FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Tue 4 SEPTEMBER 2018 - Part Theory

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

LAST + FIRST NAME:

(capital letters please)

MATRICOLA: SIGNATURE:

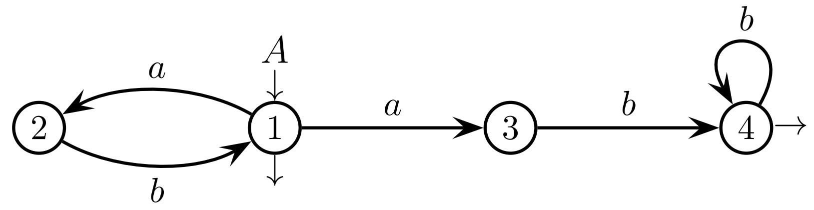
(or PERSON CODE)

INSTRUCTIONS - READ CAREFULLY:

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

1 Regular Expressions and Finite Automata 20%

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



Answer the following questions:

(a) Find the shortest strings of language $L(A)$ from length 0 (included) to length 2 (included) and write their computations.

(b) Through the Berry-Sethi method (BS), find a deterministic automaton A' equivalent to automaton A and minimize it if necessary. Verify the correctness of (the minimal version of) A' by means of the strings found at point (a).

(c) Through the node elimination method (BMC), find a regular expression R that generates language $L(A)$. Verify the correctness of R by means of the strings found at point (a).

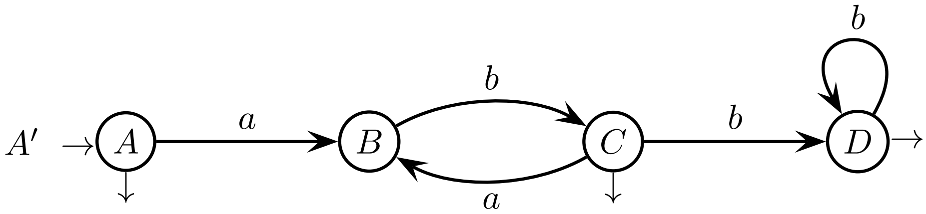
(d) Design an automaton A'' that accepts language $\overline{L(A)}$, i.e., the complement of language $L(A)$. Counterverify the correctness of A'' by means of the strings found at point (a) (all of them have to be rejected).

(e) (optional) It is well known that in some cases a nondeterministic automaton may have fewer states than the equivalent deterministic minimal one. Can you find a (nondeterministic) automaton that accepts language $L(A)$ and has fewer states than the minimal automaton A'? Explain your answer, whatever it is.

Solution

(a) There two strings of length up to 2: ε (computation: $\to 1 \to$) and $a\,b$ (computations: $\to 1 \to 2 \to 1 \to$, $\to 1 \to 3 \to 4 \to$).

(b) Rather easy even intuitively:

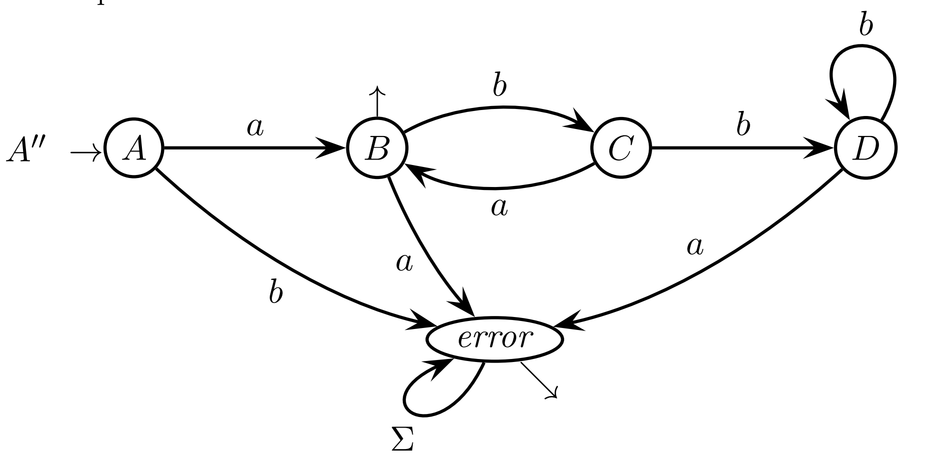


Automaton A' is minimal.

(c) Rather easy even intuitively:

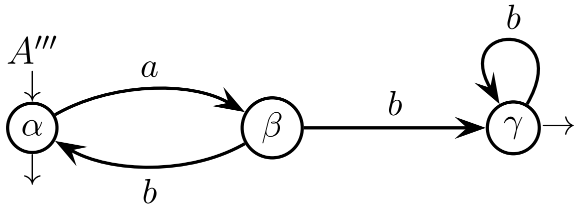
$$R = (\,a\,b\,)^{+}\ b^{*} \ \mid\ \varepsilon = \left[\ (\,a\,b\,)^{+}\ b^{*}\ \right]$$

(d) We have to start from a deterministic automaton, for instance A'. Completion and complement of A':



One may verify that the deterministic automaton A'' is minimal.

(e) The nondeterministic automaton A''' below is equivalent to automaton A and it has only three states, one state less than automaton A' (which is minimal!) has:



Through eliminating node β, one gets $R'' = \varepsilon \ \mid\ (\,a\,b\,)^{+}\ b^{*} = R$ and proves the equivalence. The five strings are accepted, and none else of length from 0 to 2.

Caveat: the nondeterministic automaton A''' is apparently obtained by merging the nodes A and C of the deterministic automaton A', yet notice that these two nodes have different outgoing arcs, thus it makes no sense to say that they are equivalent in the classical (Nerode) sense; here this happens just by chance.

2 Free Grammars and Pushdown Automata 20%

1. Consider the following language L_1 over the two-letter alphabet $\Sigma = \{ a, \ b \}$:

$$L_1 = \{ \ a^n \, b^+ \, a^m \mid \quad m \geq 0 \text{ and } \exists \, k \quad k \geq 0 \ \wedge \ n = m + 2k \ \}$$

For instance it holds:

$$a^3 \, b^2 \, a \in L_1 \qquad\qquad a^3 \, b^2 \, a^2 \notin L_1 \qquad\qquad a \, b^2 \, a^3 \notin L_1$$

Answer the following questions:

(a) Write a *BNF* grammar G_1, *not ambiguous*, that generates language L_1.

(b) Check the correctness of grammar G_1 by drawing the syntax tree of the valid sample string:

$$a^3 \, b^2 \, a$$

(c) Now consider language L_2, mirror image of language L_1, defined as follows:

$$L_2 = \{ \ a^m \, b^+ \, a^n \mid \quad m \geq 0 \text{ and } \exists \, k \quad k \geq 0 \ \wedge \ n = m + 2k \ \}$$

Write a *BNF* grammar G, *not ambiguous*, that generates language $L = L_1 \cup L_2$.

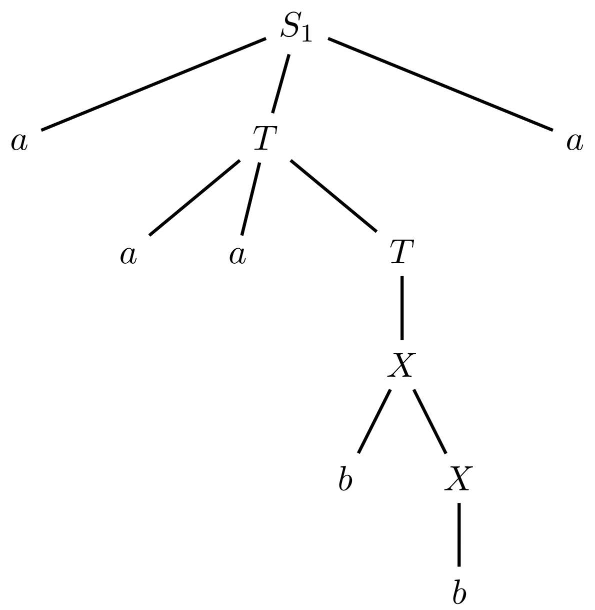
(d) (optional) Argue that grammar G_1 is not ambiguous.

Solution

(a) Here is a working *BNF* grammar G_1 (axiom S_1):

$$G_1 \begin{cases} S_1 \rightarrow a\, S_1\, a \mid T \\ T \rightarrow a\, a\, T \mid X \\ X \rightarrow b\, X \mid b \end{cases}$$

(b) Here is the tree of the valid string $a^3\, b^2\, a$:



(c) Languages L_1 and L_2 are not disjoint. All the strings with $n = m$, that is, with $k = 0$, are shared between L_1 and L_2. However, consider the language $L_3 \subset L_2$ (strict inclusion) defined below:

$$L_3 = \{\, a^m\, b^+\, a^n \mid \quad m \geq 0 \text{ and } \exists\, k \quad k \geq 1 \,\wedge\, n = m + 2k \,\}$$

One can easily see that language L is the disjoint union of language L_1 and L_3. In fact, language L_3 is a subset of language L_2 obtained by stripping off all and only the strings with $n = m$, i.e., $k = 0$.

A grammar G_3 that generates language L_3 is the following (axiom S_3)

$$G_3 \begin{cases} S_3 \rightarrow a\, S_3\, a \mid U \\ U \rightarrow U\, a\, a \mid Y\, a\, a \\ Y \rightarrow b\, Y \mid b \end{cases}$$

Hence a grammar G (axiom S) that generates language L can be obtained in the customary way as the union of grammars G_1 and G_3, as follows: $S \rightarrow S_1 \mid S_3$, ..., etc.

(d) For every string $x \in L_1$, its derivation is unique, due to the ordering S, T and X imposed on the nonterminals of grammar G_1 by the *produce* relation. Therefore, for every string $x \in L_1$ the following relation holds:

$$x = a^n \, a^{2k} \, b^h \, a^n \qquad h > 0 \qquad n, \, k \geq 0$$

and string x is necessarily derived as follows:

$$S_1 \stackrel{n}{\Rightarrow} a^n \, S_1 \, a^n \Rightarrow a^n \; T \; a^n \stackrel{k}{\Rightarrow} a^n \, a^{2k} \, T \, a^n \Rightarrow a^n \, a^{2k} \; X \; a^n \stackrel{h}{\Rightarrow} a^n \, a^{2k} \, b^h \, a^n$$

in a unique way.

2. Consider a simplified programming language, which consists of a list of assignments that feature the following syntax:

   - A program is a possibly empty list of assignment statements.
   - The assignment is denoted in a kind of *postfix* form, as follows:

     ```
     elem expr :=
     ```

     where ":=" (colon equal) is the postfix assignment operator and expr is the arithmetic expression to be assigned to element elem, as explained below.
   - The expression expr is of the type sum-of-products (2-levels). It uses the arithmetic operators "+" and "$*$" (addition and multiplication), which are binary, i.e., with two operands. The expression is denoted in *prefix* form, thus without parentheses, as follows (on the right you can see the infix notation):

     ```
     prefix form:                   infix form as a comment:
     + expr expr                    expr + expr
     + + expr expr expr             expr + expr + expr
     + expr * expr expr             expr + expr * expr
     ```

   - The elements elem that can appear in an assignment (where it makes sense) and in an expression are the following:
     - a numerical constant, schematized by terminal c
     - a named variable, schematized by terminal v
     - an array element (one- or more-dimensioned), with the following syntax:

       ```
       vect ( list_of_expressions )
       ```

       where the expressions in the list (non-empty) are separated by "," (comma), and the array name is schematized by terminal vect

   Here is a short sample program (on the right there is the usual infix representation):

   ```
   program:                             comment (infix form):
   v c :=                               v := c
   v + v c :=                           v := v + c
   v + + c v v :=                       v := c + v + v
   v + * v v v :=                       v := v * v + v
   v vect (* v v) :=                    v := vect (v * v)
   vect (c, + v v) c :=                 vect (c, v + v) := c
   ```

   Notice that the assignment operator ":=" also plays the role of statement terminator, to separate consecutive statements.

   Write a grammar, possibly of *EBNF* type, not ambiguous, that generates the language described above. To test your solution, please draw the syntax tree for the last two assignments in the sample program.

Solution

1. Here is a viable grammar (axiom $\langle \mathsf{PROG} \rangle$), *EBNF* and not ambiguous:

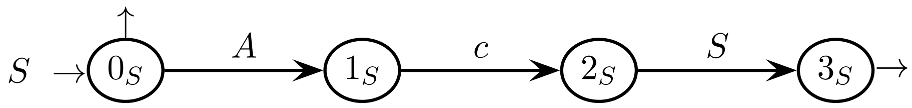
$$
\left\{
\begin{array}{rcl}
\langle \mathsf{PROG} \rangle & \to & \langle \mathsf{ASGN} \rangle^* \\
\hline
\langle \mathsf{ASGN} \rangle & \to & \langle \mathsf{ELEM} \rangle \ \langle \mathsf{EXPR} \rangle \ `\!:=\!' \\
\hline
\langle \mathsf{ELEM} \rangle & \to & \langle \mathsf{VAR} \rangle \ | \ \langle \mathsf{VECT} \rangle \\
\langle \mathsf{EXPR} \rangle & \to & `+' \ \langle \mathsf{TERM} \rangle \ \langle \mathsf{TERM} \rangle \ | \ \langle \mathsf{TERM} \rangle \\
\hline
\langle \mathsf{VAR} \rangle & \to & \mathsf{v} \\
\langle \mathsf{VECT} \rangle & \to & \mathsf{vect} \ `(' \ \langle \mathsf{LIST} \rangle \ `)' \\
\langle \mathsf{TERM} \rangle & \to & `*' \ \langle \mathsf{FACT} \rangle \ \langle \mathsf{FACT} \rangle \ | \ \langle \mathsf{FACT} \rangle \\
\hline
\langle \mathsf{LIST} \rangle & \to & \langle \mathsf{EXPR} \rangle \ (`,' \ \langle \mathsf{EXPR} \rangle)^* \\
\langle \mathsf{FACT} \rangle & \to & \langle \mathsf{CONST} \rangle \ | \ \langle \mathsf{ELEM} \rangle \\
\hline
\langle \mathsf{CONST} \rangle & \to & \mathsf{c}
\end{array}
\right.
$$

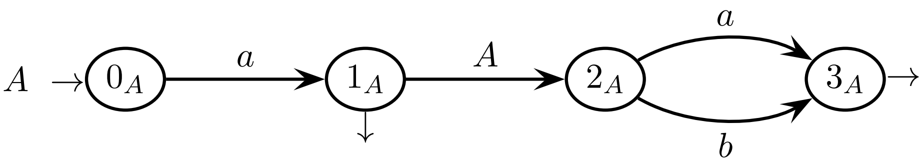
   Notice that, since the expression is prefix with binary operators, the expression structure completely defines how to associate, and this is why nothing is said about associativity in the language specifications. The expression is of the type sum-of-products (2-levels only): after passing to the product level, it is not possible to resume the addition level. The grammar is reasonably *EBNF* and it is not ambiguous.

2. TBD

3 Syntax Analysis and Parsing Methodologies 20%

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





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

(a) Draw a portion of the pilot of grammar G, sufficient to prove that grammar G is not of type $ELR(1)$. Highlight any conflict that is present in that pilot portion.

(b) Determine whether grammar G is ambiguous or not, and provide a suitable explanation.

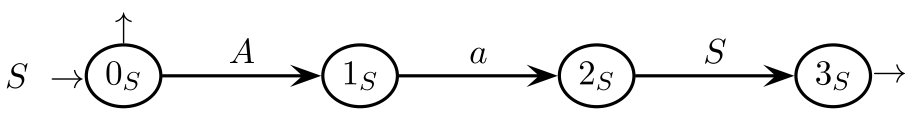
(c) Draw all the guide sets on the net of grammar G (shift arcs, call arcs and exit arrows), determine whether grammar G is of type $ELL(1)$ and justify your answer.

(d) Through the Earley algorithm for grammar G, analyze the valid string below:

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

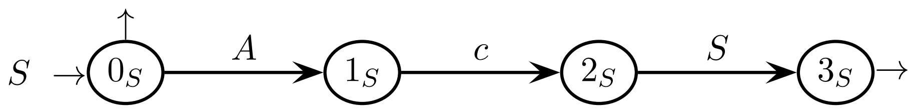
Draw the syntax tree and show which paths the net machines use to recognize.

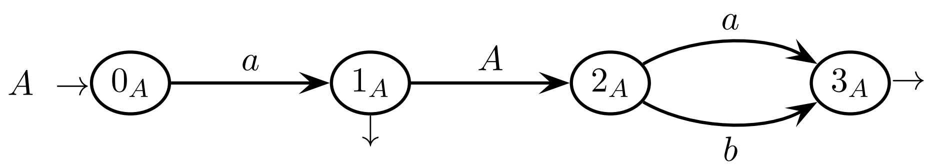
(e) (optional) Consider a variant of the above grammar G, where machine M_S is modified as shown below, while machine M_A is unchanged:



Is this new grammar $ELR(1)$? Is it nondeterministic? Is it ambiguous? Explain your answer (hint: you may refer, as a significant example, to string $a\,a\,a\,a$).

here draw the pilot of grammar G – question (a) here draw the call arcs and write all the guide sets of grammar G – question (c)





Earley vector of string $a\,a\,b\,c$ (to be filled in) – question (d)

(the number of rows is not significant)

0	a	1	a	2	b	3	c	4

space for answering questions (b) and (e)

Solution

(a) Here is the requested pilot portion:

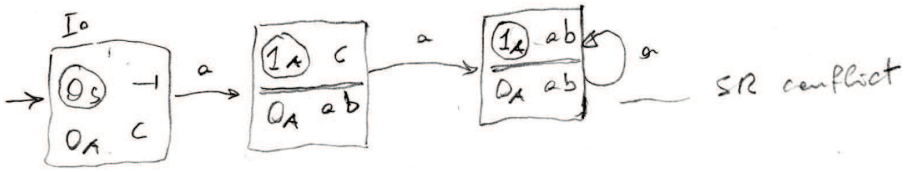


Figure 1: pilot portion

The pilot has a shift-reduce conflict, therefore the grammar is not *ELR*(1).

(b) The grammar is not ambiguous, because each substring of letters a and b followed by a letter c has an identified center, therefore the syntax tree is unique.

(c) The guide sets are not disjoint on the two arcs that exit node 1_A, hence the grammar is not of type *ELL*(1).

(d) Here is the Earley vector of the (valid) string $a\,a\,b\,c$: TBD

0	a	1	a	2	b	3	c	4
$\mathbf{0_S}$ 0		$\mathbf{1_A}$ 0		$\mathbf{1_A}$ 1		$\mathbf{3_A}$ 0		2_S 0
0_A 0		0_A 1		0_A 2		1_S 0		$\mathbf{0_S}$ 4
		1_S 0		2_A 0				$\mathbf{3_S}$ 0
								0_A 4

The acceptance condition is satisfied in the last Earley state.

(e) The string $a\,a\,a\,a$ is ambiguous (below two syntax trees for it), therefore the new grammar is also ambiguous, hence nondeterministic.

Figure 2: syntax tree for *aabc*

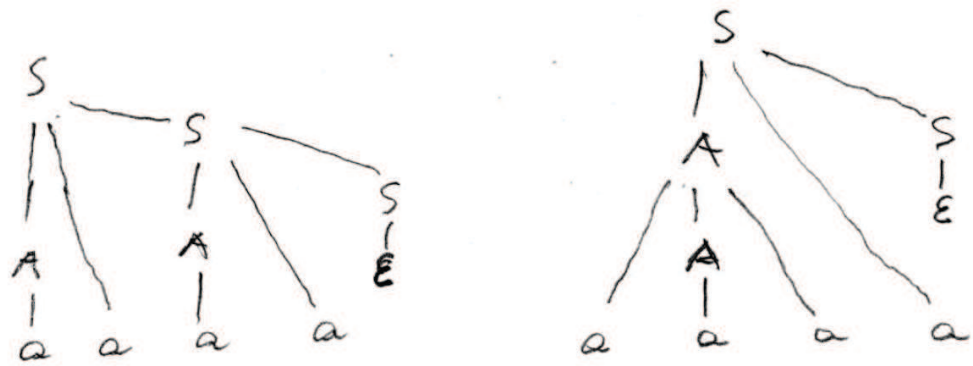
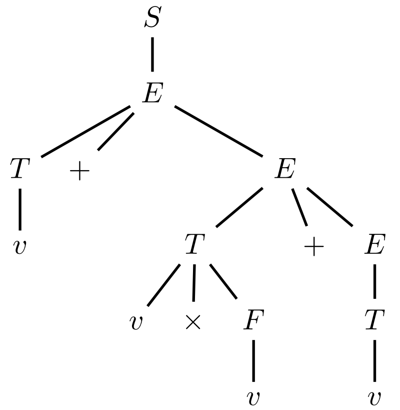


Figure 3: syntax trees for *aaaa*

4 Language Translation and Semantic Analysis 20%

1. Consider the following *BNF* non-ambiguous source grammar G (axiom S):

$$G \begin{cases} S \rightarrow E \\ E \rightarrow T + E \\ E \rightarrow T \\ T \rightarrow v \times F \\ T \rightarrow v \\ F \rightarrow v \times F \\ F \rightarrow v \end{cases}$$



Grammar G generates two-level arithmetic expressions without parentheses, e.g., v, $v+v$, $v \times v$, $v+v \times v+v$, with variables and numbers schematized by terminal v. See the sample tree above on the right.

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

(a) Consider a translation function τ_1 that transforms addition and multiplication into summatory $\sum$ and productory $\prod$, respectively:

$$\tau_1\,(v+v \times v+v) = \textstyle\sum v,\, \prod v\, v,\, v \qquad \text{and} \qquad \tau_1\,(v\,) = \sum v$$

where a comma separates the addends of summatory $\sum$.
Write a syntactic scheme (or grammar) G_1 that models translation τ_1. Draw the translation tree of the sample expression $v+v \times v+v$.

(b) Assume that the element v is the same object (variable or number) wherever it occurs in an expression. Under such an assumption, one can define a translation τ_2 that optimizes an expression in this way:

$$\tau_2\,(v+v+v \times v+v \times v \times v+v) = v+v+v\,\hat{}\,e+v\,\hat{}\,e+v$$

Translation τ_2 transforms a product of factors v into a power with an exponent e. Later a semantic translation, not to be considered here, will assign appropriate values to the exponents e, in the example $e=2$ and $e=3$ from left to right.

Write a syntactic scheme (or grammar) G_2 that models translation τ_2. Draw the translation tree of the sample expression above.

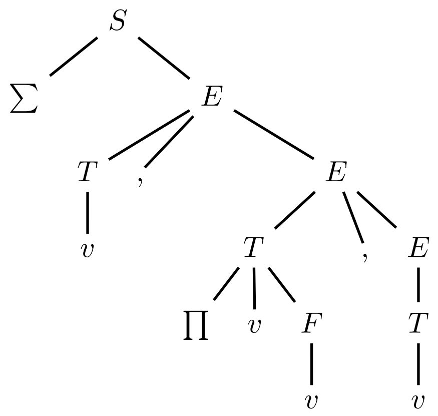
(c) Are the schemes (or grammars) G_1 and G_2 deterministic ? Explain your answers.

(d) (optional) Resume the assumption of question (b), and now define a translation τ_3 that transforms a sum of terms v into one term with a multiplicative coefficient c, e.g., $\tau_3\,(v+v+v \times v+v) = c\,v+v \times v+v$ (later it will be assigned $c=2$). Can you find a syntactic scheme (or grammar) G_3 that models translation τ_3 ? Should you change the source grammar ? Explain your answer.

Solution

(a) Almost a translitaration:

$$G_1^d \begin{cases} S \to \sum E \\ E \to T \, , \, E \\ E \to T \\ T \to \prod v \, F \\ T \to v \\ F \to v \, F \\ F \to v \end{cases}$$



The translation tree of the sample expression is aside.

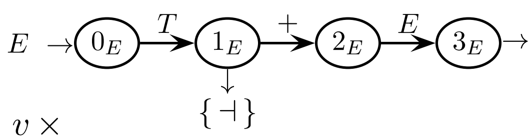
(b) Almost a translitaration:

$$G_2^d \begin{cases} S \to E \\ E \to T + E \\ E \to T \\ T \to F \\ T \to v \\ F \to F \\ F \to v \,\hat{}\, e \end{cases}$$

Tree TBD.

(c) The source grammar G is of type $LL(2)$ for the alternative rules of T and F, and of type $ELL(1)$ for machine M_E. See the disjoint guide sets of the alternative rules and of the machine M_E aside, with $k = 2$ (for M_E it suffices $k = 1$):

$$G \begin{cases} S \to E & \\ E \to T + E & \\ E \to T & \\ T \to v \times F & v \times \\ T \to v & v+, \ v \dashv \\ F \to v \times F & v \times \\ F \to v & v+, \ v \dashv \end{cases}$$



Thus both syntactic schemes G_1 and G_2 are deterministic.

(d) Yes, a scheme G_3 can be designed, but the source grammar G has to be changed to separate the cases when it generates either a sum of two or more addends v, which has to be translated into one addend with a coefficient c, or just one addend v, which is left unchanged and does not need any coefficient. This is a simple finite-state translation, yet it requires to patch in some way the rules $E \to T + E$ and $E \to T$, possibly with more nonterminals.

2. Consider the following (simplified) portion of a grammar of a programming language, with nested conditional statements if ... then ... else ... end, possibly without else branch (axiom ST):

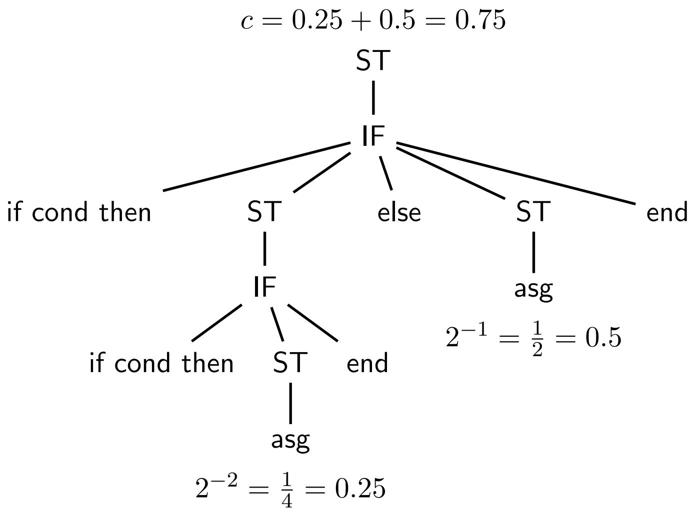
$$\begin{cases} 1: \langle \mathsf{ST} \rangle \to \langle \mathsf{IF} \rangle \\ 2: \langle \mathsf{ST} \rangle \to \mathsf{asg} \\ 3: \langle \mathsf{IF} \rangle \to \mathsf{if\ cond\ then}\ \langle \mathsf{ST} \rangle\ \mathsf{else}\ \langle \mathsf{ST} \rangle\ \mathsf{end} \\ 4: \langle \mathsf{IF} \rangle \to \mathsf{if\ cond\ then}\ \langle \mathsf{ST} \rangle\ \mathsf{end} \end{cases}$$

We intend to compute a numerical parameter c that indicates the completeness degree of the nested instructions. Parameter c tells whether all the branches of the nested conditional instructions are complete, or whether some branch else is missing.

To do so, decreasing weights are assigned to the nested branches. Therefore, each instruction asg is assigned a value equal to 2^{-n}, where $n \geq 0$ is the number of enclosing conditional instructions.

For instance, in the sample program and tree below, the first asg (3-rd line) has a value $2^{-2} = \frac{1}{4} = 0.25$, while the second asg (6-th line) has a value $2^{-1} = \frac{1}{2} = 0.5$:

```
if cond then
|  if cond then
|  |  asg
|  end
else
|  asg
end
```



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

(a) Compute the correct value of parameter c for the entire conditional statement by using only one attribute, also named c, of a suitable type. The required result must be computed as the value of c in the root node of the abstract syntax tree. In the above example, the root node ST will have an attribute $c = 2^{-2} + 2^{-1} = 0.75$.

(b) Decorate the example tree on the next page and write the values of attribute c in all the relevant nodes.

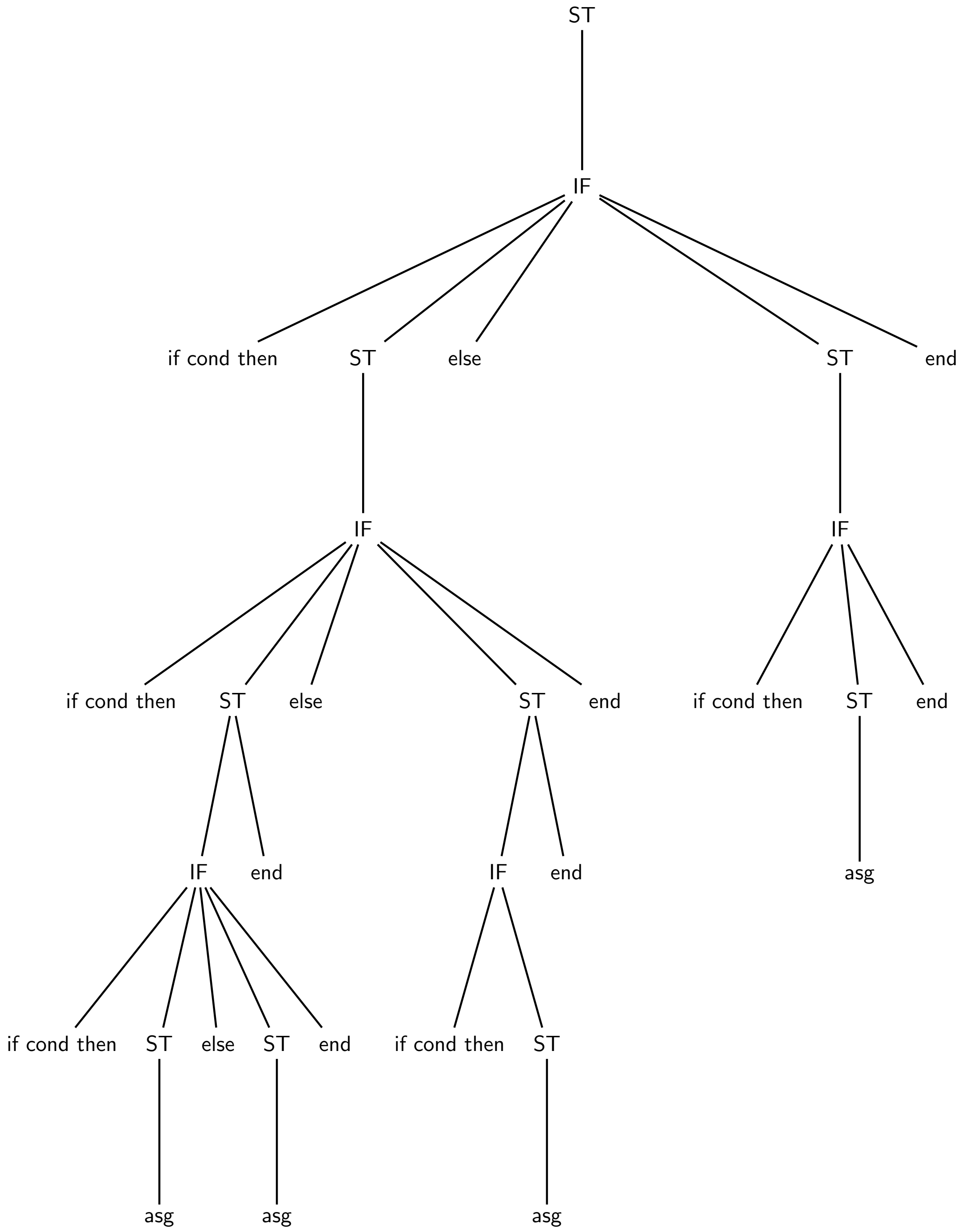
(c) Say if the defined attribute grammar is of type L. Provide an explanation for your answer and avoid generic or tautological sentences.

attribute specification to be completed and semantic functions – question (a)

name	*type*	*domain*	*symbol*
c		real	

#	*syntax*	*semantics*
1:	$\mathsf{ST}_0 \rightarrow \mathsf{IF}_1$	
2:	$\mathsf{ST}_0 \rightarrow \mathsf{asg}$	
3:	$\mathsf{IF}_0 \rightarrow \mathsf{if\ cond\ then\ ST}_1\ \mathsf{else\ ST}_2\ \mathsf{end}$	
4:	$\mathsf{IF}_0 \rightarrow \mathsf{if\ cond\ then\ ST}_1\ \mathsf{end}$	

syntax tree to be decorated – question (b)



Solution

(a) Rules of the attribute grammar:

name	*type*	*domain*	*symbol*
c	left	real	ST, IF

#	*syntax*			*semantics*
1:	ST_0	$\rightarrow$	IF_1	$c_0 := c_1$
2:	ST_0	$\rightarrow$	asg	$c_0 := 1$
3:	IF_0	$\rightarrow$	if...then ST_1 else ST_2 end	$c_0 := \frac{1}{2}c_1 + \frac{1}{2}c_2$
4:	IF_0	$\rightarrow$	if...then ST_1 end	$c_0 := \frac{1}{2}c_1$

(b) Decorated tree: TBD

(c) The grammar is of type L because it is purely synthesized, as its unique attribute c is of type left.