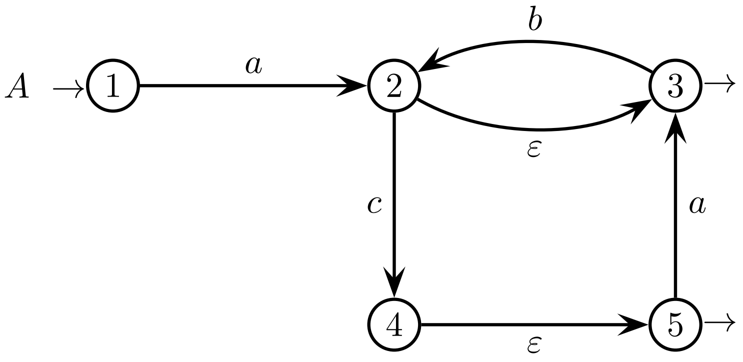
PERSON CODE: SIGNATURE:

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 finite-state nondeterministic automaton A below, over the three-letter alphabet $\Sigma = \{\ a,\ b,\ c\ \}$, with initial state 1 and final states 3, 5:



Answer the following questions:

(a) By using the Berry-Sethi method applied to automaton A, find an equivalent deterministic automaton A' and minimize it if necessary.

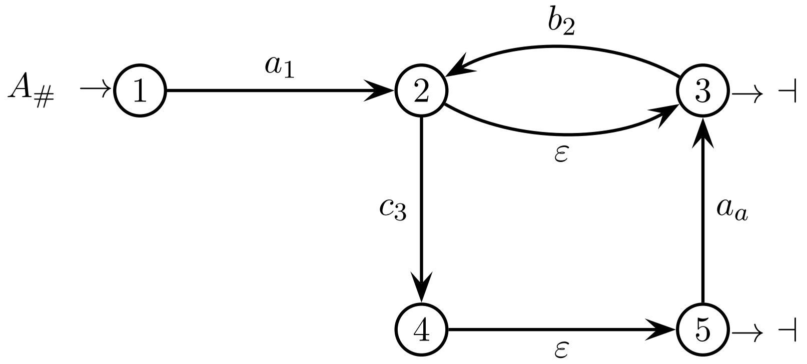
(b) By using the node elimination method (Brzozowsky) applied to automaton A or automaton A' (at choice), find an equivalent regular expression R.

(c) Examine if the regular language $L(A)$ is local or not, and justify your answer.

(d) (optional) By cutting off the spontaneous transitions (ε-transitions) from automaton A, obtain an equivalent automaton A'' (not necessarily deterministic), and test it with all the strings of length ≤ 3 that belong to language $L(A)$.

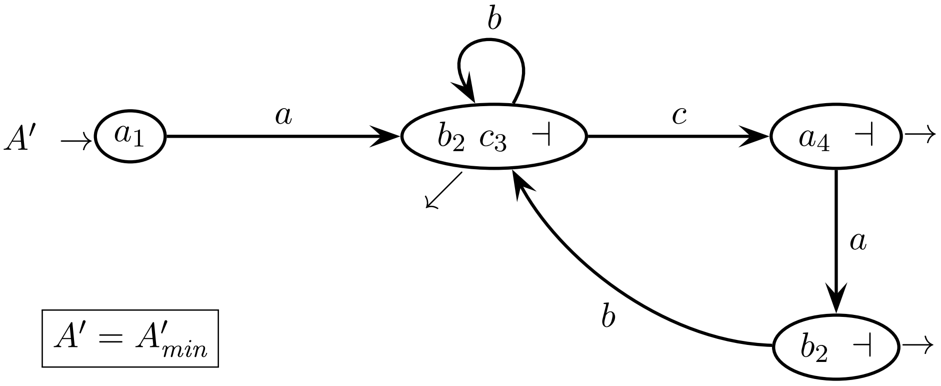
Solution

(a) First we design the numbered automaton $A_{\#}$, and we find its two sets of initials and followers:



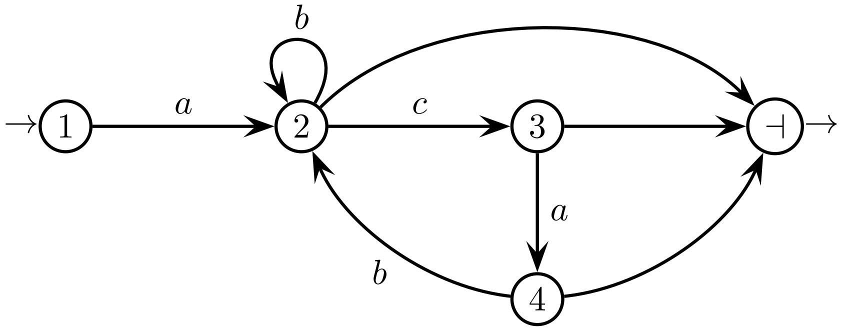
initials	a_1
terminals	**followers**
a_1	$b_2 \quad c_3 \quad \dashv$
b_2	$b_2 \quad c_3 \quad \dashv$
c_3	$a_4 \quad \dashv$
a_4	$b_2 \quad \dashv$

Then we design the *BS* automaton A that recognizes language $L(A)$:

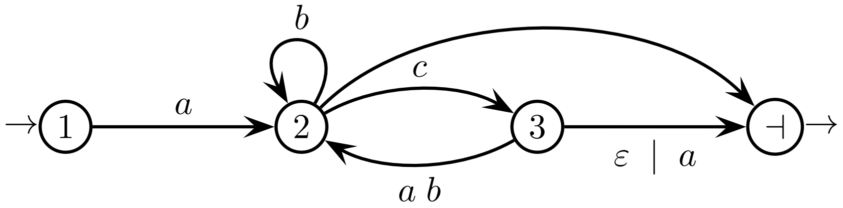


By construction, automaton A' is deterministic and in the clean form. It happens to be minimal, too (i.e., $A' = A'_{min}$): its three final states have different outgoing arc labels, so they are certainly distinguishable, and there is only one non-final state, so that in conclusion all four states are distinguishable.

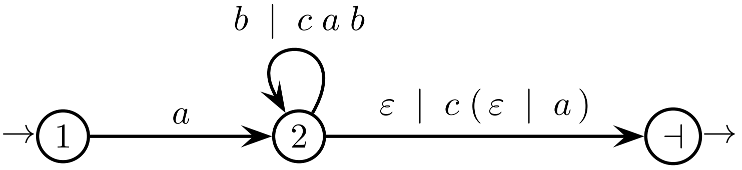
(b) We elect to eliminate the nodes of the deterministic automaton A', which has fewer states than the original automaton A (3 vs 4 states) and so may ease our work. First, to simplify, we change the state names and we put in a unique final state (notice state 1 is already unique and we do not need to modify it):



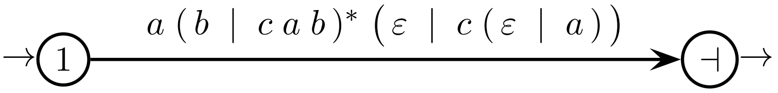
Eliminate node 4:



Eliminate node 3:



Eliminate node 2:



Here is the regular expression R in the standard form (union, concat., star):

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

or in an equivalent extended yet shorter (and likewise more readable) form:

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

where the square brackets [ ] indicate optionality, e.g., $[\,a\,] = \varepsilon \mid a$, etc.
Other forms exist, depending on the node elimination order. The form below:

$$R = a\,b^*\,\big[\,c\,(a\,b^+\,c)^*\,[\,a\,b^*\,]\big]$$

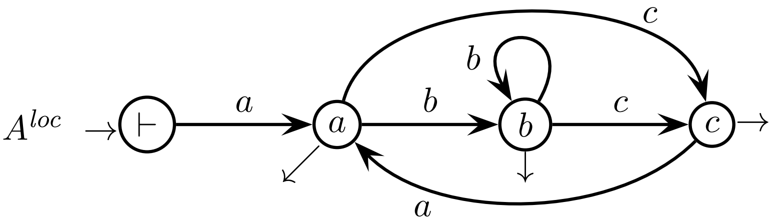
is equivalent to the previous ones, though it looks rather different from them.

(c) Examine automaton A (or A'): language $L(A)$ is not empty, its initials consist only of letter a and its finals of the entire alphabet Σ, and clearly it has both the digrams $a\,c$ and $c\,a$. Thus string $a\,c\,a\,c$ fulfills all the local constraints, yet it is rejected by automaton A, and in conclusion language $L(A)$ is not local. Coherently, notice that the regular expression R (in all the forms obtained before) is not linear, since some terminals occur twice or more times (namely at least a and c), nor could any other equivalent regular expression ever be linear.

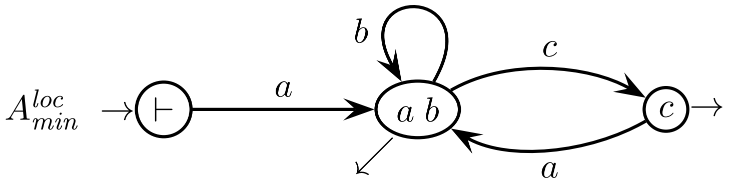
A different approach examines the local automaton derived from the local sets of language $L(A)$. The initials, finals and digrams of $L(A)$ are as follows:

$$Ini = \{\ a\ \} \qquad Fin = \Sigma \qquad Dig = \{\ a\,b,\ a\,c,\ b\,b,\ b\,c,\ c\,a\ \}$$

Here is the local automaton A^{loc} obtained from these sets, with four states:



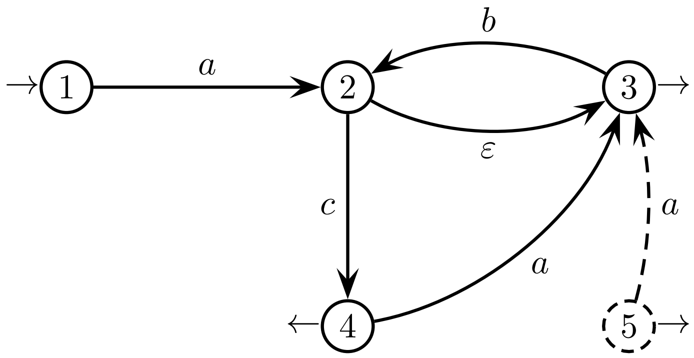
Automaton A^{loc} is clean and by construction deterministic. Clearly the states a and b are undistinguishable: both are final, go to states b and c with the same arc labels, and do not have any other outgoing arcs. The final state c has an outgoing arc with a label different from those of the other two final states, and the initial state $\vdash$ is non-final, so that both these states are distinguishable. Here is the minimized local automaton A^{loc}_{min}, with the states a and b unified:



Language $L(A)$ is local if and only if automata A (or $A' = A'_{min}$) and A^{loc} (or A^{loc}_{min}) are equivalent. Both automata A'_{min} and A^{loc}_{min} are minimal, and since the minimal automaton is unique, if they were equivalent then they would be isomorphic, i.e., identical up to state names. Yet they are not identical as they have a different number of states: 4 vs 3. Thus language $L(A)$ is not local.

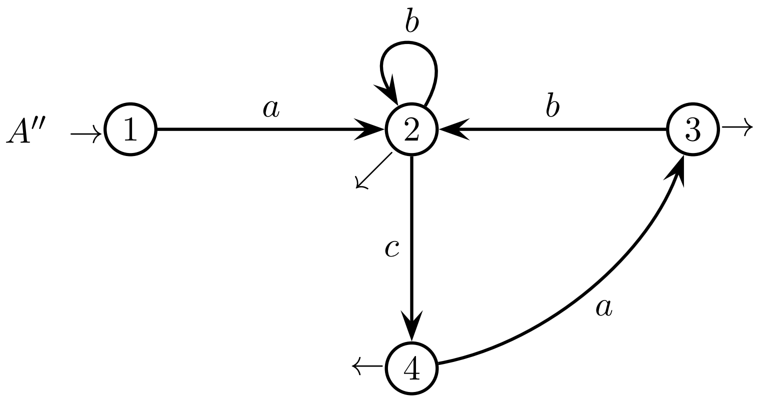
Said differently but equivalently, language $L(A)$ is not characterized by its local sets. In fact, as observed before, one can verify that automaton A^{loc} (or A^{loc}_{min}) accepts string $a\,c\,a\,c$, whereas automaton A (or A'_{min}) rejects it.

(d) Of the two spontaneous transitions of automaton A, the more immediate one to cut is $4 \xrightarrow{\varepsilon} 5$. So here is the result obtained by cutting it and back-propagating the arcs outgoing from state 5, that is, arc $5 \xrightarrow{a} 3$ and the final arrow $5 \rightarrow$:



Since state 5 gets inaccessible (from the initial one), we can cancel it and its arcs, which are already shown dashed in the state-transition graph.

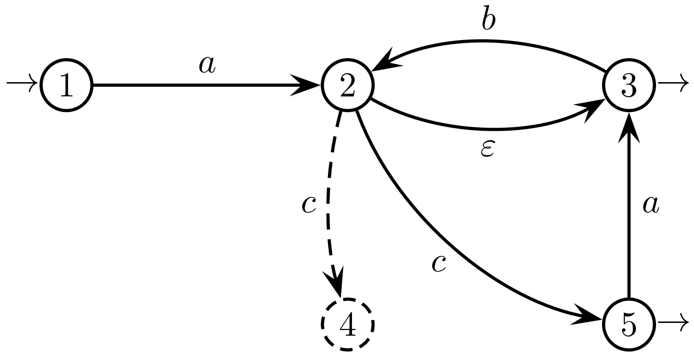
Then, and similarly, we cut transition $2 \xrightarrow{\varepsilon} 3$ and we back-propagate the outgoing arc $3 \xrightarrow{b} 2$ and the final arrow $3 \rightarrow$, to obtain an automaton A'' without spontaneous transitions:



Automaton A'' happens to be deterministic: it has one initial state, no multiple outgoing transitions (this happens by chance) and no spontaneous ones (by construction). Since it is clearly isomorphic to the (deterministic and minimal) automaton A' (this happens by chance), it is equivalent to the original (nondeterministic) automaton A as well, even without testing it with sample strings.

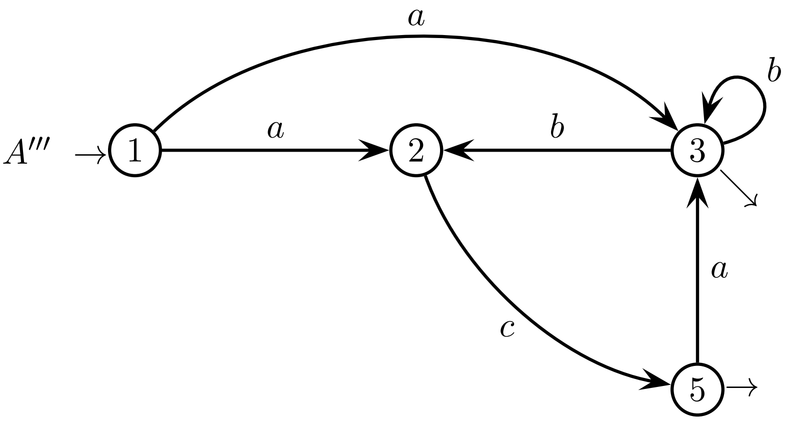
However, the strings x of language $L(A)$ with $|x| \leq 3$ are: a, $a\,b$, $a\,c$, $a\,b\,b$, $a\,b\,c$ and $a\,c\,a$ (in total 6 strings). All of them are accepted by automaton A'' as well, as expected from before, and none else of length ≤ 3 is accepted.

A different approach resorts to forward-propagation. Here is the result obtained by cutting transition $4 \overset{\varepsilon}{\to} 5$ and forward-propagating the ingoing arc $2 \overset{c}{\to} 4$:



Since state 4 becomes post-inaccessible (it does not go to any final state), we can cancel it with its arcs, which are shown dashed in the graph above.

Similarly, we cut transition $2 \overset{\varepsilon}{\to} 3$ and we forward-propagate the ingoing arcs $1 \overset{a}{\to} 2$ and $3 \overset{b}{\to} 2$, to obtain an automaton A''' without spontaneous transitions:



Automaton A''' is nondeterministic, as it has multiple transitions on its states 1 and 3. Thus formally proving that automaton A''' is equivalent to automaton A is not immediate. However, the reader can verify that it passes the same acceptance test with the six strings of length ≤ 3 listed before and that it fails with any other string of length ≤ 3, so that it is (reasonably) equivalent to A.

As automaton A''' is nondeterministic, the minimization theory of the deterministic machines does not apply to it as-is[1]. Of course, even after such a cleaning, automaton A''' still looks less convenient than automaton A''.

Notice that we could cut the spontaneous transitions by alternating back- and forward-propagation in all the combinations. In this way, we would obtain a variety of equivalent (non)deterministic automata, more or less convenient.

[1] Surprisingly automaton A''' has as many states as the deterministic automaton A'', which is minimal. In fact minimal nondeterministic automata are not unique, in general, and their minimization theory is substantially more complex than that of the deterministic ones.

2 Free Grammars and Pushdown Automata 20%

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

$$L = \left\{\ x^R\ c\ y \mid \quad x,\ y \in \{\ a,\ b\ \}^+\ \wedge\ (x = y \text{ or } x \text{ is a substring of } y)\ \right\}$$

Answer the following questions:

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

(b) Draw a syntax tree for this sample valid string of language L:

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

(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

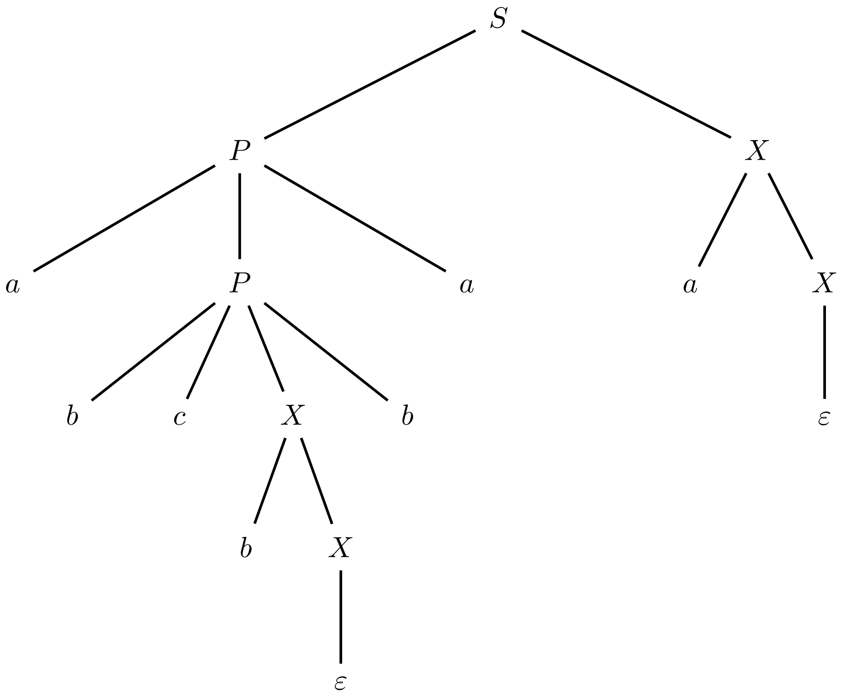
(a) Here is a working grammar G (axiom S), of type *BNF* with two nonterminals:

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

Nonterminal P generates, in the well-known auto-inclusive way, a non-empty palindrome over the letters a and b, with centre in the letter c. Instead, nonterminal X generates an arbitrary string (possibly empty) over the letters a and b, as it is expanded by two right-linear rules.

In relation with P, nonterminal X plays a double role: concatenating such an arbitrary string at the end of the palindrome, and inserting it immediately onto the right of the palindrome centre c. The overall effect is that the left part of the palindrome generated by P ends up with being a (reversed) substring of the right part thereof. This is the characteristic predicate of language L, thus grammar G is reasonably proved to be correct.

(b) Here is a syntax tree for the sample valid (see below) string $a\ b\ c\ b\ b\ a\ a$:



This syntax tree shows that nonterminal P generates the centred palindrome $\underbrace{a\,b}_{\text{left}}\ c\ \underbrace{b\,a}_{\text{right}}$ (see the P-P stem inside), while nonterminal X prepends a prefix b to the palindrome right part $b\,a$ and appends a suffix a to the palindrome end, as explained before, so that the left part $a\,b$ becomes a substring of the right part. Since nonterminal X is nullable however, grammar G can still generate all the standard centred (non-empty) palindromes over the letters a and b.

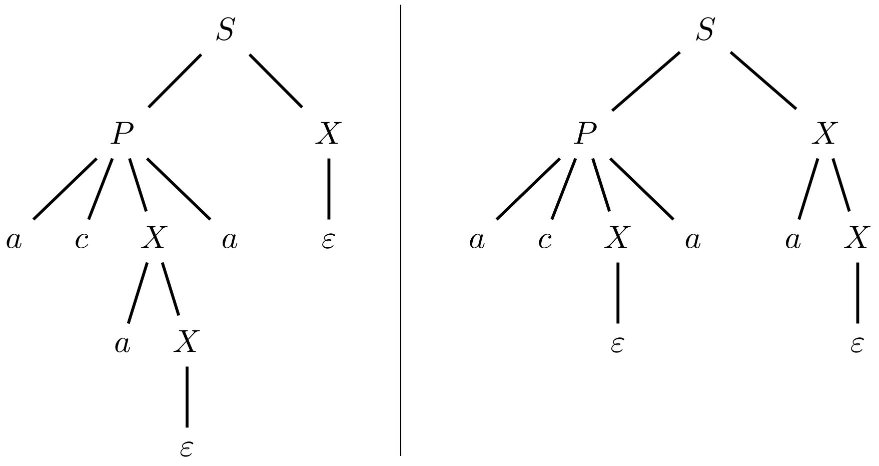
(c) Grammar G is ambiguous because so is, for instance, the (trivial) string $a\,c\,a\,a$: in fact, notice that the left part a can be thought of as a substring (reversed since $a^R = a$) of the right part $a\,a$ in two ways: as its prefix or as its suffix. Grammar G behaves consequently as it has two different leftmost derivations:

$$S \Rightarrow P\ X \Rightarrow a\ c\ X\ a\ X \Rightarrow a\ c\ a\ X\ a\ X \Rightarrow a\ c\ a\ a\ X \Rightarrow a\ c\ a\ a$$

and

$$S \Rightarrow P\ X \Rightarrow a\ c\ X\ a\ X \Rightarrow a\ c\ a\ X \Rightarrow a\ c\ a\ a\ X \Rightarrow a\ c\ a\ a$$

which clearly correspond to two different syntax trees, as shown below:



It is easy to realize that each string $a\,c\,a^n$, with $n \geq 1$, has exactly n syntax trees. In fact, such a string can be factored as follows (since again $a^R = a$):

$$\underbrace{a^R}_{x^R}\quad c\quad \overbrace{a^h\ \underbrace{a}_{x}\ a^k}^{n \text{ instances of } a} \qquad\qquad \text{with } h,\, k \geq 0 \text{ and } h + k + 1 = n$$

in exactly n different ways that correspond to as many different trees of grammar G. This proves that grammar G has an unlimited ambiguity degree: it is impossible to set a finite limit, independent of every string of language L, to the number of trees generated by G for the same string. Anyway, grammar G does not have an infinite ambiguity degree: every string of L has finitely many trees.

A longer ambiguous string of grammar G, less trivial, is $b\,a\,c\,a\,b\,a\,b$, because:

$$\underbrace{b\,a}_{x^R}\ c\ \underbrace{\underbrace{a\,b}_{x}\ a\,b}_{y} \qquad\qquad \underbrace{b\,a}_{x^R}\ c\ \underbrace{a\,b\ \underbrace{a\,b}_{x}}_{y}$$

The reader may verify by himself that the string has two syntax trees. Likewise, language L itself is inherently ambiguous, though this might be very hard to prove, if it is ever decidable. If it were confirmed, a consequence would be that every grammar that generates language L is necessarily ambiguous, in some way.

2. Consider a description language that models the layout of a graphical terminal, consisting of a desktop populated by window objects, shortly *widgets*, and application programs, shortly *apps*. The widgets may contain other widgets or apps, at a finite yet unlimited depth. Such a language features the following structures:

   - the desktop may be empty, or may contain widgets or apps (or both)
   - a widget must first specify, in this order:
     - its left upper corner position (mandatory): position x, y
     - its (diagonal) size (optional): diagonal d
     - its aspect ratio (mandatory): aspect a or the unique keyword square
     - a header (mandatory): non-empty alphanumerical string

     where x, y and d are integers, and a is a fixed-point number (all are non-negative)
   - then, a widget must contain one or more widgets or apps (or both), in any order
   - an app is a program with these arguments:
     - an executable file (pathname) in the Linux form (mandatory): absolute from the root ‘ / ’, e.g., exec /dir1/file, or relative, e.g., exec file, exec dir1/file2; the current ‘ . ’ and parent ‘ .. ’ directory symbols are allowed, e.g., exec ./file
     - an optional list of widgets to open and pop up on launch, and of functions, i.e., apps, to call on launch, in any order

   For any detail that may have been left unspecified, e.g., separators, etc, you can take inspiration from the sample description below:

```
desktop
  widget
    position 10, 20; diagonal 30; aspect 1.5; header editor;
    widget
      position 30, 40; square; header compiler;
      app ... end app
    end widget
    widget ... end widget;
    app exec ./file; end app
    app ... end app
    widget ... end widget
    ...
  end widget
  app
    exec /dir1/file;
    widget ... end widget
    app exec file; end app;
    ...
  end app
  ...
end desktop
```

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

Solution

(a) Here is a reasonable grammar G for the sketched language (axiom DTOP):

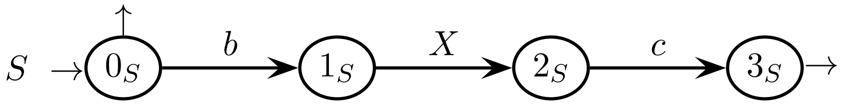
$$
G \left\{
\begin{array}{rcl}
\langle \mathsf{DTOP} \rangle & \to & [\ \langle \mathsf{OBJS} \rangle\ ] \\
\langle \mathsf{OBJS} \rangle & \to & \big((\ \langle \mathsf{WGET} \rangle \mid \langle \mathsf{APP} \rangle\ )\ [\text{‘;’}]\ \big)^{+} \\
\langle \mathsf{WGET} \rangle & \to & \mathsf{widget} \\
 & & \quad \mathsf{position}\ \langle \mathsf{INT} \rangle\ \text{‘,’}\ \langle \mathsf{INT} \rangle\ \text{‘;’} \\
 & & \quad \big[\mathsf{diagonal}\ \langle \mathsf{INT} \rangle\ \text{‘;’} \big] \\
 & & \quad \big(\mathsf{aspect}\ \langle \mathsf{FP} \rangle \mid \mathsf{square} \big)\ \text{‘;’} \\
 & & \quad \mathsf{header}\ \langle \mathsf{STR} \rangle\ \text{‘;’} \\
 & & \quad \langle \mathsf{OBJS} \rangle \\
 & & \mathsf{end\ widget} \\
\langle \mathsf{APP} \rangle & \to & \mathsf{app} \\
 & & \quad \mathsf{exec}\ \langle \mathsf{PN} \rangle\ \text{‘;’} \\
 & & \quad [\ \langle \mathsf{OBJS} \rangle\ ] \\
 & & \mathsf{end\ app} \\
\langle \mathsf{INT} \rangle & \to & [\,0 ... 9\,]^{+} \\
\langle \mathsf{FP} \rangle & \to & \langle \mathsf{INT} \rangle\ \text{‘.’}\ \langle \mathsf{INT} \rangle \\
\langle \mathsf{STR} \rangle & \to & (\,[\,A ... Z\,] \mid [\,a ... z\,] \mid [\,0 ... 9\,]\,)^{+} \\
\langle \mathsf{PN} \rangle & \to & [\text{‘/’}]\ \big((\,\langle \mathsf{STR} \rangle \mid \text{‘.’} \mid \text{‘..’}\,)\ \text{‘/’} \big)^{*}\ \langle \mathsf{STR} \rangle
\end{array}
\right.
$$

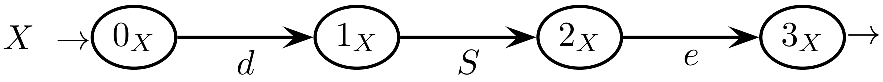
Square brackets mean optionality, as usual, whereas [...] is an interval. The sample description suggests that the semicolon separator is optional after widgets and apps, which are already clearly terminated by end ..., and mandatory after the other fields, which may overlap (particularly due to the totally free format of the header strings). Widgets and apps are placed after the other fields, and may alternate. Identifiers and numbers are modeled in the customary way. One might refine: disallow the identifiers with leading digit and allow those with underscore ‘_’, and disallow the unnecessary zeroes in the numbers.

Grammar G is reasonably correct and is unambiguous by construction, as it consists of structures known to be unambiguous. Apps may be empty, that is, not contain any more widgets or apps, so recursion can terminate and there are not any endless derivations. Other more or less similar solutions are possible.

3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the following grammar G, represented as a machine net over the four-letter terminal alphabet $\Sigma = \{ b, c, d, e\}$ and the nonterminal alphabet $V = \{ S, X \}$ (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 and determine if grammar G is of type $ELL\,(1)$, based on the guide sets, and shortly justify your answer. If you wish, you can use the figure above to add the call arcs and annotate the guide sets.

(c) Sketch a simulation of the ELR analysis of the sample valid string below:

$$b\ d\ b\ d\ e\ c\ e\ c$$

and show the sequence of stack configurations and parser moves. In the stack it suffices to show the names I_0, I_1, etc, of the pilot states stored, alternated to the grammar symbols. Continue the table prepared on the next page.

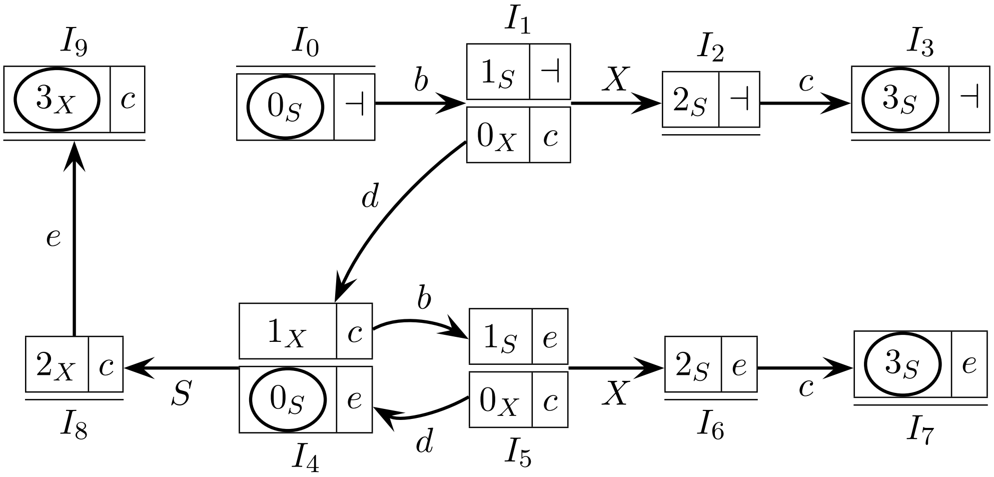
(d) (optional) Determine the computational complexity class of the function $h\,(n)$ that expresses the maximal height of the stack as a function of the length $n = |\,x\,|$ of the input string x.

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

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

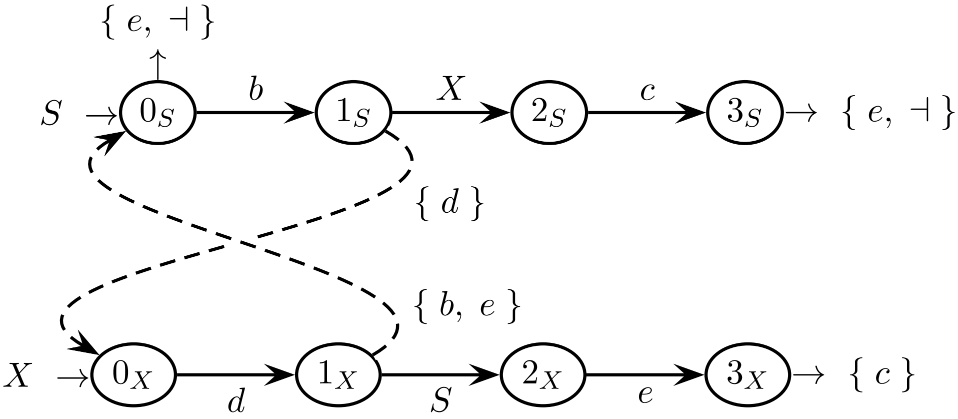
Solution

(a) Here is the pilot of grammar G, with 10 m-states (I_0 is the initial m-state):



The pilot does not have any shift-reduce or reduce-reduce conflicts, as the lookahead in the reduction m-states is always disjoint from the (symbols on the) outgoing terminal shift arcs, and as there is not any reduction m-state that contains more than one final state, respectively. Furthermore, since all the transitions are single, the pilot has the *STP* and consequently it does not have any convergence conflicts. Since the pilot is conflict-free, grammar G is of type *ELR* (1).

(b) Here are all the guide sets of the machine net of grammar G, completed with the call arcs to make the Predictive Control Flow Graph (*PCFG*):



The guide sets on the terminal shift arcs are trivial and not shown here, while those on the two call arcs and on the three exit arrows are easy to compute. Notice that the terminator $\dashv$ must not show up on the call arc $1_X \xrightarrow{\{ b, e \}} 0_S$! In fact, the terminal e in the guide set of this call arc comes from the shift arc $2_X \xrightarrow{e} 3_X$ (not from the exit arrow $0_S \xrightarrow{\{ e, \dashv \}}$), since nonterminal S is nullable (the exit guide sets do not back-propagate onto the previous call arcs).

Thus grammar G is *ELL* (1), as the guide sets at the (unique) bifurcation state 0_S are disjoint. That grammar G is *ELL* (1) can also be inferred from the pilot, since the pilot is *ELR* (1), it has the *STP* and the machine net is not left-recursive.

(c) Here is the simulation table for the *ELR* analysis of the string $b\ d\ b\ d\ e\ c\ e\ c$:

stack of alternated macro-states and grammar symbols	*input left to be scanned*	*move executed*
I_0	$b\ d\ b\ d\ e\ c\ e\ c\ \dashv$	initial configuration
$I_0\ b\ I_1$	$d\ b\ d\ e\ c\ e\ c\ \dashv$	terminal shift: $I_0 \xrightarrow{b} I_1$
$I_0\ b\ I_1\ d\ I_4$	$b\ d\ e\ c\ e\ c\ \dashv$	terminal shift: $I_1 \xrightarrow{d} I_4$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5$	$d\ e\ c\ e\ c\ \dashv$	terminal shift: $I_4 \xrightarrow{b} I_5$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5\ d\ I_4$	$e\ c\ e\ c\ \dashv$	terminal shift: $I_5 \xrightarrow{d} I_4$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5\ d\ I_4$	$S\ e\ c\ e\ c\ \dashv$	**null reduction** (on e): $\varepsilon \rightsquigarrow S$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5\ d\ I_4\ S\ I_8$	$e\ c\ e\ c\ \dashv$	nonterminal shift: $I_4 \xrightarrow{S} I_8$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5\ d\ I_4\ S\ I_8\ e\ I_9$	$c\ e\ c\ \dashv$	terminal shift: $I_8 \xrightarrow{e} I_9$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5$	$X\ c\ e\ c\ \dashv$	**reduction** (on c): $d\ S\ e \rightsquigarrow X$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5\ X\ I_6$	$c\ e\ c\ \dashv$	nonterminal shift: $I_5 \xrightarrow{X} I_6$
$I_0\ b\ I_1\ d\ I_4\ b\ I_5\ X\ I_2\ c\ I_7$	$e\ c\ \dashv$	terminal shift: $I_6 \xrightarrow{c} I_7$
$I_0\ b\ I_1\ d\ I_4$	$S\ e\ c\ \dashv$	**reduction** (on e): $b\ X\ c \rightsquigarrow S$
$I_0\ b\ I_1\ d\ I_4\ S\ I_8$	$e\ c\ \dashv$	nonterminal shift: $I_4 \xrightarrow{S} I_8$
$I_0\ b\ I_1\ d\ I_4\ S\ I_8\ e\ I_9$	$c\ \dashv$	terminal shift: $I_8 \xrightarrow{e} I_9$
$I_0\ b\ I_1$	$X\ c\ \dashv$	**reduction** (on c): $d\ S\ e \rightsquigarrow X$
$I_0\ b\ I_1\ X\ I_2$	$c\ \dashv$	nonterminal shift: $I_1 \xrightarrow{X} I_2$
$I_0\ b\ I_1\ X\ I_2\ c\ I_3$	$\dashv$	terminal shift: $I_2 \xrightarrow{c} I_3$
I_0	$S\ \dashv$	**reduction** (on $\dashv$): $b\ X\ c \rightsquigarrow S$
initial m-state	empty tape (all is reduced to S)	last move: reduction to S

At each reduction, the nonterminal reduced to is fictionally written into the input tape, to trigger the nonterminal shift that has to immediately follow each reduction. Here we do so only for clarity, it is not an effective parser action.

In the last simulation row, the acceptance condition is verified, thus the bottom-up syntax analyzer stops and accepts the string. The bottom-up list of five reduction moves (bolded) is the (rightmost) derivation of the string, namely:

$$S \Rightarrow b\ X\ c \Rightarrow b\ d\ S\ e\ c \Rightarrow b\ d\ b\ X\ c\ e\ c \Rightarrow b\ d\ b\ d\ S\ e\ c\ e\ c \Rightarrow b\ d\ b\ d\ e\ c\ e\ c$$

Equivalently, it is the bottom-up list of nodes of the corresponding syntax tree.

(d) On a general ground, it is well-known that deterministic *ELR* (and *ELL*) parsers have a linear time complexity in the input length (see the textbook). Since space complexity cannot be greater than time complexity, it is linear as well. This applies to the present case, too, since grammar G is of type $ELR\,(1)$.

However, we can try to carry out a more specific determination of the space complexity, basing on the peculiarities of grammar G. The language strings have a simple parenthetical structure, with only one nest (concatenated nests are disallowed) made of alternating bracket pairs $b\,c$ and $d\,e$. During the analysis the stack grows linearly with the depth of the nesting structure, extends to its maximal length when the string centre is reached, and then starts decreasing as the reduction moves are executed. Therefore the function $h\,(n)$ is linear in the length n of the input string x, i.e., it holds $h\,(n) \in \Theta\,(n)$.

To be even more precise, one could find a closed-form expression for $h\,(n) = \max\,\big(\,|\,\text{stack}\,|\,\big)$ for a given input string x of length n, taking inspiration from the simulation of the analysis of the sample (valid) string of point (a). For instance, observation and intuition suggest this closed-form expression for $h\,(n)$:

$$h\,(n) = \max\,\big(\,|\,\text{stack}\,|\,\big) = 2 \times |\,\text{left half of } x\,| + 4 = 2 \times \tfrac{n}{2} + 4 = n + 4$$

Notice that at its maximum extension, the stack grows up to store the left half of the (valid) string (all the letters b and d), plus the alternating shift m-states, and plus four more symbols to reach the first reduction (we can avoid counting the initial m-state I_0, which always stays at the stack bottom), after which the stack only decreases (though not monotonically). This is in line with $h\,(n) \in \Theta\,(n)$.

Notice also that basically the grammar generates a sort of palindrome, where the right half of the string is a mirrored image of the left half, though transliterated over a different subalphabet. This kind of palindrome does not have a marked centre, however the grammar happens to be deterministic (at least in the *ELR* sense) as the change of subalphabet clearly identifies the midpoint of the string.

For completeness, since grammar G is of type $ELL\,(1)$, too, we also show the simulation of the top-down analysis of the sample string $b\ d\ b\ d\ e\ c\ e\ c$:

stack of machine net states	*input left to be scanned*	*move executed*
0_S	$b\ d\ b\ d\ e\ c\ e\ c\ \dashv$	initial config. equivalent to: **call**: machine M_S
1_S	$d\ b\ d\ e\ c\ e\ c\ \dashv$	shift: $0_S \xrightarrow{b} 1_S$
$2_S\ 0_X$	$d\ b\ d\ e\ c\ e\ c\ \dashv$	**call**: machine M_X on guide d
$2_S\ 1_X$	$b\ d\ e\ c\ e\ c\ \dashv$	shift: $0_X \xrightarrow{d} 1_X$
$2_S\ 2_X\ 0_S$	$b\ d\ e\ c\ e\ c\ \dashv$	**call**: machine M_S on guide b
$2_S\ 2_X\ 1_S$	$d\ e\ c\ e\ c\ \dashv$	shift: $0_S \xrightarrow{b} 1_S$
$2_S\ 2_X\ 2_S\ 0_X$	$d\ e\ c\ e\ c\ \dashv$	**call**: machine M_X on guide d
$2_S\ 2_X\ 2_S\ 1_X$	$e\ c\ e\ c\ \dashv$	shift: $0_X \xrightarrow{d} 1_X$
$2_S\ 2_X\ 2_S\ 2_X\ 0_S$	$e\ c\ e\ c\ \dashv$	**call**: machine M_S on guide e
$2_S\ 2_X\ 2_S\ 2_X$	$e\ c\ e\ c\ \dashv$	return: from M_S on guide e
$2_S\ 2_X\ 2_S\ 3_X$	$c\ e\ c\ \dashv$	shift: $2_X \xrightarrow{e} 3_X$
$2_S\ 2_X\ 2_S$	$c\ e\ c\ \dashv$	return: from M_X on guide c
$2_S\ 2_X\ 3_S$	$e\ c\ \dashv$	shift: $2_S \xrightarrow{c} 3_S$
$2_S\ 2_X$	$e\ c\ \dashv$	return: from M_S on guide e
$2_S\ 3_X$	$c\ \dashv$	shift: $2_X \xrightarrow{e} 3_X$
2_S	$c\ \dashv$	return: from M_X on guide c
3_S	$\dashv$	shift: $2_S \xrightarrow{c} 3_S$
initial net state	empty tape	no more moves

Notice that by scanning top-down a column of net states, one can read the recognizing path in a machine call. In the last simulation row, the acceptance condition is verified, thus the top-down syntax analyzer stops and accepts the string. The top-down list of five call moves (bolded) drives the (leftmost) derivation of the string, namely:

$$S \Rightarrow b\ X\ c \Rightarrow b\ d\ S\ e\ c \Rightarrow b\ d\ b\ X\ c\ e\ c \Rightarrow b\ d\ b\ d\ S\ e\ c\ e\ c \Rightarrow b\ d\ b\ d\ e\ c\ e\ c$$

Equivalently, it builds the top-down list of nodes of the corresponding syntax tree.

Of course, the top-down pushdown analyzer is somewhat heavy and obscure, whereas an equivalent recursive-descent formulation is more handy and readable. Here it is:

```
procedure $S$                                  // axiomatic syntactic procedure
begin
    if $cc \in \{ b \}$ then                                // current state is $0_S$
        $cc := next$                      // read next char and goto state $1_S$
        if $cc \in \{ d \}$ then                            // current state is $1_S$
            call $X$                  // call machine $M_X$ then goto state $2_S$
            if $cc \in \{ c \}$ then                        // current state is $2_S$
                $cc := next$              // read next char and goto state $3_S$
                if $cc \in \{ e, \dashv \}$ then            // current state is $3_S$
                    return                      // return from final state $3_S$
                else $error$                     // invalid guide in state $3_S$
            else $error$                         // invalid guide in state $2_S$
        else $error$                             // invalid guide in state $1_S$
    else if $cc \in \{ e, \dashv \}$ then                   // current state is $0_S$
        return                                  // return from final state $0_S$
    else $error$                                 // invalid guide in state $0_S$

procedure $X$                                    // syntactic procedure of $X$
begin
    if $cc \in \{ d \}$ then                                // current state is $0_X$
        $cc := next$                      // read next char and goto state $1_X$
        if $cc \in \{ b \}$ then                            // current state is $1_X$
            call $S$                  // call machine $M_S$ then goto state $2_X$
            if $cc \in \{ e \}$ then                        // current state is $2_X$
                $cc := next$              // read next char and goto state $3_X$
                if $cc \in \{ c \}$ then                    // current state is $3_X$
                    return                      // return from final state $3_X$
                else $error$                     // invalid guide in state $3_X$
            else $error$                         // invalid guide in state $2_X$
        else $error$                             // invalid guide in state $1_X$
    else $error$                                 // invalid guide in state $0_X$

program $RECURSIVE\_DESCENT$                                  // main program
begin
    $cc := next$                               // set $cc$ to first char or $\dashv$
    call $S$                                    // call axiomatic machine $M_S$
    if $cc \in \{ \dashv \}$ then                  // test of acceptance condition
        stop and accept                                        // valid string
    else $error$                            // analysis finished before tape end
```

Algorithm: Recursive descent analyzer (written in Pascal-like pseudo-code).

As before, the series of procedure execution traces identifies the (leftmost) derivation of the accepted string. The above coding can be optimized in various ways, for instance by grouping the numerous error cases. See the next formulation:

```
procedure $S$                               // axiomatic syntactic procedure
begin
    if $cc \in \{ b \}$ then                             // current state is $0_S$
        $cc := next$                      // read next char and goto state $1_S$
        if $cc \in \{ d \}$ then                         // current state is $1_S$
            call $X$                      // call machine $M_X$ then goto state $2_S$
            if $cc \in \{ c \}$ then                     // current state is $2_S$
                $cc := next$              // read next char and goto state $3_S$
                if $cc \in \{ e, \dashv \}$ then         // current state is $3_S$
                    return                     // return from final state $3_S$
        error                       // invalid guide in state $3_S$, $2_S$ or $1_S$
    if $cc \in \{ e, \dashv \}$ then                     // current state is $0_S$
        return                                 // return from final state $0_S$
    error                                       // invalid guide in state $0_S$
```

```
procedure $X$                  // syntactic procedure of nonterminal $X$
begin
    if $cc \in \{ d \}$ then                             // current state is $0_X$
        $cc := next$                      // read next char and goto state $1_X$
        if $cc \in \{ b \}$ then                         // current state is $1_X$
            call $S$                      // call machine $M_S$ then goto state $2_X$
            if $cc \in \{ e \}$ then                     // current state is $2_X$
                $cc := next$              // read next char and goto state $3_X$
                if $cc \in \{ c \}$ then                 // current state is $3_X$
                    return                     // return from final state $3_X$
    error                   // invalid guide in state $0_X$, $1_X$, $2_X$ or $3_X$
```

```
program $RECURSIVE\_DESCENT$                              // main program
begin
    $cc := next$                                  // initialize current char
    call $S$                                   // call axiomatic machine $M_S$
    if $cc \in \{ \dashv \}$ then                   // test of acceptance condition
        stop and accept    // tape completely scanned - valid string
    error                  // tape partially scanned - invalid string
```

Algorithm: Optimized recursive descent syntactic analyzer (Pascal-like).

The *error* function might provide a diagnostic message. The non-optimized syntactic analyzer allows to make a case-by-case diagnosis, whereas the optimized one is generic. For instance, in the non-optimized analyzer we could change the following error line:

```
error    \\ invalid guide in state $3_S$
```

of the syntactic procedure S, and make it visualize the expressive diagnostic below:

fprintf (stderr, "expected char ' e ' or EOF, found char ' %c ' instead.", *cc*)

4 Language Translation and Semantic Analysis 20%

1. Consider the arithmetic expressions with the operation of addition ' $+$ ', variables and numbers schematized by letter a, and subexpressions enclosed in round brackets ' ( ) ', and at least two addends at all levels. Here is a non-ambiguous *BNF* grammar G for their language (axiom S):

$$G \begin{cases} 1\colon\ S \rightarrow E \\ 2\colon\ E \rightarrow T\ +\ E \\ 3\colon\ E \rightarrow T\ +\ T \\ 4\colon\ T \rightarrow a \\ 5\colon\ T \rightarrow (\ S\ ) \end{cases}$$

We wish to translate such arithmetic expressions into a generalized postfix form with multi-operand addition, e.g., expression $a + a$ is translated into $\langle\, a\, a\, +$, expression $a + a + a$ into $\langle\, a\, a\, a\, +$, etc. Notice that the open angle bracket ' $\langle$ ' marks the head of the list of addends, which must have a length ≥ 2 in every (sub)expression. Here is a composite example with subexpressions:

$$a + \big(\, (\, b + b\,) + (\, c + c\,)\, \big) + a \quad \mapsto \quad \Big\langle\, a\, \big\langle\, \langle\, b\, b + \langle\, c\, c + + \, a\, +$$

where various terminal letters are used as a visual help for the reader to understand.

Answer the following questions:

(a) Test the source grammar G and draw the source syntax tree of the above composite example.

(b) Use the proposed source grammar G and write a syntax translation scheme (or grammar) G_{τ} that computes the sketched translation. If you wish, you can complete the partial scheme already prepared on the next page.

(c) Test your translation scheme (or grammar) G_{τ} and draw the destination syntax tree of the above composite example.

(d) (optional) Say if your translation scheme (or grammar) G_{τ} is deterministic, and reasonably justify your answer.

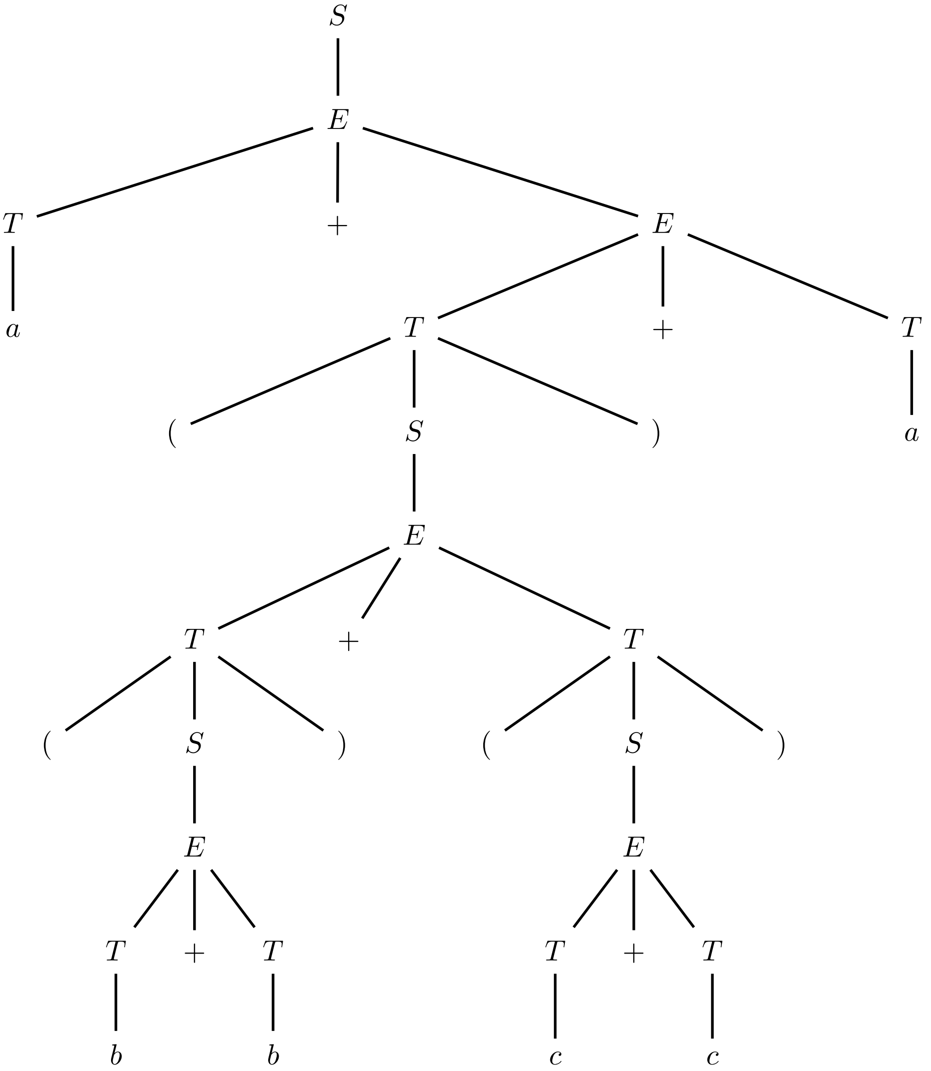
syntax translation scheme G_τ to complete - question (b)

G_τ

source grammar	*destination grammar* *(to be written)*
1: $S \rightarrow E$	
2: $E \rightarrow T + E$	
3: $E \rightarrow T + T$	
4: $T \rightarrow a$	
5: $T \rightarrow (S)$	

Solution

(a) Here is the syntax tree generated by grammar G for the sample expression $a + \big((b+b) + (c+c) \big) + a$ (for clarity we keep the differently named terminals):



This syntax tree clearly shows that the whole (outermost) expression is a 3-addend summation, which is generated by the right-linear recursive rule $E \to T + E$ applied twice. The middle addend is a bracketed subexpression, namely a 2-addend summation, which is eventually expanded (after two derivation steps) by the non-recursive rule $E \to T \ + \ T$. The same holds for the two innermost subexpressions. Clearly rule 3 ends the recursion of rule 2, so it always generates the last two addends (the rightmost ones) of a (sub)expression of three or more addends. One can easily realize that all the (sub)expressions have at least two addends and that they are right-associative (recursion is on the right).

In summary, this is a simple variation of the well-known non-ambiguous right-associative *BNF* grammar of arithmetic expressions (with one operator), slightly restricted to generate only (sub)expressions with at least two addends.

(b) Here is a working syntax translation scheme G_τ (axiom S), obtained basing on the observations made before on the syntax tree:

$$G_\tau \left\{ \begin{array}{ll} 1\colon\ S \to E & S \to \langle\ E \\ \hline 2\colon\ E \to T\ +\ E & E \to T\ E \\ \hline 3\colon\ E \to T\ +\ T & E \to T\ T\ + \\ \hline 4\colon\ T \to a & T \to a \\ \hline 5\colon\ T \to (\ S\ ) & T \to S \end{array} \right.$$

Scheme G_τ tightly follows the classical syntax translation scheme of the infix arithmetic expressions into postfix form (see the textbook). It is formally well-defined as the destination grammar has the same nonterminal structure as the source one, the only difference being in the disposition of the terminals in the rules. One can rewrite it as a translation grammar G_τ^{gram}, in two forms as follows:

$$G_\tau^{gram} \left\{ \begin{array}{l} 1\colon\ S \to \{\ \langle\ \}\ E \\ 2\colon\ E \to T\ +\ E \\ 3\colon\ E \to T\ +\ T\ \{\ +\ \} \\ 4\colon\ T \to a\ \{\ a\ \} \\ 5\colon\ T \to (\ S\ ) \end{array} \right. \qquad \left\{ \begin{array}{l} 1\colon\ S \to \frac{\varepsilon}{\langle}\ E \\ 2\colon\ E \to T\ \frac{+}{\varepsilon}\ E \\ 3\colon\ E \to T\ \frac{+}{\varepsilon}\ T\ \frac{\varepsilon}{+} \\ 4\colon\ T \to \frac{a}{a} \\ 5\colon\ T \to \frac{(}{\varepsilon}\ S\ \frac{)}{\varepsilon} \end{array} \right.$$

In the left form, the brace metasymbols '{' and '}' distinguish the destination terminals, as the source and destination alphabets are not disjoint. In the right form, the fractions distinguish the source (numerator) and destination (denominator) terminals. The two forms are just a rewriting of each other, of course.

A little thought should convince anyone that nonterminal E in the (axiomatic) rule 1 starts a new addend list, so that this the right place wherein to generate the marker angle bracket. Of course, round brackets disappear in the destination grammar. Instead rule 3, which ends an addend list, is the right place wherein to generate the postfix operator '+', and only therein. Notice in fact that (the end of) rule 2 would be a wrong place, as the operator '+' it contains may never be the last one in an addend list, as observed before on the syntax tree.

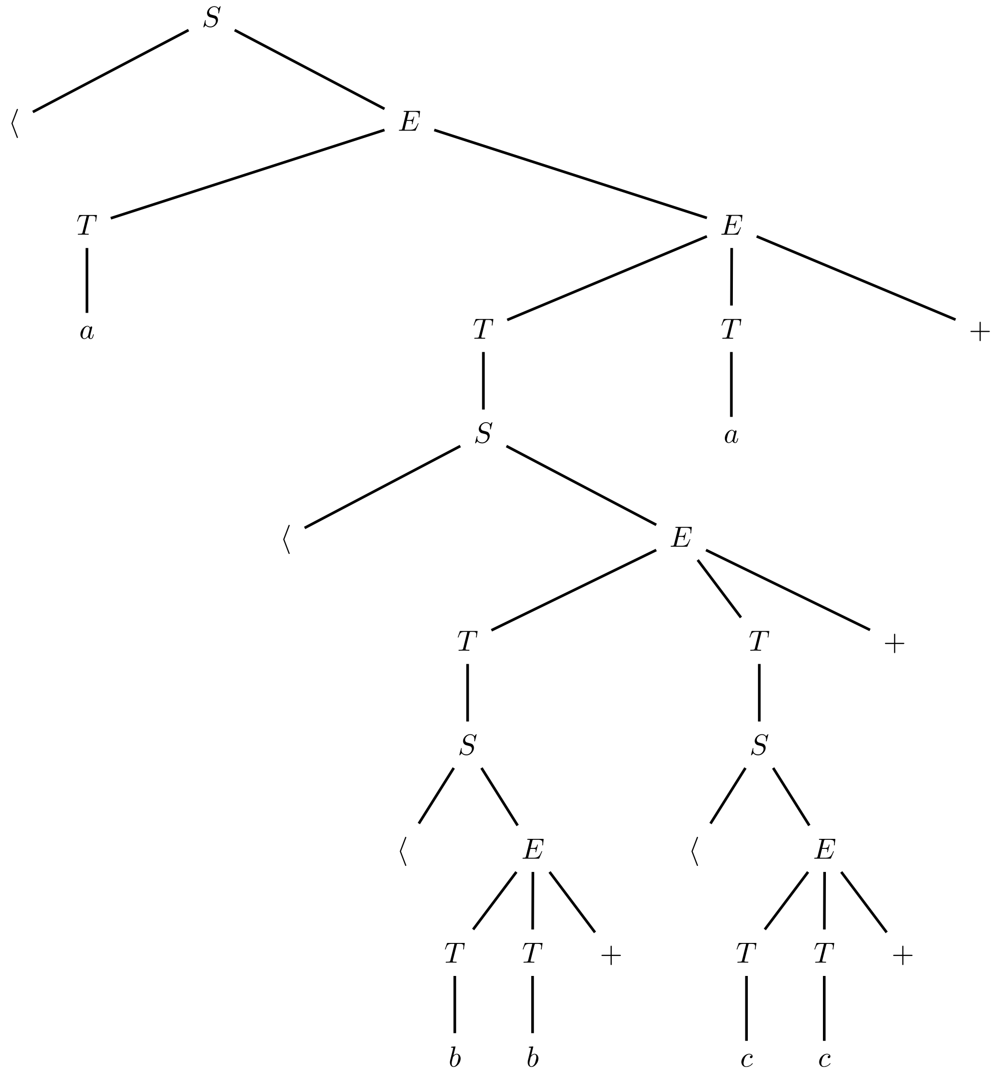
In summary, the crucial point is to distinguish the roles of rules 2 and 3, and to correctly place the postfix '+' addition operator only in the rule 3. Translating round brackets into angle brackets is simple: transliterate the open round bracket '(' into the open angle bracket '$\langle$', and just cancel the closed bracket ')'; of course, also prepend one more open angle bracket '$\langle$' to the whole expression. All these operations can be carried out altogether by moving the open bracket from rule 5 into the axiomatic rule 1).

There may be other solutions, linked less tightly than scheme G_τ to the classical solutions for translating expressions from infix to postfix. A simple one is the following translation grammar $G_{1,\tau}^{gram}$ (axiom S), equivalent to grammar G_τ^{gram}:

$$G_{1,\tau}^{gram} \quad \begin{cases} 1\colon\ S \to \{\ \langle\ \}\ E\ \{\ +\ \} \\ 2\colon\ E \to T\ +\ E \\ 3\colon\ E \to T\ +\ T \\ 4\colon\ T \to a\ \{\ a\ \} \\ 5\colon\ T \to (\ S\ ) \end{cases} \qquad \begin{cases} 1\colon\ S \to \frac{\varepsilon}{\langle}\ E\ \frac{\varepsilon}{+} \\ 2\colon\ E \to T\ \frac{+}{\varepsilon}\ E \\ 3\colon\ E \to T\ \frac{+}{\varepsilon}\ T \\ 4\colon\ T \to \frac{a}{a} \\ 5\colon\ T \to \frac{(}{\varepsilon}\ S\ \frac{)}{\varepsilon} \end{cases}$$

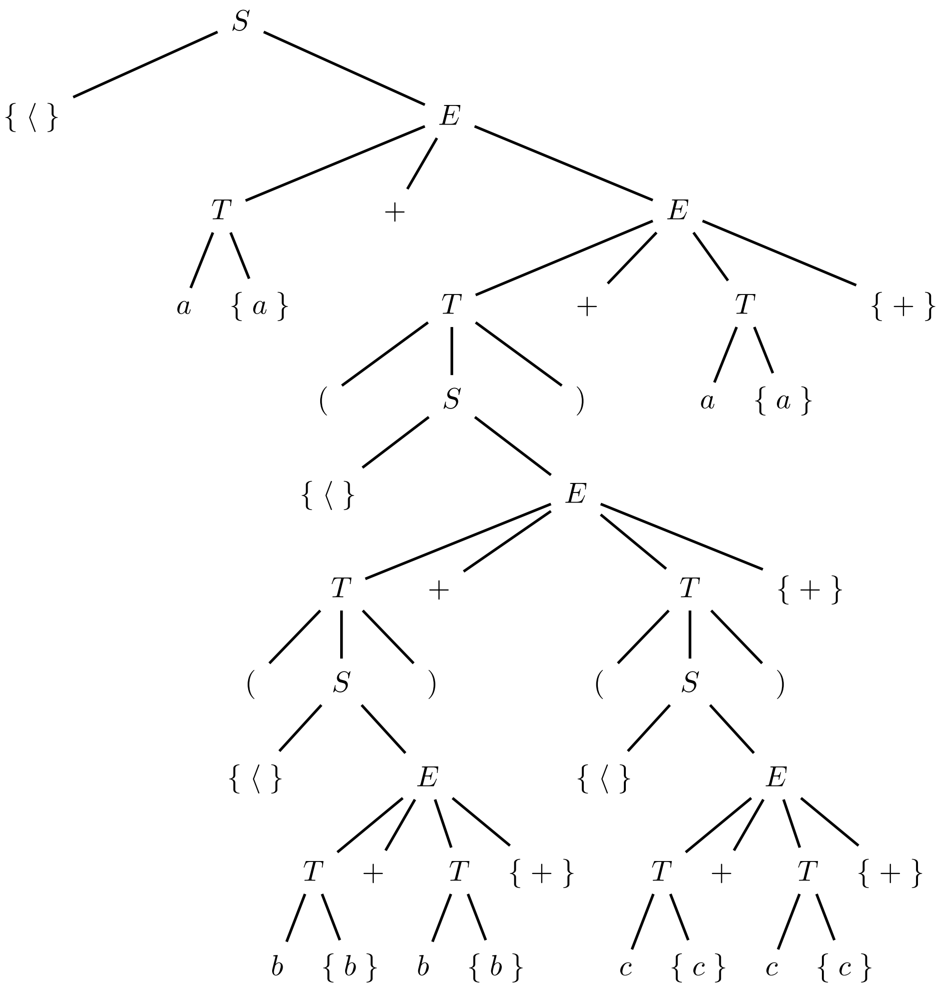
The destination addition operator '$+$' is moved from rule 3 to the (axiomatic) rule 1, as an addition has to be generated at the end of each (sub)expression.

(c) Here is the destination syntax tree generated by scheme G_τ for the sample expression translated into generalized postfix form $\Big\langle\ a\ \langle\ \langle\ b\ b+\langle\ c\ c++\ a\ +$, where for clarity we still keep the differently named terminals:



The destination tree is well-defined as it is isomorphic, i.e., superimposable, to the source one except the leaf nodes, and generates the translated expression. It uses all the grammar rules, thus it reasonably proves that the translation scheme is correct. One could rather draw the source and destination trees superimposed as one tree with the same inner nodes and two different series of leaves.

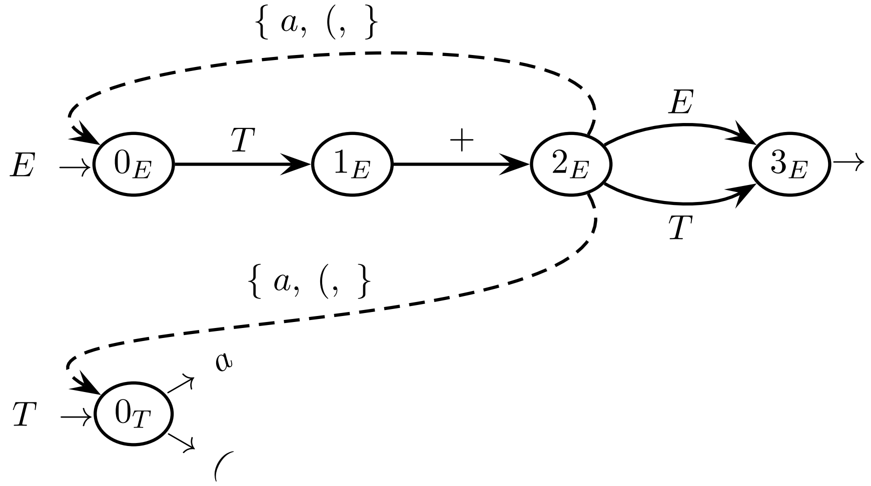
For completeness here is the source-destination superimposed tree of the sample expression above, where the destination terminals are embraced:



Now it is particularly evident which addition operators '$+$' are translated and which are just canceled, as well as how the angle brackets are placed.

The reader may wish to redraw the destination tree (superimposed or not to the source one) in the case of the other solution $G^{gram}_{1,\tau}$. The main difference from solution G^{gram}_{τ} is that every destination addition leaf '$+$' is lifted from the end of its E-E stem up to the nonterminal S at the stem start. So addition is translated at a higher level than before, yet the tree fronteer is unchanged.

(d) Clearly the proposed *BNF* source grammar G is not of type $LL(k)$ for any $k \geq 1$, as the rules 2 and 3 separately begin by the same nonterminal T, which can generate strings of arbitrary length, and so these rules have the same guide sets. Concerning *ELL*, we need to design the corresponding machine net, where the crucial machine is M_E, which unifies rules 2 and 3. Here it is:

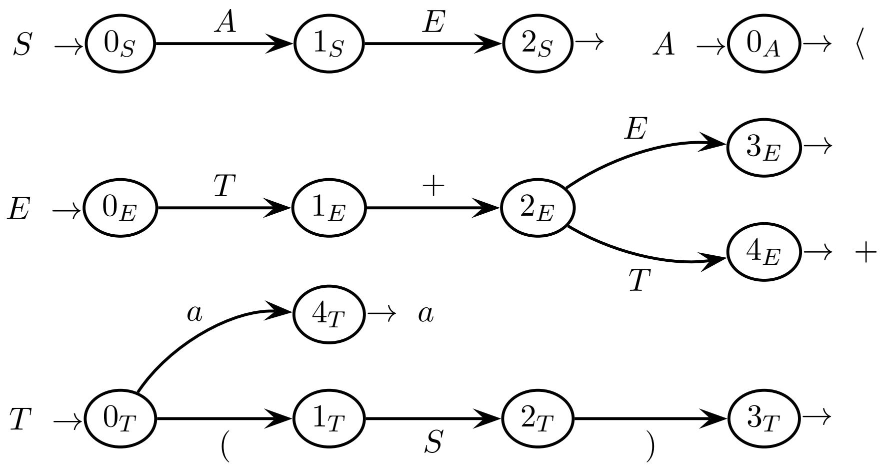


(the rest of the net is trivial and omitted) On the (unique) bifurcation state 2_E, the guide sets of the two call arcs coincide, so the source grammar G is not $ELL\,(1)$ either. It does not change with $k > 1$, for the same reason as before. In summary, here a predictive approach is unable to deterministically choose which rule of 2 or 3 to apply, or which arc of $2_E \xrightarrow{E} 3_E$ or $2_E \xrightarrow{T} 3_E$ to take.

So we need to try with LR or ELR, and before to rewrite the scheme into the postfix form. Here it is, in the form of a translation grammar G_τ^{post} (axiom S):

$$G_\tau^{post} \begin{cases} 1:\ S \to A\ E \\ 2:\ E \to T\ +\ E \\ 3:\ E \to T\ +\ T\ \{\, + \,\} \\ 4:\ T \to a\ \{\, a \,\} \\ 5:\ T \to (\ S\ ) \\ 6:\ A \to \varepsilon\ \{\, \langle \,\} \end{cases} \qquad \begin{cases} 1:\ S \to A\ E \\ 2:\ E \to T\ \frac{+}{\varepsilon}\ E \\ 3:\ E \to T\ \frac{+}{\varepsilon}\ T\ \frac{\varepsilon}{+} \\ 4:\ T \to \frac{a}{a} \\ 5:\ T \to \frac{(}{\varepsilon}\ S\ \frac{)}{\varepsilon} \\ 6:\ A \to \frac{\varepsilon}{\langle} \end{cases}$$

with an auxiliary nonterminal A. The braces ‘ { ’ and ‘ } ’ separate the destination terminals (similarly, the denominators of the fractions separate the destination terminals), all of which are placed at the end of a rule (3, 4 and 6). Some rules do not generate any destination terminal (1, 2 and 5). Notice that, after being put into the postfix form, the grammar is still not $ELL\,(1)$, as it was not before. Below is how the machine net of the translation grammar G_τ^{post} looks like now, and in particular the crucial machine M_E. All the write actions occur only on exit arrows, as the grammar is in the postfix form. Consequently both machines M_E and M_T need two final states, depending on having a write action or not:



The pilot of the translation grammar G_τ^{post} has many m-states and we skip it (indeed the reader may wish to design it for practice). One can directly notice instead that the bottom-up analyzer shifts the common prefix "$T+$" of the two rules 2 and 3 in parallel and then goes to two separate final states (3_E and 4_E), which of course will not stay in the same pilot m-state. The rest of the net is standard for expressions and well-known to be *ELR* (1), while clearly nonterminal A does not harm. In conclusion, the translation grammar G_τ^{post} (or scheme) is (likely to be) *ELR* (1) and (reasonably) the translation is deterministic.

As for the other solution $G_{1,\tau}^{gram}$, clearly it is not *LL* (1) or *ELL* (1) either, since its source grammar is the same as that of solution G_τ^{gram}. For *LR* or *ELR*, solution $G_{1,\tau}^{gram}$ also has to be put into the postfix form and the resulting machine net is the same as that of G_τ^{post}, but with the postfix write action '$+$' moved from machine M_E to machine M_S. So grammar $G_{1,\tau}^{gram}$ is of type *ELR* (1), too.

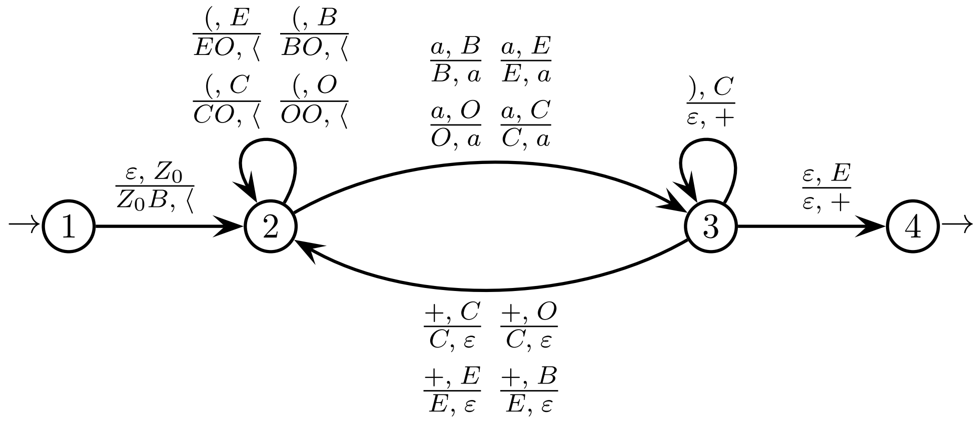
Alternatively, we can try a direct approach to find a pushdown deterministic translator. Such a device model is well-known to be able to recognize the language of the arithmetic expressions in the infix form. In whatever way it does so, we only need to make it into a translator: let it output '$\langle$' on reading '(', output '$+$' on reading ')', output a on reading a, and just read '$+$' with no output; also let it do as if the entire expression were enclosed in round brackets. See how this applies to the previous sample expression (with diversified terminals):

source		a	$+$	$\big($	(	b	$+$	b	)	$+$	(	c	$+$	c	)	$\big)$	$+$	a	
destin.	$\Big\langle$	a		$\big\langle$	$\langle$	b		b	$+$		$\langle$	c		c	$+$	$+$		a	$+$

Such a translator is equivalent to the scheme G_τ and is as deterministic as the underlying recognizer, thus in conclusion the translation is deterministic.

The reader may wish to design the state-transition graph of such a direct pushdown deterministic translator, which is not a syntax translator as it is unrelated to (the rules of) scheme G_τ and does not help finding a derivation or tree thereof. The underlying deterministic recognizer has to push each open bracket '(' on reading it, and has to pop it on reading its matching closed one ')', while only having to check that the symbols a (or a subexpression) and '$+$' alternate, starting and ending with a (or with a subexpression). This recognizer is clearly deterministic (it is driven by its input) and essentially accepts by empty stack.

Here is the translator, which accepts by empty stack as well as by final state:



$\frac{(,\ E}{EO,\ \langle}$ $\frac{(,\ B}{BO,\ \langle}$
$\frac{(,\ C}{CO,\ \langle}$ $\frac{(,\ O}{OO,\ \langle}$
$\frac{a,\ B}{B,\ a}$ $\frac{a,\ E}{E,\ a}$
$\frac{a,\ O}{O,\ a}$ $\frac{a,\ C}{C,\ a}$
$\frac{),\ C}{\varepsilon,\ +}$
$\frac{\varepsilon,\ Z_0}{Z_0 B,\ \langle}$
$\frac{\varepsilon,\ E}{\varepsilon,\ +}$
1
2
3
4
$\frac{+,\ C}{C,\ \varepsilon}$ $\frac{+,\ O}{C,\ \varepsilon}$
$\frac{+,\ E}{E,\ \varepsilon}$ $\frac{+,\ B}{E,\ \varepsilon}$

The symbols O (open) and C (closed) encode the presence of a (sub)expression and of one or more additions '$+$' in a (sub)expression, respectively. The symbols B (beginning) and E (end) play the same role for the whole expression, to be imagined as if it were embraced in brackets. The loop (2, 3) of length two ensures that addends, i.e., an a or a subexpression, and additions '$+$' alternate, entering and exiting with an addend. This grants that every (sub)expression has a length $\geq$ 2, i.e., that it has at least one addition and two addends (so grants that cases like a, $(\, a\,)$, $\big(\, (\, a\,)\, \big)$, $a + (\, a\,)$, etc, are unrecognized). The two self-loops on states 2 and 3 extend and shorten the stack on reading an open and a closed bracket, respectively. Reading an open bracket pushes a symbol O. Yet reading a closed bracket requires to pop a symbol C, so that the closed bracket necessarily matches a previous open one at the same level. Furthermore, from what said, there necessarily are one or more additions at that level in between. The symbols B and E behave in the same way for the imaginary outermost bracket pair. The rest of the state-transition graph of the recognizer is somewhat obvious.

The write actions are placed as explained before. Here is the execution trace of the translation of the sample expression above, with both the stack contents and the state at each translator move (the ε-moves are also shown):

input		ε	a	$+$	(	(	b	$+$	b	)	$+$	(	c	$+$	c	)	)	$+$	a	ε
output		$\Big\langle$	a	ε	$\langle$	$\langle$	b	ε	b	$+$	ε	$\langle$	c	ε	c	$+$	$+$	ε	a	$+$
stack						O	O	C	C			O	O	C	C					
					O	O	O	O	O	O	C	C	C	C	C	C				
		B	B	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	E	
(base)	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0
state	1	2	3	2	2	2	3	2	3	3	2	2	3	2	3	3	3	2	3	4

Scanning the stack contents horizontally from left to right is enlightening: each continuous strip of stack symbols spans a bracket pair. A strip starts by O (by B at the bottom) on reading an (imaginary) open bracket, turns to C (to E) on reading the first addition in the pair and so stays through all the subsequent ones, and ends by C (by E) on reading the matching (imaginary) closed bracket. It is evident that the translator always has a perfectly deterministic behaviour, as expected. Of course other solutions may exist, more or less optimized.

2. The phrases of a language consist of balanced sequences of nested round parentheses, e.g., $(\,)\,\Big(\,(\,(\,)\,)\;(\,)\,\Big)\,(\,)$. The grammar below generates this language (axiom S):

$$\begin{cases} 1\colon\ S \rightarrow T \\ 2\colon\ T \rightarrow T\ T \\ 3\colon\ T \rightarrow (\ T\ ) \\ 4\colon\ T \rightarrow (\ ) \end{cases}$$

The *nesting level* of a parenthesis pair is defined as usual, assuming that the value for such a level starts from 1 (for the outermost parentheses). Thus the expression:

$$(\,)\,\Big(\,(\,(\,)\,)\;(\,)\,\Big)\,(\,)$$

contains three pairs at nesting level 1, two pairs at level 2 and one pair at level 3.

We intend to compute, in two suitable attributes of the root node of the syntax tree, the number of parbaenthesis pairs that have an odd nesting level (1, 3, etc) and the number of those that have an even nesting level (2, 4, etc). For instance, in the above example there are four pairs at an odd nesting level and two pairs at an even level.

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 computation of the attributes *no* and *ne* in the tree root, as explained above. It is required to use the attributes illustrated in the table on the next page.

(b) Decorate the syntax tree of expression $(\,)\,\Big(\,(\,(\,)\,)\;(\,)\,\Big)\,(\,)$ with the attribute values. Use the syntax tree 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.

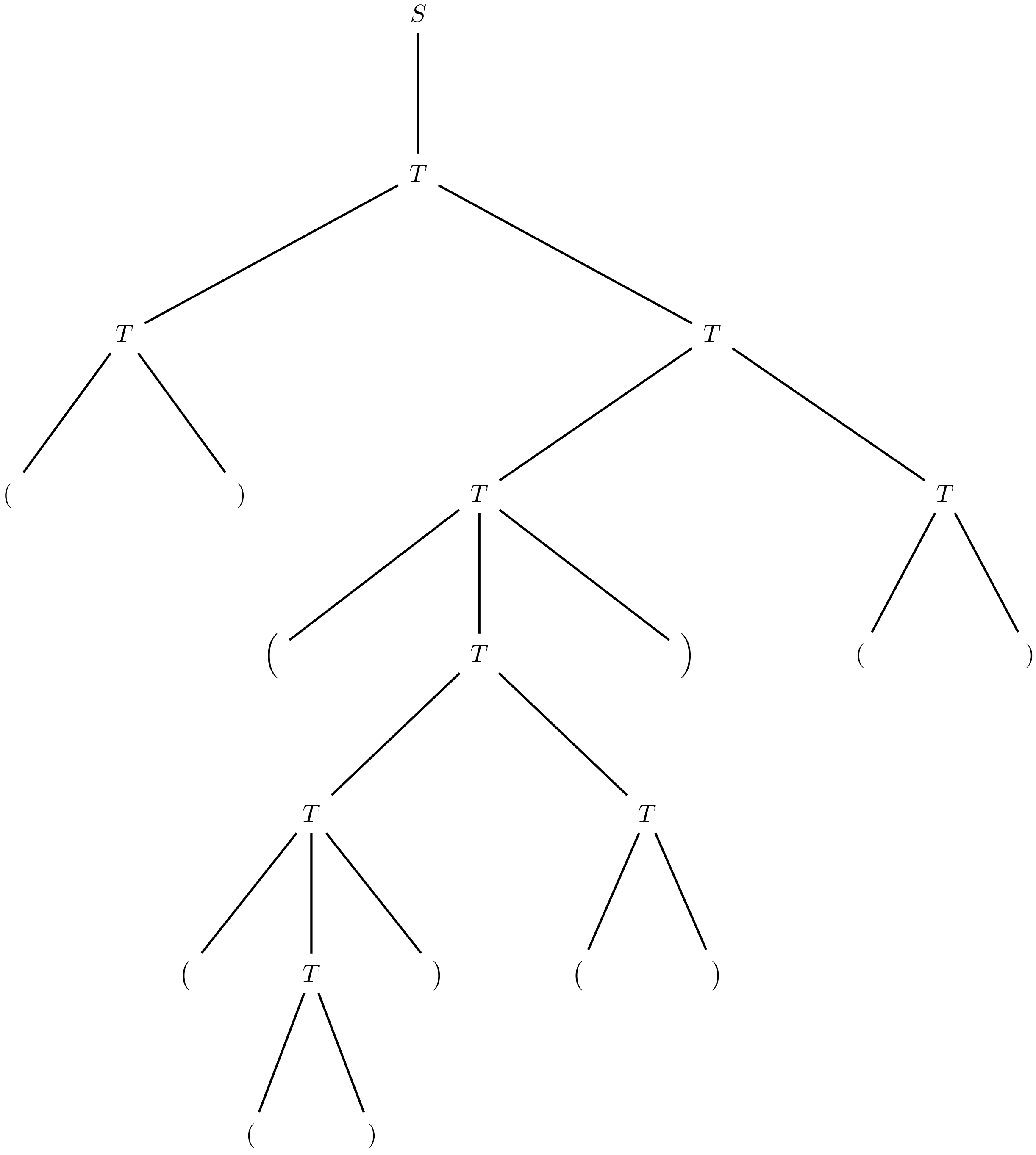
attributes assigned to be used

type	*name*	*domain*	*(non)term.*	*meaning*
right	*nep*	integer	T	number of nested parenthesis pairs that enclose (are around) the current nonterminal
left	*no*	integer	T, S	number of parenthesis pairs at an odd nesting level that are contained in the subtree rooted at the current nonterminal
left	*ne*	integer	T, S	number of parenthesis pairs at an even nesting level that are contained in the subtree rooted at the current nonterminal

attribute grammar to write - question (a)

#	*syntax*	*semantics*
1:	$S_0 \rightarrow T_1$	
2:	$T_0 \rightarrow T_1\ T_2$	
3:	$T_0 \rightarrow (\ T_1\ )$	
4:	$T_0 \rightarrow (\ )$	

syntax tree to decorate - question (b)



Solution

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

#	syntax	semantics
1:	$S_0 \to T_1$	$nep_1 := 0$ $no_0 := no_1$ $ne_0 := ne_1$
2:	$T_0 \to T_1\, T_2$	$nep_1 := nep_0$ $nep_2 := nep_0$ $no_0 := no_1 + no_2$ $ne_0 := ne_1 + ne_2$
3:	$T_0 \to (\ T_1\ )$	$nep_1 := nep_0 + 1$ $no_0 := \textsf{if } even(nep_0) \textsf{ then } no_1 + 1 \textsf{ else } no_1$ $ne_0 := \textsf{if } odd(nep_0) \textsf{ then } ne_1 + 1 \textsf{ else } ne_1$
4:	$T_0 \to (\ )$	$no_0 := \textsf{if } even(nep_0) \textsf{ then } 1 \textsf{ else } 0$ $ne_0 := \textsf{if } odd(nep_0) \textsf{ then } 1 \textsf{ else } 0$

The semantic functions of the inherited (right) attribute *nep* are listed first, then those of the two synthesized (left) ones *no* and *ne*. This attribute grammar works as follows:

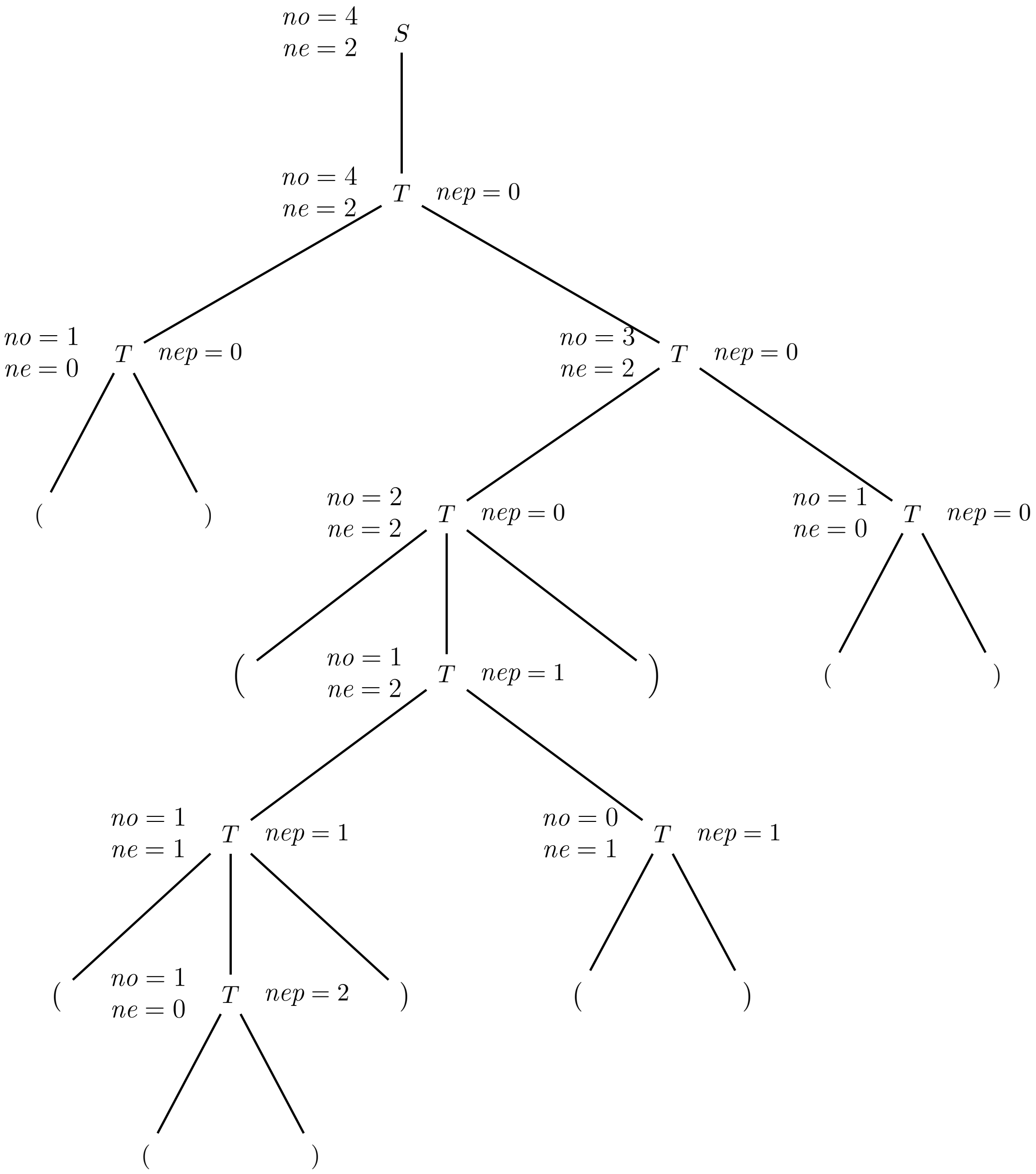
- by definition the inherited attribute *nep* counts the number of bracket pairs that enclose (are around) a nonterminal T; therefore on being passed top-down in the rule 3, which generates the root of a new T-subtree with one more bracket pair around, the attribute is incremented; and everywhere else it is either passed as-is (rule 2) or initialized (rule 1)
- by definition the synthesized attribute *no* counts the number of bracket pairs at odd nesting level that are contained in a T-subtree; therefore on being passed bottom-up in the rule 3, if the parent nonterminal T is enclosed in a bracket nest of even depth, then the attribute is incremented because the pair generated in that rule will thus be at odd depth, else it is passed as-is; following the same principle, the attribute is initialized in the rule 4; and everywhere else it is either accumulated (rule 2) or passed as-is (rule 1)
- accordingly, the synthesized attribute *ne* counts the number of bracket pairs at even nesting level that are contained in a T-subtree, so it is processed similarly to attribute *no*, though exchanging odd and even parities

The auxiliary predicates *even* and *odd* say if their argument is an even number or an odd one, respectively. One could use the remainder operator % of the C language instead, because $(even(x) == true) \iff (x \ \% \ 2 == 0)$ and $(odd(x) == true) \iff (x \ \% \ 2 == 1)$; these are implementation details.

This attribute grammar is clearly acyclic and by construction it is reasonably correct. Other solutions may exist, of course, with different attributes.

(b) Here is the decorated syntax tree of the expression $(\,)\Big(((\,))\,(\,)\Big)(\,)$:

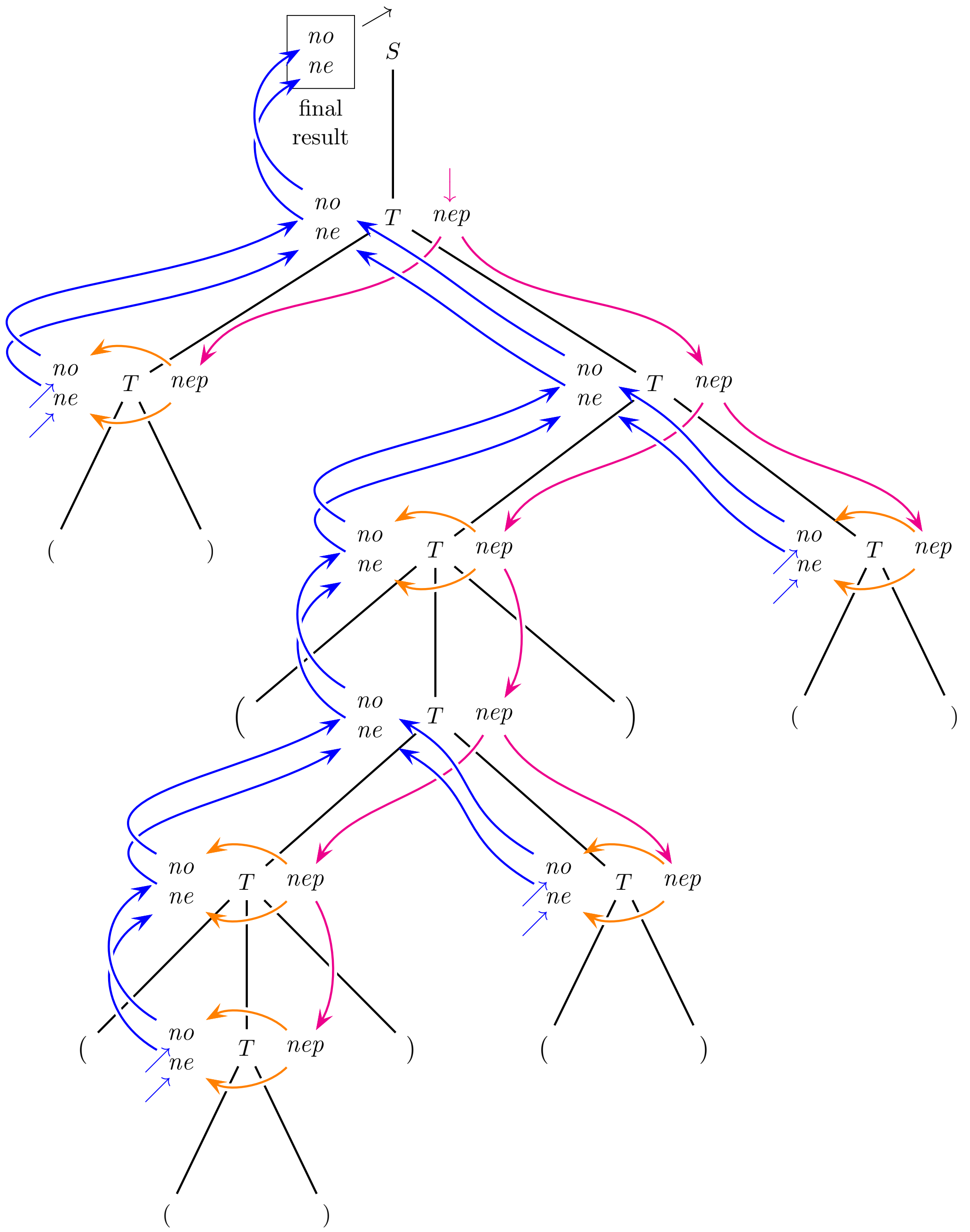
syntax tree decorated with the attribute values



As customary, the left or right attributes are placed on the left or right, respectively, of the nonterminal node they are associated with. The attributes are computed according to the grammar semantics and are visibly correct.

For completeness, here we show also an attribute dependence graph, by using the same sample syntax tree as before (without attribute values):

sample syntax tree with attribute dependences



Magenta arcs show the top-down (inherited) dependences, orange arcs show the transversal (right-to-left) ones (inherited but — in this particular case — internal to a node) and blue arcs show the bottom-up (synthesized) ones. The graph also shows that — for this attribute grammar — there are not any transversal dependences (right-to-left or left-to-right) between brother nodes (in fact the brother graph is empty, see point (c)). The entry and exit arrows indicate where the attributes are initialized and the final result is sent out, respectively.

(c) The right (inherited) attribute *nep* depends only its value in the parent node, so it satisfies the one-sweep condition. The left (synthesized) attributes *no* and *ne* depend on their values in the child nodes, and on the right attribute *nep* in the parent node (due to the predicates $even(nep_0)$ and $odd(nep_0)$ in the conditions of the if-then-else statements that compute no_0 and ne_0). Both dependence types are admissible for the one-sweep condition. Thus all the attributes satisfy the one-sweep condition, and the attribute grammar can be evaluated in one sweep (first top-down second bottom-up). See also the dependence graph before.

Concerning the L condition, notice that there are not any right attributes that depend on the values of attributes of the brother nodes, so that the brother graph is empty. As a consequence, the left-to-right syntactic ordering of the brother nodes is suited for the computation of the attributes (just like any other ordering). Therefore the attribute grammar trivially satisfies the L condition.

For completeness we show also the one-sweep pure semantic evaluator (with optimized semantic functions), which works fine provided the syntax tree is pre-built:

```
procedure T (in t, nep0; out no0, ne0)
var t1, t2, nep1, nep2, no1, no2, ne1, ne2
begin
    switch (rule of the root of tree t) do
        case 2 do
            t1 := left T-subtree of t
            nep1 := nep0
            call T (t1, nep1; no1, ne1)
            t2 := right T-subtree of t
            nep2 := nep0
            call T (t2, nep2; no2, ne2)
            no0 := no1 + no2
            ne0 := ne1 + ne2
        case 3 do
            t1 := T-subtree of t
            nep1 := nep0 +1
            call T (t1, nep1; no1, ne1)
            if (nep0 %2 == 0) then
                no0 := no1 + 1
                ne0 := ne1
            else
                no0 := no1
                ne0 := ne1 + 1
        case 4 do
            if (nep0 %2 == 0) then
                no0 := 1
                ne0 := 0
            else
                no0 := 0
                ne0 := 1
```

Algorithm: Semantic evaluator - I

```
procedure S (in t; out no0, ne0)
var t1, nep1, no1, ne1
begin
    t1 := T-subtree of t
    nep1 := 0
    call T (t1, nep1; no1, ne1)
    no0 := no1
    ne0 := ne1
```

Input: syntax tree (pre-built)
Output: attrib. *no*, *ne* in the root

```
program SEMANTIC_EVALUATOR
var t, no, ne
begin
    t := syntax tree
    call S (t; no, ne)
    print values no, ne
```

Algorithm: Semantic evaluator - II

Notice that the syntactic support is ambiguous (rule 2 has a two-sided recursion), thus it is not deterministic and it is impossible to design an integrated syntactic-semantic analyzer.