FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Tue 15 JANUARY 2019 - Part Theory

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

LAST + FIRST NAME:

(capital letters please)

MATRICOLA: SIGNATURE:

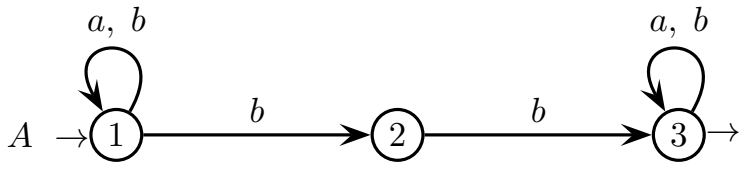
(or PERSON CODE)

INSTRUCTIONS - READ CAREFULLY:

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

1 Regular Expressions and Finite Automata 20%

1. Consider the nondeterministic finite-state automaton A below, over a two-letter alphabet $\Sigma = \{ a, b \}$:



Answer the following questions:

(a) Find a deterministic version A' of automaton A, by using the Berry-Sethi (BS) method.

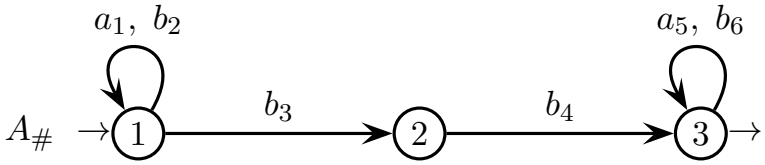
(b) By using a systematic method, determine whether the deterministic automaton A' is minimal, and if it is not so, find the equivalent minimal automaton.

(c) By using a systematic method, find an automaton $\overline{A}$ that accepts language $\overline{L}$, i.e., the set complement of language $L(A)$.

(d) (optional) Find a regular expression R that generates the complement language $\overline{L}$, by applying the Brzozowski-McCluskey (BMC) method to automaton $\overline{A}$.

Solution

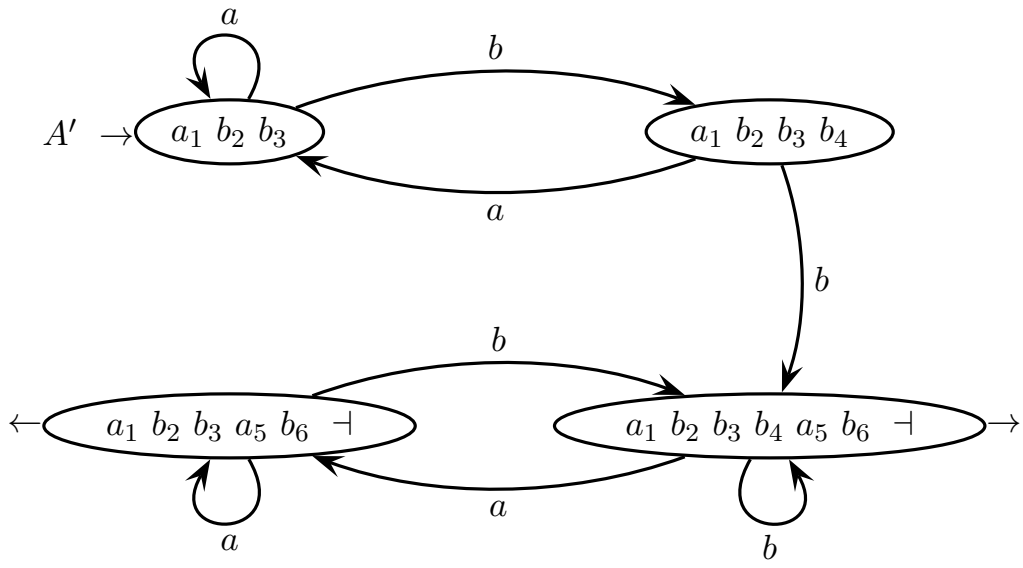
(a) Here is how to apply the Berry-Sethi (BS) method to the nondeterministic automaton A and find a deterministic version A' thereof. To start, the numbered automaton $A_\#$ is:



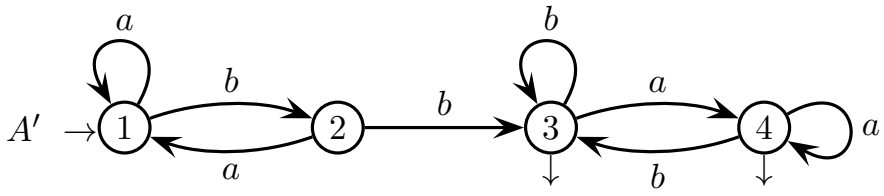
Then, the initials and followers of the numbered automaton $A_\#$ are:

initials	$a_1\ b_2\ b_3$
generators	followers
a_1	$a_1\ b_2\ b_3$
b_2	$a_1\ b_2\ b_3$
b_3	b_4
b_4	$a_5\ b_6\ \dashv$
a_5	$a_5\ b_6\ \dashv$
b_6	$a_5\ b_6\ \dashv$

Finally, the resulting deterministic BS automaton A', equivalent to A, is:

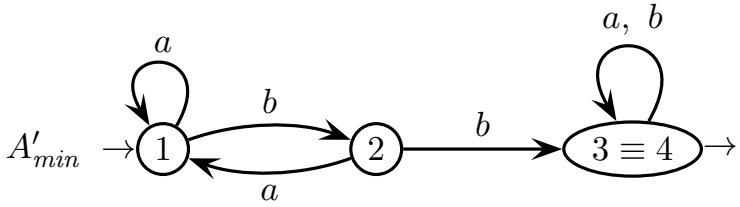


or, with a change of state names and a layout rearrangement:



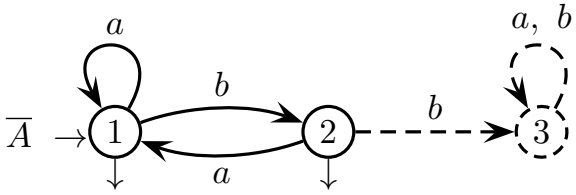
(b) The deterministic automaton A' is not minimal. We can see that the final states 3 and 4 constitute a trap group, which once entered cannot be exited any longer, and that both such states have an outgoing arc per each input letter; thus these two final states are undistinguishable. Instead, the two non-final states 1 and 2 are distinguishable, because by letter b they enter states 2 and 3, respectively, which are non-final and final, respectively, and therefore distinguishable.

The same conclusion would be obtained by using the classical triangular implication table. Anyway, here is the minimal form A'_{min} of automaton A':

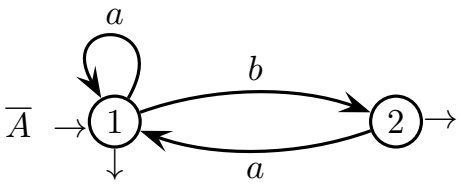


Stop for a moment and test whether automaton A'_{min} is correct. The simple (and visibly ambiguous) structure of automaton A clearly shows that A accepts any string that contains two adjacent letters b, from the latter b onwards. At the beginning, automaton A'_{min} does not accept anything, until it inputs two adjacent letters b, then it accepts everything. Thus A'_{min} is equivalent to A.

(c) The complement $\overline{A}$ of automaton A has to be computed starting from a deterministic machine, otherwise a pseudo-complement automaton is obtained, which in general is incorrect. We can choose between automata A' and A'_{min}, both equivalent to A, and of course the minimal form A'_{min} is preferable, as it has fewer states. Automaton A'_{min} is already in the naturally complete form, since every state has an outgoing arc per each input letter. Thus an error state is unnecessary, and the complement $\overline{A}$ is obtained from A'_{min} just by switching the final and non-final states. Here is the result (state $3 \equiv 4$ is shortened into 3):

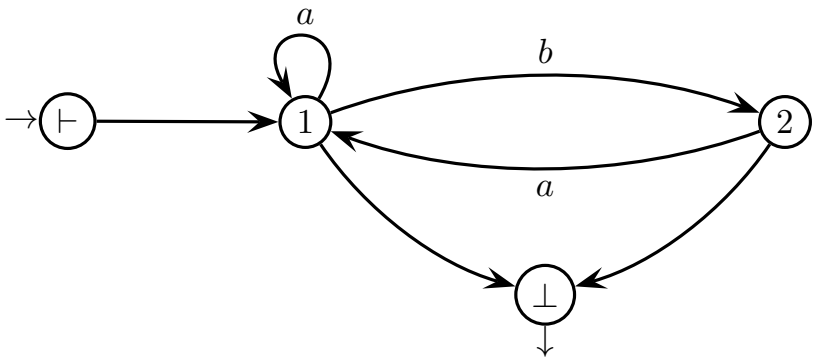


This state-transition graph of $\overline{A}$ is not in clean form. State 3 is undefined, as it cannot reach any final state, thus it is useless and eliminable, with all its arcs (dashed). States 1 and 2 are clearly useful instead. Here is the clean form of $\overline{A}$:

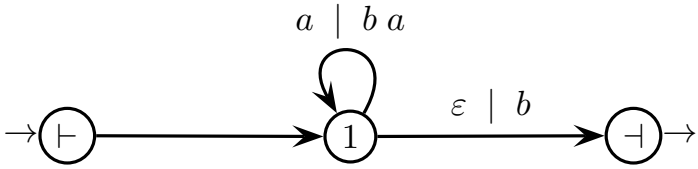


It is evident that automaton $\overline{A}$ (in clean form) is minimal as well, since state 1 has an outgoing b-arc and state 2 does not, thus such states are distinguishable. Is automaton $\overline{A}$ correct ? As it only has final states, it accepts any input, until it finds two adjacent letters b. On scanning the latter b, it falls into error (we could add an error state and make this evident) and from there on it does not accept any more input. Thus, it behaves exactly as the complement of automaton A.

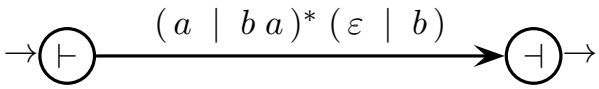
(d) It is convenient to apply the Brzozowski-McCluskey (BMC) method to the clean version of the complement automaton $\overline{A}$. Since the initial and final nodes have ingoing and outgoing arcs, respectively, the automaton has to be normalized:



Eliminate node 2:



Eliminate node 1:



In conclusion, with some simplification, a regular expression R that generates the complement language $\overline{L} = L(\overline{A}) = \overline{L(A)}$ is as follows:

$$R = (a \mid b\,a)^*\,(\varepsilon \mid b) = \big((\varepsilon \mid b)\,a\big)^*\,(\varepsilon \mid b) = \big([b]\,a\big)^*\,[b]$$

which uses the optionality operator "$[\,]$" to compact the subexpression $\varepsilon \mid b$. It does not take much to verify that the regular expression R is correct. By a quick glance at automaton A, language L consists of any string that contains two adjacent letters b. The complement language $\overline{L}$ must not have them. Expression R is able to generate any sequence of letters a (including the empty one ε), and this sequence optionally starts or ends by a single letter b, or optionally contains single letters b. Yet R is unable to generate two adjacent letters b, because at least one letter a slips in between. Thus, R generates exactly language $\overline{L}$.

2 Free Grammars and Pushdown Automata 20%

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

$$L = \Big\{\ \underbrace{a^n\ \big(\,b\,c^{k_i}\,\big)^n}_{\text{string model}}\ \Big|\ \ \underbrace{n \geq 0 \text{ and } \forall\ i \text{ with } 1 \leq i \leq n \text{ it holds } k_i \geq 1}_{\text{characteristic predicate}}\ \Big\}$$

   Five sample valid strings are:

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

   Notice that the number of consecutive letters c may differ in each substring $b\,c^{k_i}$. For instance, in relation to the sample valid string below, it holds:

$$a\ a\ \underbrace{b\ c\ c\ c}_{k_1=3}\ \underbrace{b\ c}_{k_2=1} \qquad \text{with } n = 2 \text{ thus } 1 \leq i \leq 2$$

   Answer the following questions:

   (a) Write a *BNF* grammar G, not ambiguous, that generates language L. Test grammar G by drawing the syntax tree of the fifth sample string listed above:

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

   (b) Formally prove or at least convincingly argue that grammar G is not ambiguous.

   (c) (optional) Consider the context-free language L' below:

$$L' = L\ L^R$$

   where language L^R is the mirror image of language L. Write a set representation of language L^R, by changing that of L. Then write a *BNF* grammar G', not ambiguous by construction, that generates language L', and explain why the construction is not ambiguous.

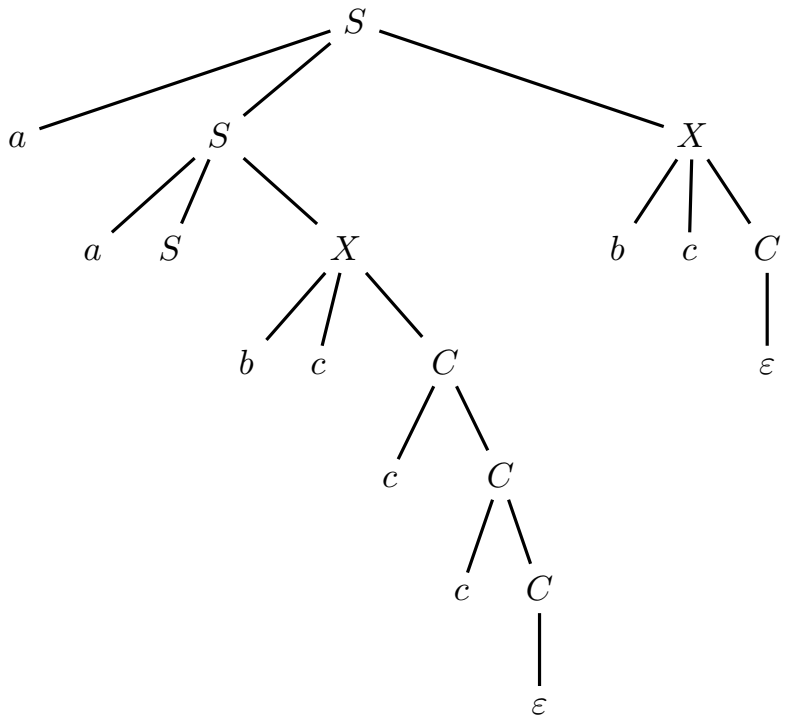
Solution

(a) Here is a possible *BNF* grammar G (axiom S) that generates language L:

$$G \begin{cases} 1\colon\ S \rightarrow a\ S\ X & // \ n \geq 1 \\ 2\colon\ S \rightarrow \varepsilon & // \ n = 0 \\ 3\colon\ X \rightarrow b\ c\ C & // \ k_i \geq 1 \\ 4\colon\ C \rightarrow c\ C & \\ 5\colon\ C \rightarrow \varepsilon & \end{cases} \quad \begin{array}{l} Gui = \{\ a\ \} \\ Gui = \{\ b,\ \dashv\ \} \\ \\ Gui = \{\ c\ \} \\ Gui = \{\ b,\ \dashv\ \} \end{array}$$

Grammar G generates a nest structure $S \stackrel{*}{\Rightarrow} a^n\ X^n$ $(n \geq 0)$ through the self-embedding recursive (axiomatic) rule 1, then it replaces each instance of nonterminal X with a start-marked list $b\ c^+$ through the right-linear recursive rule 4. Thus grammar G reasonably generates language L and is therefore correct.

Here is the syntax tree of the valid string $a\,a\,b\,c\,c\,c\,b\,c$ according to grammar G:



This syntax tree demonstrates the structure of grammar G, described above.

(b) Here is a formal proof that grammar G is not ambiguous. It is easy to see that grammar G is $LL(1)$ (see also question (a)). The guide sets of the alternative rules 1 and 2 for the axiom S are $\{\ a\ \}$ and $\{\ b,\ \dashv\ \}$, and they are disjoint. The guide sets of the alternative rules 4 and 5 for nonterminal C are $\{\ c\ \}$ and $\{\ b,\ \dashv\ \}$, and they are also disjoint. Being $LL(1)$, surely grammar G is not ambiguous.

Alternatively, grammar G generates a structure $a^n\ X^n$ $(n \geq 0)$, through the well-known non-ambiguous rule $1\colon S \rightarrow a\ S\ X$, where symbol X is replaced by the regular language $b\ c^+$, generated through the well-known non-ambiguous

right-linear rule 4: $C \to c\,C$. Furthermore, there is no concatenation ambiguity between consecutive symbols X, since each of them starts by a letter b and ends by a letter c. Thus grammar G is entirely made of non-ambiguous components, which are combined in a non-ambiguous way, therefore it is not ambiguous.

(c) Here is a possible set representation of language L^R, the mirror image of L:

$$L^R = \{\ \underbrace{\left(c^{k_i}\ b\right)^n\ a^n}_{\text{string model}}\ \mid\ \underbrace{n \geq 0 \text{ and } \forall\ i \text{ with } 1 \leq i \leq n \text{ it holds } k_i \geq 1}_{\text{characteristic predicate}}\ \}$$

Clearly a grammar G_R for language L^R is as follows (axiom S_R):

$$G_R \begin{cases} S_R \to X_R\ S_R\ a \\ S_R \to \varepsilon \\ X_R \to C_R\ c\ b & // \ k_i \geq 1 \text{ for any } i \geq 1 \\ C_R \to C_R\ c \\ C_R \to \varepsilon \end{cases}$$

obtained through mirroring all (the right parts of) the five rules of grammar G. Grammar G_R is not ambiguous either. But of course, it is *not* $LL(1)$, as it is *left-recursive* (see nonterminal C_R), and it is *not* $ELL(1)$ either (for the same reason). Thus we may not conclude as quickly as before that it is not ambiguous. However, the above alternative reasoning for proving that grammar G is unambiguous applies to G_R as well (with the obvious changes – switch left and right), and we let the reader rephrase the reasoning and provide the details.

The concatenation construction of a grammar G' for language L' is well-known. Put together both grammars G and G_R, and add the new axiomatic rule $S' \to S\ S_R$. Anyway, this construction suffers from concatenation ambiguity, since for instance string $\underbrace{a\,b\,c^3\,b\,a}_{L'}$ can be generated as $\underbrace{a\,b\,c^2}_{L} \cdot \underbrace{c\,b\,a}_{L^R}$ or $\underbrace{a\,b\,c}_{L} \cdot \underbrace{c^2\,b\,a}_{L^R}$. To eliminate ambiguity, slightly extend grammar G_R into an equivalent new one G'_R that generates only one letter c in the first position of the string (axiom S'_R):

$$G'_R \begin{cases} S'_R \to c\ b\ S_R\ a & // \ k_1 = 1 \\ S'_R \to \varepsilon & // \ n = 0 \end{cases} \qquad G_R \begin{cases} S_R \to X_R\ S_R\ a & // \ n \geq 2 \\ S_R \to \varepsilon & // \ n = 1 \\ X_R \to C_R\ c\ b & // \ k_i \geq 1 \text{ for any } i \geq 2 \\ C_R \to C_R\ c \\ C_R \to \varepsilon \end{cases}$$

The new axiomatic rule of grammar G' becomes $S' \to S\ S'_R$. It is evident that grammar G'_R is not ambiguous either, and that there is no concatenation ambiguity between grammars G and G'_R. Thus grammar G' is no longer ambiguous. Other solutions may exist, as well as simplifications of the one above, for instance by sharing some rules between grammars G and G'_R.

2. Consider a script language that defines simple regular expressions, like for instance:

$$e_1 = (\, a \;\mid\; b\,)^{+} \cdot \big(\, c \cdot [\, d\,]\,\big)^{*}$$

according to the following rules and constraints:

- an expression may include the following n-ary operators (with $n \geq 2$): alternative "$\mid$" and concatenation "$\cdot$", like $a \mid b \mid \ldots$ and $a \cdot b \cdot \ldots$
- an expression may include the following unary operators: Kleene star iterator "$*$", cross iterator "$+$" and optionality "$[\,]$"
- the immediate nesting of any operator into itself, like in $\big(\, a \;\mid\; (\, b \;\mid\; c\,)\big)$, is forbidden
- the immediate nesting of any two unary operators, like in $(\, b^{+}\,)^{*}$, is forbidden
- an expression name, like the name e_1 above, is an instance of the grammar terminal symbol id
- the alphabetical symbols of an expression, like the letters a, b, c and d in the expression e_1 above, are instances of the grammar terminal symbol sym
- a language phrase consists of a non-empty list of expression definitions separated by semicolons "$;$", and the whole list is terminated by a dot "$\cdot$"

Here is a short sample script, which just defines the sample expression e_1 above:

```
regexp e1 is
   conc
      cross
         alt
           a, b
         endalt
      endcross,
      star
         conc
            c,
            opt
               d
            endopt
         endconc
      endstar
   endconc
endregexp.
```

You are invited to infer from this sample script any information that is not explicitly provided by the rules and constraints listed above.

Write a grammar, possibly of type *EBNF* and not ambiguous, that generates the script language described above.

Solution

Here is a viable grammar, of type *EBNF* and not ambiguous (axiom RE_LST):

$$
\left\{
\begin{array}{rcl}
\langle \mathsf{RE\_LST} \rangle & \to & \langle \mathsf{RE\_DEF} \rangle \ (\ \text{`;'}\ \langle \mathsf{RE\_DEF} \rangle\ )^{*}\ \text{`.'} \\
\hline
\langle \mathsf{RE\_DEF} \rangle & \to & \mathsf{regexp\ id\ is}\ \langle \mathsf{RE\_EXP} \rangle\ \mathsf{endregexp} \\
\hline
\langle \mathsf{RE\_EXP} \rangle & \to & \langle \mathsf{N\_ARY} \rangle \mid \langle \mathsf{UNARY} \rangle \mid \mathsf{sym} \\
\hline
\langle \mathsf{N\_ARY} \rangle & \to & \langle \mathsf{ALTR} \rangle \mid \langle \mathsf{CONC} \rangle \\
\langle \mathsf{UNARY} \rangle & \to & \langle \mathsf{STAR} \rangle \mid \langle \mathsf{CROSS} \rangle \mid \langle \mathsf{OPT} \rangle \\
\hline
\langle \mathsf{ALTR} \rangle & \to & \mathsf{alt}\ \langle \mathsf{A\_ARG} \rangle\ (\ \text{`,'}\ \langle \mathsf{A\_ARG} \rangle\ )^{+}\ \mathsf{endalt} \\
\langle \mathsf{CONC} \rangle & \to & \mathsf{conc}\ \langle \mathsf{C\_ARG} \rangle\ (\ \text{`,'}\ \langle \mathsf{C\_ARG} \rangle\ )^{+}\ \mathsf{endconc} \\
\langle \mathsf{STAR} \rangle & \to & \mathsf{star}\ (\ \langle \mathsf{N\_ARY} \rangle \mid \mathsf{sym}\ )\ \mathsf{endstar} \\
\langle \mathsf{CROSS} \rangle & \to & \mathsf{cross}\ (\ \langle \mathsf{N\_ARY} \rangle \mid \mathsf{sym}\ )\ \mathsf{endcross} \\
\langle \mathsf{OPT} \rangle & \to & \mathsf{opt}\ (\ \langle \mathsf{N\_ARY} \rangle \mid \mathsf{sym}\ )\ \mathsf{endopt} \\
\hline
\langle \mathsf{A\_ARG} \rangle & \to & \langle \mathsf{CONC} \rangle \mid \langle \mathsf{UNARY} \rangle \mid \mathsf{sym} \\
\langle \mathsf{C\_ARG} \rangle & \to & \langle \mathsf{ALTR} \rangle \mid \langle \mathsf{UNARY} \rangle \mid \mathsf{sym}
\end{array}
\right.
$$

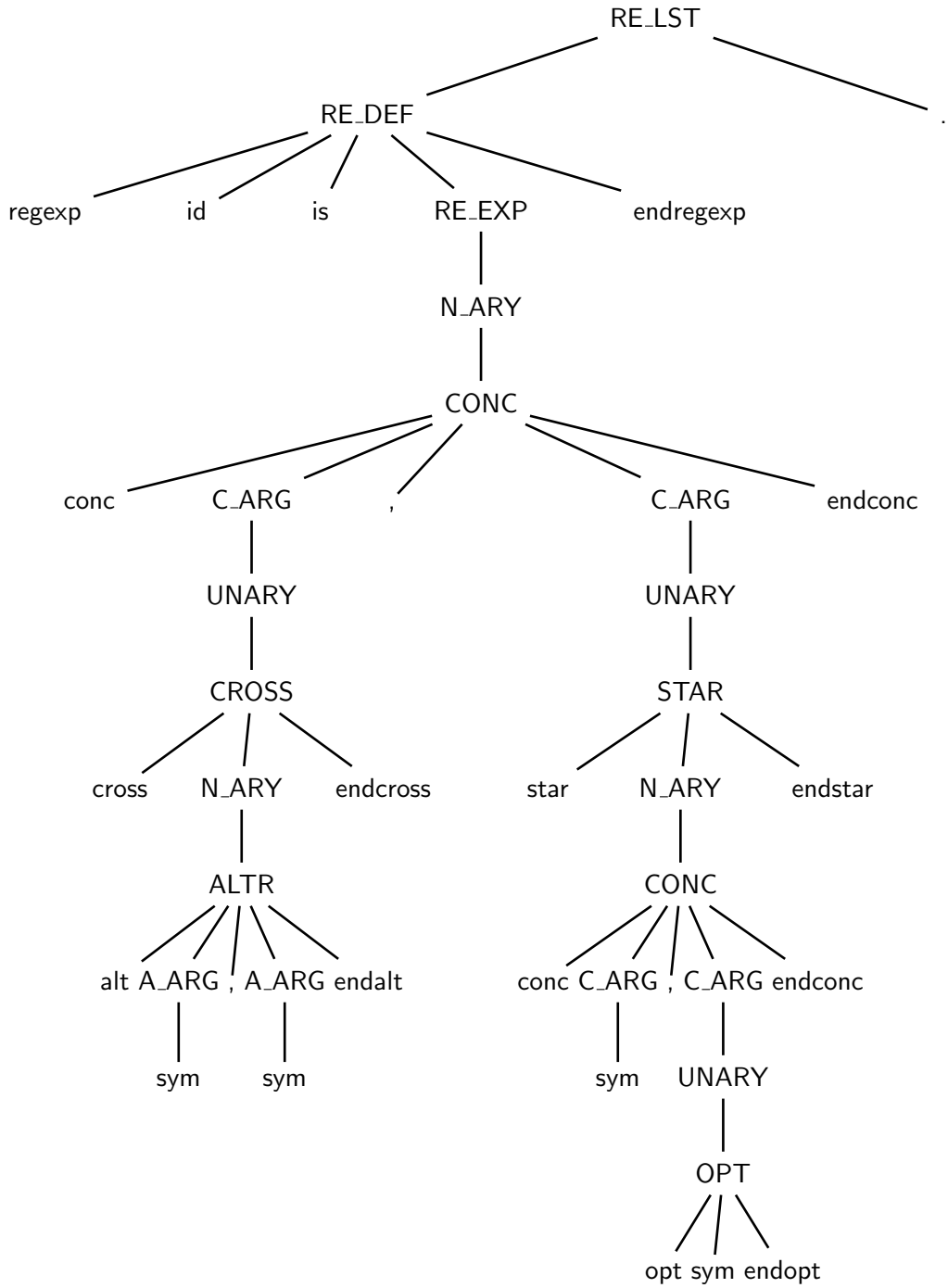
The grammar models the language specifications straightforwardly, thus it is reasonably correct. The only relevant inference from the sample script is that the two or more operands (arguments) of an alternative or concatenation operator (both are n-ary with $n \geq 2$) are denoted as a list with elements separated by commas. The forbidden immediate nestings are thoroughly modeled by the various rules that expand each operator (group ALTR–OPT) or operand (A_ARG and C_ARG) type. To verify the grammar operationally, the reader can draw the syntax tree of the sample script.

The grammar is of type *EBNF* and consists of non-ambiguous structures (lists and alternatives), which are combined in a non-ambiguous way (there is no union or concatenation ambiguity), thus it is convincingly not ambiguous. If one wants a more rigorous unambiguity proof, it does not take much to verify that the grammar is *ELL*(1), by converting the grammar rules into a machine net, computing the necessary guide sets and checking the absence of conflicts.

Other solutions are possible. For instance, combinations like ⟨N_ARY⟩ | sym and ⟨UNARY⟩ | sym are used quite often and uniformly, and it seems reasonable to categorize them with two more nonterminals and rules, like: ⟨NS⟩ → ⟨N_ARY⟩ | sym and ⟨US⟩ → ⟨UNARY⟩ | sym. Thus we could write ⟨STAR⟩ → star ⟨NS⟩ endstar (similarly for CROSS and OPT) and ⟨A_ARG⟩ → ⟨CONC⟩ | ⟨US⟩ (similarly for C_ARG). This makes the grammar less redundant, though possibly less immediate.

For completeness, here is the syntax tree of the sample script for the expression e_1:

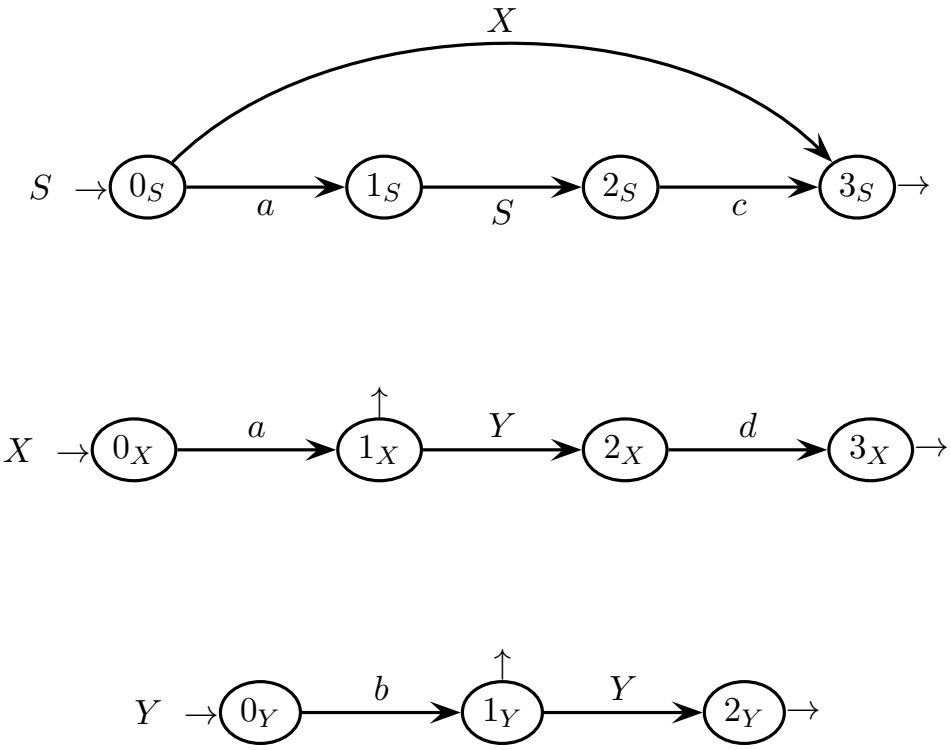
$$e_1 = (\, a \;\mid\; b\,)^{+} \cdot \big(\, c \cdot [\, d\,]\,\big)^{*}$$



This tree is correct and exerts most grammar rules, if not all of them. Thus it provides a reasonable operational evidence of the grammar correctness. In particular, notice how the immediate nesting of operators is modeled, according to the specifications, and how the forbidden nestings are excluded.

3 Syntax Analysis and Parsing Methodologies 20%

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



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

(a) Draw the complete pilot of grammar G (with all the m-states), determine whether grammar G is $ELR(1)$ and explain why yes or no.

(b) Write all the guide sets of the machine net of grammar G (on the call arcs and exit arrows), show that grammar G is *not* $ELL(1)$ and list all the guide set conflicts that are present.

(c) Determine whether grammar G is $ELL(2)$ and explain why yes or no.

(d) (optional) Give a representation of language $L(G)$ as a set of strings, like:

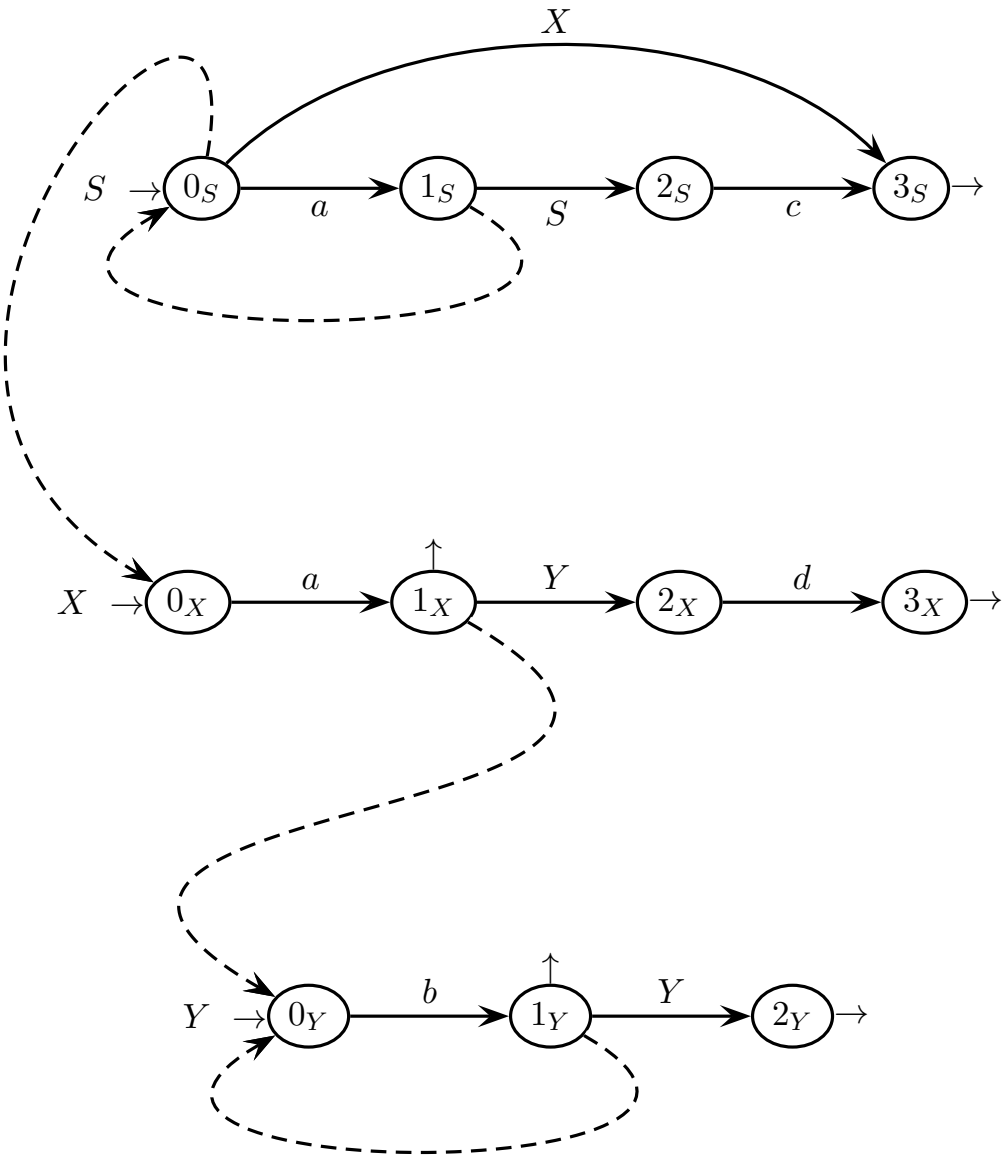
$$L(G) = \{ \text{ string model(s) } | \quad \text{characteristic predicate(s) } \}$$

In a convincing way, argue whether language $L(G)$ is regular or not.

---

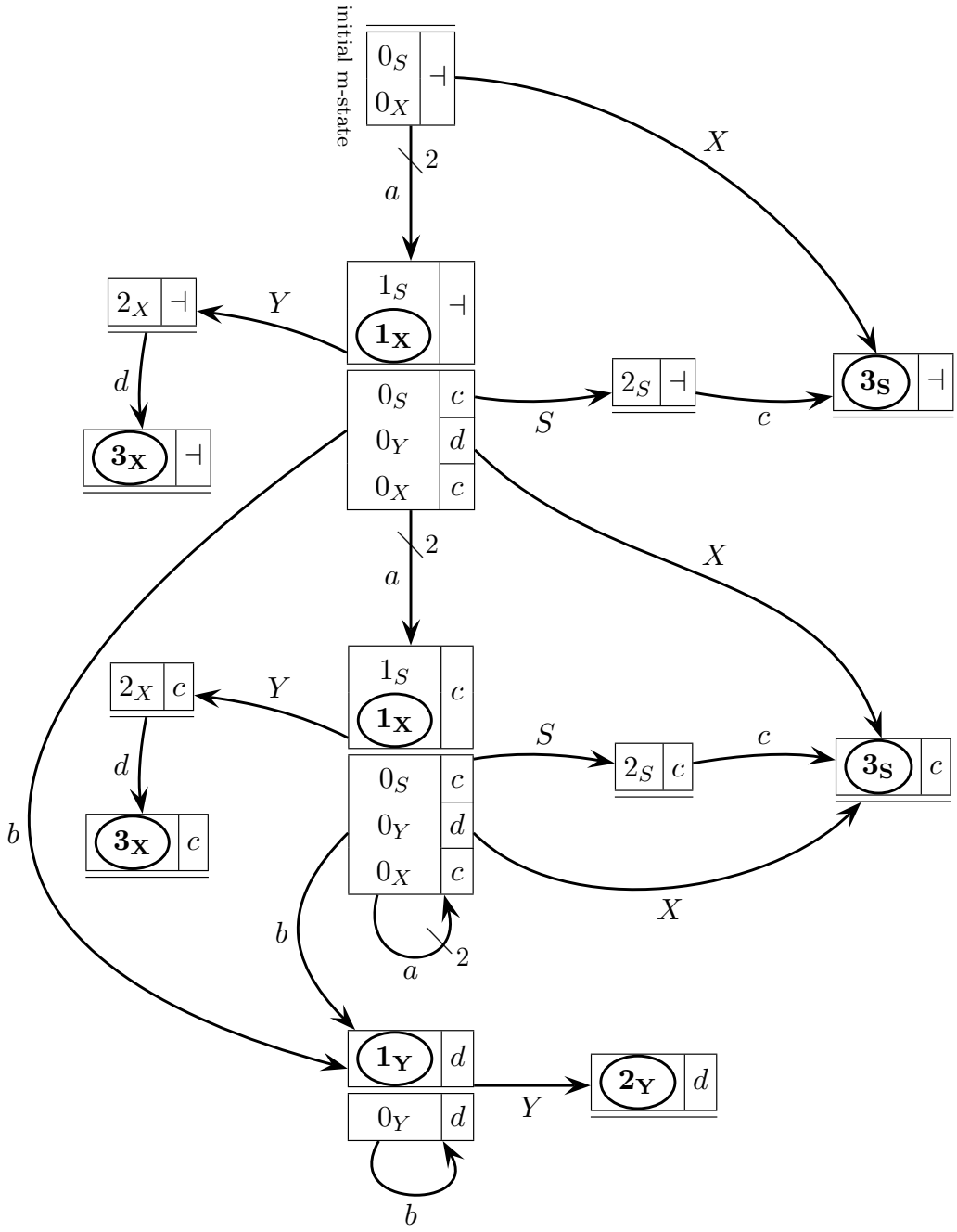
question (a) – please draw here the pilot of grammar G

question (b) – please write here all the guide sets on the call arcs and exit arrows



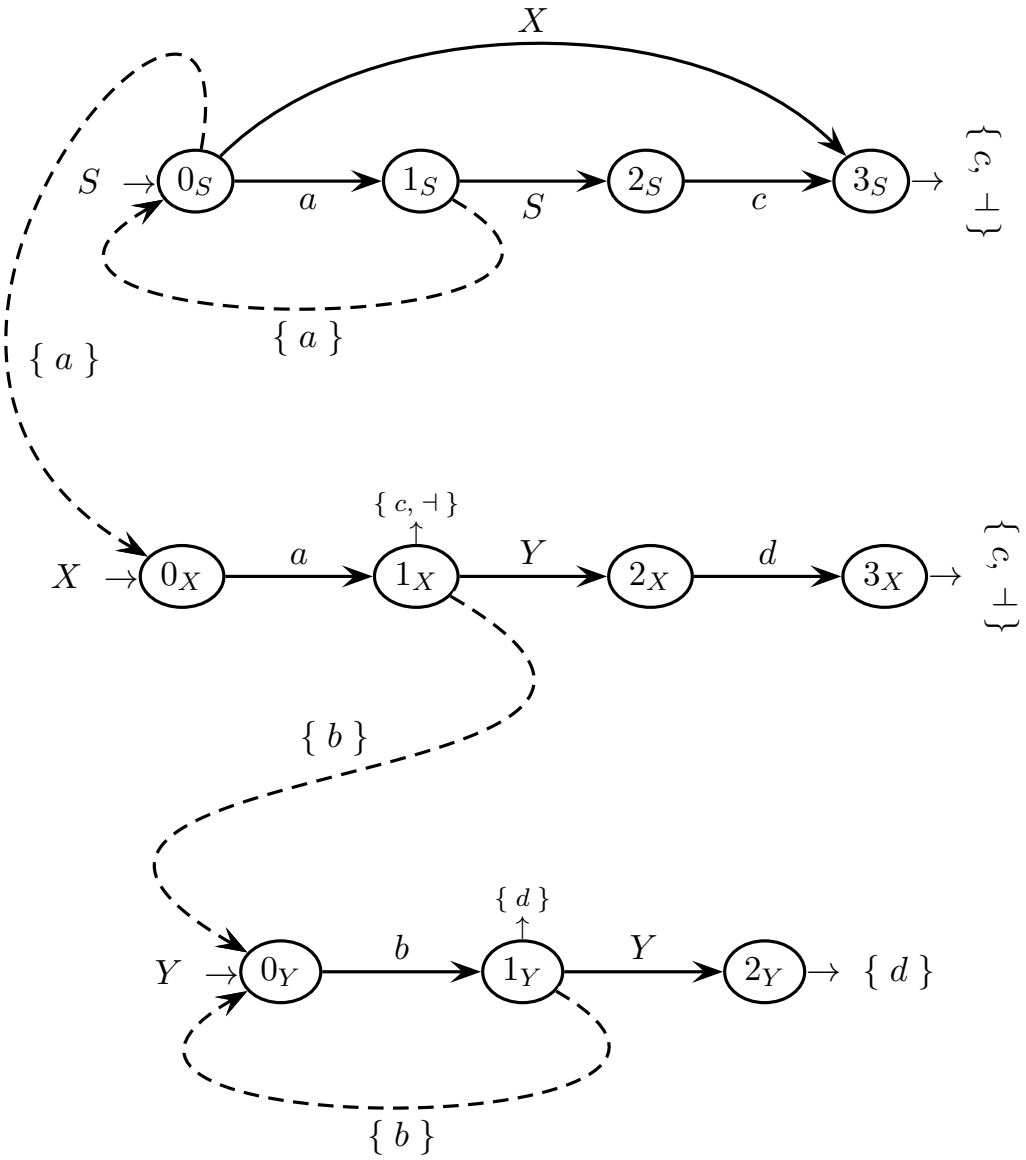
Solution

(a) Here is the complete pilot of grammar G (with 13 m-states):



The pilot is $ELR\,(1)$, as it does not exhibit any shift-reduce, reduce-reduce or convergence conflicts. Notice that it does not have the *STP*, as it has two m-states with a base that contains two items, or equivalently it has double transitions, namely the three a-arcs (though of course none of them is convergent). Being the pilot deterministic, of course grammar G is not ambiguous.

(b) Here are all the guide sets on the call arcs and exit arrows (with $k = 1$):



There is a guide set conflict on the bifurcation state 0_S, caused by letter a, namely $Gui(0_S \xrightarrow{a} 1_S) = Gui(0_S \dashrightarrow 0_X) = \{\, a \,\}$. Thus grammar G is not $ELL(1)$. This is the only conflict present in the $PCFG$ of G, anyway, as the other two bifurcation states (1_X and 1_Y) are conflict-free.

(c) From question (b), the machine net of grammar G has only one conflict, between the guide sets $Gui\,(0_S \xrightarrow{a} 1_S) = Gui\,(0_S \dashrightarrow 0_X) = \{\, a \,\}$, which have a look-ahead width $k = 1$. This suffices to make grammar G not $ELL\,(1)$. However, it turns out that grammar G is $ELL\,(2)$. In fact, in the net of G we may have:

terminal shift path of length 2	$Gui_2\,(0_S \xrightarrow{a} 1_S)$
$0_S \xrightarrow[\text{shift}]{a} 1_S \xrightarrow[\text{call } S]{} 0_S \xrightarrow[\text{shift}]{a} 1_S$	$\{\, a\,a \,\}$

where the *ELL* parser shifts twice on a with a call in between, and (3 cases):

terminal shift path of length 2	$Gui_2\,(0_S \dashrightarrow 0_X)$
1: $0_S \xrightarrow[\text{call } X]{} 0_X \xrightarrow[\text{shift}]{a} 1_X \xrightarrow[\text{call } Y]{} 0_Y \xrightarrow[\text{shift}]{b} 1_Y$	$\{\, a\,b \,\}$
2: $0_S \xrightarrow[\text{call } X]{} 0_X \xrightarrow[\text{shift}]{a} 1_X \xrightarrow[\text{exit } X]{} 3_S \xrightarrow[\text{exit } S]{} 2_S \xrightarrow[\text{shift}]{c} 3_S$	$\{\, a\,c \,\}$
3: $0_S \xrightarrow[\text{call } X]{} 0_X \xrightarrow[\text{shift}]{a} 1_X \xrightarrow[\text{exit } X]{} 3_S \xrightarrow[\text{exit } S]{} \dashv$	$\{\, a \dashv \,\}$
summary:	$\{\, a\,b,\ a\,c,\ a \dashv \,\}$

where (case 1) the *ELL* parser may do similarly to before, whereas, on exiting state 3_S, (case 2) it may return to state 2_S and shift on c, if the call to S is recursive, or (case 3) it may bump into the terminator $\dashv$, if the call to S is initial. Now, these two guide sets Gui_2 are disjoint, thus for a look-ahead width $k = 2$ the *PCFG* of G is deterministic and the grammar is $ELL\,(2)$.

(d) It turns out that the context-free language $L\,(G)$ is not regular. In fact, with a quick glance at the machine net of grammar G we can obtain:

$$
\begin{aligned}
L\,(Y) &= b^+ && \text{– by the unilinear recursive rule } Y \to b\,Y \mid b \\
L\,(X) &= a\,[\,L\,(Y)\,d\,] && \text{– by the non-recursive rule } X \to a\,Y\,d \mid a \\
L\,(S) &= \bigcup_{n \geq 0} a^n\,L\,(X)\,c^n && \text{– by the self-embedding rule } S \to a\,S\,X \mid X
\end{aligned}
$$

where "[ ]" is the optionality operator. Therefore, by the substitution of languages $L\,(Y)$ and $L\,(X)$, we have:

$$L\,(G) = \bigcup_{n \geq 0} a^n\,[\,b^+\,d\,]\,c^n$$

that is, through expanding the optionality operator "[ ]":

$$L\,(G) = \left\{\, a^n\,a\,b^k\,d\,c^n,\ a^n\,a\,c^n \mid \quad n \geq 0 \ \wedge\ k \geq 1 \,\right\}$$

or equivalently:

$$L\,(G) = \left\{\, \underbrace{a^{n+1}\,b^k\,d\,c^n,\ a^{n+1}\,c^n}_{\text{string models}} \mid \quad \underbrace{n \geq 0 \ \wedge\ k \geq 1}_{\text{characteristic predicate}} \,\right\}$$

Clearly language $L\,(G)$ is not regular, as it contains a nest structure $a^n \ldots c^n$, which cannot be generated by a finite device.

4 Language Translation and Semantic Analysis 20%

1. Consider the following source grammar G_s, over a four-letter terminal alphabet $\Sigma = \{ a,\ c,\ d,\ e \}$ and a two-symbol nonterminal alphabet $V = \{ S,\ P \}$ (axiom S):

$$G_s \begin{cases} S \ \rightarrow \ P\ S \\ S \ \rightarrow \ P \\ P \ \rightarrow \ d\ P\ a \\ P \ \rightarrow \ e\ P\ a \\ P \ \rightarrow \ c \end{cases}$$

which generates the source language L_s below:

$$L_s = \left\{ \ \left((\ d \ \mid \ e\ )^{k_i}\ c\ a^{k_i} \right)^n \ \mid \quad n \geq 1 \text{ and } \forall\ i \text{ with } 1 \leq i \leq n \text{ it holds } k_i \geq 0 \ \right\}$$

Answer the following questions (and do not change the source grammar G_s):

(a) Write a translation scheme that defines a translation function τ as follows (the symbols w_i indicate the substrings of letters d and e):

$$\tau\left(\ w_1\, c\, a^{|\, w_1\,|}\ \ldots\ w_n\, c\, a^{|\, w_n\,|}\right) = a^{|\, w_1\,|}\ f\ w_1^R\ \ldots\ a^{|\, w_n\,|}\ f\ w_n^R$$

For instance (here $n = 2$ and $k_1 = 3$, $k_2 = 2$):

$$\tau\left(\ \underbrace{e\,d\,d}_{w_1}\ c\ a\,a\,a\ \underbrace{d\,e}_{w_2}\ c\ a\,a\right) = a\,a\,a\ f\ \underbrace{d\,d\,e}_{w_1^R}\ a\,a\ f\ \underbrace{e\,d}_{w_2^R}$$

(b) Determine whether translation τ can be computed deterministically by an $ELL(1)$ or $ELR(1)$ parser enriched with suitable print actions, and motivate your answer. In the positive case, write the pseudo-code of the integrated syntax analysis and translation procedure for nonterminal P.

(c) Write a translation scheme that defines the inverse translation function τ^{-1}.

(d) (optional) Determine whether the translation function τ^{-1} can be computed deterministically by an $ELL(1)$ or $ELR(1)$ parser enriched with suitable print actions, and motivate your answer.

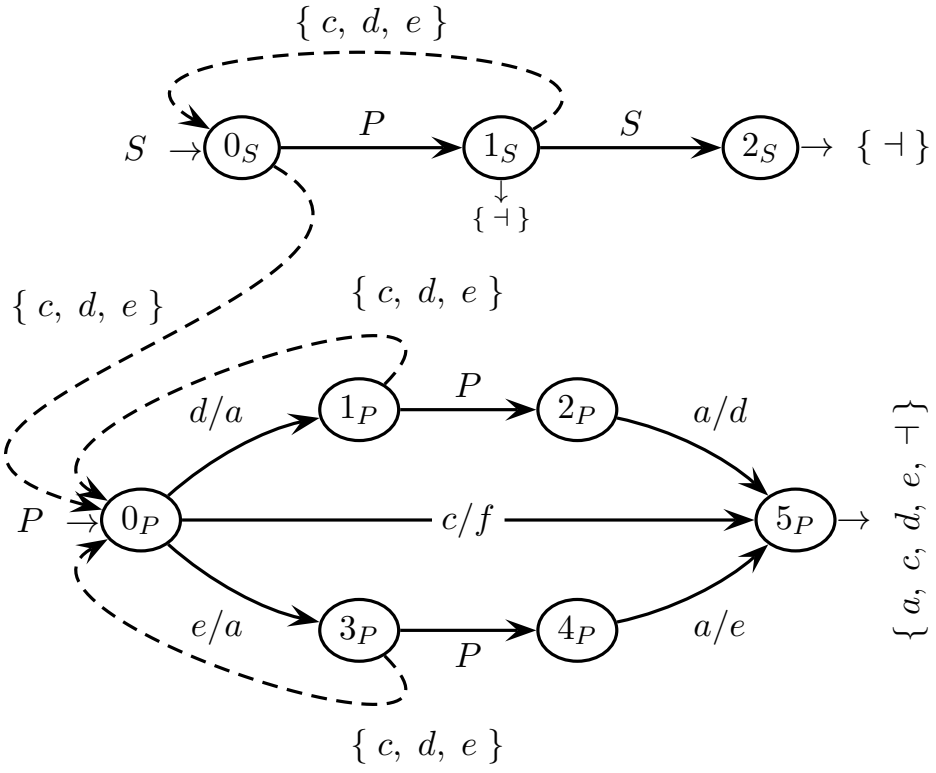
Solution

(a) Here is the destination grammar G_d for translation τ, in parallel with the source one G_s and (for clarity) with the combined translation grammar G_τ (axiom S):

$$G_s \begin{cases} S \to P\ S \\ S \to P \\ P \to d\ P\ a \\ P \to e\ P\ a \\ P \to c \end{cases} \qquad G_d \begin{cases} S \to P\ S \\ S \to P \\ P \to a\ P\ d \\ P \to a\ P\ e \\ P \to f \end{cases} \qquad G_\tau \begin{cases} S \to P\ S \\ S \to P \\ P \to \frac{d}{a}\ P\ \frac{a}{d} \\ P \to \frac{e}{a}\ P\ \frac{a}{e} \\ P \to \frac{c}{f} \end{cases}$$

Basically, the translation scheme G generates a list of nest structures $P \overset{*}{\Rightarrow} w\ c\ a^n$, with $w \in \{\ d,\ e\ \}^n$ and $n \geq 0$, through a well-known right-recursive rule $S \to P\ S$, and translates each such structure into the mirrored image, i.e., $a^n\ c\ w^R$, through well-known self-embedding recursive rules like $P \to \frac{d}{a}\ P\ \frac{a}{d}$ (same for letter e), similarly to how a palindrome can be mirrored. Scheme G can be operationally tested by drawing the translation tree of the sample string.

(b) The source grammar G_s is $ELL(1)$ (thus it is not ambiguous) and translation τ can be computed by means of a top-down parser enriched with suitable print actions. Here is the representation of scheme G as a translation net:



Consider only the source symbols: the guide sets on the bifurcation states, i.e., 1_S and 0_P, are disjoint, thus the source grammar is actually $ELL(1)$.

The recursive-descent integrated syntactic and translation procedure for nonterminal P is (with $\Sigma = \{\ a,\ c,\ d,\ e\ \}$ and $\Delta = \{\ a,\ d,\ e,\ f\ \}$):

```
procedure P                    // control flow of machine M_P
switch cc do                                        // enter 0_P
    case cc = d do                                  // look-ahead in 0_P
        cc := next                                  // go to 1_P
        write (a)                                   // and output a
        if cc ∈ { c, d, e } then                    // look-ahead in 1_P
            call P                                  // go to 2_P
            if cc = a then                          // look-ahead in 2_P
                cc := next                          // go to 5_P
                write (d)                           // and output d
                if cc ∈ Σ ∪ { ⊣ } then return       // exit 5_P
                else error                          // error in 5_P
            else error                              // error in 2_P
        else error                                  // error in 1_P
    case cc = e do                                  // look-ahead in 0_P
        cc := next                                  // go to 3_P
        write (a)                                   // and output a
        if cc ∈ { c, d, e } then                    // look-ahead in 3_P
            call P                                  // go to 4_P
            if cc = a then                          // look-ahead in 4_P
                cc := next                          // go to 5_P
                write (e)                           // and output e
                if cc ∈ Σ ∪ { ⊣ } then return       // exit 5_P
                else error                          // error in 5_P
            else error                              // error in 4_P
        else error                                  // error in 3_P
    case cc = c do                                  // look-ahead in 0_P
        cc := next                                  // go to 5_P
        write (f)                                   // and output f
        if cc ∈ Σ ∪ { ⊣ } then return               // exit 5_P
        else error                                  // error in 5_P
    otherwise do error                              // error in 0_P
```

By parametrizing each invocation of function *error* with a specific diagnostic message, related to the look-ahead mismatch, we can provide detailed error detection information and so help the programmer identify and correct the error.

Here is a procedure version with some programming optimization, by grouping the error cases at the procedure end:

```
procedure P                          // same actions as before
switch cc do
    case cc = d do
        cc := next
        write (a)
        if cc ∈ { c, d, e } then
            call P
            if cc = a then
                cc := next
                write (d)
                if cc ∈ Σ ∪ { ⊣ } then return
    case cc = e do
        cc := next
        write (a)
        if cc ∈ { c, d, e } then
            call P
            if cc = a then
                cc := next
                write (e)
                if cc ∈ Σ ∪ { ⊣ } then return
    case cc = c do
        cc := next
        write (f)
        if cc ∈ Σ ∪ { ⊣ } then return
    otherwise do error               // grouped error cases
```

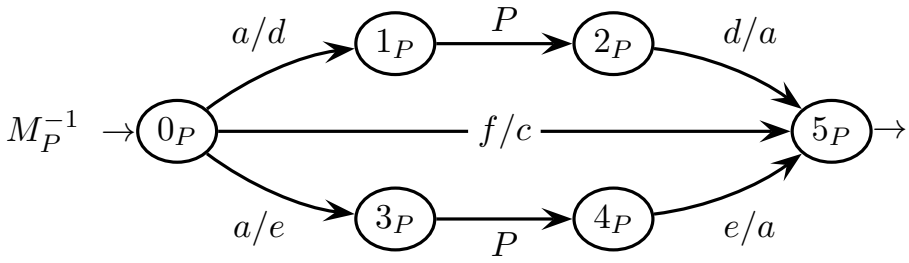
Of course, this optimized version is unable to distinguish the specific error cases.

(c) A scheme G^{-1} for the inverse translation τ^{-1} is obtained by switching the source and destination grammars G_s and G_d of the scheme G for translation τ:

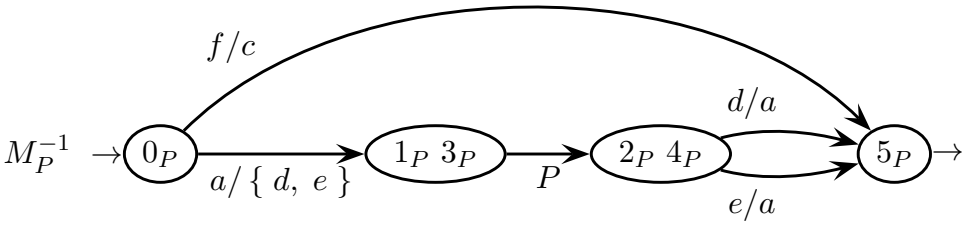
$$G_s^{-1} = G_d \begin{cases} S \to P\ S \\ S \to P \\ P \to a\ P\ d \\ P \to a\ P\ e \\ P \to f \end{cases} \qquad G_d^{-1} = G_s \begin{cases} S \to P\ S \\ S \to P \\ P \to d\ P\ a \\ P \to e\ P\ a \\ P \to c \end{cases}$$

It is easy to see that the source grammar G_s^{-1} for the inverse translation τ^{-1}, i.e., grammar G_d, is $ELL(1)$. Yet an $ELL(1)$ parser enriched with print actions is unable to compute translation τ^{-1}, because it cannot choose whether to print a character d or e immediately after scanning a character a. The reason is that the parser, i.e., procedure P, should do so ahead of the recursive call and thus before scanning in the input the corresponding character d or e.

In a more formal way, the transducer M_P^{-1} of the translation net of τ^{-1} can be derived from the transducer M_P of (the translation net of) τ (see question (b)) by switching the source and destination terminals. This corresponds to switching the source and destination grammars (see question (c)):



Yet the so-obtained transducer M_P^{-1} is not deterministic, as the underlying recognizer is not so: see the two a-arcs from state 0_P. Trying to determinize M_P^{-1} as a transducer, for instance by means of the subset construction, yields:

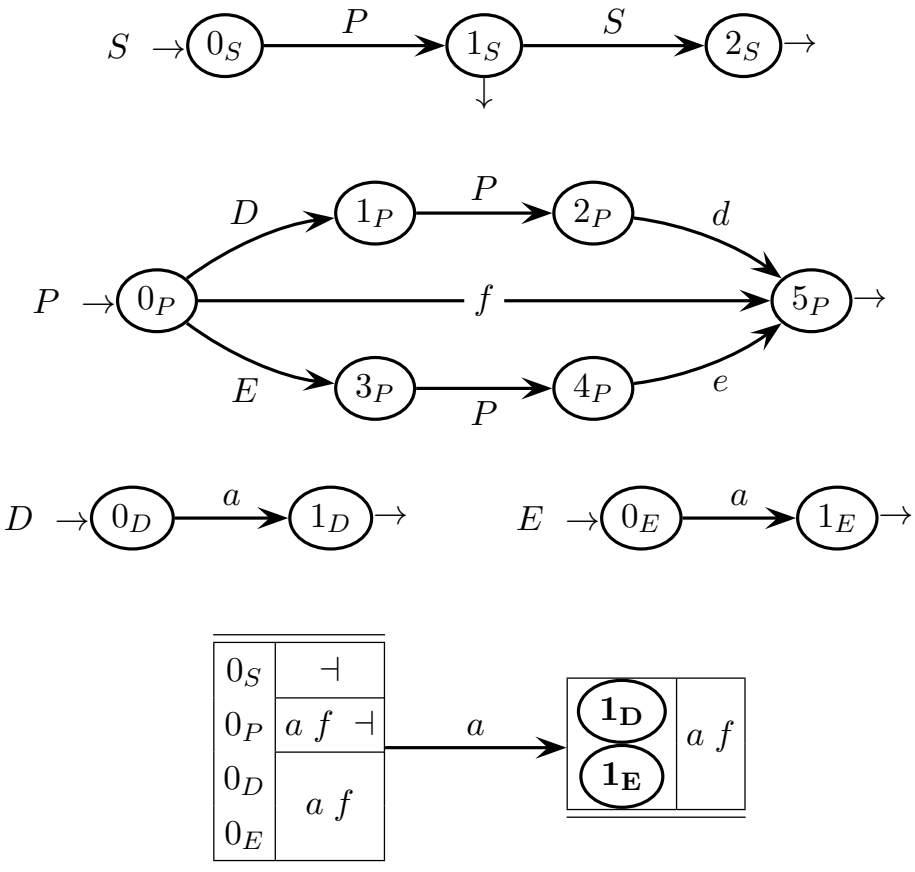


The underlying recognizer has become deterministic. Yet now there is transition $0_P \xrightarrow{a/\{\ d,\ e\ \}} \{\ 1_P,\ 3_P\ \}$ with multiple output. Thus the transformed transducer does not compute a function, from string to string, or equivalently it is still nondeterministic (on its output). In fact, it is well-known that in general the family of deterministic rational transducers is strictly contained in that of the nondeterministic ones, thus determinizing a rational transducer may be impossible.

Concerning an $ELR(1)$ parser for translation τ^{-1}, scheme G^{-1} is not in the postfix normal form. Moreover, when it is converted into postfix normal form, the $ELR(1)$ property gets lost, as shown next. Here are the postfix-normalized source and destination grammars G_{sn}^{-1} and G_{dn}^{-1}:

$$G_{sn}^{-1}\begin{cases} S \to P\ S \\ S \to P \\ P \to D\ P\ d \\ P \to E\ P\ e \\ P \to f \\ D \to a \\ E \to a \end{cases} \qquad G_{dn}^{-1}\begin{cases} S \to P\ S \\ S \to P \\ P \to D\ P\ a \\ P \to E\ P\ a \\ P \to c \\ D \to d \\ E \to e \end{cases}$$

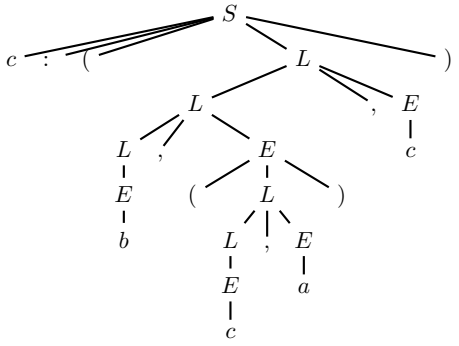
The figure below reports the relevant portions of the machines of net G_n^{-1} (with only the source symbols) and a pilot fragment (only two m-states) for the source grammar G_{sn}^{-1}, which immediately exhibits a reduce-reduce conflict:



Therefore, it seems unfeasible to compute translation τ^{-1} deterministically, at least with the known syntax-driven methods. Anyway, a generic (not derived from a syntax analyzer) deterministic pushdown transducer may still exist and be able to do so. The reader may wish to examine this last case by himself.

2. Consider a language of nested lists of arbitrary depth, which could be a fragment of the data description language *JSON*. A list (non-empty) is enclosed in round brackets, a list element is a sublist or a generic object schematized by terminal y, and the elements are separated by commas. Here are the syntax (axiom S) and a sample tree:

$$\begin{cases} S \rightarrow x : (L) \\ L \rightarrow L , E \\ L \rightarrow E \\ E \rightarrow (L) \\ E \rightarrow y \end{cases}$$



The prefix terminal x is an object to look for in the (rest of the) data structure, wherein it might occur zero, one or more times.

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

(a) Write an attribute grammar that computes whether object x occurs somewhere in the data structure. Use only the two attributes e (exists) and o (object) specified. Attribute e must be true in the tree root S if and only if object x occurs in the data structure. Sample cases:

data structure	*exists*	
$a : (b,\ a,\ c)$	yes	
$c : (b, (c,\ a),\ c)$	yes	(sample tree above)
$a : (b,\ b,\ c)$	no	

Make sure that the obtained grammar is one-sweep and explain why.

(b) Write an attribute grammar that computes the depth of the first, i.e., leftmost, occurrence of object x in the data structure, if any. Notice that the depth is the number of enclosing brackets, not the distance from the tree root. Sample cases:

data structure	*object depth (if any)*	
$a : (b,\ a,\ c)$	1	
$c : (b, (c,\ a),\ c)$	2	(sample tree above)
$a : (b,\ b,\ c)$	0	(inexistent object)

Assume that the attributes e and o of question (a) are computed as before, and that they are therefore available (do not write their semantic rules again). Use only the two additional attributes d (depth) and f (found depth) specified. Say if the obtained grammar is still one-sweep and shortly explain why yes or no.

(c) (optional) For the one-sweep attribute grammar of question (a), write the procedure L of the semantic analyzer. Specify the formal and actual parameters of L, as well as those of the procedure E invoked by L. With what actual parameters should procedure L be invoked in the axiomatic procedure S ?

attribute specifications – questions (a) and (b)

name	*type*	*domain*	*symbol*	*meaning*
e	left	boolean	S, L, E	*true* if object x occurs somewhere in the data (sub)structure rooted at the current node; else *false*
o	right	object	L, E	the object x to look for in the data (sub)structure
d	right	integer ≥ 1	L, E	the *depth* (≥ 1) of an element (sublist or object) in the data (sub)structure
f	left	integer ≥ 0	S, L, E	the *depth* (≥ 1) of the first (leftmost) instance of object x that is found in the data (sub)structure, if any; else 0

#	*syntax*	*semantics* – question (a)
1:	$S_0 \rightarrow x : (L_1)$	
2:	$L_0 \rightarrow L_1 , E_2$	
3:	$L_0 \rightarrow E_1$	
4:	$E_0 \rightarrow (L_1)$	
5:	$E_0 \rightarrow y$	

#	*syntax*	*semantics* – question (b)
1:	$S_0 \rightarrow x : (L_1)$	
2:	$L_0 \rightarrow L_1 , E_2$	
3:	$L_0 \rightarrow E_1$	
4:	$E_0 \rightarrow (L_1)$	
5:	$E_0 \rightarrow y$	

Solution

(a) Here is the attribute grammar:

#	*syntax*	*semantics*
1:	$S_0 \rightarrow x : (\ L_1\ )$	$o_1 := x$ $e_0 := e_1$
2:	$L_0 \rightarrow L_1\ ,\ E_2$	$o_1,\ o_2 := o_0$ $e_0 := e_1\ \vee\ e_2$
3:	$L_0 \rightarrow E_1$	$o_1 := o_0$ $e_0 := e_1$
4:	$E_0 \rightarrow (\ L_1\ )$	$o_1 := o_0$ $e_0 := e_1$
5:	$E_0 \rightarrow y$	**if** $o_0 = y$ **then** $e_0 :=$ *true* **else** $e_0 :=$ *false* – alternatively in the compact form $e_0 := (o_0 = y)$

The inherited attribute o propagates top-down throughout the tree the object x to look for. At the tree bottom, attribute o is compared to the leaf y and the synthesized attribute e is set to the comparison outcome (rule 5). Attribute e propagates bottom-up and its values for any two brother subtrees are logically conjuncted (rule 2). The final value of e is available at the tree root. Clearly, the root attribute e will be true if and only if (at least) one comparison succeeds, that is, if and only if the data structure contains one (or more) instance(s) of x.
First of all, the grammar does not have any circular dependence. Second, the grammar is one-sweep by construction. In fact, the right attribute o depends only on itself, and the left attribute e depends only on itself or (rule 5) on the right attribute o of the same node. Thus the one-sweep condition is satisfied.

(b) Here is the attribute grammar:

#	*syntax*	*semantics*
1:	$S_0 \rightarrow x : (\ L_1\ )$	$d_1 := 1$ $f_0 := f_1$
2:	$L_0 \rightarrow L_1\ ,\ E_2$	$d_1,\ d_2 := d_0$ **if** e_1 **then** $f_0 := f_1$ **else** $f_0 := f_2$ – alternatively a conditional assignment $f_0 := e_1\ ?\ f_1\ :\ f_2$
3:	$L_0 \rightarrow E_1$	$d_1 := d_0$ $f_0 := f_1$
4:	$E_0 \rightarrow (\ L_1\ )$	$d_1 := d_0 + 1$ $f_0 := f_1$
5:	$E_0 \rightarrow y$	**if** e_0 **then** $f_0 := d_0$ **else** $f_0 := 0$ – alternatively a conditional assignment $f_0 := e_0\ ?\ d_0\ :\ 0$

The inherited attribute d computes top-down the depth. At the tree bottom, the synthesized attribute f is set to the current depth d or to 0, depending on the value of attribute e (rule 5). Attribute f propagates bottom-up and, for any two brother subtrees, its left value is preferred, if any (rule 2). The final value of f is available at the tree root. Clearly, the root attribute f will have the depth value of the leftmost occurrence of object x in the data structure, if any, else 0.

In the rule 2, the preference of assigning f_1 instead of f_2 to f_0, unless the subtree of f_1 does not contain object x, encodes the requirement of returning the depth of the first (leftmost) occurrence of object x in the data structure.

This grammar does not have any circular dependence either and is still one-sweep, again by construction. Attributes e and o are unchanged from case (a). Notice that apparently the grammar depends on e and not on o, yet o is used to compute e, thus the grammar dependence on o is implicit. Then, the right attribute d depends only on itself, and the left attribute f depends only on itself, or (rule 2) on the left attribute e of a child node, or (rule 5) on the left and right attributes e and d of the same node. Thus the one-sweep condition is satisfied.

(c) Formal parameters of the semantic procedures L and E (the same):

L (**in** o; **out** e)

E (**in** o; **out** e)

Pseudo-code of the semantic procedure L, fully formalized:

```
procedure L ( in o; out e )
var o1, o2 : object                          // right attributes
var e1, e2 : boolean                          // left attributes
switch rule do                                 // tree node type
    case L → L , E do
        o1 := o                                    // inheritance
        o2 := o                                    // inheritance
        call L ( o1, e1 )                  // subtree computation
        call E ( o2, e2 )                  // subtree computation
        e := e1 ∨ e2                                 // synthesis
    case L → E do
        o1 := o                                    // inheritance
        call E ( o1, e1 )                  // subtree computation
        e := e1                                      // synthesis
    otherwise do error                       // unknown node type
```

or with some quite simple programming optimization, to save code:

```
procedure L ( in o; out e )
var e1, e2 : boolean                         // left attributes
switch rule do                                 // tree node type
    case L → L , E do
        call L ( o, e1 )          // inheritance / computation
        call E ( o, e2 )          // inheritance / computation
        e := e1 ∨ e2                                 // synthesis
    case L → E do  call E ( o, e )     // inherit / comp /
     synth
    otherwise do error                       // unknown node type
```

Finally, the axiomatic semantic procedure S (**out** e) must invoke procedure L with these actual parameters:

$o_1 := x$

call L (o_1, e_1)

$e := e_1$

where o_1 and e_1 are local variables, or more simply:

call L (x, e)

where e is the output parameter that will hold the final result.