FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Fri 5 JULY 2019 - Part Theory

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

LAST + FIRST NAME:

(capital letters please)

MATRICOLA: SIGNATURE:

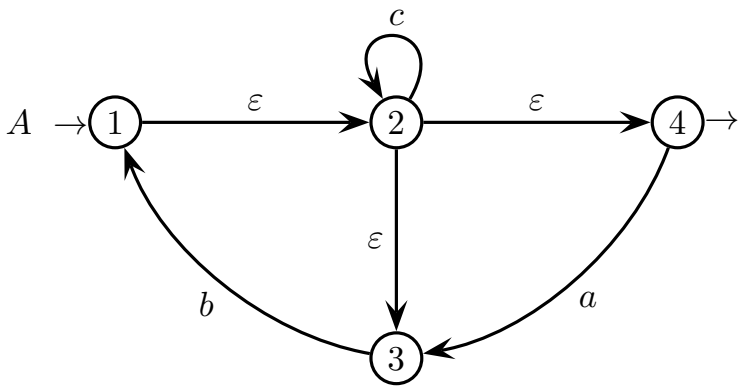
(or PERSON CODE)

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 automaton A below, with spontaneous transitions, over the three-letter alphabet $\Sigma = \{\ a,\ b,\ c\ \}$:



Answer the following questions:

(a) List all the strings of length from 0 (included) to 2 (included) that are accepted by automaton A, and separately list also all those (still of length from 0 to 2) that are rejected by A. Are the accepted strings ambiguous ? Justify your answers.

(b) Cut the spontaneous transitions of automaton A and obtain an equivalent automaton A', possibly still nondeterministic. Check whether automaton A' is minimal, and if necessary minimize it.

(c) Test the correctness of the minimal automaton A': consider all the strings (accepted and rejected) listed at point (a) that do not contain letter c, and for each of such strings write either the accepting or the rejecting computation of A'.

(d) By using the node elimination method, find a regular expression R that generates language $L(A)$. You can start from automaton A or A', as best suits you. Test the correctness of R by means of the accepted strings (again only those that do not contain letter c) found at point (a).

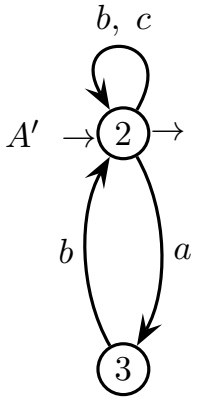
(e) Find the local sets $Ini\big(L(A)\big)$, $Fin\big(L(A)\big)$ and $Dig\big(L(A)\big)$, and say if language $L(A)$ is local. Justify your answer.

(f) (optional) Is the complement language $\overline{L(A)}$ local ? Justify your answer.

Solution

(a) Accepted strings: ε, b, c, $a\,b$, $b\,b$, $b\,c$, $c\,b$, $c\,c$; their paths go to the final state 4. Rejected strings: a, $a\,a$, $a\,c$, $b\,a$, $c\,a$; their paths do not go to the final state 4. None of the accepted strings is ambiguous, as each of them has only one accepting path (in fact the automaton A itself is not ambiguous - all the valid strings are accepted by only one path).

(b) Orderly cut the spontaneous transitions $2 \to 4$, $3 \to 1$ and $2 \to 3$:



Automaton A' is deterministic and clearly minimal, as state 2 is final and state 3 is not. It is easily seen to be correct, by examining the string paths as before.

(c) Accepted strings without letter c: ε, b, $a\,b$, $b\,b$; all of them have an accepting path on A'. Rejected strings without letter c: a, $a\,a$, $b\,a$; the corresponding paths on A' are rejecting, since they go to the non-final state 2.

(d) Regular expression R, obtained from automaton A' by node elimination (normalize, then eliminate node 3, then eliminate node 2):

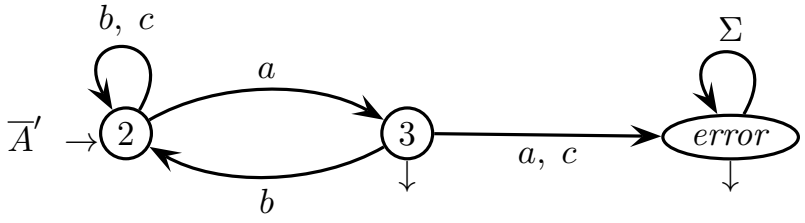
$$R = \big(\, a\,b \;\mid\; b \;\mid\; c \,\big)^* = \big(\, [\, a\,]\, b \;\mid\; c \,\big)^*$$

Expression R is easily seen to be correct, by examining the accepted strings.

(e) The local sets are as follows. All the digrams Σ^2 are admissible except the digrams $a\,a$ and $a\,c$, and all the initial and final letters Σ are admissible except the final letter a. Furthermore, the empty string ε is valid.

Language $L\,(A)$ is clearly local. In fact, automaton A', equivalent to automaton A, is local: each input letter enters exactly one state.

(f) No, the complement language is not local. Automaton A', equivalent to A, can be completed by an error state, and since it is deterministic, it can be soon complemented as follows:



This complement automaton $\overline{A}'$ is already minimal, since states 3 and *error* are distinguishable, yet it is not local as both arcs a and c enter two states; therefore the complement language is not local.

Also very simply: letter b is both initial and final for the complement automaton $\overline{A}'$ (see states 1 and *error*), thus if the complement language were local, it should contain string b, and consequently the direct language should not do; yet string b is valid for the direct language (see point (a)), therefore the complement language is not local. The same reasoning applies to letter c.

In a more general way: as seen before, for a string to belong to the direct language, a necessary and sufficient condition is that the string *does not contain* the digrams $\{\, a\,a,\ a\,c \,\}$ (none of them) **and** *does not end* with the letter a. Thus, for a string to belong to the complement language, the negation of such a condition is necessary and sufficient, i.e., that the string *contains* the digrams $\{\, a\,a,\ a\,c \,\}$ (at least one of them) **or** *ends* with the letter a (remember De Morgan: $\overline{\alpha \,\wedge\, \beta} = \overline{\alpha} \vee \overline{\beta}$). A regular expression $\overline{R}$ modeled according to the negated condition is:

$$\overline{R} = \Sigma^* \,(\, a\,a \;\mid\; a\,c \,)\, \Sigma^* \;\mid\; \Sigma^*\, a$$

The left member of the union in $\overline{R}$ admits any final letter provided there is the digram $a\,a$ or $a\,c$ (or both), whereas the right one allows such digrams to be absent provided the final letter is a. Yet, with a local formulation of the language, it is impossible to force the final letter to be exclusively a when both such digrams are absent (equivalently, how can the final letters b and c be aware that in the string, possibly far from the end, there is a digram $a\,a$ or $a\,c$?); therefore the complement language is not local.

2 Free Grammars and Pushdown Automata 20%

1. Consider the grammar G below, over a three-letter terminal alphabet $\{ a, b, c \}$ and a two-letter nonterminal alphabet $\{ X, S \}$ (axiom S):

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

   Answer the following questions:

   (a) Verify that grammar G is ambiguous, by drawing at least two syntax trees for the valid string below:

$$a\ a\ b\ c$$

   (b) For language $L(G)$, write a non-ambiguous grammar G' of the least general type.

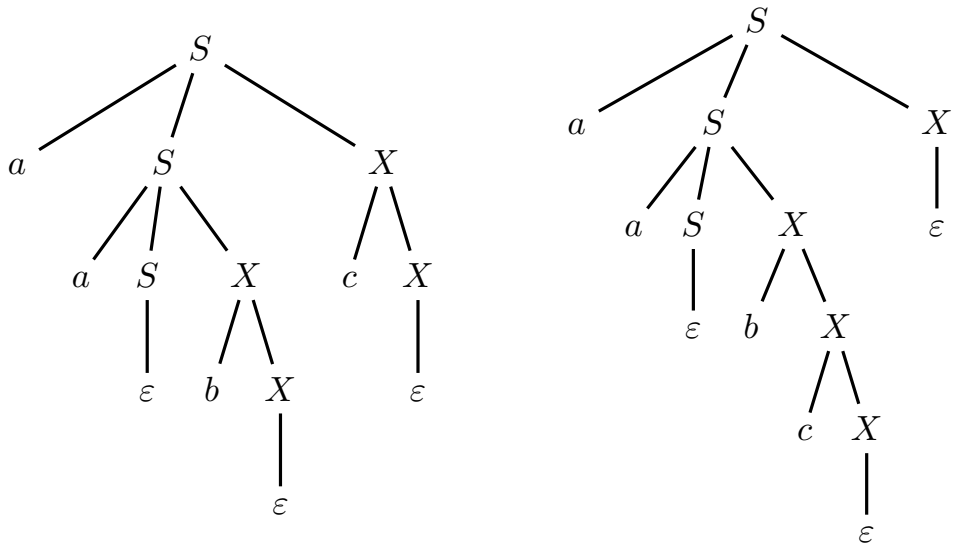
   (c) (optional) Consider now the regular expression R below:

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

   Is the intersection language $L(G) \cap L(R)$ regular ? Justify your answer.

Solution

(a) Here are two syntax trees for the valid string $a\,a\,b\,c$, where the letter c is ambiguously appended at different tree levels:



String $a\,a\,b\,c$ has only the two trees above, but longer strings have trees more and their ambiguity degree grows with their length.

(b) Language $L\,(G)$ is regular, defined by expression $a^+\ (\,b\ \mid\ c\,)^*\ \mid\ \varepsilon$. In fact, by recursion it holds $S \overset{n}{\Rightarrow} a^n\ S\ X^n$, for $n \geq 1$, or $S \Rightarrow \varepsilon$, and by the Arden identity it holds $L\,(X) = (\,b\ \mid\ c\,)^*$. Then by substitution we obtain $L\,(S) = a^n\ \big(\,(\,b\ \mid\ c\,)^*\,\big)^n\ \mid\ \varepsilon$, for $n \geq 1$. This is equivalent to $a^+\ (\,b\ \mid\ c\,)^*\ \mid\ \varepsilon$, i.e., a regular expression, since the component $(\,b\ \mid\ c\,)^*$ generates any string (including ε) over the letters b and c, and repeating such strings for $n \geq 1$ times still yields any string over the same letters. Since the language is infinite, a regular expression that contains at least one star (or a cross) is indispensable to generate it.

Therefore, being regular, the language admits a right unilinear grammar, like for instance the following one G' (axiom S):

$$G' \begin{cases} S \ \to\ a\,T \ \mid\ \varepsilon \\ T \ \to\ a\,T \ \mid\ V \\ V \ \to\ b\,V \ \mid\ c\,V \ \mid\ \varepsilon \end{cases}$$

which is not ambiguous (after removing the copy rule $T \to V$, the associated finite-state automaton is deterministic). You can test grammar G' by drawing the syntax tree (now only one) of the sample string $a\,a\,b\,c$.

(c) Language $L\,(G)\ \cap\ L\,(R)$ is the intersection of two regular languages, therefore it is regular. It is generated by the regular expression $a^+\ (\,c\ b\,)^*\ \mid\ \varepsilon$, since expression $R = a^*\ (\,c\,b\,)^*$ just additionally imposes to the expression $a^+\ (\,b\ \mid\ c\,)^*$ for G that the letters c and b alternate to each other.

2. Consider a variant of the language of the arithmetic expressions denoted in the *prefix form*, described as follows:

   - An arithmetic expression can contain these operations and symbols:
     - n-ary addition, i.e., with $n \geq 2$ operands, denoted as $+$
     - binary multiplication, denoted as $*$
     - and also parentheses (and)
   - Variables and numbers are schematized by terminal a, without expanding.
   - Both addition and multiplication are denoted in the prefix form, e.g., $+\,a\,a$ and $*\,a\,a$ for the binary case.
   - Both addition and multiplication are associative operations, as usual.
   - Binary addition, i.e., case $n = 2$, and multiplication never need parentheses, as usual with the prefix form.
   - Ternary and in general n-ary addition, i.e., case $n \geq 3$, always needs parentheses, to identify the actual number n of operands, e.g., $(\,+\,a\,a\,a\,)$ for case $n = 3$.

   Examples (ten valid expressions):

```
 1.  a

 2.  + a a             // addition is binary, not parenthesized

 3.  * a a

 4.  + a + a a         // both additions are binary, not parenthesized

 5.  (+ a a a)         // addition is ternary, parenthesized

 6.  * + a a a         // addition is binary, not parenthesized

 7.  * (+ a a a) a     // addition is ternary, parenthesized

 8.  (+ + a a a a)     // first addition is ternary, parenthesized

 9.  + (+ a a a) a     // second addition is ternary, parenthesized

10.  (+ (+ a a a) a a) // both additions are ternary, parenthesized
```

   Answer the following questions:

   (a) Write a grammar, of type *EBNF* and not ambiguous, that generates the sketched language of arithmetic expressions. The grammar generates one expression.

   (b) Test the correctness of your grammar by drawing the syntax trees of the sample arithmetic expressions n. 7, 8 and 9.

   (c) (optional) Argue (briefly but in a non-trivial and non-tautological manner) that your grammar is not ambiguous.

Solution

(a) By observing that the prefix form does not need to specify a precedence between operators, so that it is not really necessary to use a different syntactic class, i.e., nonterminal, for a (sub)expression, a term and a factor, a straightforward *EBNF* grammar, not ambiguous, that generates the sketched language is as follows::

$$\langle \mathsf{EXPR} \rangle \to + \ \langle \mathsf{EXPR} \rangle \ \langle \mathsf{EXPR} \rangle \ \mid \ \text{`(' } + \ \langle \mathsf{EXPR} \rangle^3 \ \langle \mathsf{EXPR} \rangle^* \ \text{`)'} \ \mid \ * \ \langle \mathsf{EXPR} \rangle \ \langle \mathsf{EXPR} \rangle \ \mid \ \mathsf{a}$$

or even, by using the operator $[\ ]_h^k$ of repetition from h to k times (here $k = \infty$):

$$\langle \mathsf{EXPR} \rangle \to + \ \langle \mathsf{EXPR} \rangle^2 \ \mid \ \text{`(' } + \left[ \ \langle \mathsf{EXPR} \rangle \ \right]_3^\infty \ \text{`)'} \ \mid \ * \ \langle \mathsf{EXPR} \rangle^2 \ \mid \ \mathsf{a}$$

If one prefers to proceed in a more systematic way, a viable *EBNF* grammar can be derived from that of the binary expressions in infix form (axiom EXPR), which is shown below:

$$\textit{infix} \begin{cases} \langle \mathsf{EXPR} \rangle \ \to \ \langle \mathsf{TERM} \rangle \ + \ \langle \mathsf{TERM} \rangle \ \mid \ \langle \mathsf{TERM} \rangle \\ \hline \langle \mathsf{TERM} \rangle \ \to \ \langle \mathsf{FACT} \rangle \ * \ \langle \mathsf{FACT} \rangle \ \mid \ \langle \mathsf{FACT} \rangle \\ \hline \langle \mathsf{FACT} \rangle \ \to \ \text{`(' } \langle \mathsf{EXPR} \rangle \ \text{`)'} \ \mid \ \mathsf{a} \end{cases}$$

This infix grammar already fulfills most requirements, except those of being in prefix form and of dealing with n-ary addition for $n \geq 3$. Then, by applying to it the prefix transformation, introducing n-ary addition and arranging the parentheses as requested, we obtain the following one (axiom EXPR):

$$\textit{prefix} \begin{cases} \langle \mathsf{EXPR} \rangle \ \to \ + \ \langle \mathsf{TERM} \rangle \ \langle \mathsf{TERM} \rangle \ \mid \\ \qquad\qquad \text{`(' } + \ \langle \mathsf{TERM} \rangle^3 \ \langle \mathsf{TERM} \rangle^* \ \text{`)'} \ \mid \\ \qquad\qquad \langle \mathsf{TERM} \rangle \\ \hline \langle \mathsf{TERM} \rangle \ \to \ * \ \langle \mathsf{FACT} \rangle \ \langle \mathsf{FACT} \rangle \ \mid \ \langle \mathsf{FACT} \rangle \\ \hline \langle \mathsf{FACT} \rangle \ \to \ \langle \mathsf{EXPR} \rangle \ \mid \ \mathsf{a} \end{cases}$$

This prefix grammar correctly generates the language, but it is affected by circular derivations, e.g., EXPR ⇒ TERM ⇒ FACT ⇒ EXPR, thus it is ambiguous. By cutting the circular derivations, we obtain an equivalent grammar, of type *EBNF* and not ambiguous (axiom EXPR):

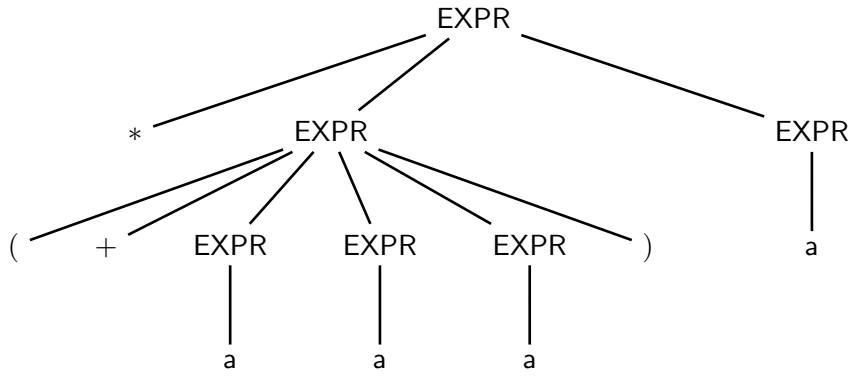
$$\textit{prefix unamb.} \begin{cases} \langle \mathsf{EXPR} \rangle \ \to \ + \ \langle \mathsf{TERM} \rangle \ \langle \mathsf{TERM} \rangle \ \mid \\ \qquad\qquad \text{`(' } + \ \langle \mathsf{TERM} \rangle^3 \ \langle \mathsf{TERM} \rangle^* \ \text{`)'} \ \mid \\ \qquad\qquad * \ \langle \mathsf{FACT} \rangle \ \langle \mathsf{FACT} \rangle \ \mid \ \mathsf{a} \\ \hline \langle \mathsf{TERM} \rangle \ \to \ \langle \mathsf{FACT} \rangle \\ \hline \langle \mathsf{FACT} \rangle \ \to \ \langle \mathsf{EXPR} \rangle \end{cases}$$

To break the circularity, it suffices to remove the copy rule EXPR → TERM. This grammar is correct and satisfies all the requirements. Furthermore, it preserves the syntactic distinction between a (sub)expression, a term and a factor.

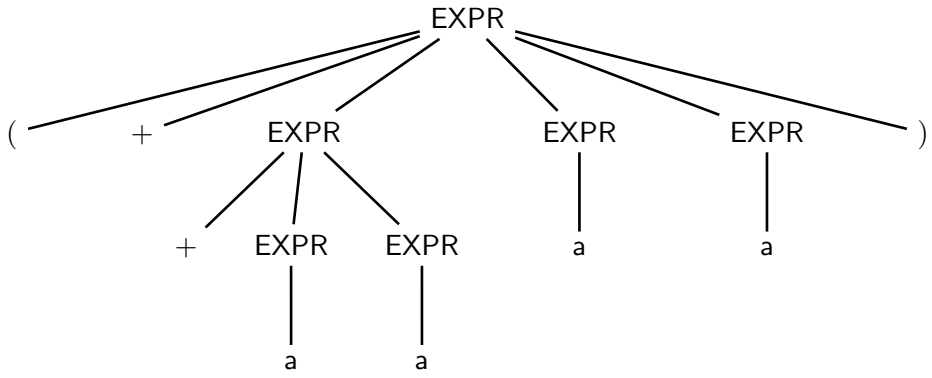
Notice that, if we also remove the two copy rules still left and we consequently eliminate the nonterminals TERM and FACT, then the syntactic distinction between a (sub)expression, a term and a factor vanishes, and we obtain again the straightforward solution informally provided at the beginning.

(b) Here are the trees, with the straightforward grammar:

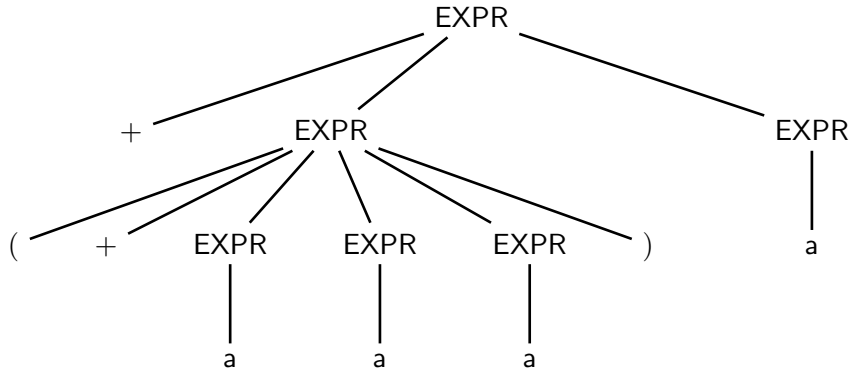
$$* \; (\; + \; a \; a \; a \;) \; a$$



$$(\; + \; + \; a \; a \; a \; a \;)$$



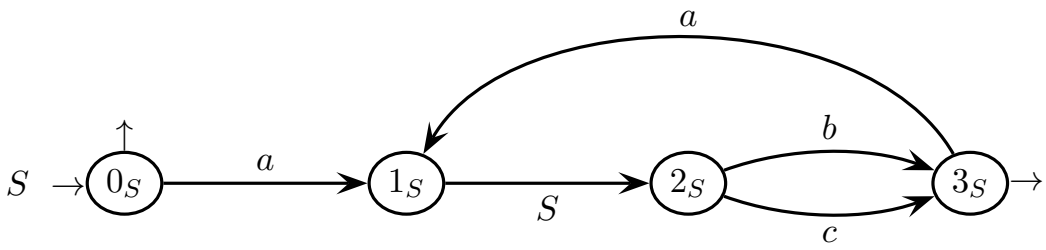
$$+ \; (\; + \; a \; a \; a \;) \; a$$



(c) The grammar (straightforward) is of type $ELL(1)$, as it is easy to see by examining the first terminal generated by the alternative rules, and controlling the number of iterations of the Kleene star. Thus it is clearly unambiguous. Alternatively, draw the machine net and check the guide sets.

3 Syntax Analysis and Parsing Methodologies 20%

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



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

(a) Draw the complete pilot of grammar G and show that grammar G is $ELR\,(1)$.

(b) Write all the guide sets of the machine net of grammar G (on the call arcs and exit arrows) and show that grammar G is $ELL\,(1)$.

(c) Write a simulation of the bottom-up analysis of the valid string below:

$$a\ a\ b\ a\ c\ b \in L\,(G)$$

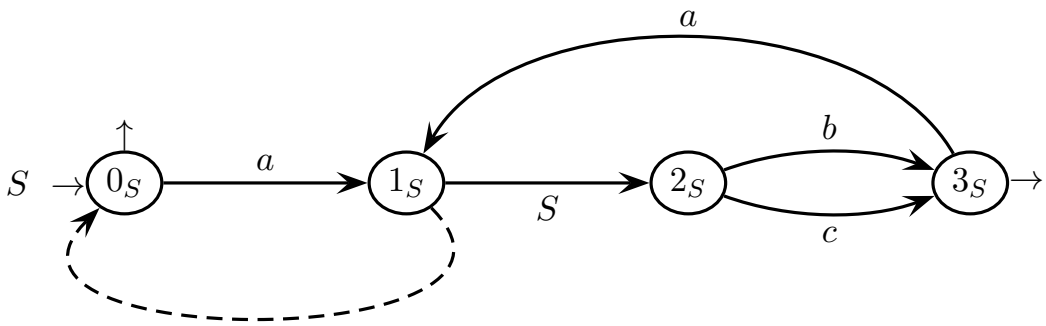
Show at least the stack contents, the parsing actions and the resulting syntax tree.

(d) (optional) Write the recursive descent procedure for nonterminal S, based on the *ELL* method.

---

please draw here the complete pilot – question (a)

please write here all the guide sets – question (b)



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

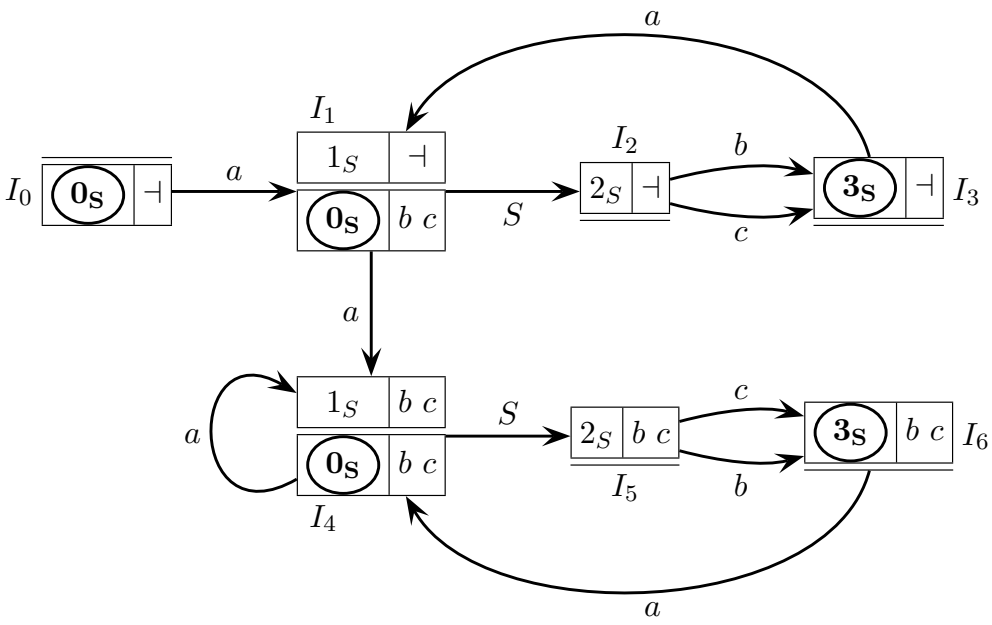
stack of alternated macro-states and grammar symbols	*input left to be scanned*	*move executed*
I_0	$a\ a\ b\ a\ c\ b\ \dashv$	initial configuration
$I_0\ a\ I_{...}$	$a\ b\ a\ c\ b\ \dashv$	terminal shift: $I_0 \xrightarrow{a} I_{...}$

draw here the piecewise tree construction

please write here the recursive descent procedure of nonterminal S – question (d)

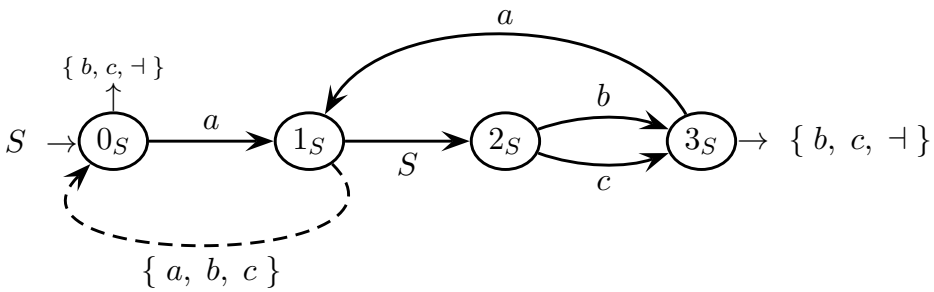
Solution

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



The pilot structure is quite regular. There are no conflicts of any type, thus the grammar is *ELR*(1). Notice that the pilot has the *STP*.

(b) Here are all the guide sets on the call arcs and exit arrows of the machine net of grammar G:

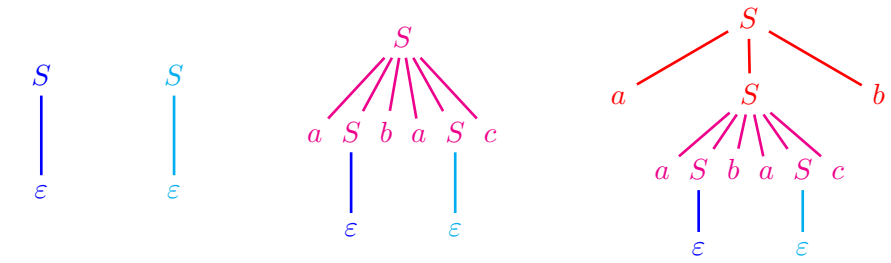


There are no conflicts between the guide sets on the bifurcation states 0_S, 2_S and 3_S, thus the grammar is *ELL*(1). This is expected since the pilot satisfies the *ELL* condition: it is *ELR*, has the *STP* and does not have left recursion.

(c) Here is the simulation of the *ELR* analysis of $a\,a\,b\,a\,c\,b$ with the previous pilot:

stack of alternated macro-states and grammar symbols	*input left to be scanned*	*move executed*
I_0	$a\ a\ b\ a\ c\ b\ \dashv$	initial configuration
$I_0\ a\ I_1$	$a\ b\ a\ c\ b\ \dashv$	terminal shift: $I_0 \xrightarrow{a} I_1$
$I_0\ a\ I_1\ a\ I_4$	$b\ a\ c\ b\ \dashv$	terminal shift: $I_1 \xrightarrow{a} I_4$
$I_0\ a\ I_1\ a\ I_4$	$b\ a\ c\ b\ \dashv$	**reduce** 1: $\varepsilon \rightsquigarrow S$
$I_0\ a\ I_1\ a\ I_4\ S\ I_5$	$b\ a\ c\ b\ \dashv$	nonterm. shift: $I_4 \xrightarrow{S} I_5$
$I_0\ a\ I_1\ a\ I_4\ S\ I_5\ b\ I_6$	$a\ c\ b\ \dashv$	terminal shift: $I_5 \xrightarrow{b} I_6$
$I_0\ a\ I_1\ a\ I_4\ S\ I_5\ b\ I_6\ a\ I_4$	$c\ b\ \dashv$	terminal shift: $I_6 \xrightarrow{a} I_4$
$I_0\ a\ I_1\ a\ I_4\ S\ I_5\ b\ I_6\ a\ I_4$	$c\ b\ \dashv$	**reduce** 2: $\varepsilon \rightsquigarrow S$
$I_0\ a\ I_1\ a\ I_4\ S\ I_5\ b\ I_6\ a\ I_4\ S\ I_5$	$c\ b\ \dashv$	nonterm. shift: $I_4 \xrightarrow{S} I_5$
$I_0\ a\ I_1\ a\ I_4\ S\ I_5\ b\ I_6\ a\ I_4\ S\ I_5\ c\ I_6$	$b\ \dashv$	terminal shift: $I_5 \xrightarrow{c} I_6$
$I_0\ a\ I_1$	$b\ \dashv$	**reduce** 3: $a\,S\,b\,a\,S\,c \rightsquigarrow S$
$I_0\ a\ I_1\ S\ I_2$	$b\ \dashv$	nonterm. shift: $I_1 \xrightarrow{S} I_2$
$I_0\ a\ I_1\ S\ I_2\ b\ I_3$	$\dashv$	terminal shift: $I_2 \xrightarrow{b} I_3$
I_0	$\dashv$	**reduce** 4: $a\,S\,b \rightsquigarrow S$
initial configuration	empty tape	reduced to axiom, accept

Here is the piecewise tree construction (colors help visualize the various steps):



red. 1 red. 2 red. 3 red. 4

The simulation is scanned forward from top to bottom, and the subtree of each reduction move is built and connected to the other subtrees.

For completeness, the rightmost derivation that corresponds to the tree is as follows (the rightmost instance of S to derive is underlined):

$$S \overset{4}{\Rightarrow} a\ \underline{S}\ b \overset{3}{\Rightarrow} a\ a\ S\ b\ a\ \underline{S}\ c\ b \overset{2}{\Rightarrow} a\ a\ \underline{S}\ b\ a\ c\ b \overset{1}{\Rightarrow} a\ a\ b\ a\ c\ b$$

This derivation is obtained by backward scanning the simulation, from bottom to top, and by collecting the reduction moves encountered.

(d) Here is the requested procedure, with a faithful encoding of the control graph:

```
procedure S
if cc ∈ { a } then                                   // state 0
    cc := next                                         // 0 -a-> 1
    if cc ∈ { a, b, c } then                           // state 1
        LOOP: call S                                     // 1 -S-> 2
        if cc ∈ { b, c } then                          // state 2
            cc := next                                  // 2 -b,c-> 3
            if cc ∈ { a } then                         // state 3
                cc := next
                goto LOOP                                // 3 -a-> 1
            else if cc ∈ { b, c, ⊣ } then return       // state 3
            else error                                 // error in 3
        else error                                     // error in 2
    else error                                         // error in 1
else if cc ∈ { b, c, ⊣ } then return                   // state 0
else error                                             // error in 0
end procedure
```

With some optimization, i.e., loop rolling, and by moving (and grouping) all the error cases to the end, the procedure can be recoded as follows:

```
procedure S
while cc ∈ { a } do                                  // state 0 or 3
    cc := next                                         // 0 -a-> 1 or 3 -a-> 1
    if cc ∈ { a, b, c } then                           // state 1
        call S                                           // 1 -S-> 2
        if cc ∈ { b, c } then                          // state 2
            cc := next                                  // 2 -b,c-> 3
            if cc ∈ { b, c, ⊣ } then return            // state 3
if cc ∈ { b, c, ⊣ } then return                        // state 0
error                                                  // all errors
end procedure
```

The main program calls the procedure after initializing the current character *cc*. At the end, the main program must check that the input tape is empty.

4 Language Translation and Semantic Analysis 20%

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

$$L_s = \left\{\ a^h\ b^k \mid \quad h,\ k \geq 0\ \wedge\ \left(k \leq h \leq 2k\ \vee\ k > h\right)\ \right\}$$

A source grammar G_s that generates language L_s is (axiom S):

$$G_s \begin{cases} S\ \rightarrow\ X\ \mid\ Y \\ X\ \rightarrow\ a\ a\ X\ b\ \mid\ a\ X\ b\ \mid\ \varepsilon \\ Y\ \rightarrow\ a\ Y\ b\ \mid\ Y\ b\ \mid\ b \end{cases}$$

Answer the following questions:

(a) Write a syntax scheme G_τ (or a translation grammar) for the translation τ over the source language L_s defined as follows:

$$\tau\left(a^h\ b^k\right) = a^s\ b^s \qquad\qquad s = \min\left(h,\ k\right)$$

without changing the source grammar G_s.

Furthermore, draw the syntax translation trees of the two valid source strings:

$$a\,a\,b \qquad\qquad a\,b\,b$$

(b) Is scheme (or grammar) G_τ a deterministic translation ? Explain your answer.

Solution

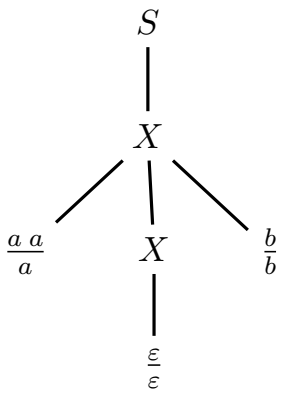
(a) The source strings can be informally characterized as those that consist of letters a followed by letters b, i.e., $a^h\, b^k$, where the number h of letters a is not larger than twice the number k of letters b.

A possible translation grammar G_τ is (axiom S), which computes the destination string $a^s\, b^s$ with $s = \min(h,\ k)$:

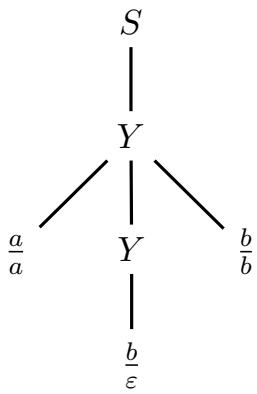
$$G_\tau \begin{cases} S \ \to\ X \ \mid\ Y & \\ X \ \to\ \frac{a\,a}{a}\ X\ \frac{b}{b} \ \mid\ \frac{a}{a}\ X\ \frac{b}{b} \ \mid\ \frac{\varepsilon}{\varepsilon} & (1) \text{ if } h \geq k, \text{ i.e., } s = k \\ Y \ \to\ \frac{a}{a}\ Y\ \frac{b}{b} \ \mid\ Y\ \frac{b}{\varepsilon} \ \mid\ \frac{b}{\varepsilon} & (2) \text{ if } h < k, \text{ i.e., } s = h \end{cases}$$

In the case 1 the number s of letters a output is equal to the number k of input letters b, while in the case 2 the number s of letters b output is equal to the number h of input letters a. This is equivalent to computing $s = \min(h,\ k)$.

Here are the two translation trees:



$\tau(a\,a\,b) = a\,b$



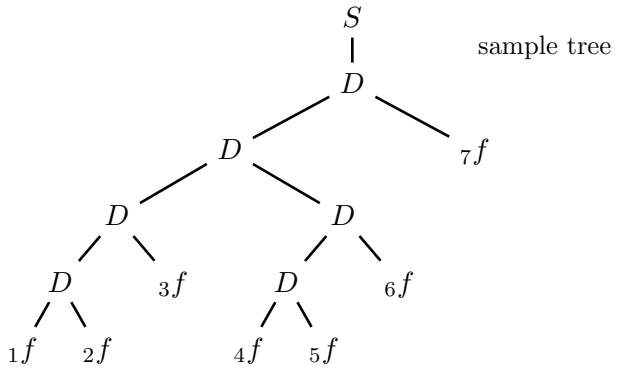
$\tau(a\,b\,b) = a\,b$

(b) Translation τ is a function from a string to one string, since the exponent s is the unique minimum value between the exponents h and k.

However, the source grammar G_s is clearly ambiguous, as both nonterminals X and Y generate strings ambiguously. For instance, each string $S \Rightarrow X \stackrel{*}{\Rightarrow} a\,a\,a\,b\,b$ and $S \Rightarrow Y \stackrel{*}{\Rightarrow} a\,b\,b\,b$ has two syntax trees, thus both are ambiguous. Therefore the syntax scheme (or translation grammar) G_τ is certainly nondeterministic.

2. The grammar below (axiom S), over a one-letter terminal alphabet $\{\ f\ \}$, models the abstract syntax of the binary trees where all the internal nodes have two child nodes (simplified for the sake of brevity). A sample tree is reported on the right (a prefix subscript index is associated to every leaf node only for future reference):

$$\begin{cases} S \rightarrow D \\ D \rightarrow D\ D \\ D \rightarrow D\ f \\ D \rightarrow f\ f \end{cases}$$



sample tree

This grammar just models an *abstract* syntax, which is unsuitable for syntax analysis.

We call *distance* between two tree nodes n_1 and n_2, denoted as $dist(n_1, n_2)$, the length of the shortest path between them; for instance $dist({}_2f,\ {}_4f) = 6$ and $dist({}_2f,\ {}_7f) = 5$. The depth of any tree node is its distance from the root, hence in the sample tree the node depth ranges from 0 (for the root S) to 5 (for the leaf nodes ${}_1f$, ${}_2f$, ${}_4f$ and ${}_5f$).

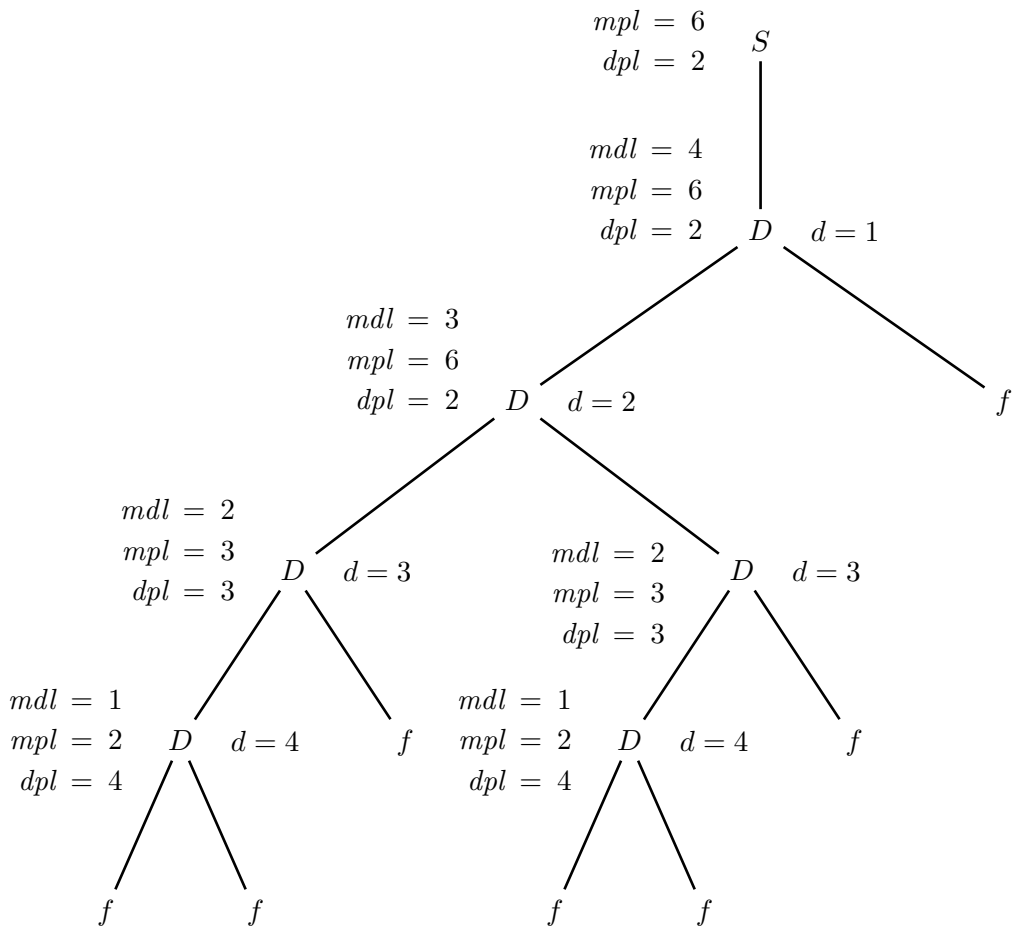
We intend to design an attribute grammar that supports the computation of the maximal distance between any two tree leaf nodes. In the above sample tree the maximal distance is 6, because for instance $dist({}_2f,\ {}_4f) = 6$.

The grammar must also allow for the computation of the depth of the deepest internal node that is a common ancestor of any two leaf nodes that have maximal distance. In the above example the depth is 2, because the deepest common ancestor of the leaf nodes ${}_2f$ and ${}_4f$ has depth 2 (it is the second node D from the root).

Consider the table below, which lists a set of attributes for the grammar:

name	*type*	*domain*	*symbol*	*meaning*
d	right	integer	D	***d***epth of the current node
mdl	left	integer	D	***m***aximal ***d***istance, from the current node, of a ***l***eaf node descendant of the current node
mpl	left	integer	D, S	***m***aximal distance between any ***p***air of ***l***eaf nodes both descendant of the current node
dpl	left	integer	D, S	***d***epth of the deepest internal node that is a common ancestor of a ***p***air of ***l***eaf nodes that are both descendant of the current node and have a maximal distance

The picture below shows the previous sample tree decorated with the attribute values:



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

(a) Write the semantic rules to compute attribute d.

(b) Write the semantic rules to compute attribute *mdl*. Decorate the next tree with the attribute value, according to the provided semantic rules.

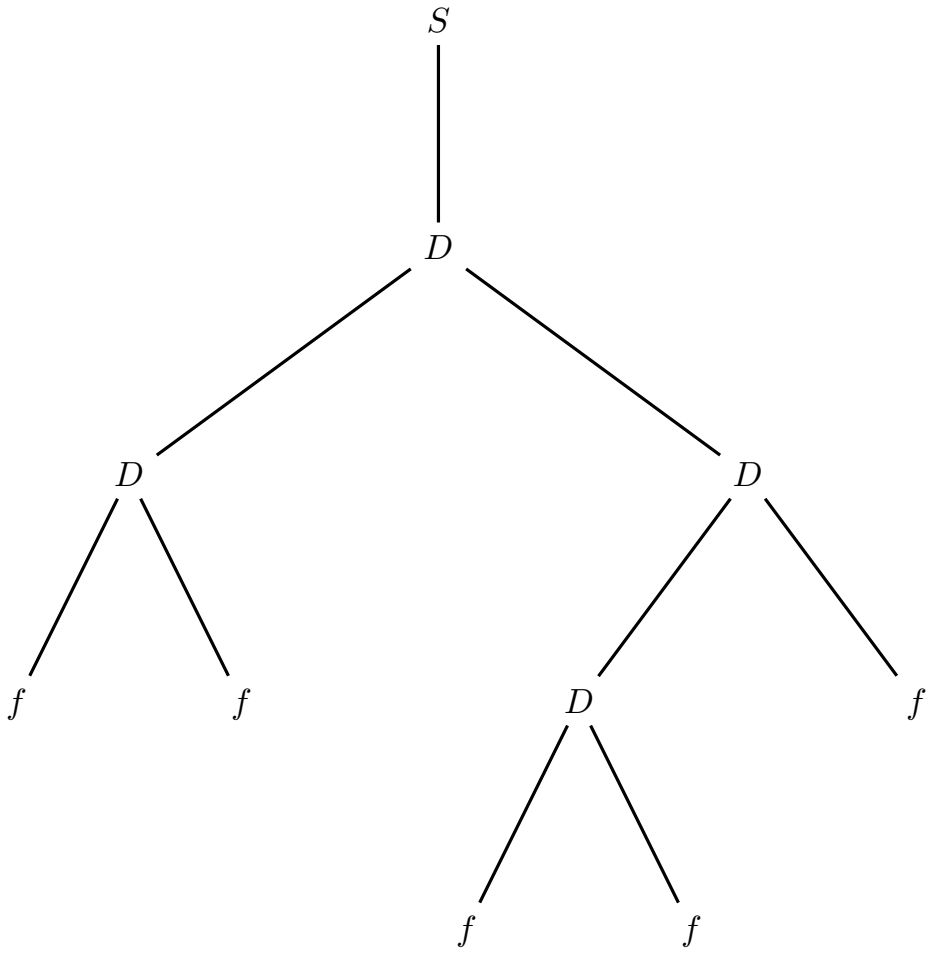
(c) Write the semantic rules to compute attribute *mpl*. Decorate the next tree with the attribute value, according to the provided semantic rules.

(d) (optional) Write the semantic rules to compute attribute *dpl*. Decorate the next tree with the attribute value, according to the provided semantic rules.

When writing the semantic rules you can use functions like $\max(v_1, \ldots, v_n)$, for any $n \geq 2$, to denote the maximum of a set of values $v_1, \ldots, v_n$.

---

tree to be decorated for questions (b-c-d)



semantic rules for attribute d – question (a)

#	*syntax*	*semantics*
1:	$S_0 \rightarrow D_1$	
2:	$D_0 \rightarrow D_1\ D_2$	
3:	$D_0 \rightarrow D_1\ f$	
4:	$D_0 \rightarrow f\ f$	

semantic rules for attribute mdl – question (b)

#	*syntax*	*semantics*
1:	$S_0 \rightarrow D_1$	
2:	$D_0 \rightarrow D_1\ D_2$	
3:	$D_0 \rightarrow D_1\ f$	
4:	$D_0 \rightarrow f\ f$	

semantic rules for attribute *mpl* – question (c)

#	*syntax*	*semantics*
1:	$S_0 \rightarrow D_1$	
2:	$D_0 \rightarrow D_1\ D_2$	
3:	$D_0 \rightarrow D_1\ f$	
4:	$D_0 \rightarrow f\ f$	

semantic rules for attribute *dpl* – question (d)

#	*syntax*	*semantics*
1:	$S_0 \rightarrow D_1$	
2:	$D_0 \rightarrow D_1\ D_2$	
3:	$D_0 \rightarrow D_1\ f$	
4:	$D_0 \rightarrow f\ f$	

Solution

(a) Here are the semantic rules for attribute d:

#	*syntax*			*semantics*
1:	S_0	$\rightarrow$	D_1	$d_1 := 1$
2:	D_0	$\rightarrow$	$D_1\ D_2$	$d_1,\ d_2 := d_0 + 1$
3:	D_0	$\rightarrow$	$D_1\ f$	$d_1 := d_0 + 1$
4:	D_0	$\rightarrow$	$f\ f$	

This is quite standard a computation: the node depth is incremented from top to bottom. Rule 1 initializes.

(b) Here are the semantic rules for attribute mdl:

#	*syntax*			*semantics*
1:	S_0	$\rightarrow$	D_1	
2:	D_0	$\rightarrow$	$D_1\ D_2$	$mdl_0 := \max(mdl_1,\ mdl_2) + 1$
3:	D_0	$\rightarrow$	$D_1\ f$	$mdl_0 := mdl_1 + 1$
4:	D_0	$\rightarrow$	$f\ f$	$mdl_0 := 1$

In the rule 2, the maximum root-to-leaf distance in the two subtrees is taken, plus one to count the parent node. In the rule 3, there is only one subtree, thus taking the maximum is unnecessary. Rule 4 initializes.

(c) Here are the semantic rules for attribute mpl:

#	*syntax*			*semantics*
1:	S_0	$\rightarrow$	D_1	$mpl_0 := mpl_1$
2:	D_0	$\rightarrow$	$D_1\ D_2$	$mpl_0 := \max(mpl_1,\ mpl_2,\ mdl_1 + mdl_2 + 2)$
3:	D_0	$\rightarrow$	$D_1\ f$	$mpl_0 := \max(mpl_1,\ mdl_1 + 2)$
4:	D_0	$\rightarrow$	$f\ f$	$mpl_0 := 2$

In the rule 2, it is taken the maximum of the leaf-to-leaf distances in the two subtrees and of the distance sum plus two for the two leaves to belong to different subtrees. In the rule 3, there is only one subtree, thus the maximum is simplified. Rule 4 initializes.

(d) Here are the semantic rules for attribute *dpl*:

#	*syntax*	*semantics*
1:	$S_0 \rightarrow D_1$	$dpl_0 := dpl_1$

2:	$D_0 \rightarrow D_1\ D_2$	**if** $(mpl_0 = mpl_1)$ **then**
		$dpl_0 := dpl_1$
		else if $(mpl_0 = mpl_2)$ **then**
		$dpl_0 := dpl_2$
		else
		$dpl_0 := d_0$
		end if

3:	$D_0 \rightarrow D_1\ f$	**if** $(mpl_0 = mpl_1)$ **then** – equiv. to $mpl_1 \geq mdl_1+2$
		$dpl_0 := dpl_1$
		else
		$dpl_0 := d_0$
		end if

4:	$D_0 \rightarrow f\ f$	$dpl_0 := d_0$

In the rules 2 and 3 (with some simplification) the max distances of letter pairs are compared and the parent node depth is set accordingly. Rule 4 initializes.

tree decorated – all questions