

2. The grammar below (axiom S) generates (abstract syntax) trees with nodes of three types, that is, internal binary B , internal unary U and leaf f :

$$\left\{ \begin{array}{l} 1: S \rightarrow B \\ 2: B \rightarrow B B \\ 3: B \rightarrow U B \\ 4: B \rightarrow U U \end{array} \right. \quad \left\{ \begin{array}{l} 5: U \rightarrow U \\ 6: U \rightarrow B \\ 7: U \rightarrow f \end{array} \right.$$

A sample tree is shown on the next page.

For such trees we define the *depth* of a node of type B , U or f , to be the distance from the root node S minus one. For instance, in the sample tree the largest depth of the unary internal nodes (of type U) is 4.

We want to define an attribute grammar, based on the above syntax, to compute in the root node S the *number of internal unary nodes* that have *maximal depth*. This is the number of nodes of type U with a depth such that no other node of type U in the tree has a larger depth. In the sample tree, the number is 4 because there are four nodes of type U at depth 4, and no other node of type U at a larger depth.

Here are the attributes permitted to use (no other attributes are allowed):

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
d	right	integer	B, U	depth ≥ 0 of a node
dp	left	integer	B, U	depth of the deepest node of type U that is a descendant of the current node; conventionally 0 if no such node exists
ndp	left	integer	S, B, U	number of nodes of type U , among the descendants of the current node, that have maximal depth; conventionally 0 if no such node exists

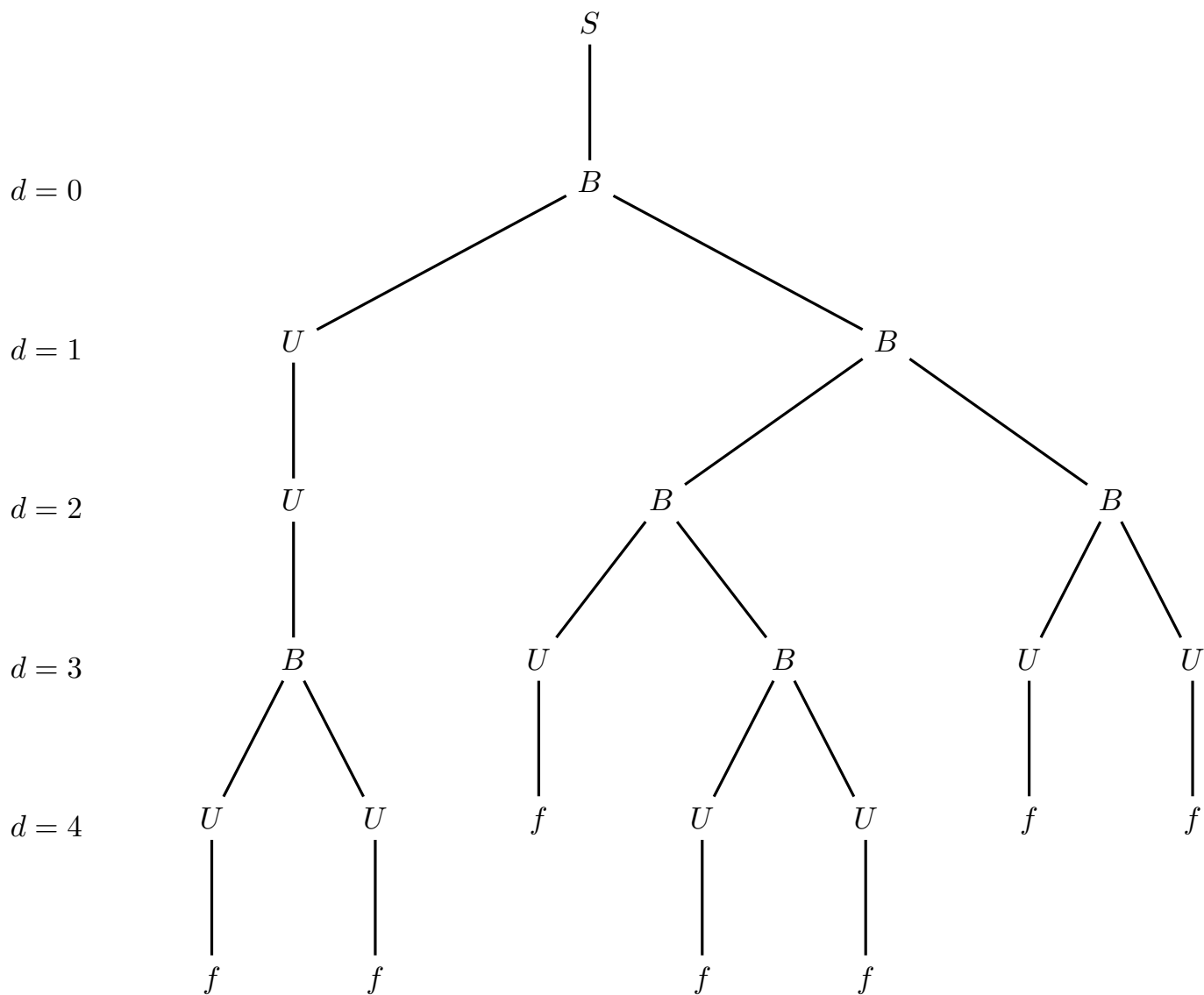
Answer the following questions (use the tables / trees / spaces on the next pages):

- Decorate the sample tree (next page) with the values of attributes dp and ndp .
- Write an attribute grammar, based on the syntax above and using only the permitted attributes (do not change the attribute types), that computes the required value of attribute ndp in the root node S of the tree. The attribute grammar must be of type one-sweep. Proceed stepwise and separately write:
 - the semantic rules for computing attribute d into table 1
 - those for attribute dp into table 2
 - and those for attribute ndp into table 3

Please, do not copy the rules already written in a table into the next one.

- (optional) Prove that your attribute grammar is of type one-sweep.

sample tree to decorate – question (a)



syntax *semantics* – question (b) point (i) – TAB 1

1: $S_0 \rightarrow B_1$

2: $B_0 \rightarrow B_1 B_2$

3: $B_0 \rightarrow U_1 B_2$

4: $B_0 \rightarrow U_1 U_2$

5: $U_0 \rightarrow U_1$

6: $U_0 \rightarrow B_1$

7: $U_0 \rightarrow f$

syntax

semantics – question (b) point (ii) – TAB 2

1: $S_0 \rightarrow B_1$

2: $B_0 \rightarrow B_1 B_2$

3: $B_0 \rightarrow U_1 B_2$

4: $B_0 \rightarrow U_1 U_2$

5: $U_0 \rightarrow U_1$

6: $U_0 \rightarrow B_1$

7: $U_0 \rightarrow f$

syntax *semantics* – question (b) point (iii) – TAB 3

1: $S_0 \rightarrow B_1$

2: $B_0 \rightarrow B_1 B_2$

3: $B_0 \rightarrow U_1 B_2$

4: $B_0 \rightarrow U_1 U_2$

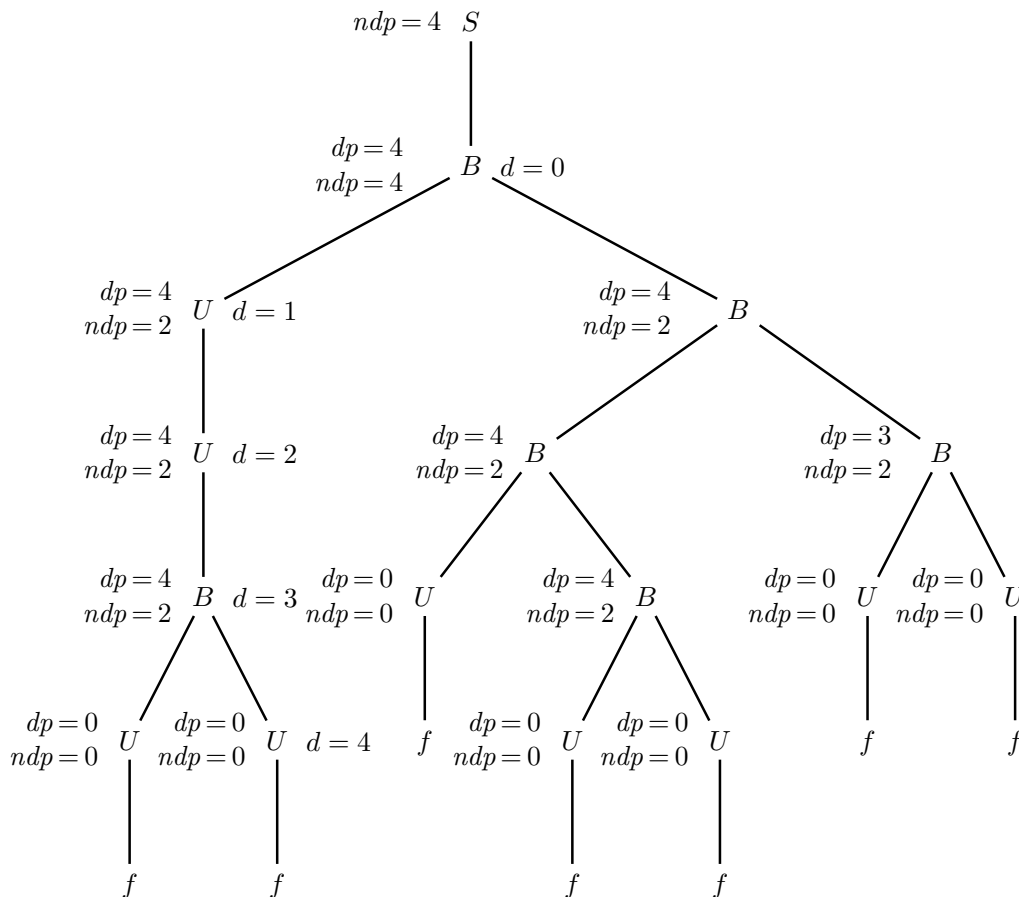
5: $U_0 \rightarrow U_1$

6: $U_0 \rightarrow B_1$

7: $U_0 \rightarrow f$

Solution

- (a) Here is the decorated syntax tree, where the attributes dp and ndp (attribute d is obvious and partially shown) are assigned a value according to their specifications given in the exercise text:



As customary, left and the right attributes are written on the left and right of the tree node they refer to, respectively. See also the semantic functions.

The computation deserves no special comment. Only notice that attribute dp takes the depth value (computed from the tree root S) of the deepest node U in the subtree and simply transports it upwards unchanged, unless it is superseded by a deeper node U . Therefore, on moving from bottom to top, attribute dp never decreases: it keeps its value or increases. Attribute ndp behaves similarly.

(b) Here is the complete attribute grammar (all semantic functions together):

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow B_1$	$d_1 := 0$ $ndp_0 := ndp_1$
2:	$B_0 \rightarrow B_1 B_2$	$d_1, d_2 := d_0 + 1$ $dp_0 := \max(dp_1, dp_2)$ $ndp_0 := \text{if } (dp_1 == dp_2) \text{ then } ndp_1 + ndp_2$ $\text{else if } (dp_1 > dp_2) \text{ then } ndp_1$ $\text{else } ndp_2 \text{ endif}$
3:	$B_0 \rightarrow U_1 B_2$	$d_1, d_2 := d_0 + 1$ $dp_0 := \text{if } (dp_1 == 0) \text{ then } dp_2 \text{ (not } d_2 \text{ here)}$ $\text{else } \max(dp_1, dp_2) \text{ endif}$ $ndp_0 := \text{if } (dp_1 == 0) \text{ then } ndp_2$ $\text{else if } (dp_1 == dp_2) \text{ then } ndp_1 + ndp_2$ $\text{else if } (dp_1 > dp_2) \text{ then } ndp_1$ $\text{else } ndp_2 \text{ endif}$
4:	$B_0 \rightarrow U_1 U_2$	$d_1, d_2 := d_0 + 1$ $dp_0 := \text{if } (dp_1 == 0 \text{ and } dp_2 == 0) \text{ then } d_1 \text{ (or } d_2)$ $\text{else } \max(dp_1, dp_2) \text{ endif}$ $ndp_0 := \text{if } (dp_1 == 0 \text{ and } dp_2 == 0) \text{ then } 2$ $\text{else if } (dp_1 == dp_2) \text{ then } ndp_1 + ndp_2$ $\text{else if } (dp_1 > dp_2) \text{ then } ndp_1$ $\text{else } ndp_2 \text{ endif}$
5:	$U_0 \rightarrow U_1$	$d_1 := d_0 + 1$ $dp_0 := \text{if } (dp_1 == 0) \text{ then } d_1 \text{ else } dp_1 \text{ endif}$ $ndp_0 := \text{if } (dp_1 == 0) \text{ then } 1 \text{ else } ndp_1 \text{ endif}$
6:	$U_0 \rightarrow B_1$	$d_1 := d_0 + 1$ $dp_0 := dp_1$ $ndp_0 := ndp_1$
7:	$U_0 \rightarrow f$	$dp_0 := 0$ $ndp_0 := 0$

These semantic functions are coherent with the decorated sample tree. They are rather intuitive and do not deserve any specific comments. Just notice that nonterminal B unavoidably has nonterminals U appended in its subtree (else it could not have leaves f appended therein either), thus in the rules that contain instances of B in their right parts (namely rules 2, 3 and 6), it is unnecessary to test if the attribute dp of B is zero or not, since obviously it may not be zero.

Separately and written as a program:

#	<i>syntax</i>	<i>semantics</i> – question (b) point (i) – TAB 1
1:	$S_0 \rightarrow B_1$	$d_1 := 0$
2:	$B_0 \rightarrow B_1 B_2$	$d_1, d_2 := d_0 + 1$
3:	$B_0 \rightarrow U_1 B_2$	$d_1, d_2 := d_0 + 1$
4:	$B_0 \rightarrow U_1 U_2$	$d_1, d_2 := d_0 + 1$
5:	$U_0 \rightarrow U_1$	$d_1 := d_0 + 1$
6:	$U_0 \rightarrow B_1$	$d_1 := d_0 + 1$
7:	$U_0 \rightarrow f$	

#	<i>syntax</i>	<i>semantics</i> – question (b) point (ii) – TAB 2
1:	$S_0 \rightarrow B_1$	
2:	$B_0 \rightarrow B_1 B_2$	$dp_0 := \max(dp_1, dp_2)$
3:	$B_0 \rightarrow U_1 B_2$	if ($dp_1 == 0$) then $dp_0 := dp_2$ else $dp_0 := \max(dp_1, dp_2)$ endif // equivalent form: $dp_0 := \max(dp_1, dp_2)$
4:	$B_0 \rightarrow U_1 U_2$	if ($dp_1 == 0$ and $dp_2 == 0$) then $dp_0 := d_1$ (or $dp_0 := d_2$) else $dp_0 := \max(dp_1, dp_2)$ endif // equivalent form: $dp_0 := \max(dp_1, dp_2, d_1)$
5:	$U_0 \rightarrow U_1$	if ($dp_1 == 0$) then $dp_0 := d_1$ else $dp_0 := dp_1$ endif // equivalent form: $dp_0 := \max(dp_1, d_1)$
6:	$U_0 \rightarrow B_1$	$dp_0 := dp_1$
7:	$U_0 \rightarrow f$	$dp_0 := 0$

Since the attribute d of the internal nodes U is necessarily > 0 and the attribute dp of any node (U or B) is ≥ 0 , equivalent short forms that use an operator \max are possible, as they also express the nullity tests of attribute dp . Such forms are correct, yet they are rather implicit and less immediately understandable.

#	<i>syntax</i>	<i>semantics</i> – question (b) point (iii) – TAB 3
1:	$S_0 \rightarrow B_1$	$ndp_0 := ndp_1$
2:	$B_0 \rightarrow B_1 B_2$	if ($dp_1 == dp_2$) then $ndp_0 := ndp_1 + ndp_2$ else if ($dp_1 > dp_2$) then $ndp_0 := ndp_1$ else $ndp_0 := ndp_2$ endif
3:	$B_0 \rightarrow U_1 B_2$	if ($dp_1 == 0$) then $ndp_0 := ndp_2$ else if ($dp_1 == dp_2$) then $ndp_0 := ndp_1 + ndp_2$ else if ($dp_1 > dp_2$) then $ndp_0 := ndp_1$ else $ndp_0 := ndp_2$ endif
4:	$B_0 \rightarrow U_1 U_2$	if ($dp_1 == 0$ and $dp_2 == 0$) then $ndp_0 := 2$ else if ($dp_1 == dp_2$) then $ndp_0 := ndp_1 + ndp_2$ else if ($dp_1 > dp_2$) then $ndp_0 := ndp_1$ else $ndp_0 := ndp_2$ endif
5:	$U_0 \rightarrow U_1$	if ($dp_1 == 0$) then $ndp_0 := 1$ else $ndp_0 := ndp_1$ endif
6:	$U_0 \rightarrow B_1$	$ndp_0 := ndp_1$
7:	$U_0 \rightarrow f$	$ndp_0 := 0$

Some short equivalent forms might exist for attribute ndp , too, similarly to what has been done for attribute dp , though even less readable.

- (c) The attribute grammar is acyclic, hence it is correct. It is also of type one-sweep: the attributes can be computed through a depth-first tree visit, because the right attributes depend only on right ones in the parent node, and the left attributes depend only on attributes (both right and left) in the siblings.

Specifically, these dependencies hold: the right attribute d depends only on itself in its parent node; furthermore, the left attribute dp depends on itself in its siblings and on the right attribute d in its siblings; finally, the left attribute ndp depends on itself in its siblings and on the left attribute dp in its siblings. Thus the one-sweep condition is easily satisfied.

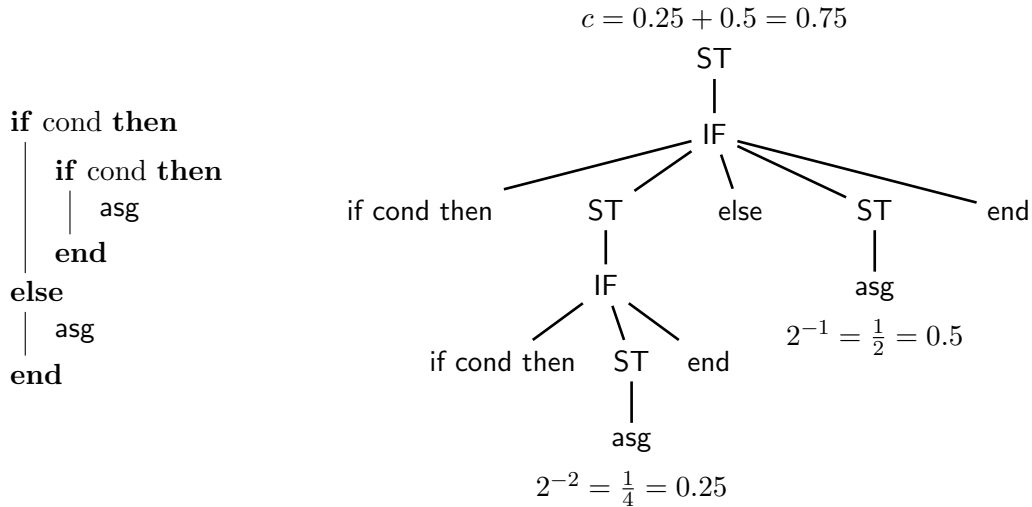
2. Consider the following (simplified) portion of a grammar of a programming language, with nested conditional statements `if ... then ... else ... end`, possibly without `else` branch (axiom `ST`):

$$\left\{ \begin{array}{l} 1: \langle \text{ST} \rangle \rightarrow \langle \text{IF} \rangle \\ 2: \langle \text{ST} \rangle \rightarrow \text{asg} \\ 3: \langle \text{IF} \rangle \rightarrow \text{if cond then } \langle \text{ST} \rangle \text{ else } \langle \text{ST} \rangle \text{ end} \\ 4: \langle \text{IF} \rangle \rightarrow \text{if cond then } \langle \text{ST} \rangle \text{ end} \end{array} \right.$$

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 $2^{-1} = \frac{1}{2} = 0.5$:



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

- 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 $c = 2^{-2} + 2^{-1} = 0.75$.
- Decorate the example tree on the next page and write the values of attribute c in all the relevant nodes.
- 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)

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>
<i>c</i>		real	

#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

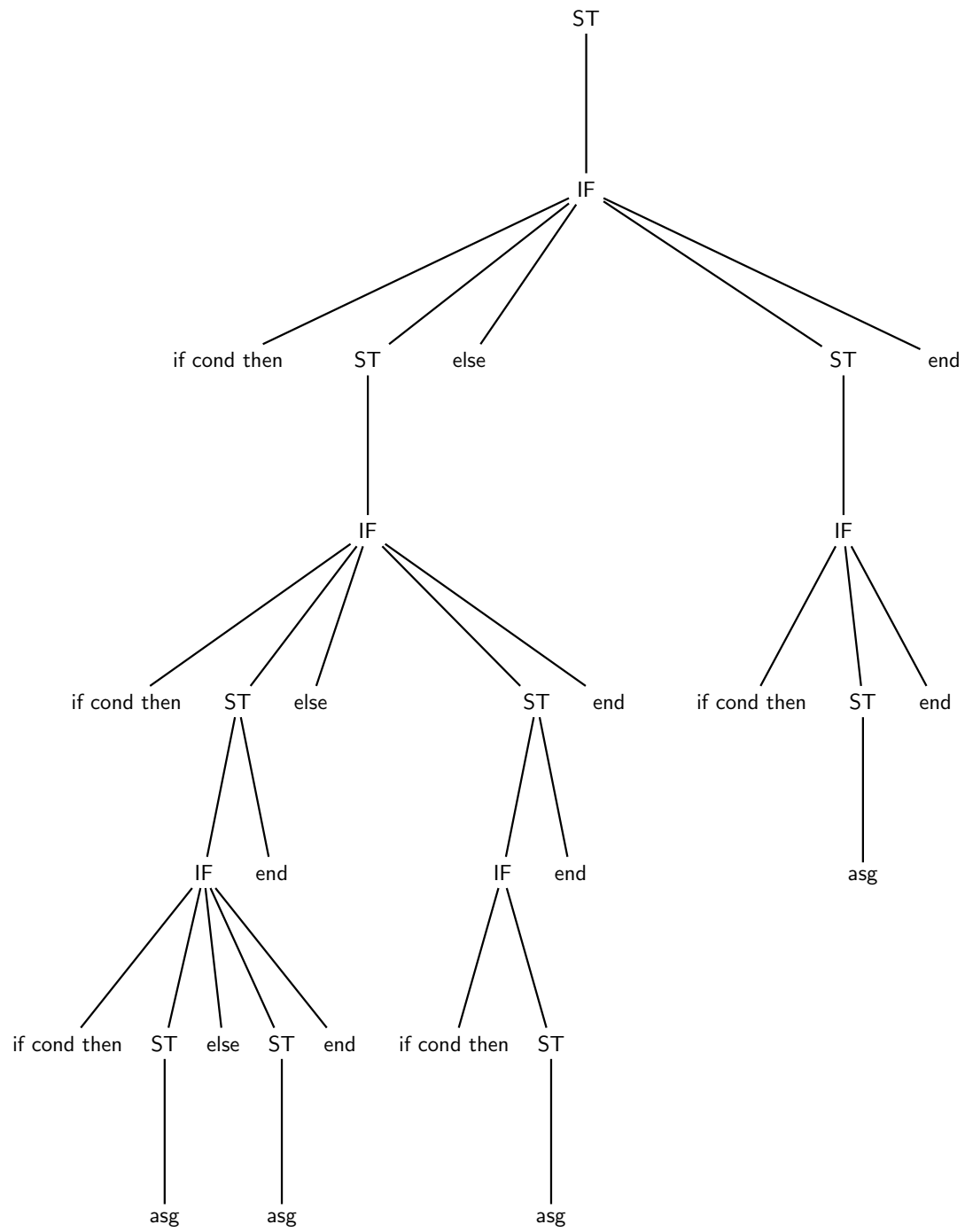
1: $ST_0 \rightarrow IF_1$

2: $ST_0 \rightarrow \text{asg}$

3: $IF_0 \rightarrow \text{if cond then } ST_1 \text{ else } ST_2 \text{ end}$

4: $IF_0 \rightarrow \text{if cond then } ST_1 \text{ end}$

syntax tree to be decorated – question (b)



Solution

- (a) Rules of the attribute grammar:

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>
c	left	real	ST, IF

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

- (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.

- {

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

}

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

- | <i>data structure</i> | <i>exists</i> |
|-----------------------|-------------------------|
| $a : (b, a, c)$ | yes |
| $c : (b, (c, a), c)$ | yes (sample tree above) |
| $a : (b, b, c)$ | no |

(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:

<i>data structure</i>	<i>object depth (if any)</i>
$a : (b, a, c)$	1
$c : (b, (c, a), c)$	2 (sample tree above)
$a : (b, b, c)$	0 (inexistent object)

(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)

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

#	<i>syntax</i>	<i>semantics</i> – question (a)
---	---------------	---------------------------------

$$1: \quad S_0 \quad \rightarrow \quad x : (L_1)$$
$$2: \quad L_0 \rightarrow L_1, E_2$$
$$3: \quad L_0 \rightarrow E_1$$
$$4: \quad E_0 \quad \rightarrow \quad (L_1)$$
$$5: \quad E_0 \rightarrow y$$

#	<i>syntax</i>	<i>semantics</i> – question (b)
---	---------------	---------------------------------

$$1: \quad S_0 \quad \rightarrow \quad x : (L_1)$$
$$2: \quad L_0 \rightarrow L_1, E_2$$
$$3: \quad L_0 \rightarrow E_1$$
$$4: \quad E_0 \quad \rightarrow \quad (L_1)$$
$$5: \quad E_0 \rightarrow y$$

Solution

(a) Here is the attribute grammar:

#	<i>syntax</i>	<i>semantics</i>
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:

#	<i>syntax</i>	<i>semantics</i>
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):

$L(\text{in } o; \text{out } e)$

$E(\text{in } o; \text{out } e)$

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

```

procedure  $L$  ( in  $o$ ; out  $e$  )
var  $o_1, o_2$  : object                // right attributes
var  $e_1, e_2$  : boolean              // left attributes
switch rule do                        // tree node type
  case  $L \rightarrow L, E$  do
     $o_1 := o$                           // inheritance
     $o_2 := o$                           // inheritance
    call  $L(o_1, e_1)$                   // subtree computation
    call  $E(o_2, e_2)$                   // subtree computation
     $e := e_1 \vee e_2$                   // synthesis
  case  $L \rightarrow E$  do
     $o_1 := o$                           // inheritance
    call  $E(o_1, e_1)$                   // subtree computation
     $e := e_1$                           // synthesis
  otherwise do error                  // unknown node type

```

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

```

procedure  $L$  ( in  $o$ ; out  $e$  )
var  $e_1, e_2$  : boolean                // left attributes
switch rule do                        // tree node type
  case  $L \rightarrow L, E$  do
    call  $L(o, e_1)$                   // inheritance / computation
    call  $E(o, e_2)$                   // inheritance / computation
     $e := e_1 \vee e_2$                   // synthesis
  case  $L \rightarrow E$  do call  $E(o, e)$     // inherit / comp /
    synth
  otherwise do error                  // unknown node type

```

Finally, the axiomatic semantic procedure $S(\text{out } e)$ must invoke procedure L with these actual parameters:

$o_1 := x$

call $L(o_1, e_1)$

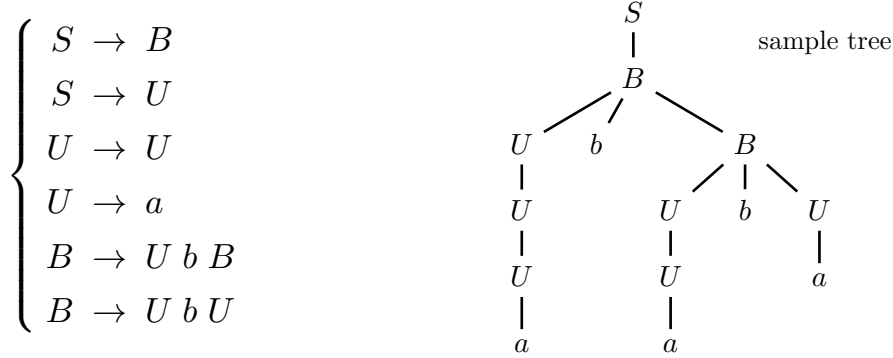
$e := e_1$

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

call $L(x, e)$

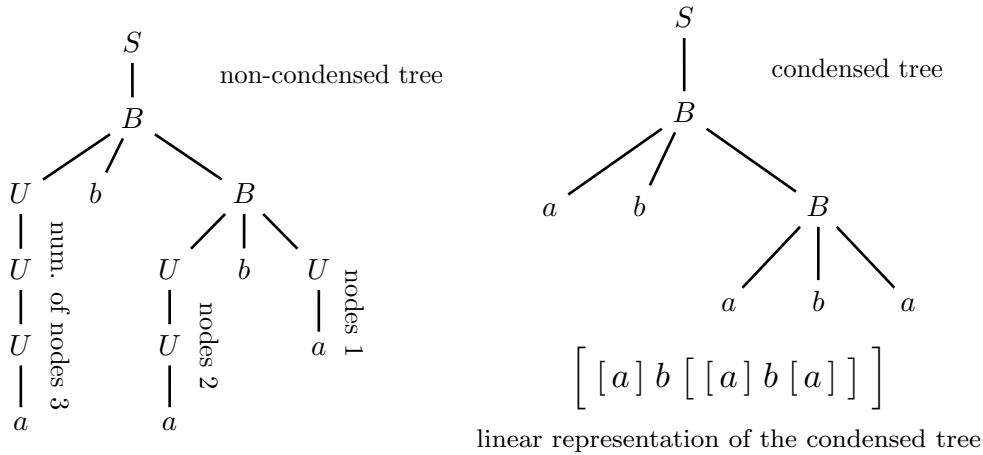
where e is the output parameter that will hold the final result.

2. The grammar below (axiom S), over a two-letter terminal alphabet $\{a, b\}$, models the abstract syntax of (simplified) expressions with binary operator b and atomic operand a . The copy rule $U \rightarrow U$ allows the trees of such expressions to have non-branching paths of arbitrary length, as exemplified in the sample tree on the right:



This grammar models an *abstract* syntax, thus it is unsuitable for syntax analysis.

A *condensed* syntax tree is one where the internal nodes of a non-branching path are canceled, as shown below (related to the same sample tree as above):



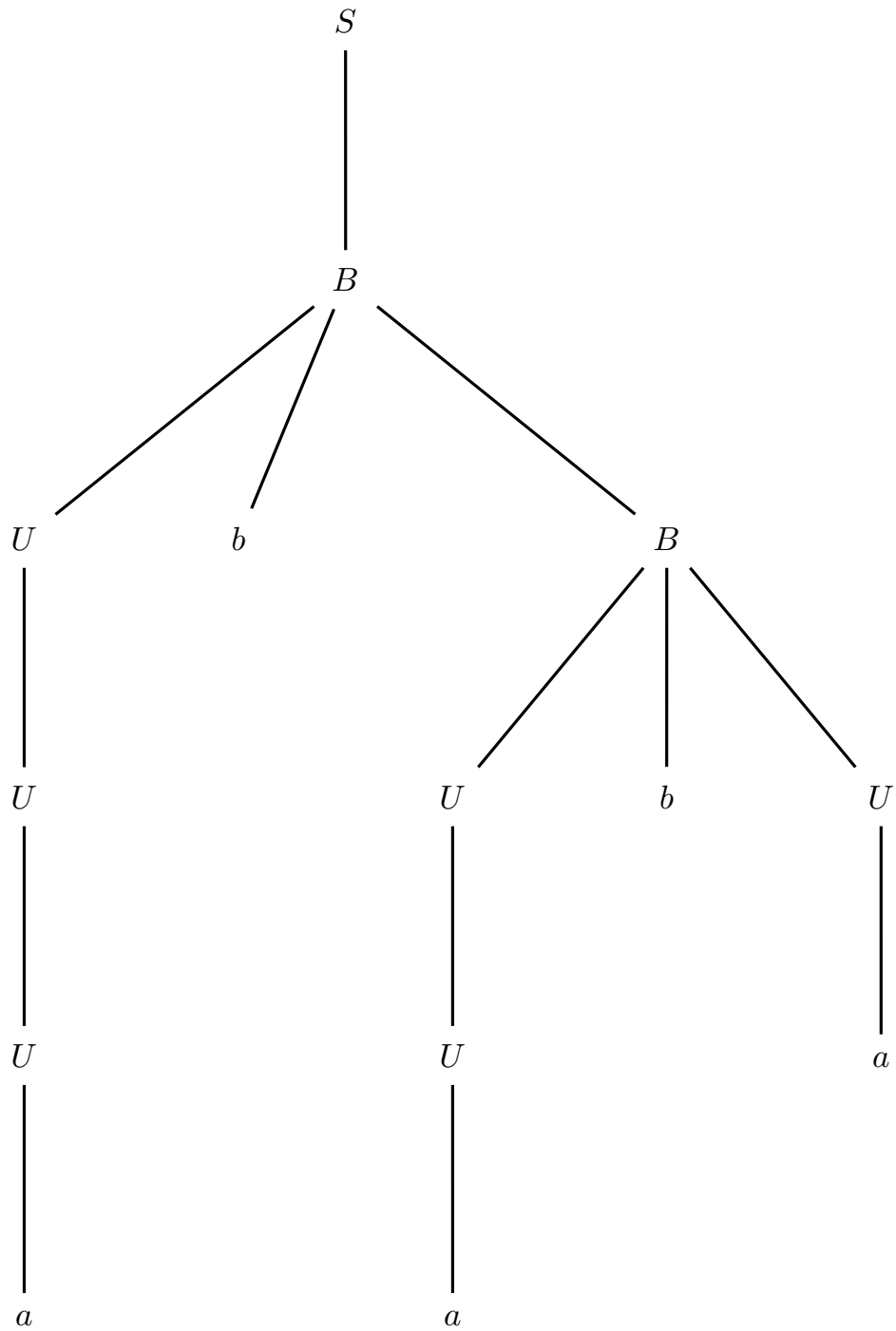
Answer the following questions (use the tables / drawings / spaces on the next pages):

- Write an attribute grammar that computes, in the tree root, a string constituting the linear (parenthesized) representation of the *condensed* syntax tree. Use only the attribute s specified, and, to compute it, use the concatenation operator “.”. Make sure that the grammar is one-sweep and explain why.
- Write an attribute grammar that computes, in the tree root, an *extended* linear representation of the condensed syntax tree, where an integer is inserted next to each substring “[a]”. Such an integer indicates the number of consecutive nodes U occurring in the non-branching path immediately above the corresponding leaf node “ a ”. The extended linear representation of the sample tree will thus be “[[a]3 b [[a]2 b [a]1]]”. Use both the attributes s and n specified. To insert such integers into the extended linear representation, you may use the function $itos(i)$, which converts an integer i into its string representation.
- (optional) Decorate the sample tree with the values of attributes s and n .

attribute specifications – questions (a) and (b)

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
<i>s</i>	left	string	<i>S, B, U</i>	the linear representation of the condensed (sub)tree rooted at the current node
<i>n</i>	left	integer ≥ 1	<i>U</i>	the number of consecutive nodes <i>U</i> in a vertical downward path starting from the current node (included); for instance, for the three branches of nodes <i>U</i> in the sample tree (from left to right), attribute <i>n</i> is equal to 3, 2 and 1

question (c) – decorate here the sample syntax tree with attributes s and n



Solution

- (a) The idea is to encode the operand a between square brackets at the leaf level (rule 4), to propagate the partial linear representations upwards (rule 3), and to concatenate them whenever an operator node b is encountered (rules 5 and 6). The final result emerges on top (rules 1 and 2). This is similar to how an arithmetic expression is computed bottom-up. Here is the attribute grammar, quite simple:

#	<i>syntax</i>	<i>semantics</i> – question (a)
1:	$S_0 \rightarrow B_1$	$s_0 := s_1$
2:	$S_0 \rightarrow U_1$	$s_0 := s_1$
3:	$U_0 \rightarrow U_1$	$s_0 := s_1$
4:	$U_0 \rightarrow a$	$s_0 := [\cdot a \cdot]$
5:	$B_0 \rightarrow U_1 b B_2$	$s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$
6:	$B_0 \rightarrow U_1 b U_2$	$s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$

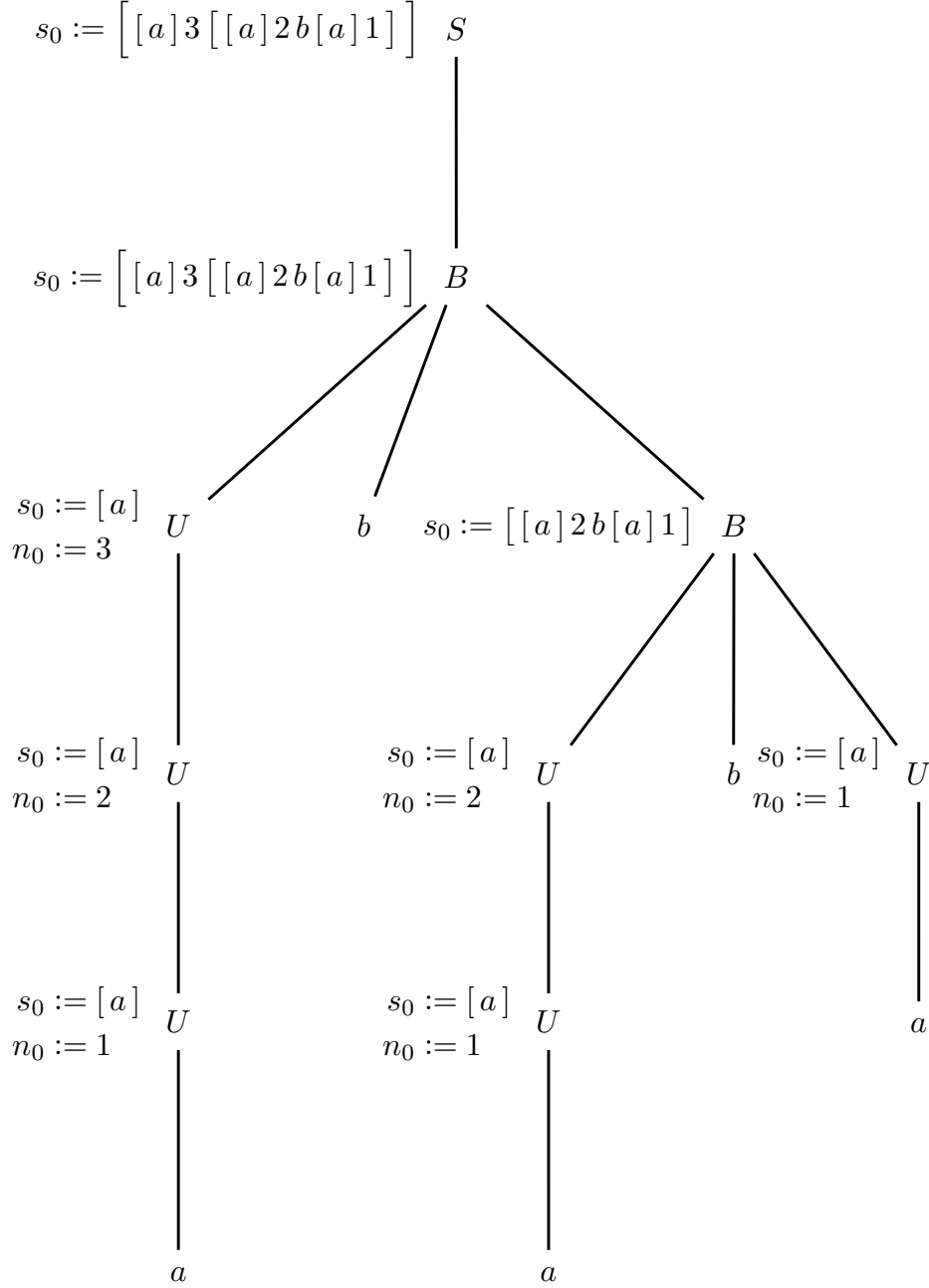
The attribute grammar does not have any circular dependence, because it uses only one attribute, thus it is correct. Furthermore, the grammar is one-sweep because it is purely synthesized.

- (b) The idea is similar to the one before, but additionally the depth is computed bottom-up (rules 3 and 4) and is inserted next to the corresponding operand a whenever an operator node b is encountered (rules 5 and 6) or the root is reached (rule 2). Here is the attribute grammar, almost as simple as before:

#	<i>syntax</i>	<i>semantics</i> – question (b)
1:	$S_0 \rightarrow B_1$	$s_0 := s_1$
2:	$S_0 \rightarrow U_1$	$s_0 := s_1 \cdot itos(n_1)$
3:	$U_0 \rightarrow U_1$	$s_0 := s_1$ $n_0 := n_1 + 1$
4:	$U_0 \rightarrow a$	$s_0 := [\cdot a \cdot]$ $n_0 := 1$
5:	$B_0 \rightarrow U_1 b B_2$	$s_0 := [\cdot s_1 \cdot itos(n_1) \cdot b \cdot s_2 \cdot]$
6:	$B_0 \rightarrow U_1 b U_2$	$s_0 := [\cdot s_1 \cdot itos(n_1) \cdot b \cdot s_2 \cdot itos(n_2) \cdot]$

The decorated tree of question (c) provides a computation example. The attribute grammar does not have any circular dependence, because the only interaction between attributes s and n is in the rules 2, 5 and 6, and clearly it is not circular. Thus the grammar is correct. Furthermore, it is one-sweep because it is purely synthesized.

(c) Here is the tree decoration for question (b), quite immediate:

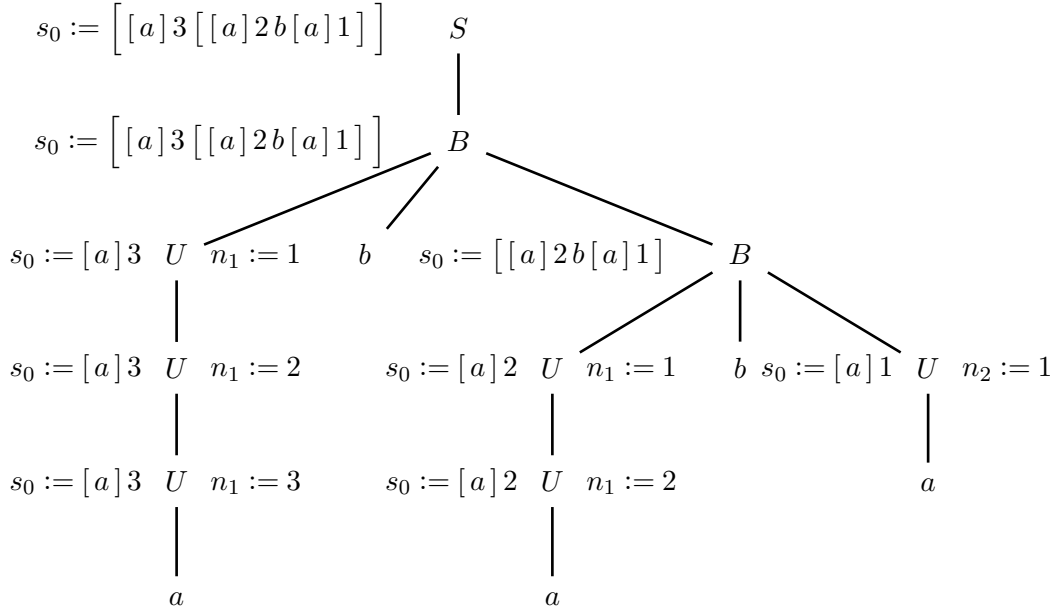


Both the left attributes s and n are initialized and propagated upwards together. The integer is inserted next to the operand when the operator b is also included. According to the usual conventions, both attributes, which are synthesized (left), are written on the left side of the node they refer to.

As a further development, notice that question (b) could be as well solved by means of a *right* attribute n , instead of left. Here is the modified solution:

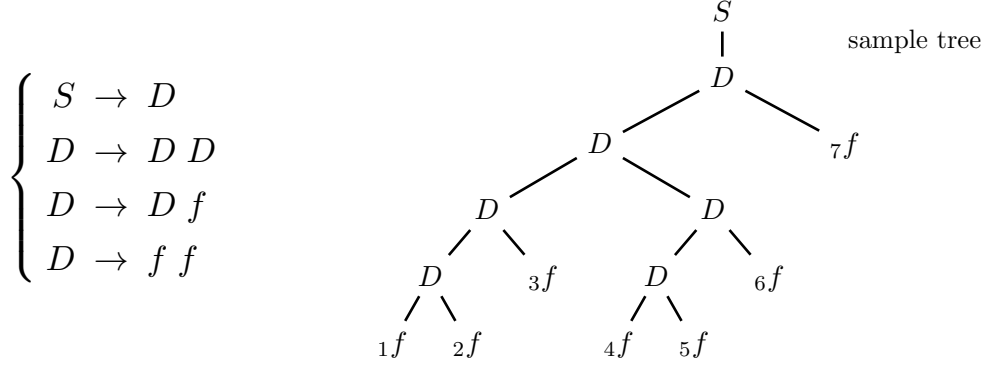
#	<i>syntax</i>	<i>semantics</i> – question (b) with attribute n right
1:	$S_0 \rightarrow B_1$	$s_0 := s_1$
2:	$S_0 \rightarrow U_1$	$n_1 := 1$ $s_0 := s_1$
3:	$U_0 \rightarrow U_1$	$n_1 := n_0 + 1$ $s_0 := s_1$
4:	$U_0 \rightarrow a$	$s_0 := [\cdot a \cdot] \cdot itos(n_0)$
5:	$B_0 \rightarrow U_1 b B_2$	$n_1 := 1$ $s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$
6:	$B_0 \rightarrow U_1 b U_2$	$n_1, n_2 := 1$ $s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$

The difference with respect to grammar (b) is that the depth is inserted next to the related operand a in the rule (4), immediately when the operand itself is encoded. The attribute grammar does not have any circular dependence, because the only interaction between attributes s and n is in the rule 4, and clearly it is not circular. Thus the grammar is correct. Furthermore, the grammar is one-sweep, too, because the right attribute n depends only on itself, and the left attribute s depends on itself and on the right attribute n , but in the same node (rule 4). Thus the one-sweep condition is satisfied. The decorated tree becomes as follows (now attribute n , right, is written on the right of the node it refers to):



The final result emerges at the root and is the same as before, but the integer is first computed downwards, then (at the tree bottom) it is immediately inserted next to the operand, and finally it is propagated upwards encoded together with the operand.

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):



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 $5f$).

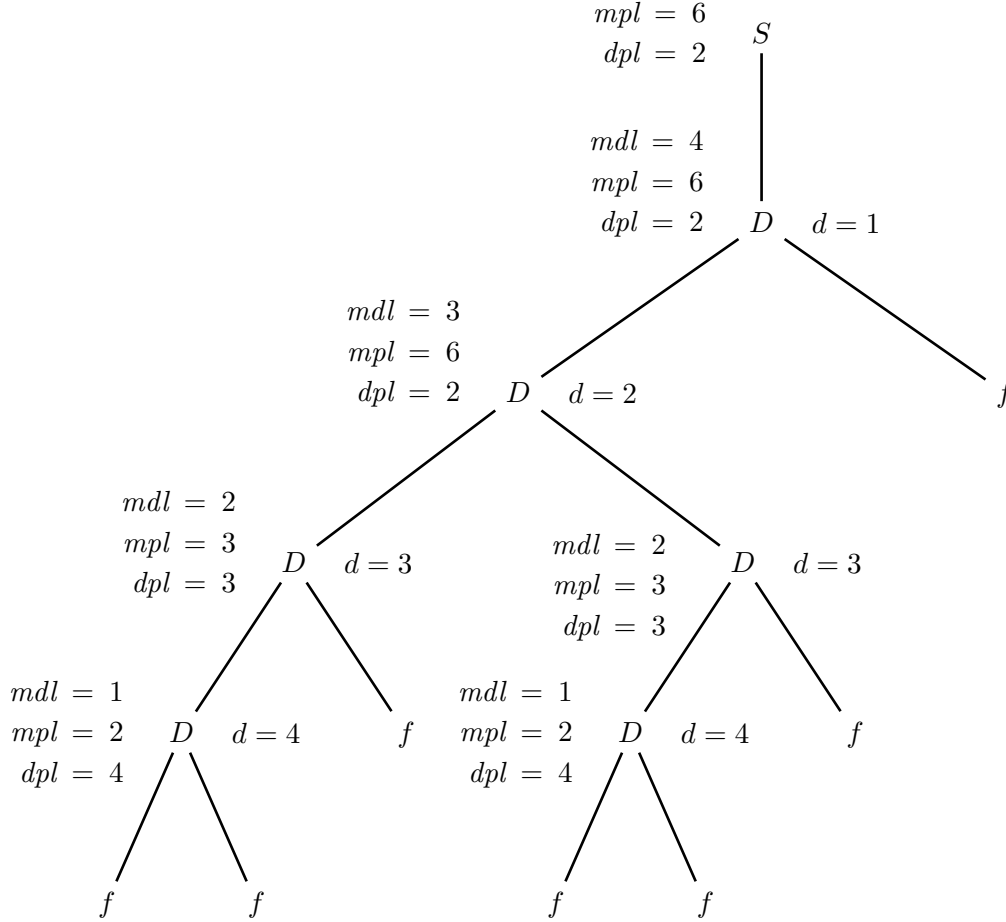
We intend to design an attribute grammar that supports the computation of the maximal distance between any two tree leaf nodes. In the above sample tree the maximal distance is 6, because for instance $dist(2f, 4f) = 6$.

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:

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
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:

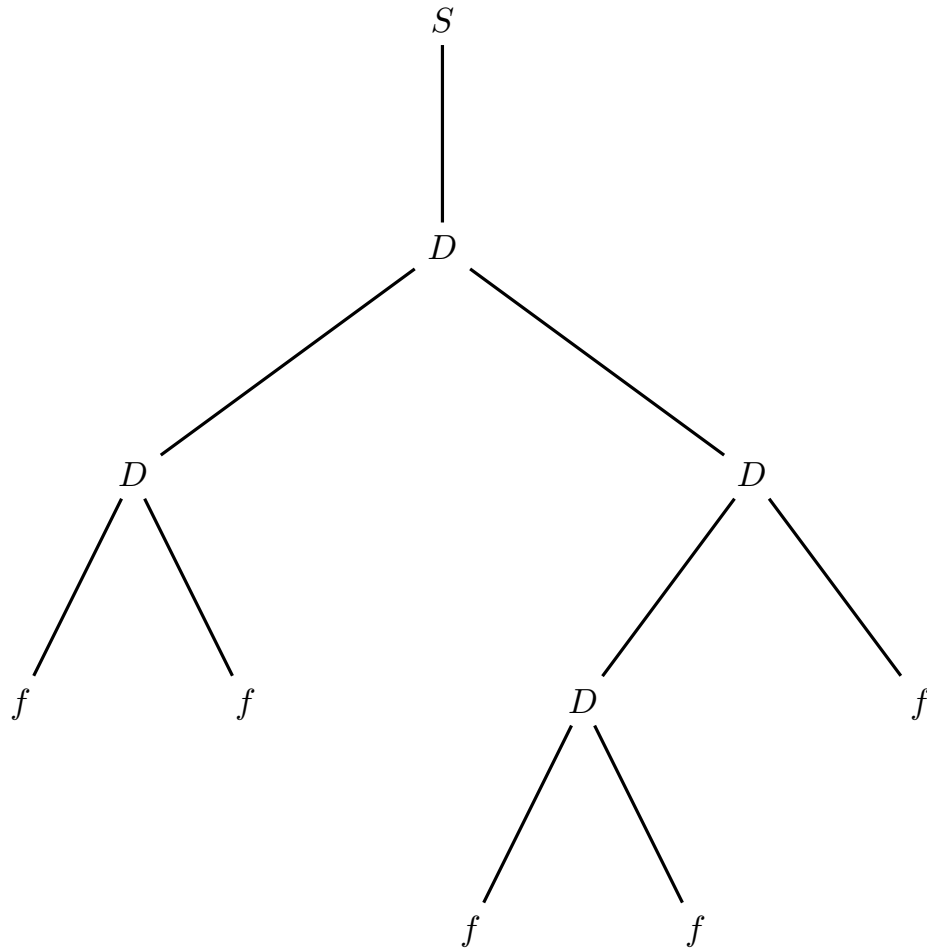


Answer the following questions (use the tables / drawings / spaces on the next pages):

- Write the semantic rules to compute attribute d .
- Write the semantic rules to compute attribute mdl . Decorate the next tree with the attribute value, according to the provided semantic rules.
- Write the semantic rules to compute attribute mpl . Decorate the next tree with the attribute value, according to the provided semantic rules.
- (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 $\max(v_1, \dots, v_n)$, for any $n \geq 2$, to denote the maximum of a set of values v_1, \dots, v_n .

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



semantic rules for attribute d – question (a)

#	<i>syntax</i>	<i>semantics</i>
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)

#	<i>syntax</i>	<i>semantics</i>
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)

#	<i>syntax</i>	<i>semantics</i>
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)

#	<i>syntax</i>	<i>semantics</i>
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 :

#	<i>syntax</i>	<i>semantics</i>
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 :

#	<i>syntax</i>	<i>semantics</i>
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 :

#	<i>syntax</i>	<i>semantics</i>
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:	$D_0 \rightarrow f f$	$mpl_0 := 2$

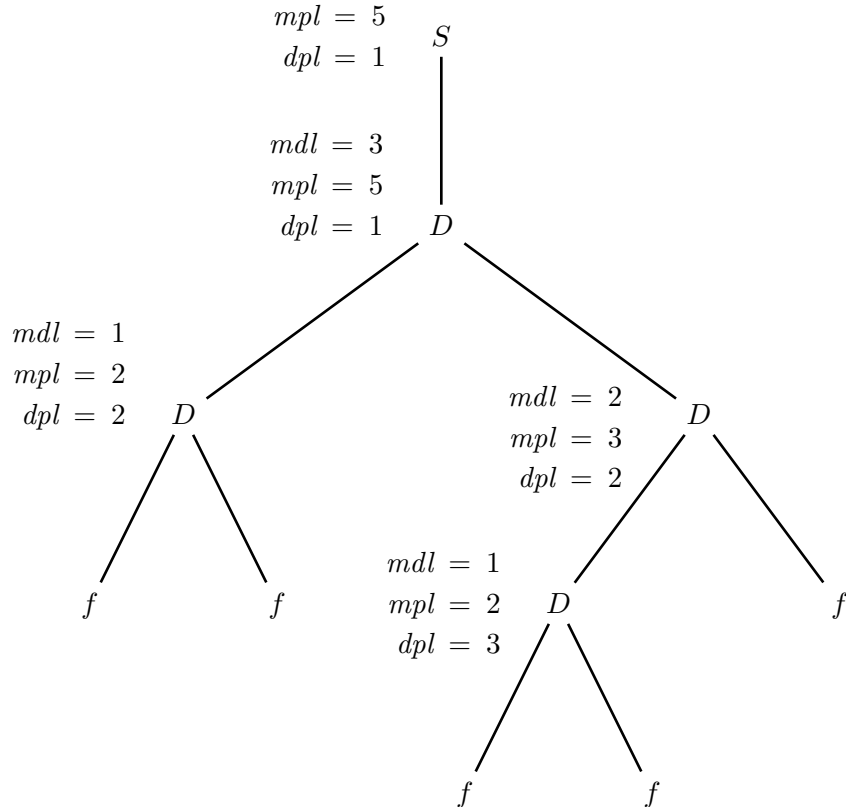
In the rule 2, it is taken the maximum of the leaf-to-leaf distances in the two subtrees and of the distance sum plus two for the two leaves to belong to different subtrees. In the rule 3, there is only one subtree, thus the maximum is simplified. Rule 4 initializes.

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

#	<i>syntax</i>	<i>semantics</i>
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



2. We intend to compute a semantic translation τ of an arithmetic expression from the infix form into a partial postfix form, by using an attribute grammar. Below it is reported the syntax of the (infix) expression to be translated (axiom S), which is right-associative, where terminal a represents any variable or constant:

$$\left\{ \begin{array}{l} 1: S \rightarrow E \\ 2: E \rightarrow T + E \\ 3: E \rightarrow T \\ 4: T \rightarrow (E) \\ 5: T \rightarrow a \end{array} \right.$$

Translation τ keeps all the parentheses as they are and works on the rest as follows:

- it leaves in *infix form* every (sub)expression (immediately nested in a parenthesis pair or at the outermost level) that has an *odd* number of terms (1, 3, ...)
- it puts in *postfix form* every (sub)expression (immediately nested in a parenthesis pair or at the outermost level) that has an *even* number of terms (2, 4, ...)

Here are two sample translations of increasing complexity:

$$\tau \left(\overset{o}{\boxed{a}} +_1 \left(\overset{e}{\boxed{\overset{o}{\boxed{a}} +_2 \overset{e}{\boxed{a}}}} \right) +_3 \overset{o}{\boxed{a}} \right) = a +_1 (a a +_2) +_3 a$$

$$\tau \left(a +_1 \left(a +_2 \left(a +_3 a \right) +_4 a \right) \right) = a \left(a +_2 \left(a a +_3 \right) +_4 a \right) +_1$$

The operator “+” is numbered to visualize how translation τ works. The position of each term is tagged odd or even (o or e) in each (sub)expression (top sample only) for the same purpose. Such annotations are not part of the attribute grammar.

The translated expression is represented by an attribute τ of type string in the root node of the syntax tree. On the next page, there is a list of grammar attributes (including τ) to use. Do not add or modify any attribute. There is also a decorated tree that illustrates how to use such attributes to translate a string (bottom sample).

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

- Write the semantic functions for computing attribute *odd*.
- Write the semantic functions for computing attribute *totOdd*.
- Write the semantic functions for computing attribute τ .
- Write the translation of this expression (different from the two samples above):

$$(a + a + a) + a$$

and decorate its tree with all the appropriate attribute values in each node.

- (optional) Draw the dependency graph for each grammar rule.

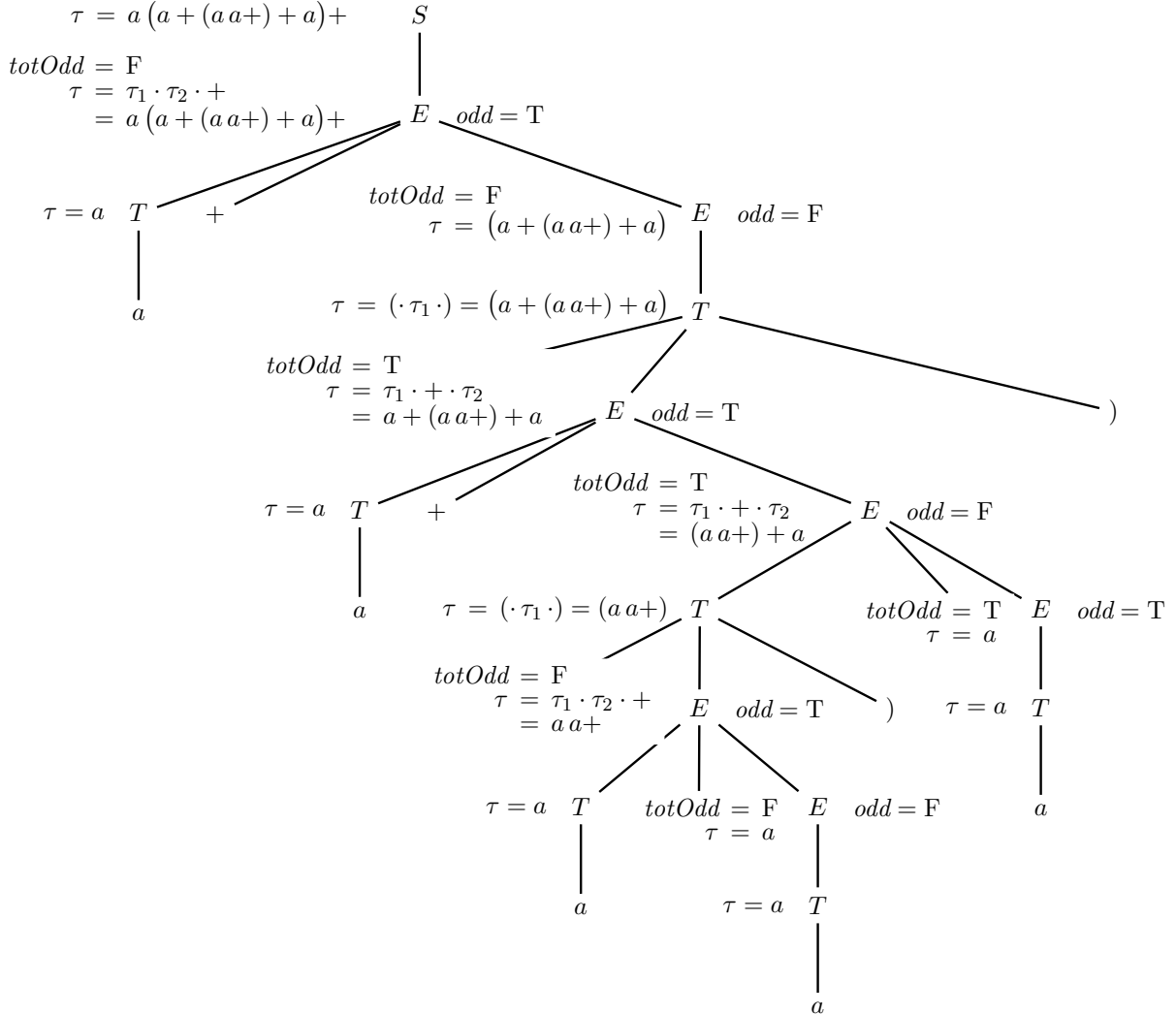
attribute list and sample decorated tree illustrating the attribute use

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
<i>odd</i>	right	boolean	<i>E</i>	true if the node occupies an odd position (first, third, etc) in the current sequence of nodes of type <i>E</i>
<i>totOdd</i>	left	boolean	<i>E</i>	true if the total number of nodes of type <i>E</i> in the current sequence is odd
τ	left	string	<i>T, E, S</i>	in the root, the translation of the entire tree; in the other nodes, suitable fragments of the overall translation

How to use the attributes to translate the bottom sample string:

$$\tau \left(a + (a + (a + a) + a) \right) = a (a + (a a +) + a) +$$

is shown by this tree (some leaves are masked by the attributes):



tables for answering questions (a-b-c)
only the rules relevant for each attribute are reported

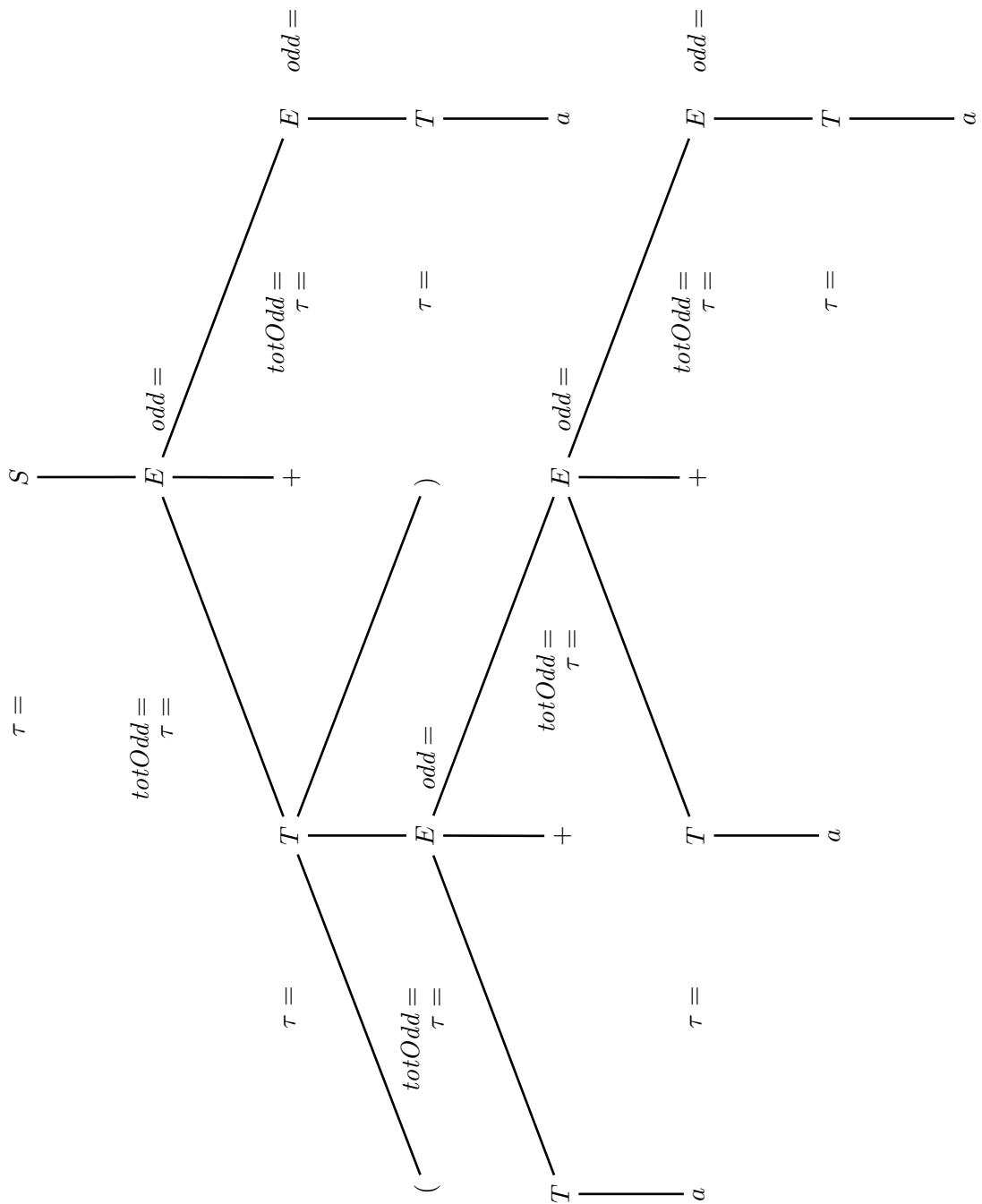
#	<i>syntax</i>	<i>semantics of the <u>right</u> attribute “odd” – question (a)</i>
1:	$S_0 \rightarrow E_1$	
2:	$E_0 \rightarrow T_1 + E_2$	
4:	$T_0 \rightarrow (E_1)$	
#	<i>syntax</i>	<i>semantics of the <u>left</u> attribute “totOdd” – question (b)</i>
2:	$E_0 \rightarrow T_1 + E_2$	
3:	$E_0 \rightarrow T_1$	
#	<i>syntax</i>	<i>semantics of the <u>left</u> attribute “τ” – question (c)</i>
1:	$S_0 \rightarrow E_1$	
2:	$E_0 \rightarrow T_1 + E_2$	
3:	$E_0 \rightarrow T_1$	
4:	$T_0 \rightarrow (E_1)$	
5:	$T_0 \rightarrow a$	

space and tree for answering question (d)

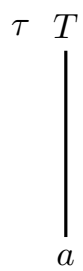
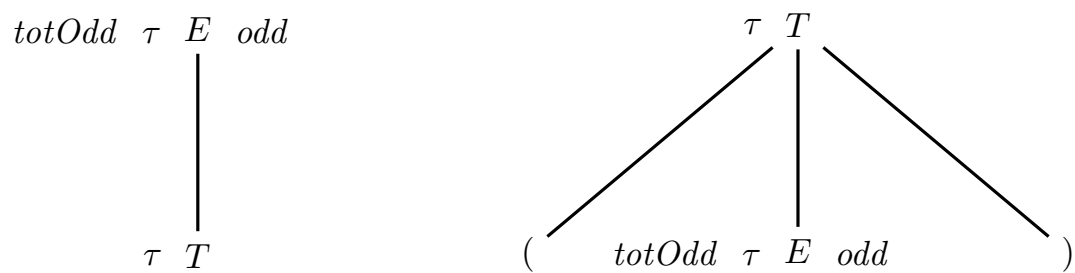
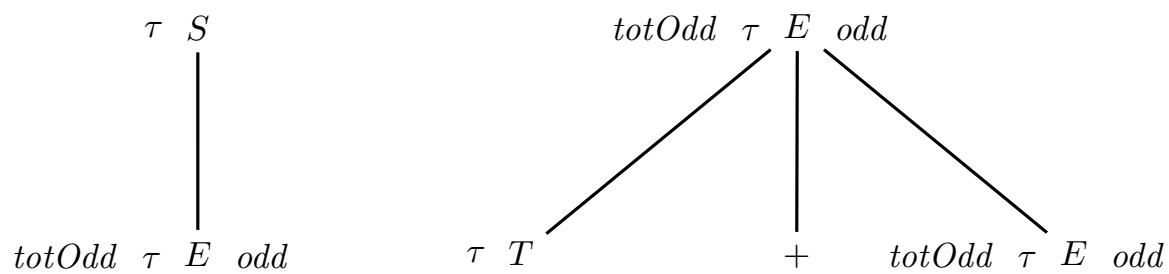
Write here the translation of the proposed string “ $(a + a + a) + a$ ”:

$$\tau \left((a + a + a) + a \right) =$$

Decorate with all the attribute values the syntax tree of such a string:



elementary grammar trees for answering question (e)
draw here the dependency graph of each grammar rule



Solution

To help understanding, some elementary cases of τ are: $\tau(a) = a$, $\tau(a + b) = a b +$, $\tau(a + b + c) = a + b + c$ and $\tau(a + b + c + d) = a b c d + + +$, with named variables.

- (a) Here are the semantic functions for the right (inherited) attribute *odd*:

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow E_1$	$odd_1 := T$
2:	$E_0 \rightarrow T_1 + E_2$	$odd_2 := \underbrace{\neg odd_0}_{\text{toggled}}$
4:	$T_0 \rightarrow (E_1)$	$odd_1 := T$

The attribute is initialized to *true* at the first term of a (sub)expression (rules 1-4), and is toggled when moving on to the next term (rule 3), i.e., when moving down through the tree, until the last term of the (sub)expression is reached.

- (b) Here are the semantic functions for the left (synthesized) attribute *totOdd*:

#	<i>syntax</i>	<i>semantics</i>
2:	$E_0 \rightarrow T_1 + E_2$	$totOdd_0 := \underbrace{totOdd_2}_{\text{propagated}}$
3:	$E_0 \rightarrow T_1$	$totOdd_0 := \underbrace{odd_0}_{\text{init}}$

The attribute is initialized to the value assigned to attribute *odd* at the last term of a (sub)expression (rule 3), and is propagated unchanged when moving back to the previous term (rule 2), i.e., when moving up through the tree, until the first term of the (sub)expression is reached.

- (c) Here are the semantic functions for the left (synthesized) attribute τ :

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow E_1$	$\tau_0 := \tau_1$
2:	$E_0 \rightarrow T_1 + E_2$	$\tau_0 := \text{if } \underbrace{totOdd_0}_{\text{select form}} \text{ then } \underbrace{\tau_1 \cdot + \cdot \tau_2}_{\text{infix form}} \text{ else } \underbrace{\tau_1 \cdot \tau_2 \cdot +}_{\text{postfix form}} \text{ end}$
3:	$E_0 \rightarrow T_1$	$\tau_0 := \tau_1$
4:	$T_0 \rightarrow (E_1)$	$\tau_0 := \underbrace{(\cdot \tau_1 \cdot)}_{\text{unchanged}}$
5:	$T_0 \rightarrow a$	$\tau_0 := a$

The attribute will stepwise accumulate the translation. It grows by concatenating a term and (the rest of) a subexpression when moving up through the tree, and by putting them in the infix or postfix form according to attribute *totOdd* (rule 2), while the parentheses do not change (rule 4). It is initialized to an elementary term (rule 5) and, as for the rest, it is propagated upwards (rules 1-3). In the root, it will have thus taken its final value, i.e., the complete translation.

- $$\tau\left((a + a + a) + a\right) = (a + a + a)a +$$

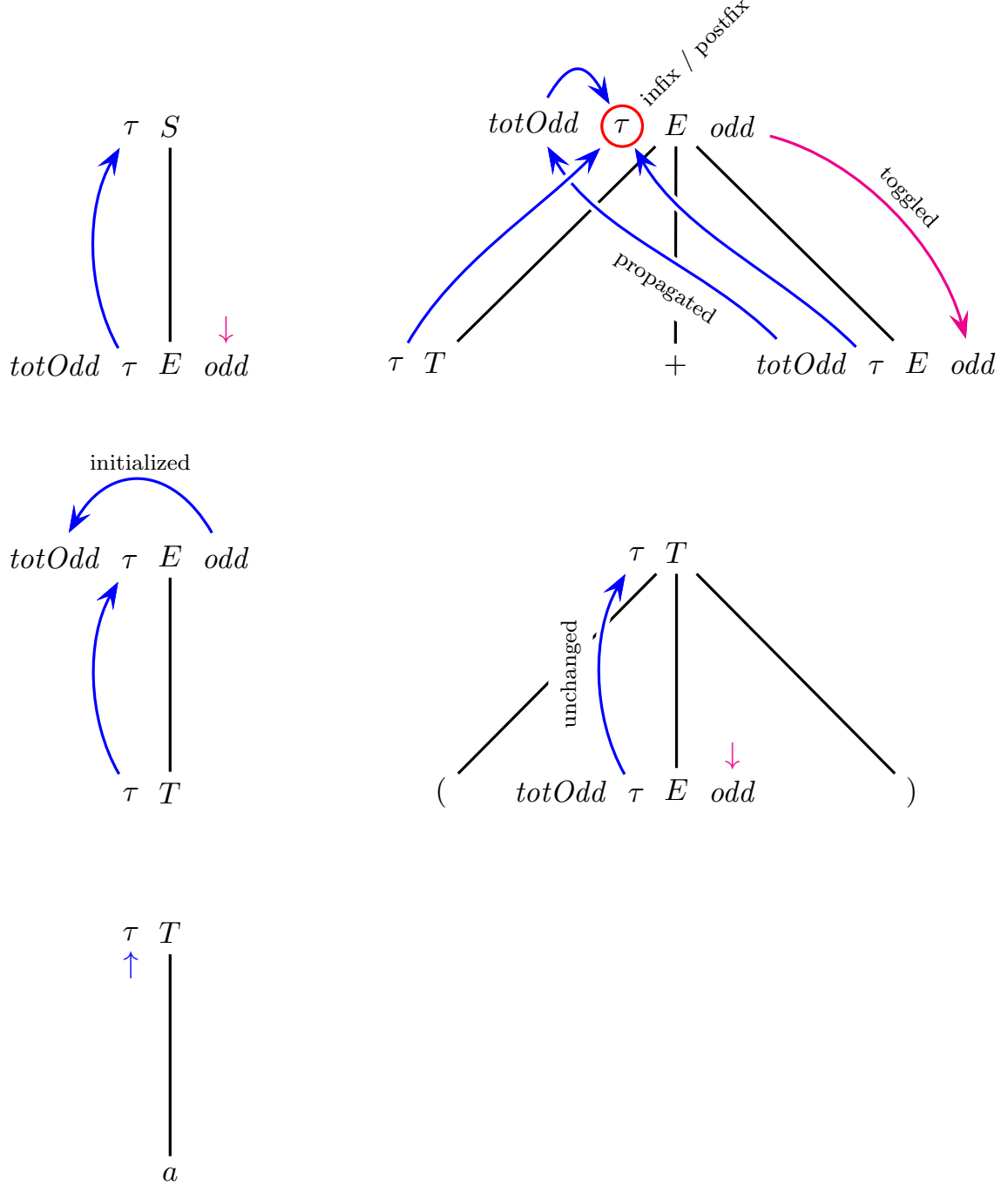
The graph in Figure 10 consists of 12 nodes and 18 directed edges. The nodes are labeled as follows:

- Top-left: $odd = T, totOdd = F$
- Top-middle: $odd = T, totOdd = T$
- Top-right: $odd = F, totOdd = F$
- Middle-left: $odd = T, totOdd = T$
- Middle-middle: $odd = F, totOdd = T$
- Middle-right: $odd = T, totOdd = F$
- Bottom-left: $odd = T, totOdd = T$
- Bottom-middle: $odd = F, totOdd = T$
- Bottom-right: $odd = T, totOdd = F$
- Far-left: $odd = T, totOdd = F$
- Far-middle: $odd = T, totOdd = T$
- Far-right: $odd = F, totOdd = F$

The edges are labeled with logical expressions involving 'a' and 'tau':

- From Top-left to Top-middle: a
- From Top-left to Top-right: $\tau = a$
- From Top-middle to Top-right: $\tau = a$
- From Middle-left to Middle-middle: a
- From Middle-left to Middle-right: $\tau = a$
- From Middle-middle to Middle-right: $\tau = a$
- From Bottom-left to Bottom-middle: a
- From Bottom-left to Bottom-right: $\tau = a$
- From Bottom-middle to Bottom-right: $\tau = a$
- From Far-left to Far-middle: a
- From Far-left to Far-right: $\tau = a$
- From Far-middle to Far-right: $\tau = a$
- From Top-left to Middle-left: a
- From Top-middle to Middle-middle: a
- From Top-right to Middle-right: a
- From Middle-left to Bottom-left: a
- From Middle-middle to Bottom-middle: a
- From Middle-right to Bottom-right: a

(e) Here are the dependency graphs of all the rules of the attribute grammar:



Colors magenta and blue represent inheritance and synthesis, respectively. The short vertical arrows represent initialization. The crucial point is the computation of attribute τ (encircled in red) in the parent node E of the elementary tree of rule $E \rightarrow T + E$, where the contributions of attributes τ_1 , τ_2 and $totOdd_2$ are collected from the child nodes and either the infix or the postfix form is built. These elementary graphs are acyclic, and it is impossible to form a cycle by composing them on a tree. Thus the attribute grammar is (reasonably) correct.

PART 4 - Language translation and semantic analysis

The following **source grammar** G_S (axiom S):

$$S \rightarrow XS$$

$$S \rightarrow d$$

$$X \rightarrow aX$$

$$X \rightarrow bX$$

$$X \rightarrow c$$

generates a language of **two-level lists**. The top-level list is terminated by a letter d , while each bottom-level list has letters a and b as its elements, and is terminated by a letter c . The top level list may be empty, but its terminator is always present; the same applies to the bottom-level lists.

With reference to the **source language** defined by the grammar G_S above, a **translation** τ is defined as follows:

- in each bottom-level list, the elements a and b are ordered, first a then b
- each bottom-level list is initiated by a letter c , and is not terminated
- the top-level list is unchanged

Example: $\tau(abbcbaacabcbbaacd) = cabbcabcbcaabd$

Write a **target grammar** G that, combined with G_S , provides a **syntactic translation scheme** that defines translation τ .

36.

ANS:

target grammar G_t

$$S \rightarrow cXS$$

$$S \rightarrow d$$

$$X \rightarrow aX$$

$$X \rightarrow Xb$$

$$X \rightarrow \epsilon$$

or scheme G_τ

$$S \rightarrow \epsilon/cXS$$

$$S \rightarrow d/d$$

$$X \rightarrow a/aX\epsilon/\epsilon$$

$$X \rightarrow b/\epsilon X\epsilon/b$$

$$X \rightarrow c/\epsilon$$

37. Is the translation τ defined before **rational**, i.e., regular, definable through a **rational translation** expression ?
- Select one:
- ☐ True
- ☐ False
- ANS: FALSE

38. Shortly **justify** your answer to the previous question, whether translation τ is **rational** or not.

Scheme G_τ clearly shows that, for ordering the elements a and b while keeping their number in each bottom-level list, a recursive self-embedding rule, i.e., $X \rightarrow b/\epsilon X \epsilon/b$, is necessary, which cannot be emulated by a rational expression.

ANS:

39. Is the translation τ defined before **deterministic**, i.e., the proposed scheme is of type **(E)LL** or **(E)LR** for some k ?
- Select one:
- ☐ True
- ☐ False
- ANS: TRUE

40. Shortly **justify** your answer to the previous question, whether translation τ is **deterministic** or not.

ANS:

The source grammar G_s is clearly of type **LL** for $k = 1$, since each alternative rule for nonterminal X is immediately initiated by a different terminal symbol, while the alternative rules for the axiom S have guide sets $\{a, b, c\}$ and $\{d\}$, respectively, which are disjoint. Furthermore, translation τ is a function (one-valued) on strings. Therefore the scheme can be computed by a recursive-descent syntactic translator and the translation is deterministic.

41. Now consider the **inverse translation** τ^{-1} . Is it **deterministic** ? Shortly **justify** your answer.

The inverse translation is definitely **not deterministic**. In fact, it is not a function either, since it is multi-valued, e.g., $\tau^{-1}(cabd) = \{abcd, bacd\}$. Therefore it cannot be deterministic.

ANS:

Now, again with reference to the **source language** defined by the grammar G_S above, a **new translation τ'** is defined as follows:

- a bottom-level list is left unchanged (elements **a** and **b**, and terminator **c**), provided it contains the same numbers of elements
- otherwise, a bottom-level list is completely removed (including its terminator **c**)
- the top-level list is unchanged (except for the removed bottom-level lists)

Example: $\tau'(abbcbaacbaacd) = bacabcd$

Write a **attribute grammar G_A** that, basing on the syntactic support G_S , defines such a new **semantic translation τ'** .

To this end, use the following **four attributes**:

- **na** and **nb** are integer **right** attributes associated with **X**, and respectively count how many elements **a** and **b** a bottom-level list contains
- **e** is a boolean **left** attribute associated with **X**, and tells if a bottom-level list contains the same numbers of elements **a** and **b**
- **t** is a string **left** attribute associated with **X** and **S**, and provides the final translation **τ'** in the root node (axiom **S**) of the syntax tree

Please **write**, in the template provided below, the **equations** (semantic functions) for the attributes associated with each rule of the attribute grammar G_A .

42.

- | | |
|------------------------------|-------|
| 1: $S_0 \rightarrow X_1 S_2$ | |
| 2: $S_0 \rightarrow d$ | |
| 3: $X_0 \rightarrow a X_1$ | |
| 4: $X_0 \rightarrow b X_1$ | |
| 5: $X_0 \rightarrow c$ | |

ANS:

1: $S_0 \rightarrow X_1 S_2$	$na_1, nb_1 := 0$	$t_0 := t_1 t_2$	
2: $S_0 \rightarrow d$	$t_0 := "d"$		
3: $X_0 \rightarrow a X_1$	$na_1 := na_0 + 1$	$e_0 := e_1$	<u>if</u> $e_0 == \text{true}$ <u>then</u> $t_0 := "a" t_1$ <u>else</u> $t_0 := \varepsilon$
4: $X_0 \rightarrow b X_1$	$nb_1 := nb_0 + 1$	$e_0 := e_1$	<u>if</u> $e_0 == \text{true}$ <u>then</u> $t_0 := "b" t_1$ <u>else</u> $t_0 := \varepsilon$
5: $X_0 \rightarrow c$		$e_0 := (na_0 == nb_0)$	<u>if</u> $e_0 == \text{true}$ <u>then</u> $t_0 := "c"$ <u>else</u> $t_0 := \varepsilon$

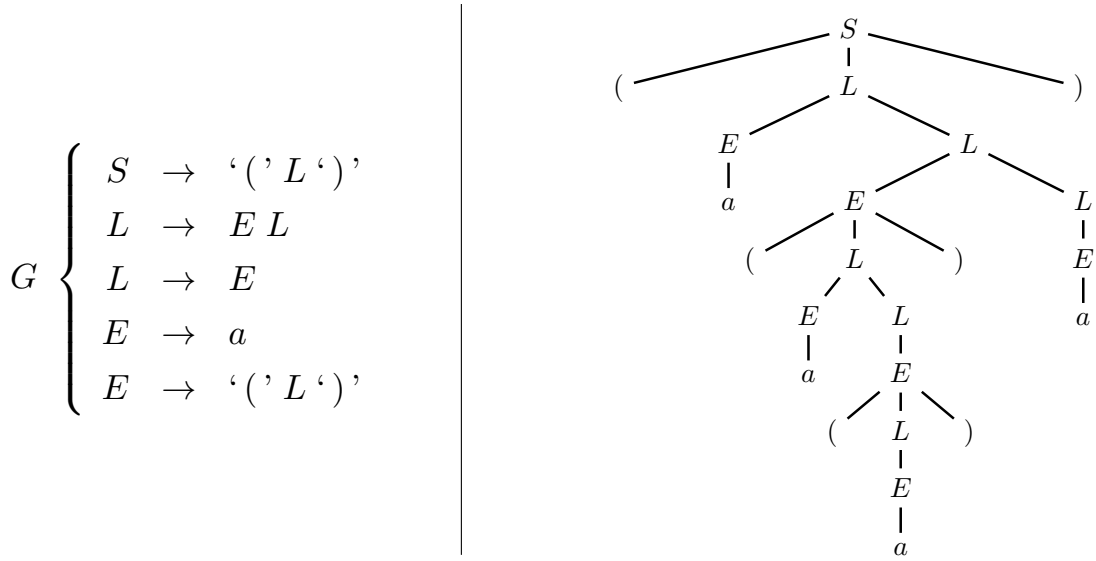
2. A grammar G generates expressions that consist of a non-empty list delimited by round brackets '(' and ')'. Such a list contains one or more elements of these two types: an atom represented by terminal a or a non-empty sublist delimited by brackets (as before); and so on recursively down to an arbitrary sublist nesting depth.

The nesting level of an element (atom or sublist) is the number of bracket pairs that enclose the element. Here is a sample expression e :

$$e = \left(a \left(a \left(a \right) \right) a \right)$$

The expression e has three elements at level 1, i.e., the 1st and 4th atom a , and the sublist $(a(a))$; it has two at level 2, i.e., the 2nd atom a and the sublist (a) ; and only one at level 3, i.e., the 3rd atom a . It has a *total* number of elements $3+2+1=6$. The sublist $(a(a))$ has three elements in total: two atoms a and the sublist (a) .

Here is the grammar G (axiom S) of the expressions, and the syntax tree of e :



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

- Write an attribute grammar G_a based on the syntactic support G . Grammar G_a computes an integer attribute $n \geq 1$ that expresses the total number of elements (atoms and sublists) in the expression (i.e., the number of nonterminals E). In the tree root of e it holds $n = 6$. Decorate the tree of e with the values of n .
- By means of an integer attribute $d \geq 1$, associate to each element (atom or sublist) the respective nesting level. Decorate the tree of e with the values of d .
- (optional) By means of a boolean attribute v , and possibly of more ones if they help, verify if in the expression there is a proper sublist (i.e., not coincident with the entire expression) that has a total number of elements equal to its nesting level. If there is, in the tree root it holds $v = T$, otherwise it holds $v = F$. The expression e has $v = F$ in the root. Instead, the expression $e' = (a(a)a)$ (different from e) has $v = T$, because its proper sublist (a) has only one element and is at level 1. Decorate the tree of e with the attribute values.

attributes already assigned to be used for grammar G_a

write the field *type*

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
	n	integer ≥ 1	S, L, E	total number of elements (atoms and sublists)
	d	integer ≥ 1	L, E	nesting level of an element (atom or sublist)
	v	boolean	S, L, E	this predicate is true if and only if in the expression there is a proper sublist (i.e., not coincident with the entire expression) that has a total number of elements (atoms and sublists) equal to its nesting level

possible auxiliary attributes to be added for question c (if they help)

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>

syntax

semantics - question *a*

$$\mathbf{n}_0 = \mathbf{n}_1$$

$$1: S_0 \rightarrow (L_1)$$

$$2: L_0 \rightarrow E_1 L_2$$

$$3: L_0 \rightarrow E_1$$

$$4: E_0 \rightarrow a$$

$$5: E_0 \rightarrow (L_1)$$

syntax

semantics - question *b*

d₁ = 1

1: $S_0 \rightarrow (L_1)$

2: $L_0 \rightarrow E_1 L_2$

3: $L_0 \rightarrow E_1$

4: $E_0 \rightarrow a$

5: $E_0 \rightarrow (L_1)$

syntax

semantics - question *c*

$$\mathbf{v}_0 = \mathbf{v}_1$$

$$1: S_0 \rightarrow (L_1)$$

$$2: L_0 \rightarrow E_1 L_2$$

$$3: L_0 \rightarrow E_1$$

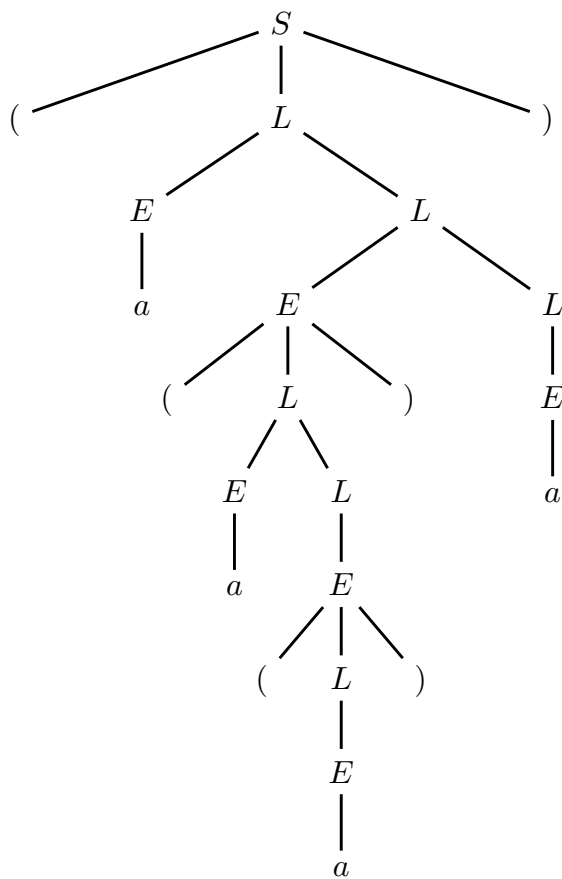
$$4: E_0 \rightarrow a$$

$$5: E_0 \rightarrow (L_1)$$

[illegible]

question *a*

question *b*



question *c*

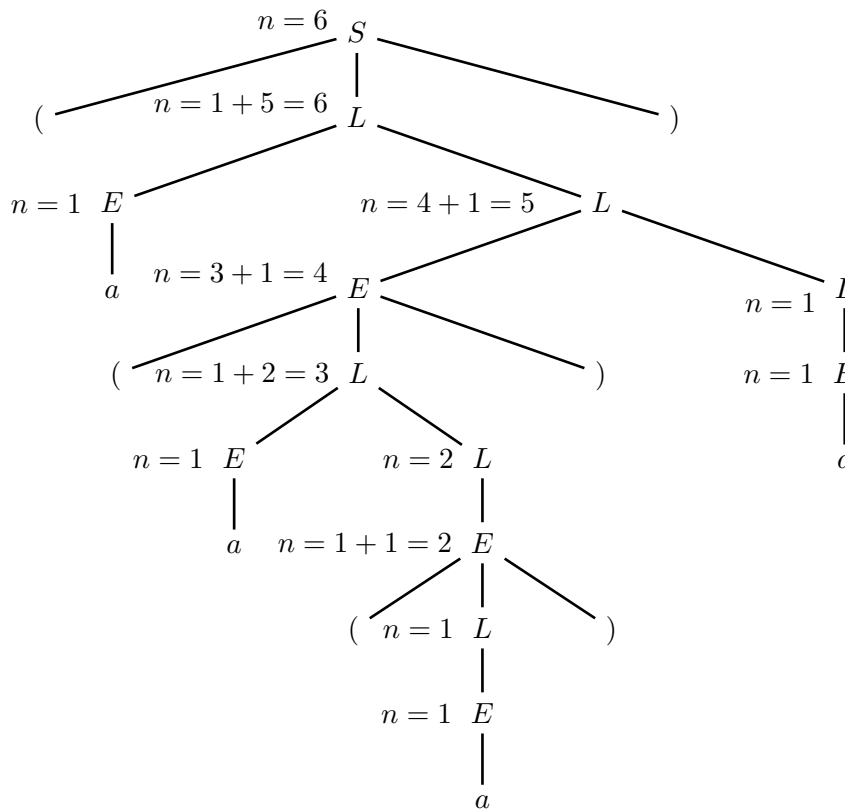
Solution

(a) Here are the left attribute n and its semantic functions:

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
left	n	integer ≥ 1	S, L, E	total number of elements (atoms and sublists)

#	<i>syntax</i>	<i>semantics</i> - question <i>a</i>
1:	$S_0 \rightarrow (L_1)$	$n_0 = n_1$
2:	$L_0 \rightarrow E_1 L_2$	$n_0 = n_1 + n_2$
3:	$L_0 \rightarrow E_1$	$n_0 = n_1$
4:	$E_0 \rightarrow a$	$n_0 = 1$
5:	$E_0 \rightarrow (L_1)$	$n_0 = n_1 + 1$

And here is the syntax tree decorated with the values of the left attribute n :

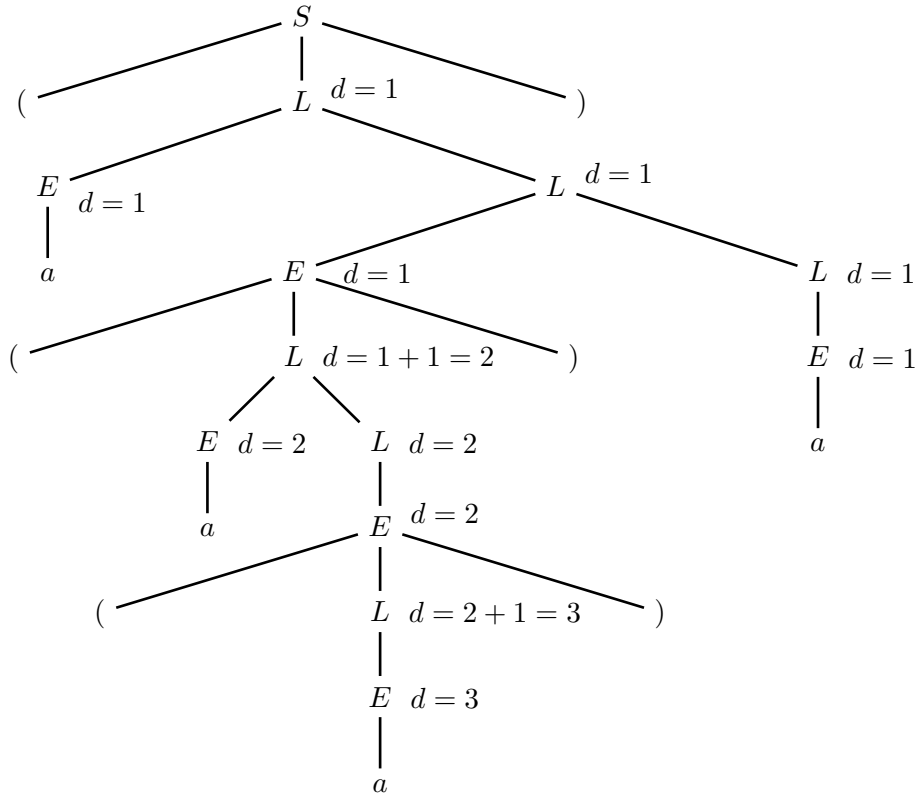


(b) Here are the right attribute d and its semantic functions:

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
right	d	integer ≥ 1	L, E	nesting level of an element (atom or sublist)

#	<i>syntax</i>	<i>semantics</i> - question b
1:	$S_0 \rightarrow (L_1)$	$d_1 = 1$
2:	$L_0 \rightarrow E_1 L_2$	$d_1 = d_0$ $d_2 = d_0$
3:	$L_0 \rightarrow E_1$	$d_1 = d_0$
4:	$E_0 \rightarrow a$	none (or formally $a_1 = d_0$) ⁰
5:	$E_0 \rightarrow (L_1)$	$d_1 = d_0 + 1$

And here is the syntax tree decorated with the values of the right attribute d :



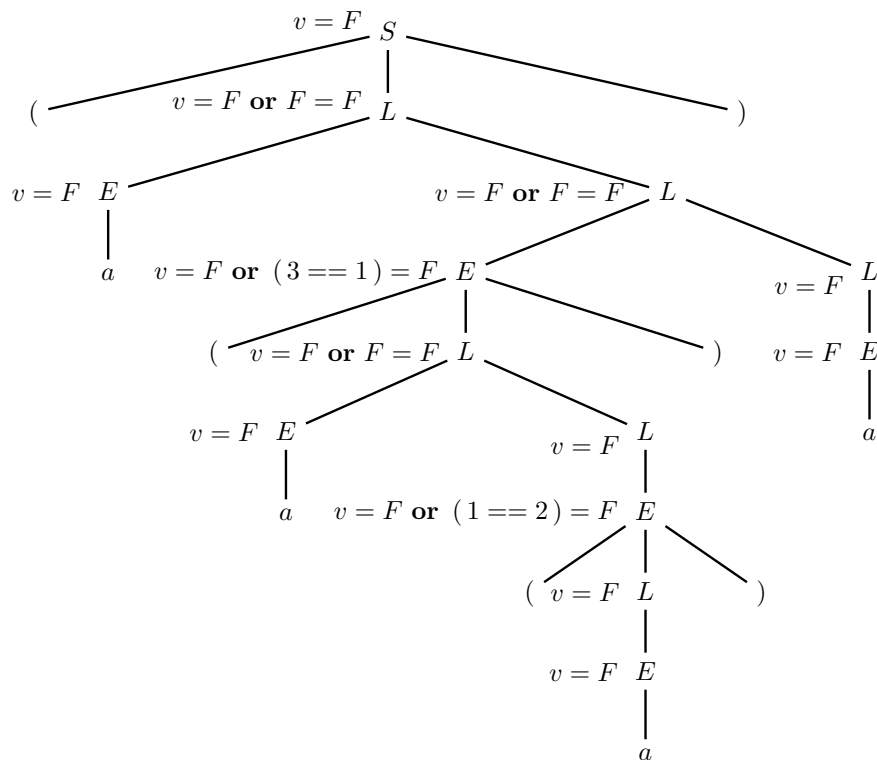
⁰The attribute grammar model allows one to associate a right attribute to a terminal, although here such a position plays a purely formal role and might only be justified for completeness.

(c) Here are the left attribute v and its semantic functions:

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
left	v	boolean	S, L, E	this predicate is true if and only if in the expression there is a proper sublist (i.e., not coincident with the entire expression) that has a total number of elements (atoms and sublists) equal to its nesting level

#	<i>syntax</i>	<i>semantics</i> - question <i>c</i>
1:	$S_0 \rightarrow (L_1)$	$v_0 = v_1$
2:	$L_0 \rightarrow E_1 L_2$	$v_0 = v_1$ or v_2
3:	$L_0 \rightarrow E_1$	$v_0 = v_1$
4:	$E_0 \rightarrow a$	$v_0 = F$
5:	$E_0 \rightarrow (L_1)$	$v_0 = v_1$ or $(n_1 == d_0)$

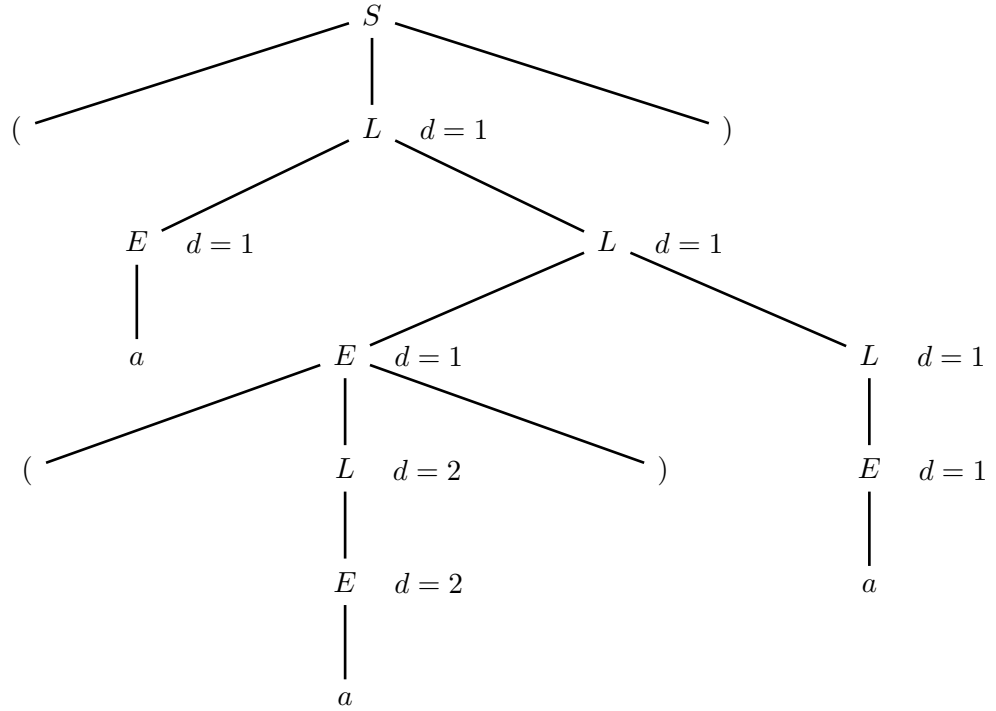
And here is the syntax tree decorated with the values of the left attribute v :

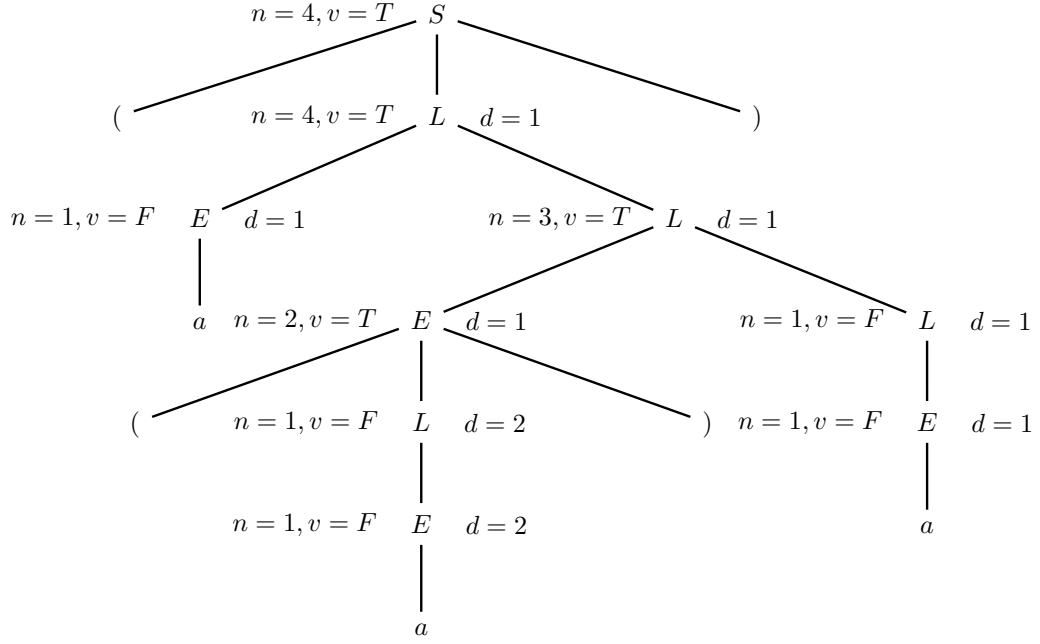


We list here a few additional (not requested) observations. First of all, it is not necessary to add any more auxiliary attributes to the three ones already given. Of course, there may be other solutions that use four or even more attributes.

Second, notice that the full attribute grammar G_a , with all the three attributes and their semantic functions, is of type one-sweep. In fact, the attributes n and d are of type left and right, respectively, they are independent of each other, and the latter (d) depends only on itself in the parent node; and the attribute v is of type left and depends on itself (left) in the child nodes, on attribute n (left) in the child nodes, and on attribute d (right) in the parent node. Thus grammar G_a satisfies the one-sweep condition. Therefore the attributes n , d and v are computable in this order: first d from top to bottom, then n and v altogether from bottom to top (using the values of d already computed). The evaluation order of the child nodes E and L in the rule 2 is free, as their right attributes are independent of each other.

For instance, consider the new expression $e' = (a(a)a)$. In total expression e' has four elements, namely the sublist (a) and three atoms a . The sublist (a) is at level 1 and has one element (the atom a). Therefore the whole expression e' has $v = T$. Here is the one-sweep evaluation (first top-to-bottom and then bottom-to-top):





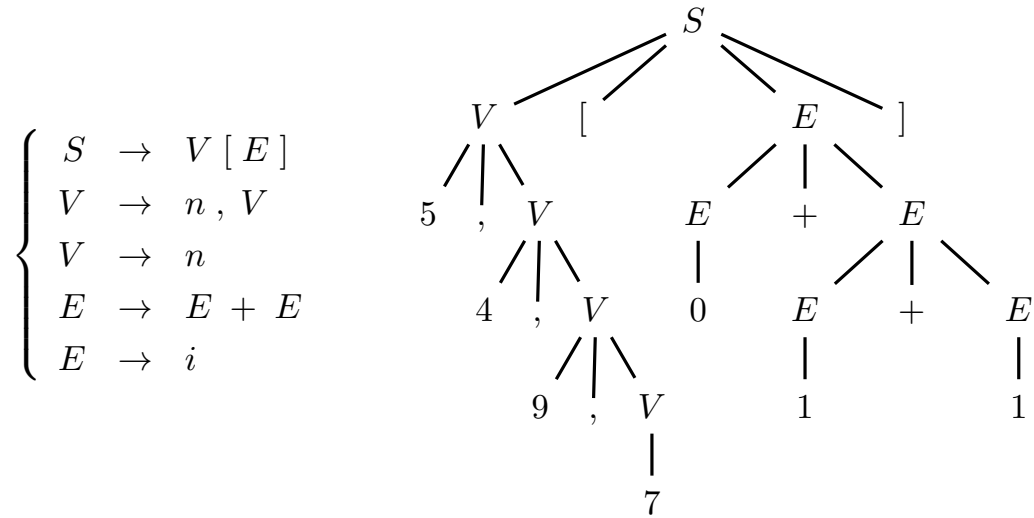
Third, since the subtree evaluation order in the rule 2 (the only one with two nonterminals in the right part) is free (since attribute d depends only on itself in the parent node), it can be taken left-to-right. Anyway, the *BNF* syntactic support G is not of type $LL(k)$ for any $k \geq 1$. In fact, the guide sets of the two alternative rules 2 and 3 overlap for any $k \geq 1$, as both sets contain a string of k open round brackets “ $((\dots ($ ”, due to the existence of the recursive derivation $E \Rightarrow (L) \Rightarrow (E)$ for the initial nonterminal E of the two rules. Thus the attribute grammar G_a is not of type L and it does not have an integrated recursive descent semantic analyzer.

2. Take a vector V of numbers and a summation E of positive or null integers. Consider the problem of finding the following element of V :

$$V [i_1 + i_2 + \dots + i_k]$$

where i_1, i_2, \dots, i_k (with $k \geq 1$) are the integers of summation E . Suppose vector V is indexed as in the language C , starting from index 0.

The problem above is modeled by the following syntax (with axiom S), which generates the vector V (modeled as a list of numerical elements) and concatenates to V the summation E (modeled as an expression with addition):



The terminals n and i schematize a generic element of vector V and a generic integer of summation E , respectively. Commas and brackets are just “syntactic sugar”.

For instance suppose vector V has the four elements 5, 4, 9, 7. If one wishes one knew the value of $V[0 + 1 + 1]$, then the result is element 9 of V . The string that models this instance of the problem, is “ $\underbrace{5, 4, 9, 7}_{\text{vector}} [\underbrace{0 + 1 + 1}_{\text{summation}}]$ ”; see its tree above.

Answer the following questions:

- Write an attribute grammar that, as it is requested by the problem above, brings the wanted vector element to the tree root S . Define the necessary attributes and write the semantic functions coupled with the grammar rules (please use the tables already prepared on the next pages). In the semantic functions, use the terminal names n and i to mean the values of the terminals themselves.
- Say if the attribute grammar designed is of type one-sweep, and explain why. In the case the grammar is one-sweep, write the semantic evaluation procedure of axiom S , say if the grammar is of type L as well, and explain why.
- (optional) If the attribute grammar designed is of type one-sweep, write the semantic evaluation procedure of nonterminal V ; otherwise compute a correct evaluation order of the attributes for the sample syntax tree shown above.

attributes to be used for the grammar

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>

syntax

semantics - to be completed

$$1: S_0 \rightarrow V_1 [E_2]$$

$$2: V_0 \rightarrow n, V_1$$

$$3: V_0 \rightarrow n$$

$$4: E_0 \rightarrow E_1 + E_2$$

$$5: E_0 \rightarrow i$$

Solution

- (a) A solution is to use three attributes: a left attribute α for computing E , a right attribute β for propagating the value of E down through vector V and for accessing the wanted vector element, and one more left attribute γ for transporting the element value up to the root. It is easy to see that such an attribute choice and evaluation scheme will allow us to write a one-sweep attribute grammar.

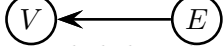
<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
left	α	integer	E	summation value to be computed and transported upward to the tree root
right	β	integer	V	index value to be propagated downward through the vector
left	γ	integer	V, S	value of the wanted vector element, to be transported upward to the tree root

Here are the semantic functions:

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow V_1 [E_2]$	$\beta_1 = \alpha_2$ $\gamma_0 = \gamma_1$
2:	$V_0 \rightarrow n, V_1$	$\beta_1 = \beta_0 - 1$ if $\beta_0 > 0$ then $\gamma_0 = \gamma_1$ else if $\beta_0 == 0$ then $\gamma_0 = n$ else - - i.e. $\beta_0 < 0$ $\gamma_0 = nil$ endif
3:	$V_0 \rightarrow n$	if $\beta_0 > 0$ then vector access error else if $\beta_0 == 0$ then $\gamma_0 = n$ else - - i.e. $\beta_0 < 0$ $\gamma_0 = nil$ endif
4:	$E_0 \rightarrow E_1 + E_2$	$\alpha_0 = \alpha_1 + \alpha_2$
5:	$E_0 \rightarrow i$	$\alpha_0 = i$

Remember that formally all the attributes must be assigned wherever their associated nonterminals occur. Thus attribute γ , which is associated to nonterminal V , must be assigned in all the inner nodes V of the tree, that is, also after the attribute evaluation flow has walked across the wanted vector element (it happens if attribute β becomes negative); in such cases attribute γ is assigned value *nil*; but in a practical implementation, such a useless assignment ought to be simply omitted. The vector access error can be dealt with, either by stopping the evaluation, or by sending out a diagnostic and still assigning *nil* to γ .

There may be other solutions, more or less similar. For instance one might send down along the vector subtree, the expression value and an index starting from 0 and incrementing at each node. When the index equals the expression value, the terminal is sent up to the root. It needs one more attribute, anyway.

- (b) The attribute grammar is one-sweep. This can be argued informally (by words), as done before, or by checking the one-sweep condition. In the latter case: the left attribute α of a parent node E depends on the left attributes α (itself) of its two child nodes E , so it OK; the right attribute β of a child node V depends on the right attribute β (itself) of its parent node V , or (only in the tree root) on the left attribute α of its brother node E , so it is OK; and the left attribute γ of a parent node V (or S) depends on the right attribute β of the parent node V itself (not for S) and on the left attribute γ of its one child node V , so it is OK. Since the attribute grammar is one-sweep, one can draw the sibling graph of the two child nodes of the axiom S , which is:  Now the semantic procedure of S and can be easily designed, provided that the semantic evaluation order is first E and then V (opposite w.r.t. the syntax). Here it is:

```

procedure  $S$  (out  $\gamma$ )
  variable
     $\alpha', \beta', \gamma'$ : integer
  call  $E$  ( $\alpha'$ )
   $\beta' := \alpha'$ 
  call  $V$  ( $\beta', \gamma'$ )
   $\gamma := \gamma'$ 
end procedure

```

Yet the grammar is not of type L, for there are two obstacles: first, because the grammar is ambiguous (and so it is not *LL* either) due to the two-sided recursion in the rule “ $E \rightarrow E + E$ ”; and second, because the syntactic ordering of the nonterminals V and E in the axiomatic rule is different (in particular it is reversed) from the semantic evaluation order of these two nonterminals.

The former obstacle could be circumvented by making the grammar unambiguous, for instance by generating the summation with a right recursive rule like “ $E \rightarrow n + E$ ” (the other rules are left unchanged). Then one would easily see that the grammar becomes *LL*. But the latter obstacle is hard to pass over, unless the language itself is changed by swapping the list of vector elements and the summation; of course, one might instead try to drastically change the attributes and the semantic scheme, though one should pay attention to avoid big attributes and an exceedingly complex scheme; it also seems hard to do, anyway.

- (c) Since the attribute grammar is one-sweep, the semantic procedure of V is easy to write. Here it is:

```

procedure  $V$  ( in  $\beta$ ; out  $\gamma$  )
  variable
     $\beta', \gamma'$ : integer
  case rule alternative of
    " $V \rightarrow n, V$ " : begin
       $\beta' := \beta - 1$ 
      call  $V$  (  $\beta', \gamma'$  )
      if  $\beta > 0$  then
         $\gamma := \gamma'$ 
      else if  $\beta = 0$  then
         $\gamma := n$ 
      else
         $\gamma := nil$ 
      end if
    end
    " $V \rightarrow n$ " : begin
      if  $\beta > 0$  then
        vector access error
      else if  $\beta = 0$  then
         $\gamma := n$ 
      else
         $\gamma := nil$ 
      end if
    end
  end case
end procedure

```

- - optimized

- - optimized

v.a.e.

The procedure is written systematically, see the textbook or the lecture slides for how to do. In practice the *nil* assignment - and the whole “**else**” branch that contains it - could be omitted; but in this event the recursive call that appears in the case “ $V \rightarrow n, V$ ”, should be executed only if parameter β is not negative yet; otherwise parameter γ would just be left unassigned, which is a misbehaviour since it is output by the procedure and therefore is expected to have a value. To optimize so properly, it suffices to take both the assignment to parameter β' and the subsequent call to procedure V , and shift them into the “**then**” branch of the “**if** $\beta > 0$ ” conditional, ahead of the assignment to parameter γ (see above); and to optimize the case “ $V \rightarrow n$ ” accordingly.

Of course the semantic procedure of nonterminal E has one output parameter (the reader may wish to write this procedure as well, or see below).

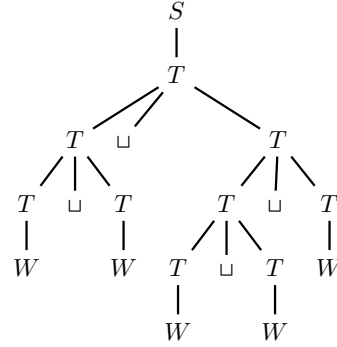
```

procedure  $E$  ( out  $\alpha$  )
  variable
     $\alpha', \alpha''$ : integer
  case rule alternative of
    “ $E \rightarrow E + E$ ” : begin
      call  $E$  (  $\alpha'$  )
      call  $E$  (  $\alpha''$  )
       $\alpha := \alpha' + \alpha''$ 
    end
    “ $E \rightarrow i$ ”      : begin
       $\alpha := i$ 
    end
  end case
end procedure

```


2. A syntactic support G generates a text (not empty) made of words W (it is not necessary to expand this nonterminal), which are separated by a blank character '□'. Here is the support G of the text (axiom S), with a sample syntax tree aside:

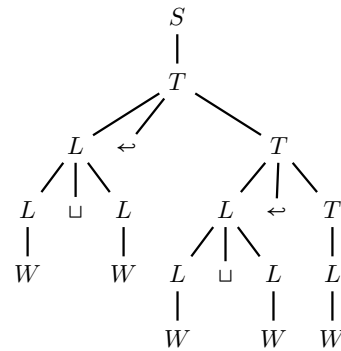
$$G \left\{ \begin{array}{l} S \rightarrow T \\ T \rightarrow T \sqcup T \\ T \rightarrow W \end{array} \right.$$



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

- (a) By means of an integer attribute $w \geq 1$, write an attribute grammar G_1 that computes how many words there are in the text. The result is expressed by w in the root S . Decorate the sample syntax tree with the values of w .
- (b) By means of the attribute w as before and of a new integer attribute $p \geq 0$, write an attribute grammar G_2 that computes how many words precede each word. The results are expressed by p in the respective leaves W . Decorate the sample syntax tree with the values of w e p .
Furthermore say if the attribute grammar G_2 is of type one-sweep or not, and shortly explain why.
- (c) (optional) The syntactic support G is replaced by a new support G' that can split the text into lines (not empty), as follows (axiom S), where the terminal ' \leftarrow ' represents a newline (there is a sample tree aside):

$$G' \left\{ \begin{array}{l} S \rightarrow T \\ T \rightarrow L \leftarrow T \\ T \rightarrow L \\ L \rightarrow L \sqcup L \\ L \rightarrow W \end{array} \right.$$



By means of the attribute w as before and of a new integer attribute $r \geq 0$ (and even more if it helps), write an attribute grammar G_3 (on the support G') that computes how many words there are in the text and how many lines precede each word. The results are expressed by w in the root S and by r in the respective leaves W . Decorate the sample syntax tree with the values of w and r .

Furthermore say if the attribute grammar G_3 is of type one-sweep or not, and shortly explain why.

attributes already assigned to be used for support G

fill the field *type*

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
	w	integer ≥ 1	S, T	number of words
	p	integer ≥ 0	T, W	number of words that precede the current word

attributes already assigned to be used for support G'

and possible auxiliary attributes to be added (if it helps)

fill the field *type*

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
	<i>w</i>	integer ≥ 1	<i>S, T, L</i>	number of words
	<i>r</i>	integer ≥ 0	<i>T, L, W</i>	number of lines that precede (the line where it is located) the current word

$$1: S_0 \rightarrow T_1$$

$$2: T_0 \rightarrow T_1 \sqcup T_2$$

$$3: T_0 \rightarrow W_1$$

$$1: S_0 \rightarrow T_1$$

$$2: T_0 \rightarrow T_1 \sqcup T_2$$

$$3: T_0 \rightarrow W_1$$

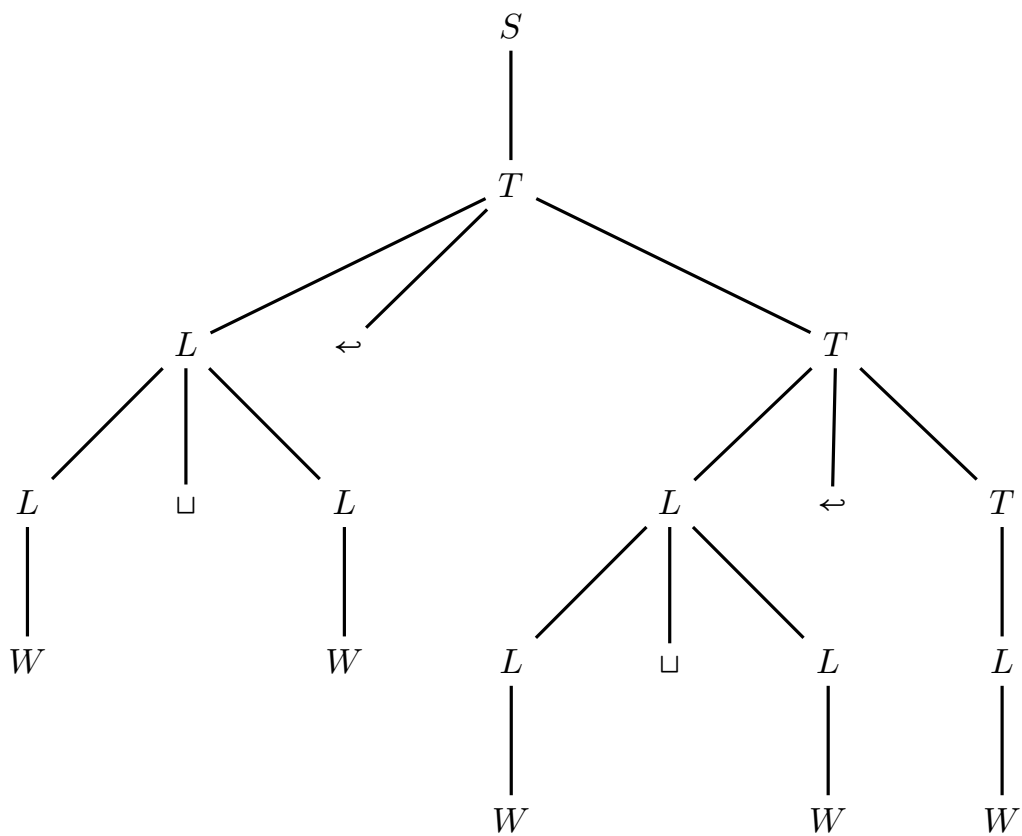
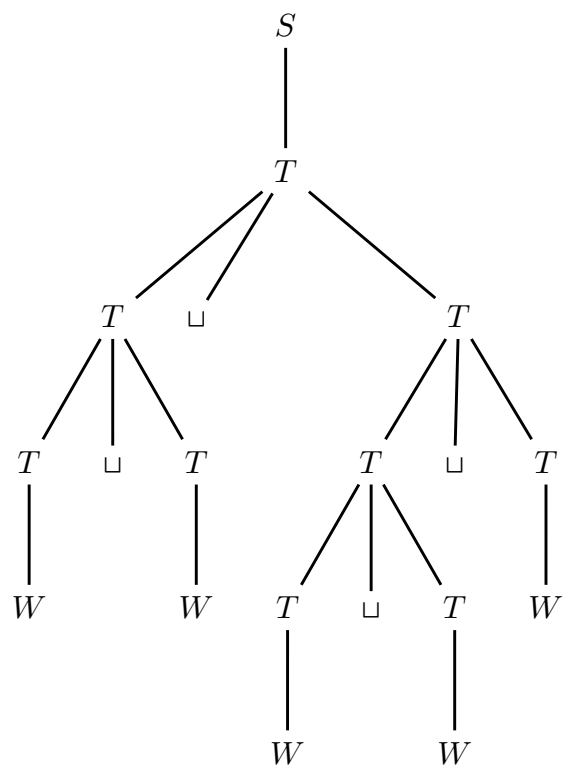
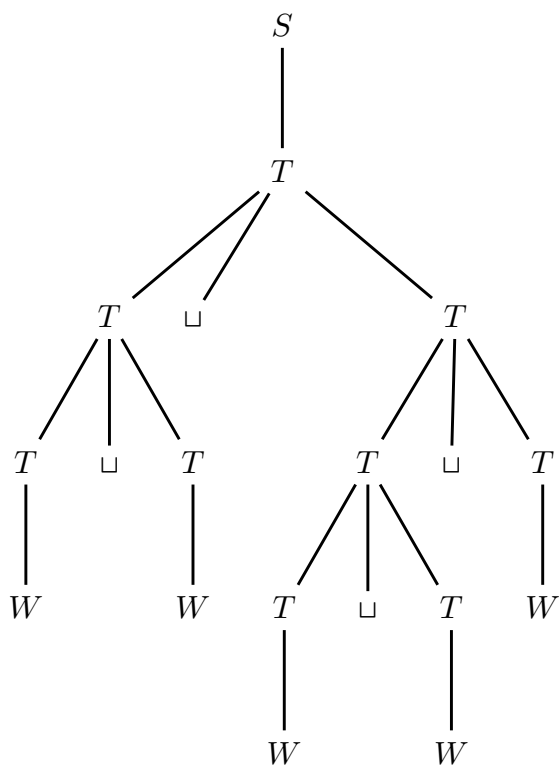
$$1: S_0 \rightarrow T_1$$

$$2: T_0 \rightarrow L_1 \leftrightarrow T_2$$

$$3: T_0 \rightarrow L_1$$

$$4: L_0 \rightarrow L_1 \sqcup L_2$$

$$5: L_0 \rightarrow W_1$$



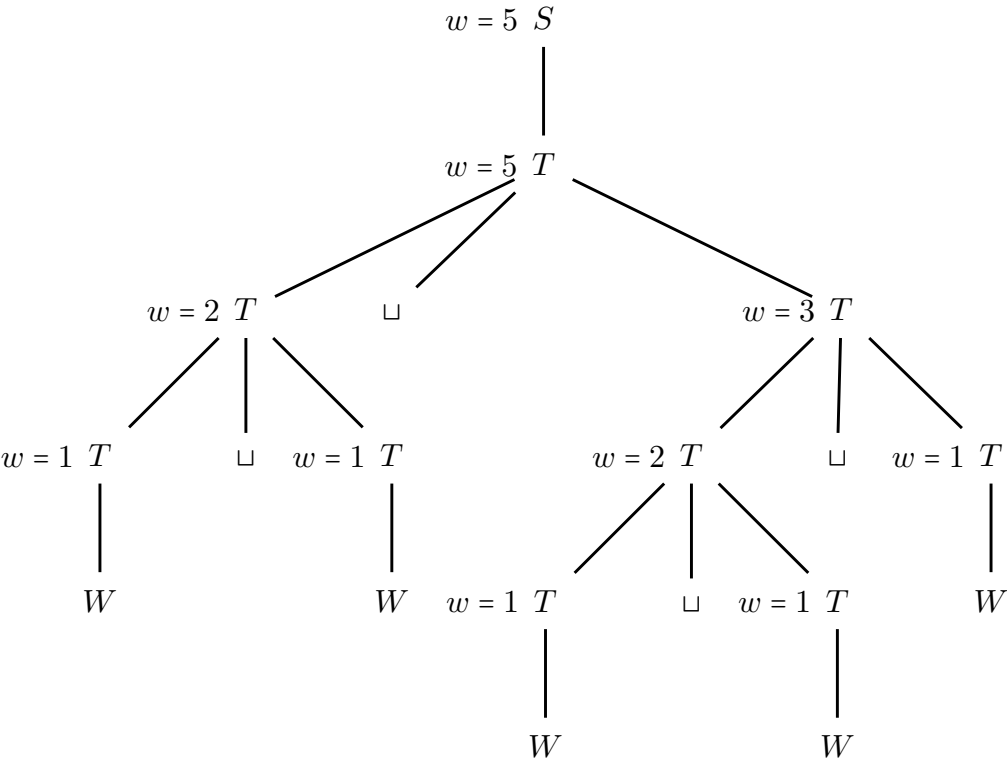
Solution

(a) Here is grammar G_1 (on the support G) with the computation of attribute w :

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
left	w	integer ≥ 1	S, T	number of words

#	<i>syntax</i>	<i>semantics</i> - question a
1:	$S_0 \rightarrow T_1$	$w_0 = w_1$ (result)
2:	$T_0 \rightarrow T_1 \sqcup T_2$	$w_0 = w_1 + w_2$
3:	$T_0 \rightarrow W_1$	$w_0 = 1$

Here is the decoration of the first syntax tree with attribute w (left):



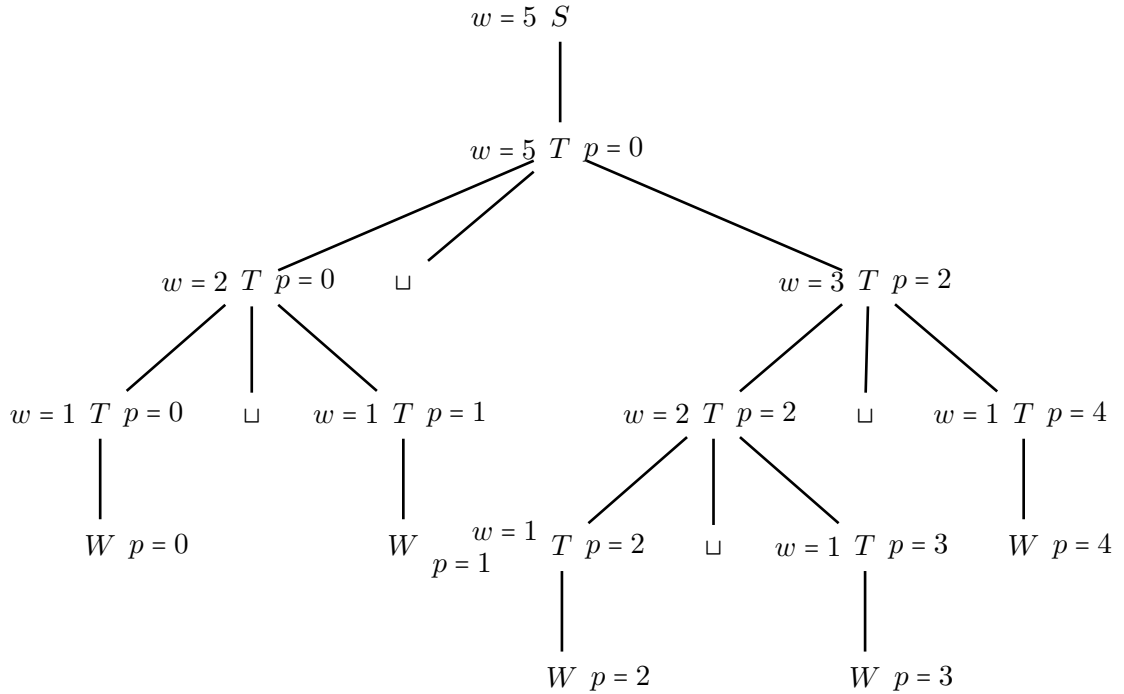
The final result, i.e., the total number of words in the text, is shown by the attribute w associated with the tree root S .

- (b) Here is grammar G_2 (on the support G) with the computation of attribute p (along with that of attribute w , which is repeated as before):

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
right	p	integer ≥ 0	T, W	number of words that precede the current word

#	<i>syntax</i>	<i>semantics</i> - question b
1:	$S_0 \rightarrow T_1$	$p_1 = 0$ $w_0 = w_1$
2:	$T_0 \rightarrow T_1 \sqcup T_2$	$p_1 = p_0$ $p_2 = p_0 + w_1$ $w_0 = w_1 + w_2$
3:	$T_0 \rightarrow W_1$	$p_1 = p_0$ (result) $w_0 = 1$

Here is the decoration of the second syntax tree with attribute w (left), which coincides with what is shown before on the first tree, and attribute p (right):



The final results, i.e., the total number of words in the text and the numbers of words that precede each word of the text, are shown by the attribute w associated with the tree root S and by the attributes p associated with each tree leaf W .

Yes, grammar G_2 is one-sweep: the crucial point is the semantic rule $p_2 = p_0 + w_1$, but attribute p_2 can be computed after exploring the child subtree T_1 on the left

and thus computing attribute w_1 ; in all the other semantic rules, the attributes p and w are independent, and therefore they are computable in any order, so for instance first (top-down) the right attribute p and then (bottom-up) the left attribute w , as it is required by the one-sweep grammar model. More formally, one could check the one-sweep condition and see it is satisfied.

- (c) Here is grammar G_3 (on the support G') with the computation of attribute r (along with that of attribute w , which is essentially the same as before):

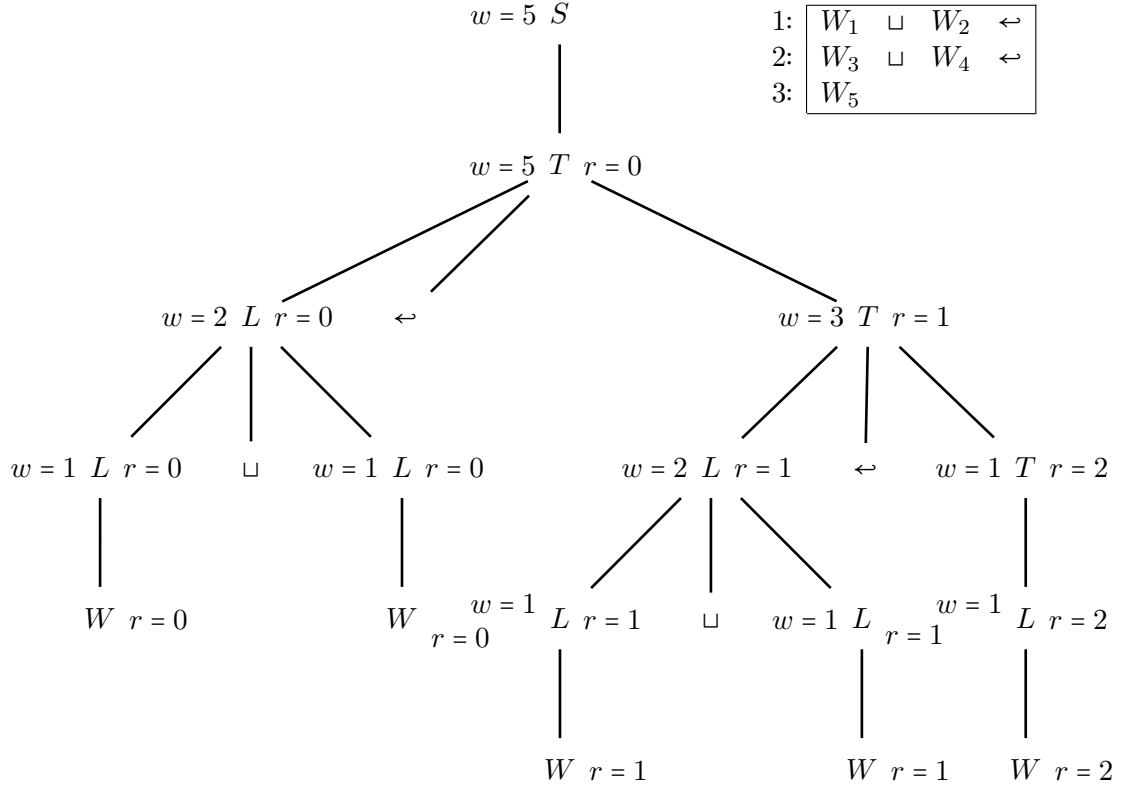
<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
	w	integer ≥ 1	S, T, L	number of words
	r	integer ≥ 0	T, L, W	number of lines that precede the (line where it is located the) current word

#	<i>syntax</i>	<i>semantics</i> - question c
1:	$S_0 \rightarrow T_1$	$r_1 = 0$ $w_0 = w_1$ (result)
2:	$T_0 \rightarrow L_1 \leftarrow T_2$	$r_1 = r_0$ $r_2 = r_0 + 1$ $w_0 = w_1 + w_2$
3:	$T_0 \rightarrow L_1$	$r_1 = r_0$ $w_0 = w_1$
4:	$L_0 \rightarrow L_1 \sqcup L_2$	$r_1 = r_0$ $r_2 = r_0$ $w_0 = w_1 + w_2$
5:	$L_0 \rightarrow W_1$	$r_1 = r_0$ (result) $w_0 = 1$

It is not necessary to introduce any more auxiliary attributes.

Yes, grammar G_3 is one-sweep: the two attributes w (left) and r (right) are fully independent, so it is possible to compute first attribute r and then attribute w , as it is required by the one-sweep grammar model.

Here is the decoration of the third syntax tree, with attribute w (left) and attribute r (right) (aside it is shown the separation and numbering of the text into lines and words):



The final results, i.e., the total number of words in the text and the numbers of lines that precede each word of the text, are shown by the attribute w associated with the tree root S and by the attributes r associated with each tree leaf W .

2. The rules below (axiom S) define an abstract syntax for boolean expressions with the usual operators of logical product (**and**) and negation (**not**), as well as propositional letters schematized by terminal **a**:

- 1: $S \rightarrow F$
- 2: $F \rightarrow F \text{ and } F$
- 3: $F \rightarrow \text{not } F$
- 4: $F \rightarrow \mathbf{a}$

As customary for an abstract syntax, the grammar above does not adequately represent the operator precedence, yet this fact is not relevant here. For instance, the sample boolean expression “**a and not a and not a**” has the syntax tree shown in the figure (next page).

- (a) Write an attribute grammar that computes, for each tree node of type F and for the tree root S , the number of propositional letters of the subtree rooted at that node, that in the overall tree, are in the scope of an even number of negations or of an odd number of negations. In the above example, two letters are in the scope of an even number of negations (first and third letter) and one letter is in the scope of an odd number of negations (second letter).

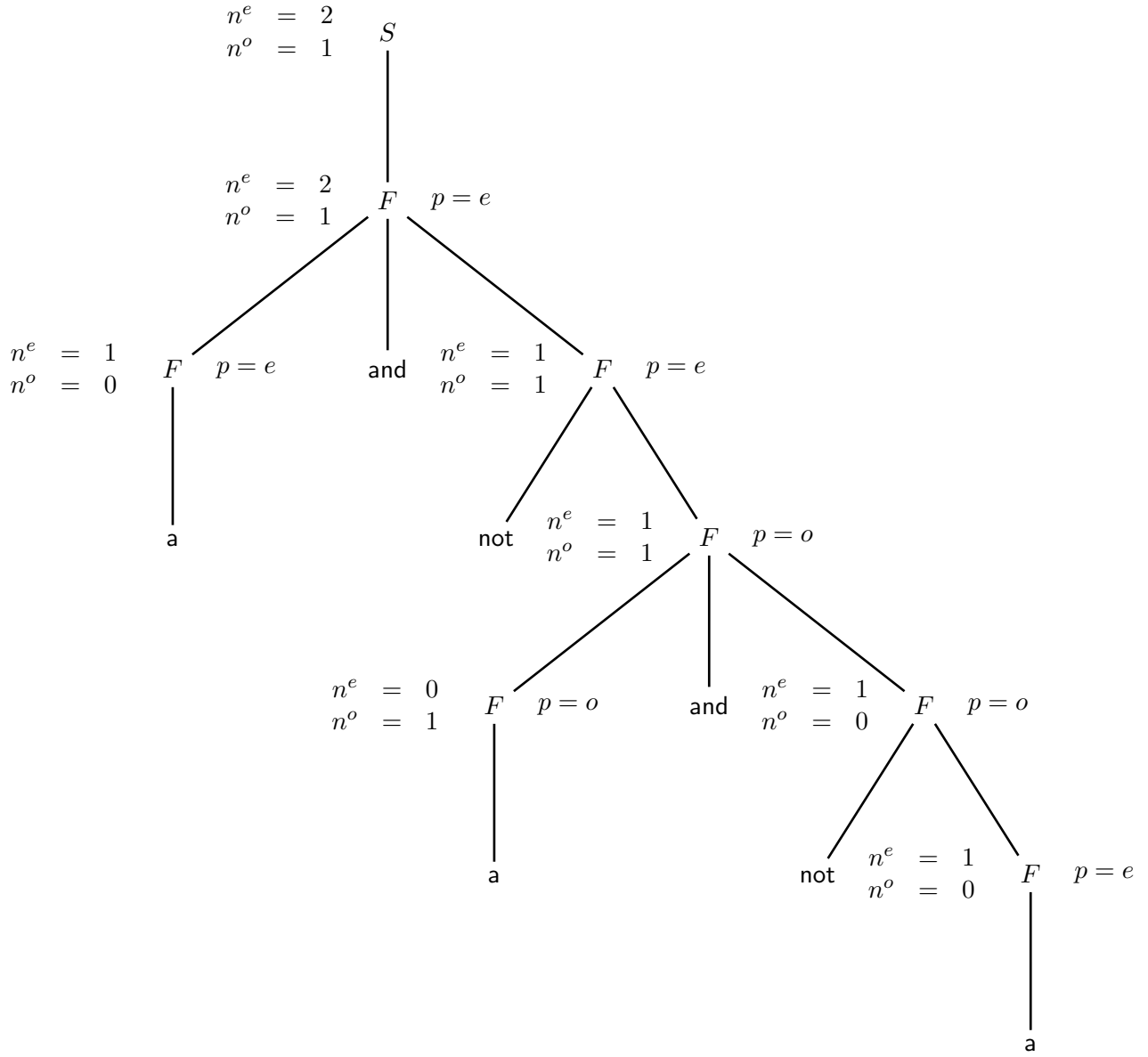
The grammar should use the following three attributes:

- $p \in \{e, o\}$ is a right attribute that indicates if the associated nonterminal is in the scope of an even or odd number of negations
- n^e and n^o are two left integer attributes that count the number of letters in the scope of an even or odd number of negations, respectively.

The values of these three attributes are exemplified in the tree figure (next page). See also the attribute table on the subsequent page.

- (b) (optional) Write a different attribute grammar that computes, in two attributes of the root node S that are called *even* and *odd*, the number of propositional letters that, in the entire expression, are in the scope of an even or of an odd number of negations. The grammar must use only left attributes. Notice that, if necessary, other grammar symbols besides S might have the attributes *even* and *odd*.

decorated tree for question *a*



attributes assigned to be used for question (a)

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
right	p	$\{e, o\}$	F	parity of the number of negations F is in the scope of
left	n^e	integer ≥ 0	S, F	number of propositional letters that are in the scope of an even number of negations
left	n^o	integer ≥ 0	S, F	number of propositional letters that are in the scope of an odd number of negations

attributes to be completed or added for question (b)

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
left	<i>even</i>	integer ≥ 0	$S \dots$	number of propositional letters that are in the scope of an even number of negations
left	<i>odd</i>	integer ≥ 0	$S \dots$	number of propositional letters that are in the scope of an odd number of negations

1: $S_0 \rightarrow F_1$

2: $F_0 \rightarrow F_1 \text{ and } F_2$

3: $F_0 \rightarrow \text{not } F_1$

4: $F_0 \rightarrow \text{a}$

1: $S_0 \rightarrow F_1$

2: $F_0 \rightarrow F_1 \text{ and } F_2$

3: $F_0 \rightarrow \text{not } F_1$

4: $F_0 \rightarrow \text{a}$

Solution

- (a) Here is the attribute grammar with inherited and synthesized attributes:

#	<i>syntax</i>	<i>semantics</i> - question <i>a</i>
1:	$S_0 \rightarrow F_1$	$p_1 := e$ $n_0^o := n_1^o$ $n_0^e := n_1^e$
2:	$F_0 \rightarrow F_1 \text{ and } F_2$	$p_1 := p_0$ $p_2 := p_0$ $n_0^o := n_1^o + n_2^o$ $n_0^e := n_1^e + n_2^e$
3:	$F_0 \rightarrow \text{not } F_1$	$p_1 := \text{if } (p_0 = e) \text{ then } o \text{ else } e \text{ endif}$ $n_0^o := n_1^o$ $n_0^e := n_1^e$
4:	$F_0 \rightarrow a$	$n_0^e := \text{if } (p_0 = e) \text{ then } 1 \text{ else } 0 \text{ endif}$ $n_0^o := \text{if } (p_0 = o) \text{ then } 1 \text{ else } 0 \text{ endif}$

Notice that the attributes n^e and n^o give the correct numbers of letters for the subtree they are associated to, as they are computed basing on attribute p . The reader may notice this attribute grammar is of type one-sweep. Anyway, since the syntactic support is ambiguous (see the two-sided recursive rule $F \rightarrow F \text{ and } F$), the attribute grammar is not of type L and the semantic analyzer cannot be integrated with the syntax analyzer (which is not deterministic).

- (b) As for the attributes, those already assigned, namely *even* and *odd*, suffice, but they have to be associated to the nonterminal F as well (besides the axiom S). Here is the attribute grammar with synthesized attributes only:

#	<i>syntax</i>	<i>semantics</i> - question <i>b</i>
1:	$S_0 \rightarrow F_1$	$even_0 := even_1$ $odd_0 := odd_1$
2:	$F_0 \rightarrow F_1 \text{ and } F_2$	$even_0 := even_1 + even_2$ $odd_0 := odd_1 + odd_2$
3:	$F_0 \rightarrow \text{not } F_1$	$even_0 := odd_1$ $odd_0 := even_1$
4:	$F_0 \rightarrow a$	$even_0 := 1$ $odd_0 := 0$

Notice that the attributes *even* and *odd* give the correct numbers of letters only for the tree root. In the internal nodes, the letter counting may be wrong, as it does not yet consider the total number of negations that rule the node, i.e., the negations from the root to the node. Obviously this grammar is of type one-sweep, as it is purely synthesized. As before, it is not of type L either.

2. Lists of nested sequences of dots, such as $((\dots))(\dots)$, are used to encode positive integer numbers as sums of powers. For instance, this list:

$$\underbrace{\left(\left(\left(\overbrace{\dots}^{2 \text{ dots}} \right) \right) \right)}_{\text{depth 3}} \underbrace{\left(\left(\overbrace{\cdot}^{1 \text{ dot}} \right) \right)}_{\text{depth 2}}$$

encodes the number $3^2 + 2^1 = 9 + 2 = 11$, while the list $((\dots))(((((\dots))))))$ encodes $2^5 + 4^3 = 32 + 64 = 96$. Hence the number of dots in a sequence represents the exponent, and its nesting depth the base of each power in the sum.

The following grammar generates the list (axiom L):

$$\left\{ \begin{array}{l} L \rightarrow E L \mid E \\ E \rightarrow (E) \mid (P) \\ P \rightarrow .P \mid . \end{array} \right.$$

- (a) Write an attribute grammar that computes the value of the number encoded by the list. The final value is associated to the tree root. It is suggested to use the following attributes:
- d is the nesting level of each sublist
 - n is the number of dots in each sublist
 - v is the numerical value encoded
- (b) Decorate the syntax tree of the sentence $((\dots))(\dots)$, by adding to the tree the values of all the attributes. Use the tree prepared on the next page.
- (c) (optional) Determine whether the attribute grammar written is one-sweep and adequately justify your answer.
-

attributes assigned to be used (complete the specification)

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
	<i>d</i>	integer		nesting level of each sublist
	<i>n</i>	integer		number of dots in each sublist
	<i>v</i>	integer		numerical value encoded

attributes to be added, if any

<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>

syntax

semantics

1: $L_0 \rightarrow E_1 L_2$

2: $L_0 \rightarrow E_1$

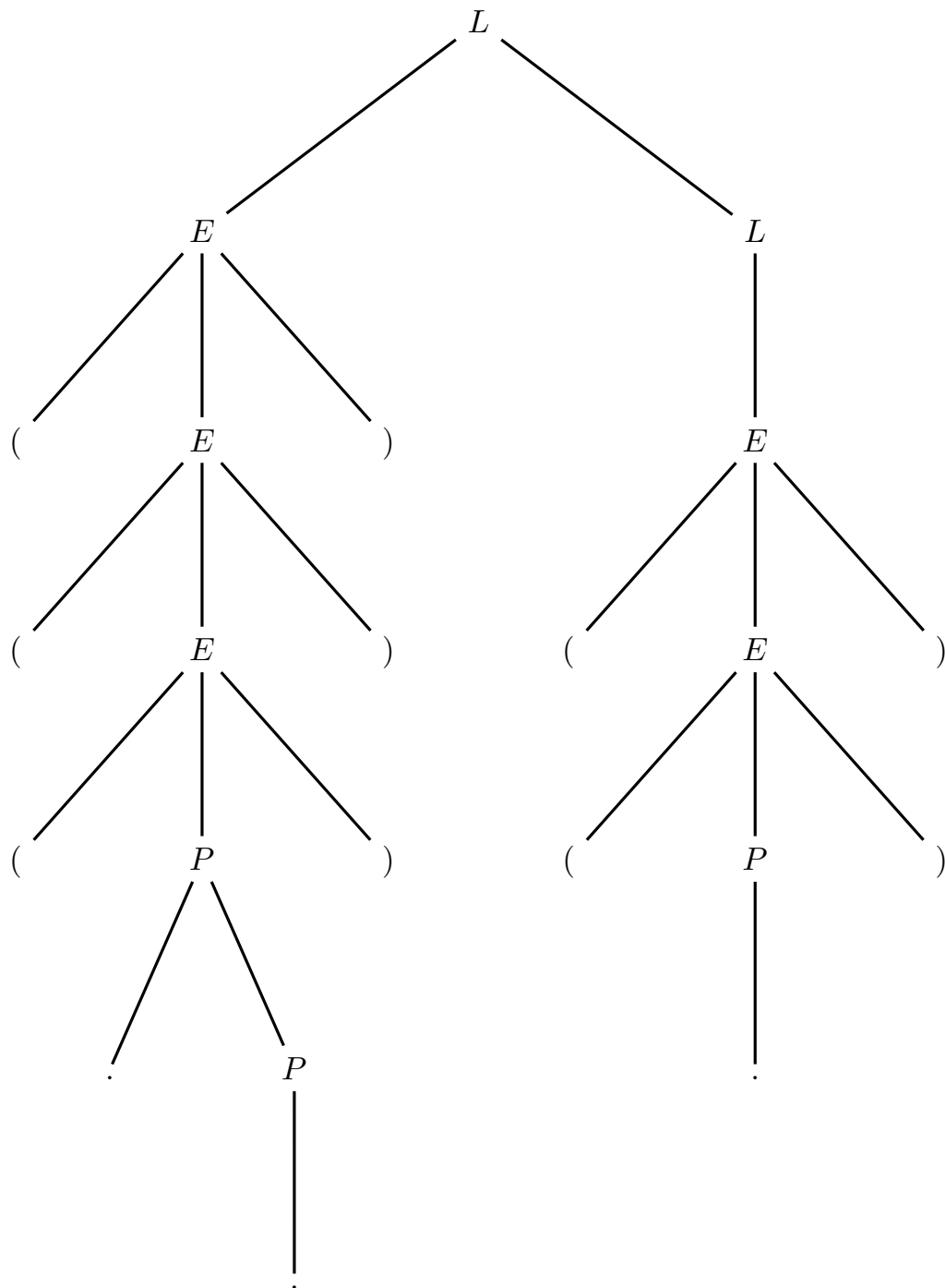
3: $E_0 \rightarrow (E_1)$

4: $E_0 \rightarrow (P_1)$

5: $P_0 \rightarrow . P_1$

6: $P_0 \rightarrow .$

syntax tree to be decorated



Solution

(a) Attributes (the assigned ones):

attributes assigned to be used (specification completed)				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
right	d	integer	E	nesting level of each sublist
left	n	integer	P	number of dots in each sublist
left	v	integer	L, E	numerical value encoded

No need of more attributes.

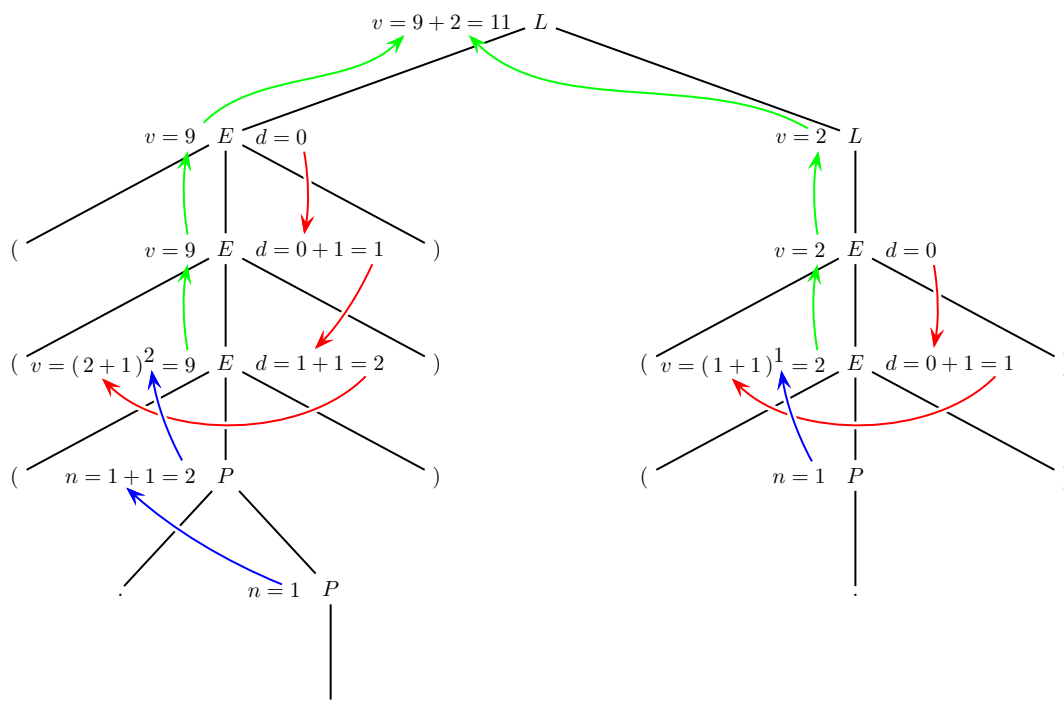
Semantic functions (axiom L_0):

#	<i>syntax</i>	<i>semantics</i>	<i>comment</i>
1:	$L_0 \rightarrow E_1 L_2$	$d_1 = 0$ $v_0 = v_1 + v_2$	inherited (initialized) synthesized
2:	$L_0 \rightarrow E_1$	$d_1 = 0$ $v_0 = v_1$	inherited (initialized) synthesized
3:	$E_0 \rightarrow (E_1)$	$d_1 = d_0 + 1$ $v_0 = v_1$	inherited synthesized
4:	$E_0 \rightarrow (P_1)$	$v_0 = (d_0 + 1)^{n_1}$	synthesized
5:	$P_0 \rightarrow . P_1$	$n_0 = n_1 + 1$	synthesized
6:	$P_0 \rightarrow .$	$n_0 = 1$	synthesized (initialized)

This attribute grammar is quite intuitive and deserves few comments: the dots are counted in the rules 6 (initialization) and 5; the levels are counted in the rules 1 and 2 (initialization), 3 and 4; the power is computed in the rule 4 and the summation of the powers is computed in the axiomatic rule 1. The rest of the semantic functions is just for attribute propagation.

Notice: in the rule 4, the inherited attribute d (coming from above in the tree) and the synthesized one n (coming from below in the tree) meet in the formula $(d + 1)^n$ and merge themselves into computing the synthesized attribute v , which climbs the tree and eventually reaches the root with the final result.

- (b) Here is the decorated tree, enhanced with dependence arrows to show the value flow of the attributes (this enhancement was not requested):



The red, blue and green dependence arrows are for the value flows of attributes d (inherited), n and v (both synthesized), respectively.

- (c) Intuitively, the one-sweep condition is satisfied: attribute d is computed top-down, until it reaches rule 4; attribute n is computed bottom-up, until it also reaches rule 4; so attributes d and n meet at the rule 4: $E \rightarrow (P)$, where they merge themselves into the power computed through the semantic formula $(d + 1)^n$, whose value is saved into the attribute v ; then attribute v is propagated bottom-up, until it reaches the axiomatic rule 1, where it is accumulated and eventually carried up to the tree root. Hence the whole tree is swept only once, first top-down and second bottom-up, as required by the one-sweep condition.

More formally, we should verify, rule by rule, the various parts of the one-sweep condition (as of the course textbook). Here we can notice these facts:

- the right attribute d depends only on itself in all the rules
- the left attribute n depends only on itself in all the rules
- so both attributes d and v trivially satisfy the one-sweep condition
- the left attribute v depends, in all the rules except rule 4, only on itself; and it depends, only in the rule 4, on a right attribute of the parent node E , i.e., d , and on a left attribute of the child node P , i.e., n
- so also attribute v satisfies (not trivially) the one-sweep condition

In conclusion, the whole attribute grammar is of type one-sweep.

Notice that in this grammar there does not happen the case of a rule with a right attribute of a child node depending on some attributes of its brother nodes.

If the definition of the attribute d is modified, a different solution exists, purely synthesized. We sketch it here.

Attributes (the assigned ones - with attribute d redefined):

attributes assigned to be used (specification completed)				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>nonterm.</i>	<i>meaning</i>
left	d	integer	E	<u>reverse</u> nesting level of each sublist
left	n	integer	E, P	number of dots in each sublist
left	v	integer	L	numerical value encoded

Notice that here attribute d specifies the nesting level in reverse order, i.e., the inner-most round brackets are at level 1. This is a change of the exercise text. There is no need of more attributes.

Semantic functions (axiom L_0):

#	<i>syntax</i>	<i>semantics</i>	<i>comment</i>
1:	$L_0 \rightarrow E_1 L_2$	$v_0 = d_1^{n_1} + v_2$	
2:	$L_0 \rightarrow E_1$	$v_0 = d_1^{n_1}$	
3:	$E_0 \rightarrow (E_1)$	$d_0 = d_1 + 1$ $n_0 = n_1$	
4:	$E_0 \rightarrow (P_1)$	$d_0 = 1$ $n_0 = n_1$	initalized
5:	$P_0 \rightarrow . P_1$	$n_0 = n_1 + 1$	
6:	$P_0 \rightarrow .$	$n_0 = 1$	initialized

The difference with respect to the previous solution is that the nesting level is computed upwards instead of downwards. Of course, in this way the nesting level of the internal brackets is different from the standard one. However, at the top the nesting level is correctly known and therefore can be used in the arithmetic computation. This implies that the numbers of dots are propagated upwards at a longer distance, to reach the points where the powers are computed and immediately accumulated. The decorated tree is left to the reader. As this solution is purely synthesized, it is obviously of type one-sweep.

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

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

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

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

- (a) The *nesting level* of an addend is the number of parenthesis pairs around the addend. By using a synthesized attribute h , compute the *height* of the whole expression, i.e., 1 plus the maximal nesting level of all the atomic addends. In the sample expression $a + (a + a) + a$, the value of h is 2. Decorate the syntax tree of the sample expression above with the values of h .
- (b) For each atomic addend a , compute the attribute nl that expresses the nesting level of a (see the definition above). The minimal value of nl is 0.

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

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

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

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

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

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

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

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

attributes assigned to be used (complete the specification)

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

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

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

Question (a):

syntax

semantics - question (a)

1: $S_0 \rightarrow E_1$

2: $E_0 \rightarrow E_1 + E_2$

3: $E_0 \rightarrow (E_1)$

4: $E_0 \rightarrow a$

Question (b):

syntax

semantics - question (b)

1: $S_0 \rightarrow E_1$

2: $E_0 \rightarrow E_1 + E_2$

3: $E_0 \rightarrow (E_1)$

4: $E_0 \rightarrow a$

Question (c):

syntax

semantics - question (c)

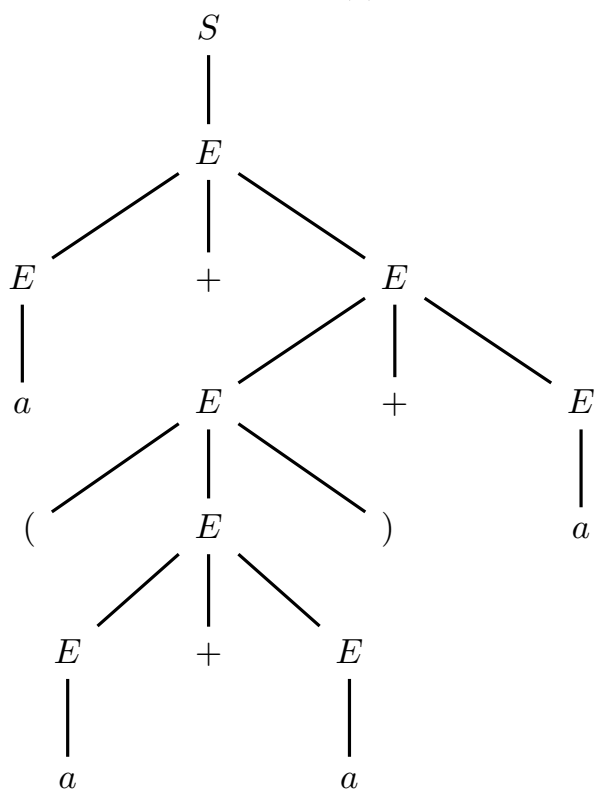
$$1: \quad S_0 \rightarrow E_1$$

$$2: \quad E_0 \rightarrow E_1 + E_2$$

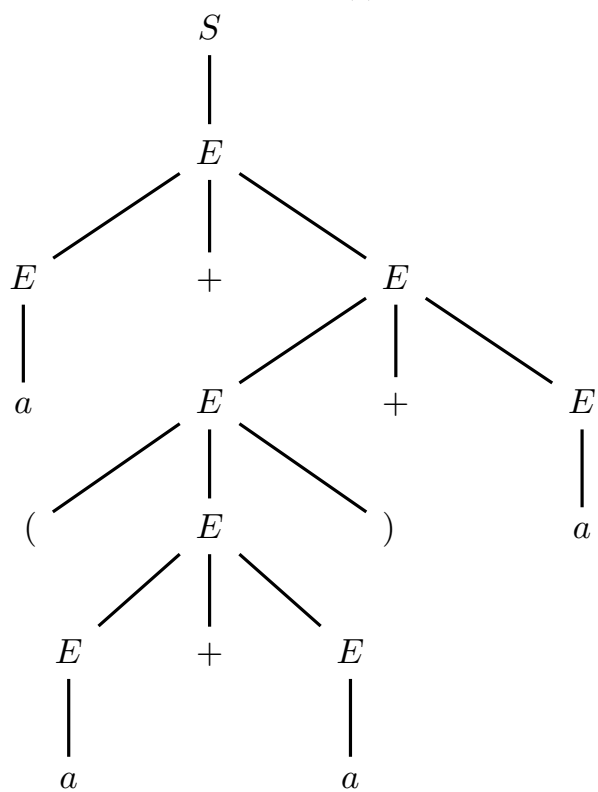
$$3: \quad E_0 \rightarrow (E_1)$$

$$4: \quad E_0 \rightarrow a$$

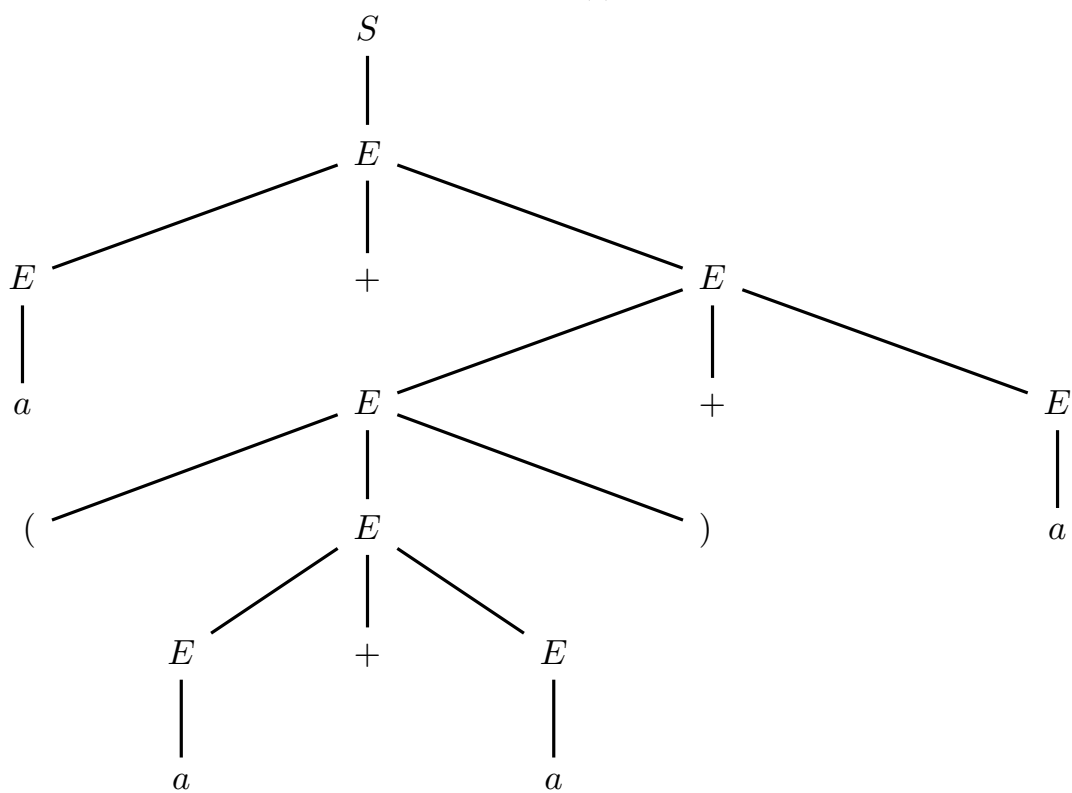
question (a)



question (b)



question (c)



Solution

(a) Attributes and semantics:

attributes assigned to be used				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
left	h	integer ≥ 1	S, E	height of the whole expression

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

The rationale of this attribute grammar is at all obvious.

(b) Attributes and semantics:

attributes assigned to be used				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
right	nl	integer ≥ 0	E, a	number of parenthesis pairs around an addend

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

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

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

(c) Attributes and semantics:

attributes assigned to be used				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
right	p	integer ≥ 0	E, a	position of an addend in the list of addends at the same nesting level in a (sub)expression

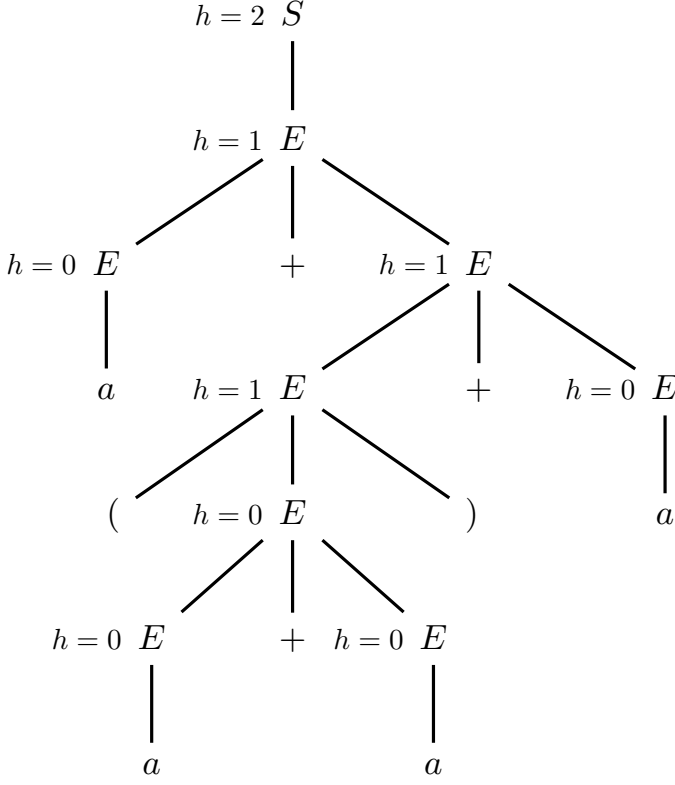
attributes to be added, if any				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
left	na	integer ≥ 1	E	number of addends at the same nesting level in a (sub)expression

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

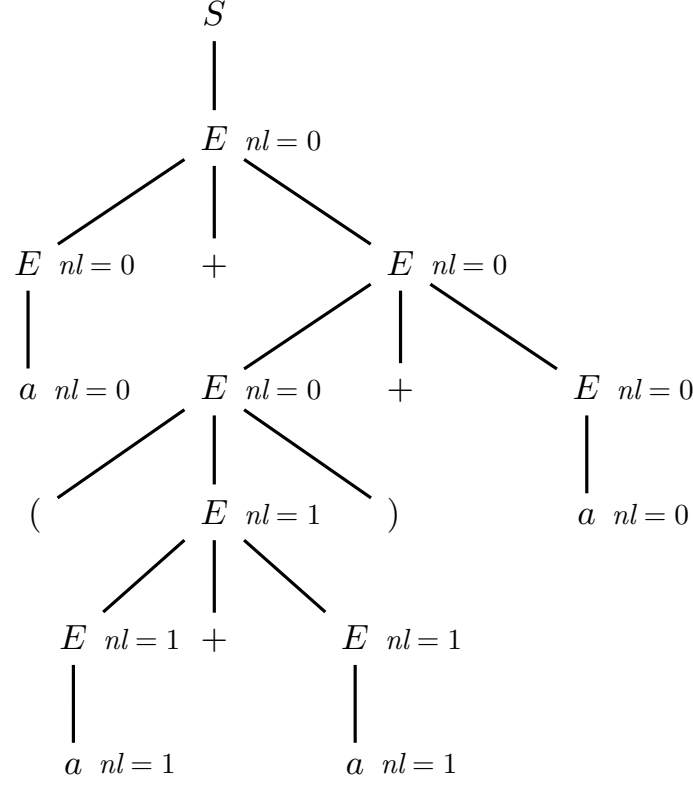
The rationale of this attribute grammar is a little less obvious: attribute na counts the total number of addends (atomic or not) at the same level inside of each parenthesized (sub)expression; attribute p expresses the positions (starting from 0 on the left) of the addends (atomic or not) inside of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

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

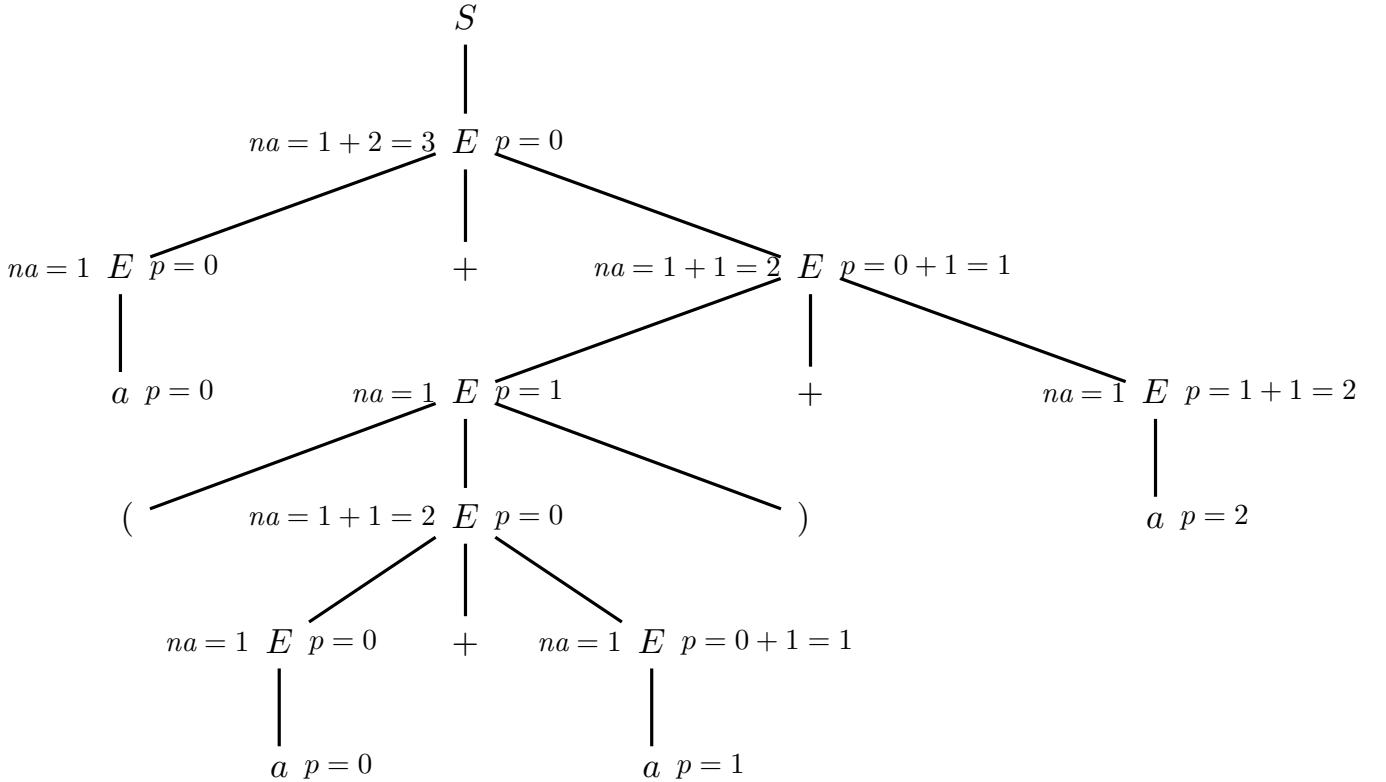
question (a)



question (b)



question (c)



2. Consider the Dyck language with two parenthesis types, namely $a b$ (open closed) and $c d$ (open closed), generated by the known syntactic support below (axiom S):

$$\left\{ \begin{array}{lcl} 0: & S & \rightarrow D \\ 1: & D & \rightarrow a D b D \\ 2: & D & \rightarrow c D d D \\ 3: & D & \rightarrow \varepsilon \end{array} \right.$$

One wishes to analyze a syntax tree by means of attribute grammars, to check if the Dyck string contains certain pairs of adjacent letters, and to compute and output the nesting depth of the pair, if any, that occurs leftmost in the string, or to output 0 if there is no such pair. To answer the questions below, one has to use (partly or totally) the attributes already prepared on the next page.

Answer the following questions:

- (a) Write an attribute grammar G_1 that checks if the Dyck string contains an adjacent letter pair of type ac .
- (b) Write an attribute grammar G_2 that checks if the Dyck string contains an adjacent letter pair of type bd .
- (c) (optional) Reconsider grammar G_1 and extend it, in such a way that it outputs (at the tree root) the nesting depth (always ≥ 1) of the leftmost occurrence of the pair ac (more precisely the depth of letter c in the pair), if the Dyck string has some, or else that outputs 0. Samples:
 - $acdb$, yes (one pair), depth 1
 - $abacdb$, yes (one pair), depth 1
 - $aabacdbb$, yes (one pair), depth 2
 - $acdaacdbb$, yes (two pairs), depth 1
 - $abcd$, none, depth 0

Furthermore, say if the so-extended attribute grammar is of type one-sweep and justify your answer. To answer question (c), you can (and actually have to) define more attributes, for dealing with the nesting depth. Complete the attribute list already given on the next page.

attributes assigned to be used (complete the specification if necessary)

<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
left	b	boolean	D	true if and only if nonterminal D generates a string that ends with letter b
left	c	boolean	D	true if and only if nonterminal D generates a string that starts with letter c
left	e	boolean	D	true if and only if nonterminal D generates the empty string
left	ac	boolean	S, D	true at the tree root if and only if letter pair ac occurs in the Dyck (sub)string generated by S (or by D)
left	bd	boolean	S, D	true at the tree root if and only if letter pair bd occurs in the Dyck (sub)string generated by S (or by D)

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

<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>

$$0: S_0 \rightarrow D_1$$

$$1: D_0 \rightarrow a D_1 b D_2$$

$$2: D_0 \rightarrow c D_1 d D_2$$

$$3: D_0 \rightarrow \varepsilon$$

$$0: S_0 \rightarrow D_1$$

$$1: D_0 \rightarrow a D_1 b D_2$$

$$2: D_0 \rightarrow c D_1 d D_2$$

$$3: D_0 \rightarrow \varepsilon$$

$$0: S_0 \rightarrow D_1$$

$$1: D_0 \rightarrow a D_1 b D_2$$

$$2: D_0 \rightarrow c D_1 d D_2$$

$$3: D_0 \rightarrow \varepsilon$$

Solution

- (a) Here is the solution, purely synthesized, where attributes c and ac suffice:

#	<i>syntax</i>	<i>semantics</i> - question (a)
0:	$S_0 \rightarrow D_1$	$ac_0 = ac_1$
1:	$D_0 \rightarrow a D_1 b D_2$	$c_0 = false$ $ac_0 = c_1 \vee ac_1 \vee ac_2$
2:	$D_0 \rightarrow c D_1 d D_2$	$c_0 = true$ $ac_0 = ac_1 \vee ac_2$
3:	$D_0 \rightarrow \varepsilon$	$c_0 = false$ $ac_0 = false$

Remember that a connective “ \vee ” means logical sum (disjunction). Basically, see rule 1, a letter pair ac occurs in the Dyck (sub)string generated by D_0 if nonterminal D_1 immediately generates letter c , or if a pair ac already occurs in the Dyck substrings generated by D_1 or D_2 . Similarly for rule 2, except that in this case an immediate pair ac may not show up, as the rule does not contain letter a . The other semantic functions are obvious.

- (b) Here is the solution, purely synthesized, where attributes b , e and bd suffice:

#	<i>syntax</i>	<i>semantics</i> - question (b)
0:	$S_0 \rightarrow D_1$	$bd_0 = bd_1$
1:	$D_0 \rightarrow a D_1 b D_2$	$b_0 = b_2 \vee e_2$ $e_0 = false$ $bd_0 = bd_1 \vee bd_2$
2:	$D_0 \rightarrow c D_1 d D_2$	$b_0 = b_2$ $e_0 = false$ $bd_0 = b_1 \vee bd_1 \vee bd_2$
3:	$D_0 \rightarrow \varepsilon$	$b_0 = false$ $e_0 = true$ $bd_0 = false$

Attribute grammar G_2 is quite similar to grammar G_1 , but computing attribute b is (only) slightly more difficult than computing attribute c . Basically, see rule 2, a letter pair bd occurs in the Dyck (sub)string generated by D_0 if nonterminal D_1 generates a final letter b , or if a pair bd already occurs in the Dyck substrings generated by D_1 or D_2 . Similarly for rule 1, except that in this case an immediate pair bd may not show up, as the rule does not contain letter d .

To check if nonterminal D_0 generates a final letter b : in the rule 1, this happens if nonterminal D_2 does so or if it is nullable; while in the rule 2 this happens only if nonterminal D_2 does so. The other semantic functions are obvious.

- (c) Here is the solution, with inheritance, where more attributes (one of which is inherited) have to be added to those of grammar G_1 . Here they are:

attributes to be added, if any (complete the specification)				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
right	<i>depth</i>	integer ≥ 0	D	standard nesting depth (downward computation)
left	<i>up</i>	integer ≥ 0	S, D	nesting depth of pair $a\ c$ (if any) at the tree root (upward propagation)

And here is grammar G_1 extended; one can skip the semantic functions of G_1 that detect pair $a\ c$, which are here included unchanged from question (a):

#	<i>syntax</i>	<i>semantics</i> - question (c)
0:	$S_0 \rightarrow D_1$	$depth_1 = 0$ $ac_0 = ac_1$ $up_0 = up_1$
1:	$D_0 \rightarrow a\ D_1\ b\ D_2$	$depth_1 = depth_0 + 1$ $depth_2 = depth_0$ $c_0 = false$ $ac_0 = c_1 \vee ac_1 \vee ac_2$ if $c_1 == true$ then $up_0 = depth_1$ else if $ac_1 == true$ then $up_0 = up_1$ else if $ac_2 == true$ then $up_0 = up_2$ else $up_0 = 0$ end if
2:	$D_0 \rightarrow c\ D_1\ d\ D_2$	$depth_1 = depth_0 + 1$ $depth_2 = depth_0$ $c_0 = true$ $ac_0 = ac_1 \vee ac_2$ if $ac_1 == true$ then $up_0 = up_1$ else if $ac_2 == true$ then $up_0 = up_2$ else $up_0 = 0$ end if
3:	$D_0 \rightarrow \varepsilon$	$c_0 = false$ $ac_0 = false$ $up_0 = 0$

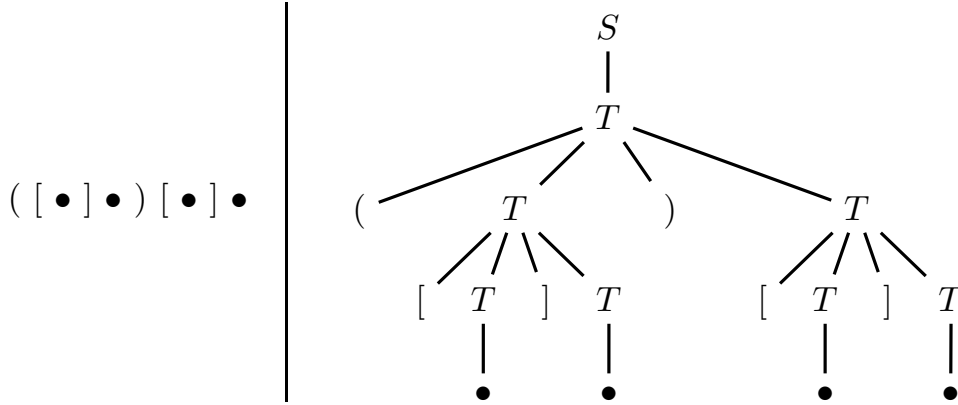
The so-extended attribute grammar G_1 works as follows. The nesting depth of each parenthesis structure $a b$ or $c d$ is computed moving downwards in the usual way by means of the right attribute *depth*, with the outermost parentheses at depth 0 as suggested by the given samples. The adjacent letter pairs $a c$ are detected moving upwards by means of the left attributes c and ac , as it is done in the question (a). The nesting depth of the detected leftmost letter pair $a c$ is transported upwards by means of the left attribute *up*: this attribute is first initialized with 0 at the innermost parenthesis structures, then it is (re)assigned the depth of letter c whenever a new letter pair $a c$ is found leftmost, else it is simply maintained the depth of the already detected inner leftmost pair, if any. Clearly the nesting depth of the leftmost pair $a c$ is always given preference to be assigned to attribute *up*, see the semantic functions of the rules 1 and 2.

The so-extended attribute grammar G_1 is of type one-sweep. The reason is that the only right attribute is *depth*, which can be computed independently of all the others, and that the remaining attributes c , ac and *up* are all left and depend only on left ones, except attribute *up*, which depends on the left attributes c , ac and *up* (itself) of the child nodes, but (in the rule 1) also on the right attribute *depth* of a child node (this dependence is crucial as it transfers the inherited information flow to the synthesized one); the last however is a dependence type that is admitted for the left attributes of a one-sweep grammar.

2. Consider the following abstract syntax, which generates well balanced parenthetical expressions (with two parenthesis types: round and square), where the bullets “•” are enclosed in a number ≥ 0 of round or square parenthesis pairs (axiom S):

$$\left\{ \begin{array}{l} 1: S \rightarrow T \\ 2: T \rightarrow (T)T \\ 3: T \rightarrow [T]T \\ 4: T \rightarrow \bullet \end{array} \right.$$

For each bullet in the expression, let r and s be the numbers of enclosing *round* and *square* parenthesis pairs, respectively. Suppose each bullet is given a value v , such that $v = 2 \times r + 3 \times s$. The entire expression is also given a value, defined as the sum of the values of all its bullets. For instance, in the sample expression below:



the syntax tree of which is reported on the right, the value v of the expression is:

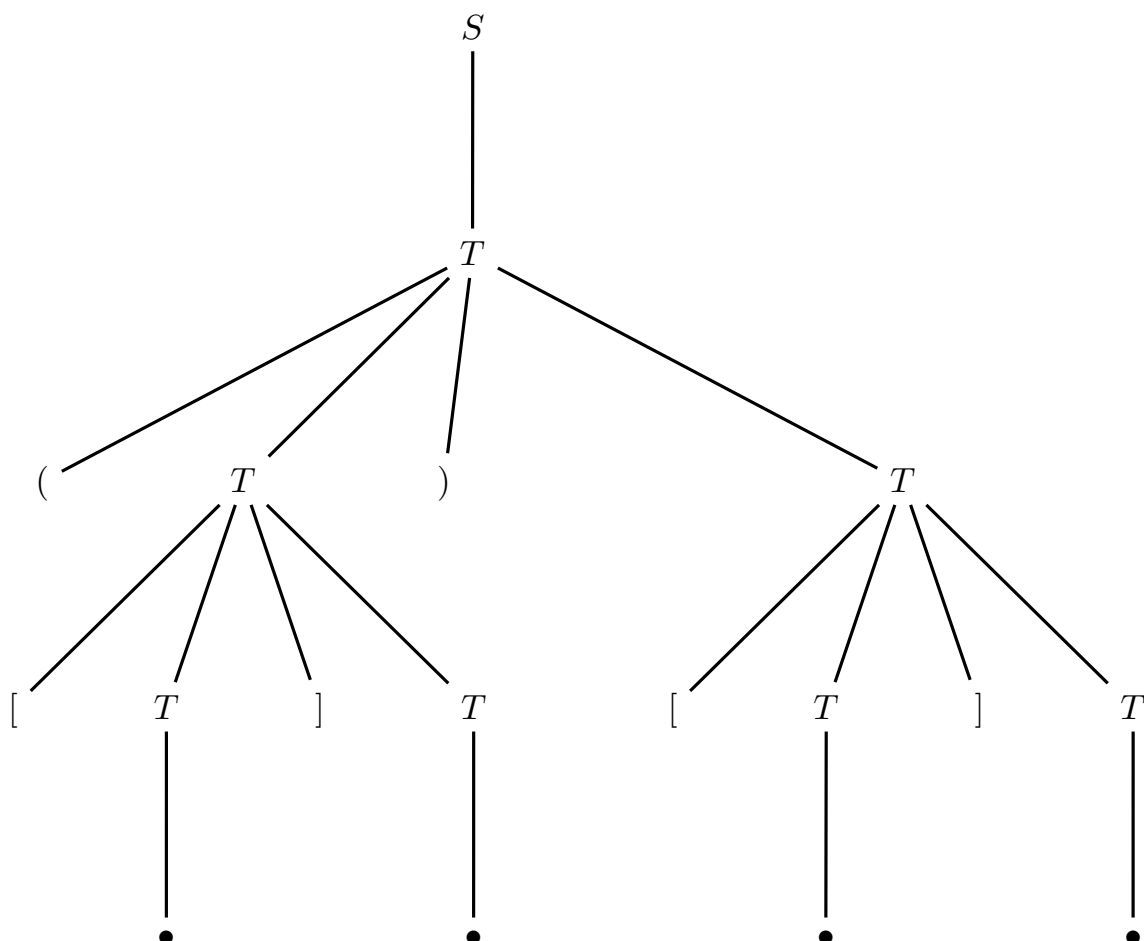
$$v = \underbrace{2+3}_{\text{first bullet}} + \underbrace{2+0}_{\text{second bullet}} + \underbrace{0+3}_{\text{third bullet}} + \underbrace{0+0}_{\text{fourth bullet}} = 10$$

Answer the following questions:

- Define an attribute grammar, based on the above syntax, that defines the computation of the value v for the entire expression, as explained above. It is suggested to use three attributes r , s , and v with the meaning previously explained. See the attribute table on the next page.
- Decorate the syntax tree of the sample expression $([\bullet]\bullet)[\bullet]\bullet$ with the values of the attributes. Use the syntax tree prepared on the next page.
- (optional) Determine if the attribute grammar is of type one-sweep and if it satisfies the L condition, and reasonably justify your answers.

attributes assigned to be used				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
right	<i>r</i>	integer	<i>T</i>	number ≥ 0 of pairs of round brackets that enclose the bullet or (sub)expression
right	<i>s</i>	integer	<i>T</i>	number ≥ 0 of pairs of square brackets that enclose the bullet or (sub)expression
left	<i>v</i>	integer	<i>S, T</i>	value ≥ 0 given to the bullet or (sub)expression

tree prepared for decoration with attributes



#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

1: $S_0 \rightarrow T_1$

2: $T_0 \rightarrow (T_1) T_2$

3: $T_0 \rightarrow [T_1] T_2$

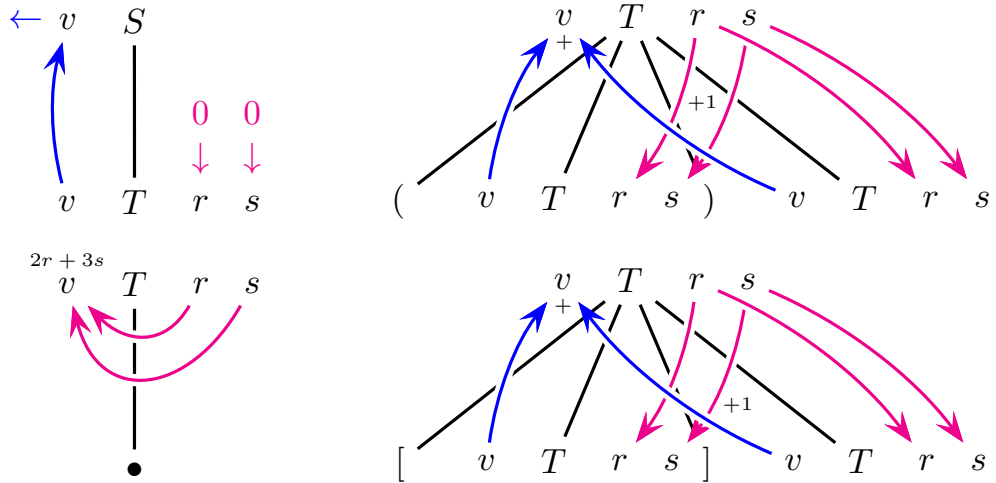
4: $T_0 \rightarrow \bullet$

Solution

- (a) Here is the requested attribute grammar, with the three attributes r , s and v :

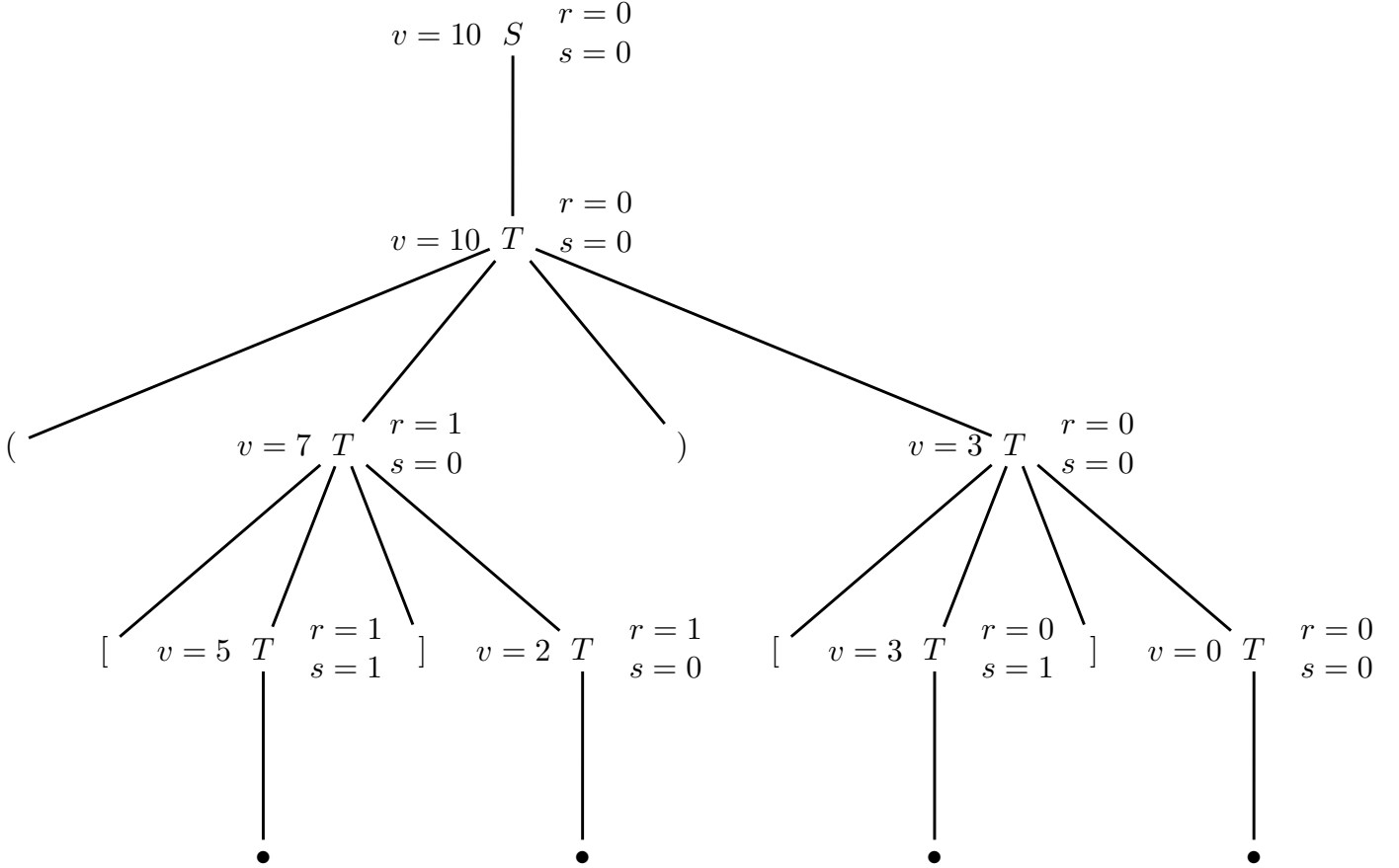
#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow T_1$	$r_1 := 0$ $s_1 := 0$ $v_0 := v_1$
2:	$T_0 \rightarrow (T_1)T_2$	$r_1 := r_0 + 1$ $r_2 := r_0$ $s_1 := s_0$ $s_2 := s_0$ $v_0 := v_1 + v_2$
3:	$T_0 \rightarrow [T_1]T_2$	$r_1 := r_0$ $r_2 := r_0$ $s_1 := s_0 + 1$ $s_2 := s_0$ $v_0 := v_1 + v_2$
4:	$T_0 \rightarrow \bullet$	$v_0 := 2 \times r_0 + 3 \times s_0$

The computation of the nested round and square parenthesis pairs is carried out by means of the right (inherited) attributes r and s alone, respectively, in a somewhat standard way. Each such attribute is incremented in the rule where the corresponding parenthesis type (round or square) is generated. The left (synthesized) attribute v computes and stores the bullet value in the rule 4, and then all the bullet values are accumulated and transported upwards by means of attribute v , as far as the final value of the expression reaches the tree root. The attribute grammar is clearly correct: all the attributes are computed in the rules where they are defined and (intuitively) the dependence graphs of all possible syntactic trees are always acyclic (here this property is granted by construction). For completeness, here are the dependence graphs of each single grammar rule:



Magenta and blue arrows denote inheritance and synthesis dependences, respectively. The top-down and transversal information flows, and the bottom-up one, are quite visible. Input / output values are injected / extracted at the top.

(b) Here is the sample tree decorated according to the given attribute grammar:



(c) The attribute grammar is of type one-sweep. The right attributes r and s depend only on right attributes, a dependence type that is admitted for one-sweep grammars. The left attribute v depends only on itself (as in a purely synthesized solution, which is immediately one-sweep), except in the rule 4 where it depends on right attributes of the parent node. Also the last is a dependence that is admitted for one-sweep grammars. Thus the whole grammar is one-sweep.

Even intuitively, first attributes r and s can be computed top-down (and independently of each other), then attribute v can be computed bottom-up, until it reaches the tree root and the evaluation terminates with the final result.

Concerning the L condition for integrating the syntactic and semantic analyzers. First, the attribute grammar is one-sweep. Second, notice that clearly a possible evaluation order of the child subtrees T_1 and T_2 in both rules 2 and 3 is the syntactic order from left to right: first visit and evaluate subtree T_1 , then subtree T_2 . One can also notice that the sibling graphs of both rules 2 and 3 do not have any arcs, so that their nodes, i.e., the nonterminal occurrences T_1 and T_2 , can be linearly ordered in any way. Thus the L condition is easily satisfied.

Furthermore notice that the *BNF* syntactic support is basically the well known non-ambiguous form of the Dyck grammar, with right recursion. It is known that this grammar is of type $LL(1)$. In any case, one can immediately see that the three

alternative rules 2, 3 and 4 have disjoint guide sets. Since the syntactic support is $LL(1)$ and the L condition is satisfied, it is possible to write a top-down parser, e.g., a recursive descent one, that also integrates the semantic evaluation of the attributes.

For completeness, here it is shown a recursive-descent evaluator for the attribute grammar designed. It consists of a main program and two procedures S and T , with their attribute parameters. First, it is more convenient to show the pure parser underlying the evaluator, and then to complete it with the attribute declarations and semantic functions. Here is the parser (mostly uncommented):

program PARSER

$cc = next$

call S

if $cc \in \{ \vdash \}$

 | **accept**

else

 | **reject**

endif

end program

procedure S

if $cc \in \{ '(', '[', '\bullet' \}$

 | **call** T

 | **return** // scanned 1

else

 | **error** // error case

endif

end procedure

procedure T

if $cc \in \{ ' ' \}$ // 1st way: scan rule 2

$cc = next$

if $cc \in \{ '(', '[', '\bullet' \}$

 | **call** T

if $cc \in \{ ') ' \}$

 | $cc = next$

if $cc \in \{ '(', '[', '\bullet' \}$

 | **call** T

 | **return**

 // scanned 2

endif

endif

endif

else if $cc \in \{ '[' \}$ // 2nd way: scan rule 3

$cc = next$

if $cc \in \{ '(', '[', '\bullet' \}$

 | **call** T

if $cc \in \{ ']' \}$

 | $cc = next$

if $cc \in \{ '(', '[', '\bullet' \}$

 | **call** T

 | **return**

 // scanned 3

endif

endif

endif

else if $cc \in \{ '\bullet' \}$ // 3rd way: scan rule 4

$cc = next$

 | **return**

 // scanned 4

endif

error

 // grouped error cases

end procedure

For brevity, in the two procedures all the error cases are grouped at the end: if anyone of the if-conditions fails to be true, execution drops to the final error statement. Most work is carried out by procedure T , which at the top level has a 3-way conditional that processes the three alternative rules of T , plus their grouped final error cases.

This recursive-descent parser is designed informally as the syntactic support is purely BNF and quite simple. However, those who prefer a systematic design can transform the grammar into a machine net and proceed with the ELL methodology, which would produce a possibly different but equivalent parser. Just one notice: since this grammar is BNF , it is not necessary, when a rule has been completely scanned, to

verify the rule exit condition (which in practice is the rule prospect set), as at this point exiting is the only possibility; thus there is no check before a **return** statement. The error, if any, will be anyway discovered at a higher level in the calling procedure. To conclude, here is the full integrated syntactic-semantic recursive-descent evaluator:

program EVALUATOR

```

var result
cc = next
call S (result)
if cc ∈ {⊥}
    accept
    print (result)
else
    reject
endif
end program

```

```

procedure S (out v)
if cc ∈ { '(', '[', '•' }
    call T (0, 0; v)
    return // scanned 1
else
    error // error case
endif
end procedure

```

```

procedure T (in r, s; out v)
var v1, v2
if cc ∈ { '(' } // 1st way: scan rule 2
    cc = next
    if cc ∈ { '(', '[', '•' }
        call T (r + 1, s; v1)
        if cc ∈ { ')' }
            cc = next
            if cc ∈ { '(', '[', '•' }
                call T (r, s; v2)
                v = v1 + v2
                return // scanned 2
            endif
        endif
    endif
else if cc ∈ { '[' } // 2nd way: scan rule 3
    cc = next
    if cc ∈ { '(', '[', '•' }
        call T (r, s + 1; v1)
        if cc ∈ { ']' }
            cc = next
            if cc ∈ { '(', '[', '•' }
                call T (r, s; v2)
                v = v1 + v2
                return // scanned 3
            endif
        endif
    endif
else if cc ∈ { '•' } // 3rd way: scan rule 4
    cc = next
    v = 2 × r + 3 × s
    return // scanned 4
endif
error // grouped error cases
end procedure

```

The headers of the two procedures are completed with their input (right) and output (left) parameters (attributes). Procedure *T* has two auxiliary local variables v_1 and v_2 to compute the left attribute v . The right attributes r and s are directly computed when passed to the calls (one might use auxiliary local variables for them, too). Procedure *S* initializes (to zero) the right attributes r and s , and just receives the final value of the left attribute v . The main program has a local variable *result* to store the final value of the expression (remember that variable *cc* stores the current input character and is global). Immediately after signaling to accept the source string, it prints the final value, which represents the output of the expression evaluation, i.e., of the semantic translation. The rest of the evaluator is clear.

2. An e-commerce application computes the overall discount and the final total price for an ordered list of purchased items. Starting from the list head (on the left), the discount rate is 3% for all the purchased items the total value of which is less than or equal to 100 euros, and it grows to 5% for the following items in the list, if any.

Thus, for instance, if the list contains items with prices equal to 20, 50, 40 and 30 euros, then the discount is equal to:

$$20 \times 0.03 + 50 \times 0.03 + 40 \times 0.05 + 30 \times 0.05 = 0.6 + 1.5 + 2.0 + 1.5 = 5.6 \text{ EUR}$$

and the final total price will be $140 - 5.6 = 134.4$ EUR.

Notice that the order of the items in the list is relevant. For a list that contains items with prices equal to 20, 50, 30 and 40 euros, the discount is equal to:

$$20 \times 0.03 + 50 \times 0.03 + 30 \times 0.03 + 40 \times 0.05 = 0.6 + 1.5 + 0.9 + 2.0 = 5.0 \text{ EUR}$$

and the final total price will be $140 - 5 = 135$ EUR.

The list of purchased items is modeled by the following abstract syntax (axiom S):

$$\left\{ \begin{array}{l} 1: S \rightarrow L \\ 2: L \rightarrow i L \\ 3: L \rightarrow i \end{array} \right.$$

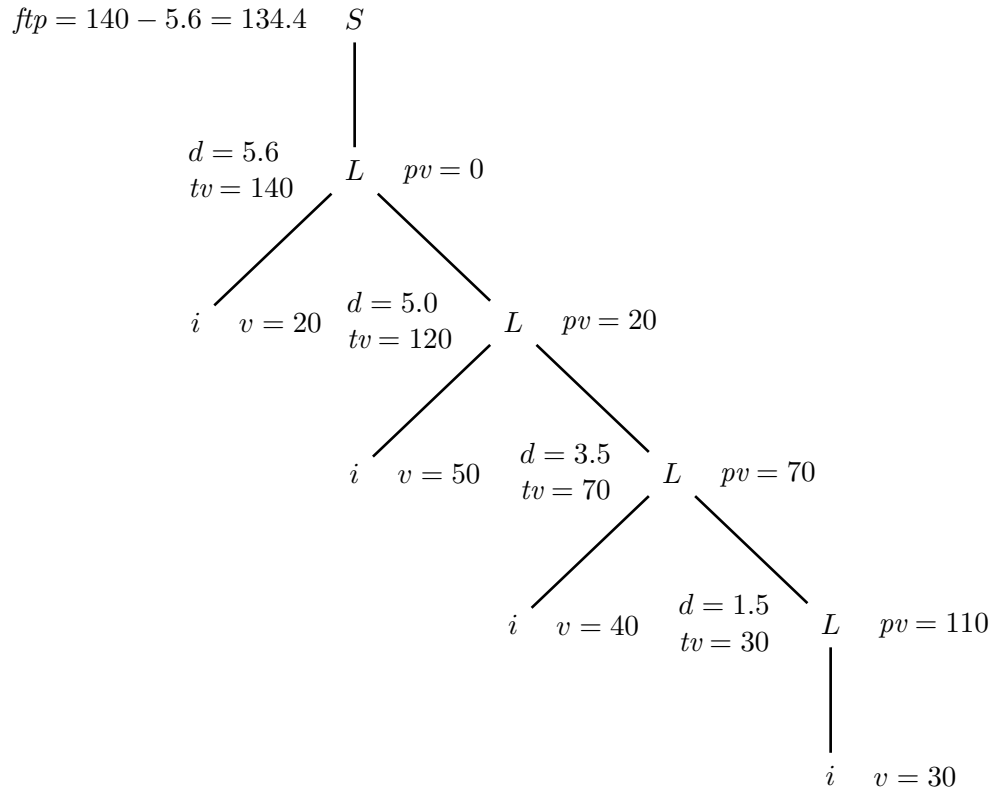
where the terminal symbol i represents a purchased item, having a real-valued attribute v (the functional attribute v is pre-computed during lexical analysis and is conventionally assumed to be right) that represents the item price in euros.

The attributes to use are already assigned. See the table below. Other attributes are unnecessary and should not be added.

attributes assigned to be used				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
right	<i>pv</i>	real	L	<i>previous value</i> : total value of the items occurring in the previous list positions
left	<i>d</i>	real	L	<i>discount</i> : discount for the items in the subtree rooted in L
left	<i>tv</i>	real	L	<i>total value</i> : total value of the items in the subtree rooted in L
left	<i>ftp</i>	real	S	<i>final total price</i> = total value of the items in the whole tree – overall discount
right	<i>v</i>	real	i	<i>value</i> : value of the item (this attribute of a terminal is pre-computed by lexical analysis)

Answer the following questions:

- (a) Write an attribute grammar, based on the above syntax, that defines the computation of the final price ftp in the tree root, as explained above. It is suggested to use the attributes v , pv , d and tv with the meaning explained (see also the table above). An example for the item list $[20, 50, 40, 30]$ is provided below.



- (b) Decorate the syntax tree of the item list $[20, 50, 30, 40]$ with the values of the attributes. Use the syntax tree prepared on the next page.
- (c) Determine if the attribute grammar is of type one-sweep and if it satisfies the L condition, and reasonably justify your answers.
- (d) (optional) Write the semantic evaluation procedure for the nonterminal L .

attribute grammar to write - question (a)
(for clarity also the terminals are indexed)

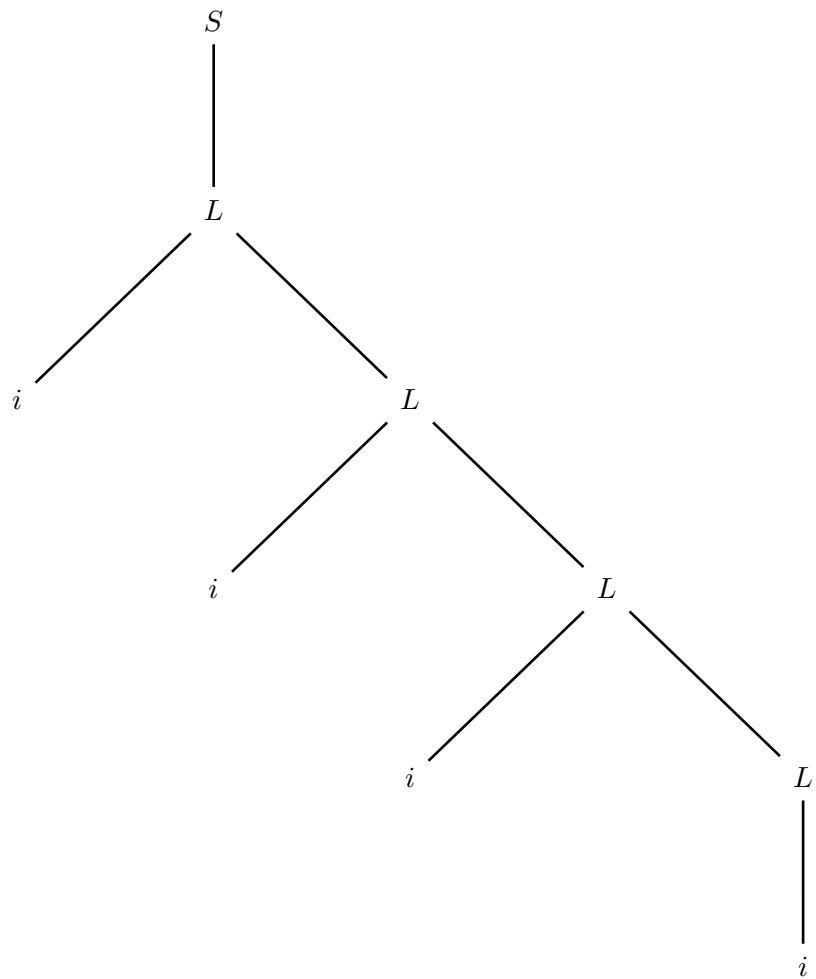
#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

1:	$S_0 \rightarrow L_1$	
----	-----------------------	--

2:	$L_0 \rightarrow i_1 L_2$	
----	---------------------------	--

3:	$L_0 \rightarrow i_1$	
----	-----------------------	--

syntax tree to decorate - question (b)



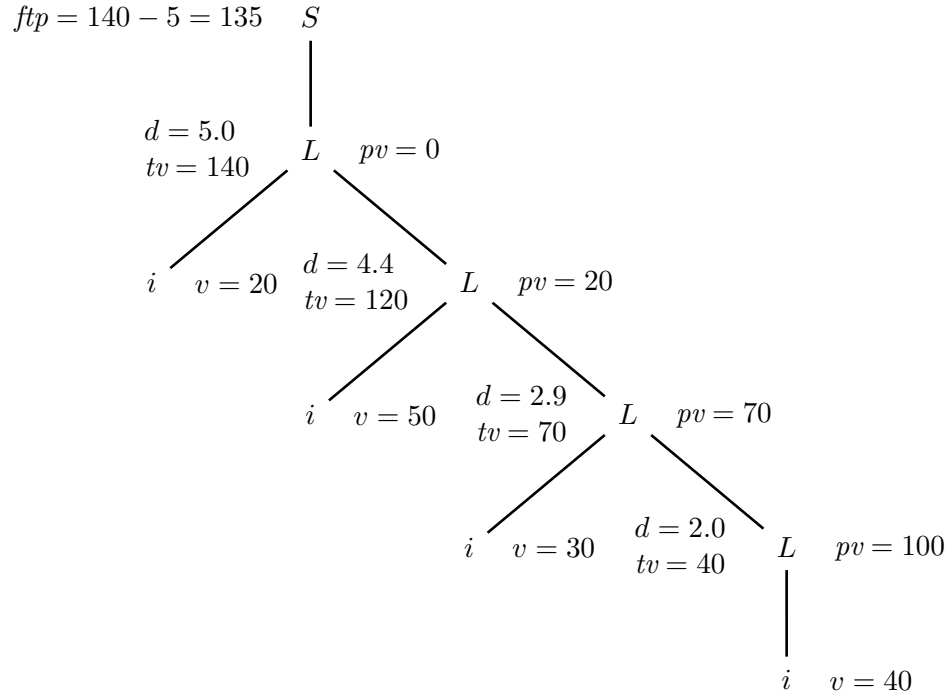
Solution

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

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow L_1$	$pv_1 = 0$ $ftp_0 = tv_1 - d_1$
2:	$L_0 \rightarrow i_1 L_2$	$pv_2 = pv_0 + v_1$ if $(pv_0 + v_1 \leq 100)$ then $d_0 = d_2 + v_1 \times 0.03$ else $d_0 = d_2 + v_1 \times 0.05$ endif $tv_0 = tv_2 + v_1$
3:	$L_0 \rightarrow i_1$	if $(pv_0 + v_1 \leq 100)$ then $d_0 = v_1 \times 0.03$ else $d_0 = v_1 \times 0.05$ endif $tv_0 := v_1$

This grammar is clearly acyclic by construction and reasonably correct.

- (b) Here is the decorated syntax tree for the proposed item price list [20, 50, 30, 40]:



As customary, the left or right attributes are placed on the left or right, respectively, of the nonterminal node they are associated with. The pre-computed (conventionally) right attribute v is similarly placed by the terminal node i .

- (c) The attribute grammar is of type one-sweep, because the right attribute pv depends only on itself (and on the pre-computed attribute v) and thus it can be computed one-way downwards, and because all the other (left) attributes depend only on themselves, i.e., on left attributes, and on the already computed attribute pv , and thus they can be computed one-way upwards (attribute v is pre-computed and does not matter). Here there are not any dependencies between brother nodes (as there are no brothers !), thus the analysis of the one-sweep condition is straightforward, as the only possible violation might come from a right attribute depending on a left attribute of the same node.

The attribute grammar is also of type L , because the syntactic order of the nonterminals in the right parts of the rules coincides with the semantic evaluation order, as the syntactic support is right-linear. Said differently, here there are not any brother nonterminal nodes, therefore the syntactic and semantic evaluation orders are trivial (only one node to process) and thus necessarily the same.

For a more formal approach, one may wish to apply the whole one-sweep and L conditions, as detailed in the textbook.

Furthermore, the syntactic support is of type $LL(2)$, as the guide sets with $k = 2$ of the alternative rules 2 and 3 are $\{ ii \}$ and $\{ i \neg \}$, respectively, and they are disjoint. Therefore it is possible to write an integrated syntactic (of the recursive descent type) and semantic parser / evaluator.

- (d) It is not difficult first to write the recursive descent syntactic analyzer and then to complete it with the attribute parameters and the semantic functions. Here is the semantic procedure for nonterminal L (the syntactic part is just sketched):

```

var  $cc1, cc2$ : lexical token - - global text window of size two

procedure  $L$  (in  $pv_0$ : real; out  $d_0, tv_0$ : real)
  var  $pv_2, d_2, tv_2$ : real    - - local vars for updating the attributes
  - - checks look-ahead ( $k = 2$ )
  case  $\langle cc1, cc2 \rangle$  of          - - syntax of type  $LL(2)$ 
    ' $ii$ ' : begin                - - (recursive) rule  $L \rightarrow i L$ 
      next                      - - computes  $v_1$  then shifts  $\langle cc1, cc2 \rangle$ 
      - - updates right attributes to pass downwards
       $pv_2 = pv_0 + v_1$ 
      call  $L(pv_2; d_2, tv_2)$  - - recursive invocation of  $L$ 
      - - updates left attributes to pass upwards
      if  $(pv_0 + v_1 \leq 100)$  then
         $d_0 = d_2 + v_1 \times 0.03$ 
      else
         $d_0 = d_2 + v_1 \times 0.05$ 
      end if
       $tv_0 = tv_2 + v_1$ 
    end
    ' $i \dashv$ ' : begin            - - (terminal) rule  $L \rightarrow i$ 
      next                      - - computes  $v_1$  then shifts  $\langle cc1, cc2 \rangle$ 
      - - updates left attributes to pass upwards
      if  $(pv_0 + v_1 \leq 100)$  then
         $d_0 = v_1 \times 0.03$ 
      else
         $d_0 = v_1 \times 0.05$ 
      end if
       $tv_0 = v_1$ 
    end
  otherwise error              - - incorrect window
  end case
end procedure

```

The function **next** is the lexical analyzer. It updates the current text window of size two $\langle cc1, cc2 \rangle$, which consists of the two lexical tokens $cc1$ and $cc2$, by shifting the window of one position (i.e., one token) to the right. The main program initializes the global variables $cc1$ and $cc2$, by placing them on the initial and second lexical tokens of the input tape, respectively. Then it invokes the axiomatic procedure (shown below) and finally verifies acceptance.

Furthermore, the attribute v_1 is pre-computed by **next** and can be passed to the parser as an output parameter of **next**. Attribute v_1 refers to token $cc1$, as token $cc2$ is a look-ahead and will be evaluated at the next shift.

For completeness here is the rest of the semantic evaluator:

<pre> procedure S (out ftp_0: real) var pv_1, d_1, tv_1: real - - checks look-ahead ($k = 1$) if ($cc1 == 'i'$) then - - updates right attributes $pv_1 = 0$ call L ($pv_1; d_1, tv_1$) - - updates left attributes $ftp_0 = tv_1 - d_1$ else error - - token $\neq i$ end if end procedure procedure next (out v_0: real) - - computes term. attrib. $v_0 = \text{value of } cc1$ - - shifts window $cc1 = cc2$ if ($cc2 \neq '+'$) read ($cc2$) end procedure </pre>	<pre> program <i>EVALUATOR</i> var ftp_0: real - - translation - - initializes window read ($cc2$) $cc1 = cc2$ if ($cc2 \neq '+'$) read ($cc2$) - - launches analysis call S (ftp_0) - - verifies acceptance if ($\langle cc1, cc2 \rangle == '+\mid'$) then accept and output ftp_0 else reject (no output) end if end program </pre>
--	---

Of course, the function **next** should be invoked in the procedure L as **next** (v_1), and the real local variable v_1 should be added to the others of L . The analyzer does not deserve any further comments. There are a few optimizations possible, as well as other possible pseudo-code variants, more or less similar.

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

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

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

$$() \left(((()) ()) \right) ()$$

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

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

The attributes to use are already assigned. See the table on the next page. Other attributes are unnecessary and should not be added.

Answer the following questions:

- Write an attribute grammar (in the table prepared on the next page), based on the above syntax, that defines the computation of the attributes no and ne in the tree root, as explained above. It is required to use the attributes illustrated in the table on the next page.
- Decorate the syntax tree of expression $() \left(((()) ()) \right) ()$ with the attribute values. Use the syntax tree prepared on the next page.
- (optional) Determine if the attribute grammar is of type one-sweep and if it satisfies the L condition, and reasonably justify your answers.

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

attribute grammar to write - question (a)

#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

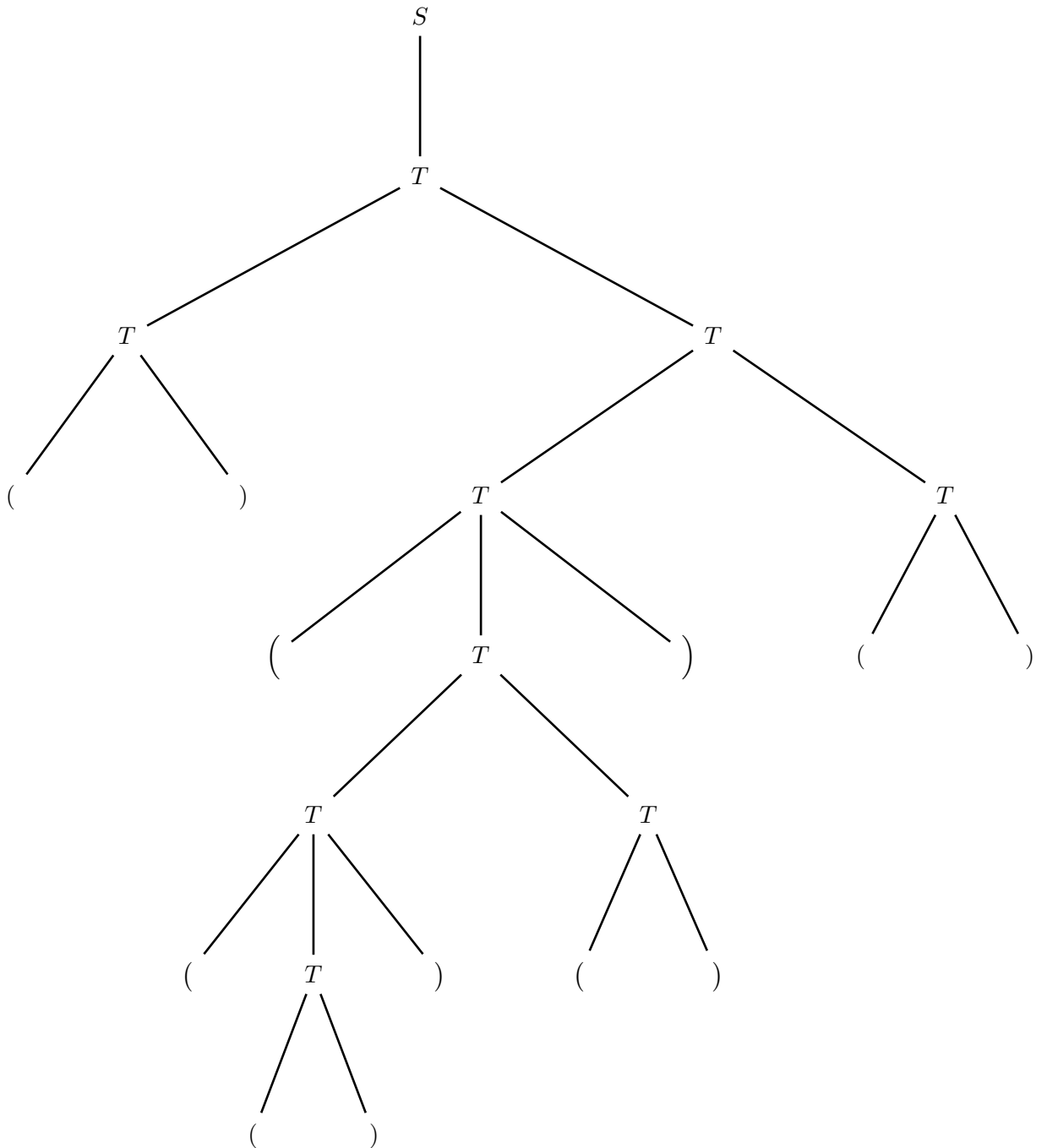
1:	$S_0 \rightarrow T_1$	
----	-----------------------	--

2:	$T_0 \rightarrow T_1 T_2$	
----	---------------------------	--

3:	$T_0 \rightarrow (T_1)$	
----	---------------------------	--

4:	$T_0 \rightarrow ()$	
----	-----------------------	--

syntax tree to decorate - question (b)



Solution

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

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow T_1$	$nep_1 := 0$ $no_0 := no_1$ $ne_0 := ne_1$
2:	$T_0 \rightarrow T_1 T_2$	$nep_1 := nep_0$ $nep_2 := nep_0$ $no_0 := no_1 + no_2$ $ne_0 := ne_1 + ne_2$
3:	$T_0 \rightarrow (T_1)$	$nep_1 := nep_0 + 1$ $no_0 := \text{if } even(nep_0) \text{ then } no_1 + 1 \text{ else } no_1$ $ne_0 := \text{if } odd(nep_0) \text{ then } ne_1 + 1 \text{ else } ne_1$
4:	$T_0 \rightarrow ()$	$no_0 := \text{if } even(nep_0) \text{ then } 1 \text{ else } 0$ $ne_0 := \text{if } odd(nep_0) \text{ then } 1 \text{ else } 0$

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

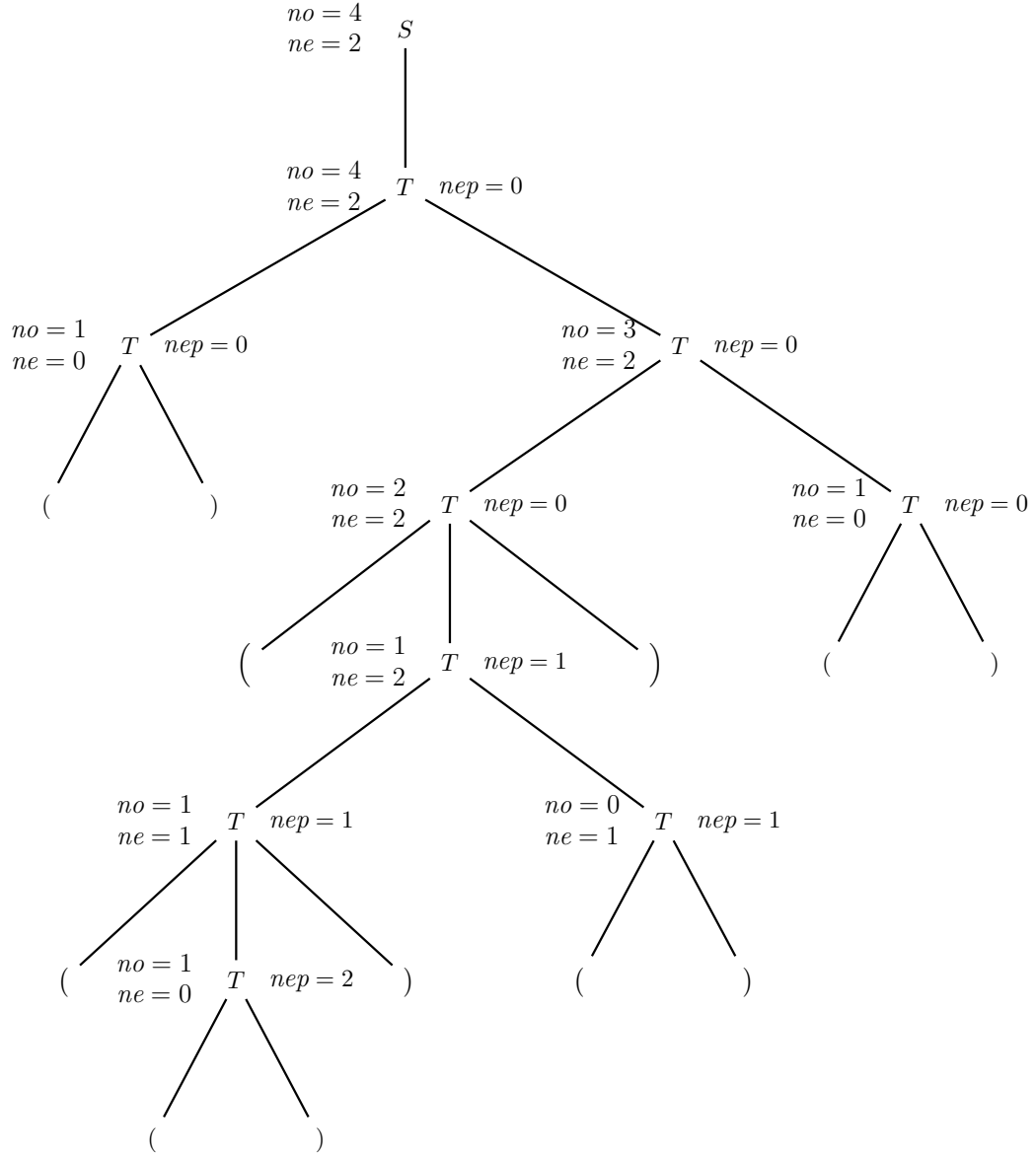
- by definition the inherited attribute *nep* counts the number of bracket pairs that enclose (are around) a nonterminal *T*; therefore on being passed top-down in the rule 3, which generates the root of a new *T*-subtree with one more bracket pair around, the attribute is incremented; and everywhere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

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

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

(b) Here is the decorated syntax tree of the expression $() \left((()) () \right) ()$:

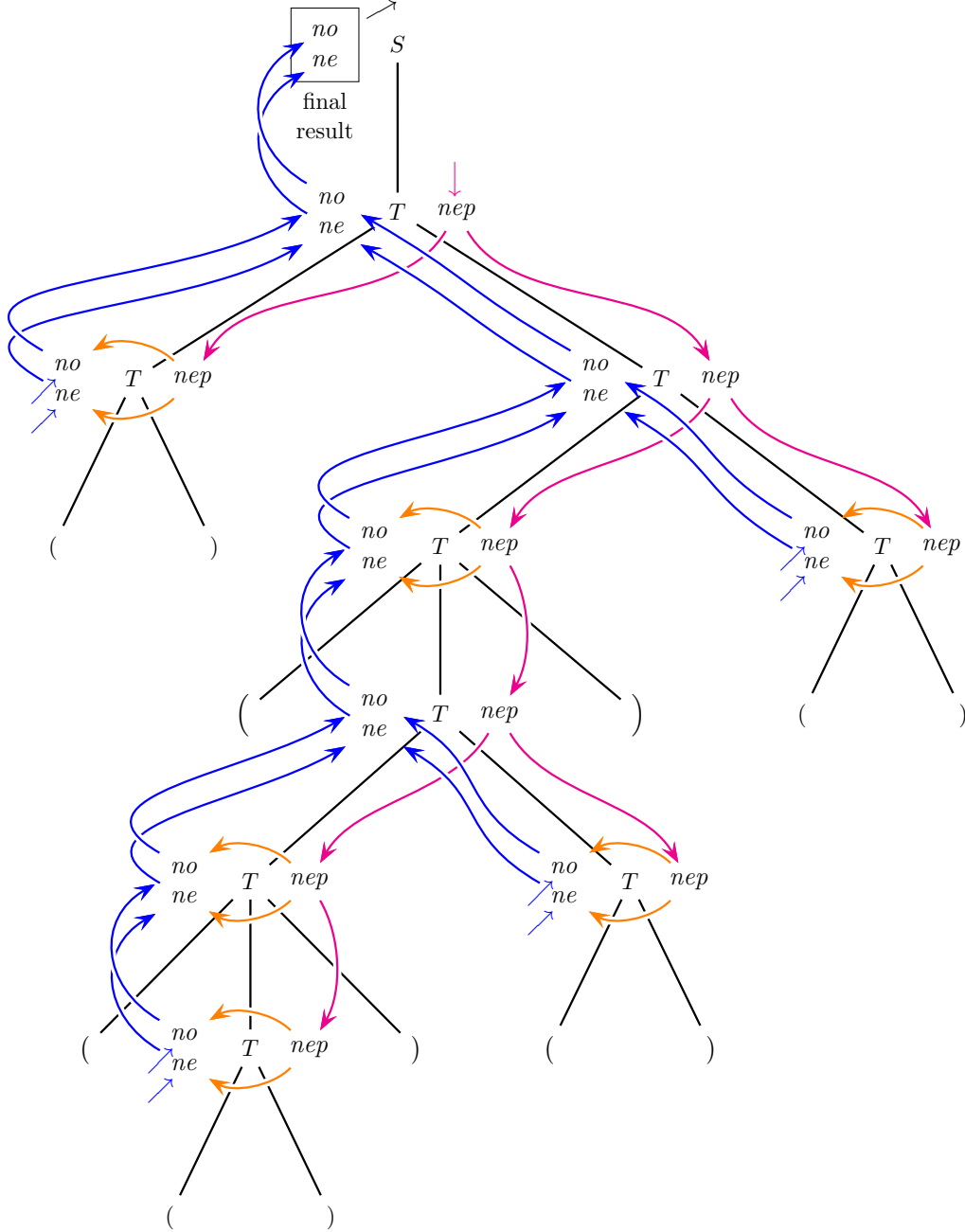
syntax tree decorated with the attribute values



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

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

sample syntax tree with attribute dependences



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

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

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

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

```

procedure  $T$  (in  $t$ ,  $nep_0$ ; out  $no_0$ ,  $ne_0$ )
var  $t_1$ ,  $t_2$ ,  $nep_1$ ,  $nep_2$ ,  $no_1$ ,  $no_2$ ,  $ne_1$ ,  $ne_2$ 
begin
  switch (rule of the root of tree  $t$ ) do
    case 2 do
       $t_1 :=$  left  $T$ -subtree of  $t$ 
       $nep_1 := nep_0$ 
      call  $T$  ( $t_1$ ,  $nep_1$ ;  $no_1$ ,  $ne_1$ )
       $t_2 :=$  right  $T$ -subtree of  $t$ 
       $nep_2 := nep_0$ 
      call  $T$  ( $t_2$ ,  $nep_2$ ;  $no_2$ ,  $ne_2$ )
       $no_0 := no_1 + no_2$ 
       $ne_0 := ne_1 + ne_2$ 
    case 3 do
       $t_1 := T$ -subtree of  $t$ 
       $nep_1 := nep_0 + 1$ 
      call  $T$  ( $t_1$ ,  $nep_1$ ;  $no_1$ ,  $ne_1$ )
      if ( $nep_0 \% 2 == 0$ ) then
         $no_0 := no_1 + 1$ 
         $ne_0 := ne_1$ 
      else
         $no_0 := no_1$ 
         $ne_0 := ne_1 + 1$ 
    case 4 do
      if ( $nep_0 \% 2 == 0$ ) then
         $no_0 := 1$ 
         $ne_0 := 0$ 
      else
         $no_0 := 0$ 
         $ne_0 := 1$ 

```

Algorithm: Semantic evaluator - I

```

procedure  $S$  (in  $t$ ; out  $no_0$ ,  $ne_0$ )
var  $t_1$ ,  $nep_1$ ,  $no_1$ ,  $ne_1$ 
begin
   $t_1 := T$ -subtree of  $t$ 
   $nep_1 := 0$ 
  call  $T$  ( $t_1$ ,  $nep_1$ ;  $no_1$ ,  $ne_1$ )
   $no_0 := no_1$ 
   $ne_0 := ne_1$ 

```

Input: syntax tree (pre-built)

Output: attrib. no , ne in the root

program SEMANTIC-EVALUATOR

var t , no , ne

begin

$t :=$ syntax tree

call S (t ; no , ne)

 print values no , ne

Algorithm: Semantic evaluator - II

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

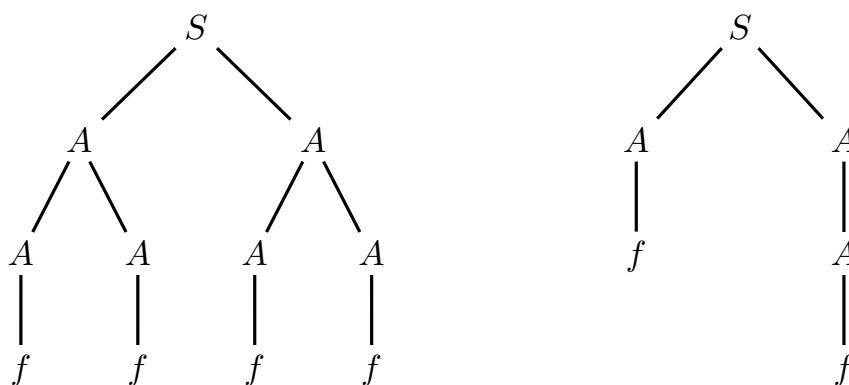
2. A binary tree is said to be *triangular* if and only if both these conditions are satisfied:

- all its leaves are at the same *depth*, where the depth of a node is the length of the path from the root to that node
- the number of leaves is equal to 2^{height} , where the height of the tree is the length of the path from the root to the deepest non-leaf node

We encode binary trees through syntax trees generated by this grammar (axiom S):

$$\left\{ \begin{array}{l} 1: S \rightarrow A A \\ 2: A \rightarrow A A \\ 3: A \rightarrow A \\ 4: A \rightarrow f \end{array} \right.$$

The figure below shows a triangular tree (left) and a non-triangular tree (right).



Notice that, according to the above definitions, the tree on the right side of the figure has one leaf at depth 2 and one leaf at depth 3, and that the height of both trees is 2.

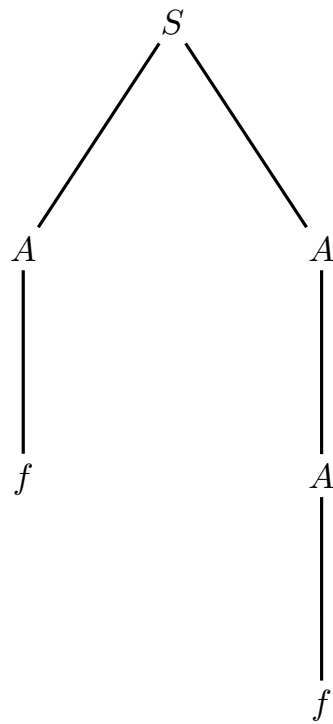
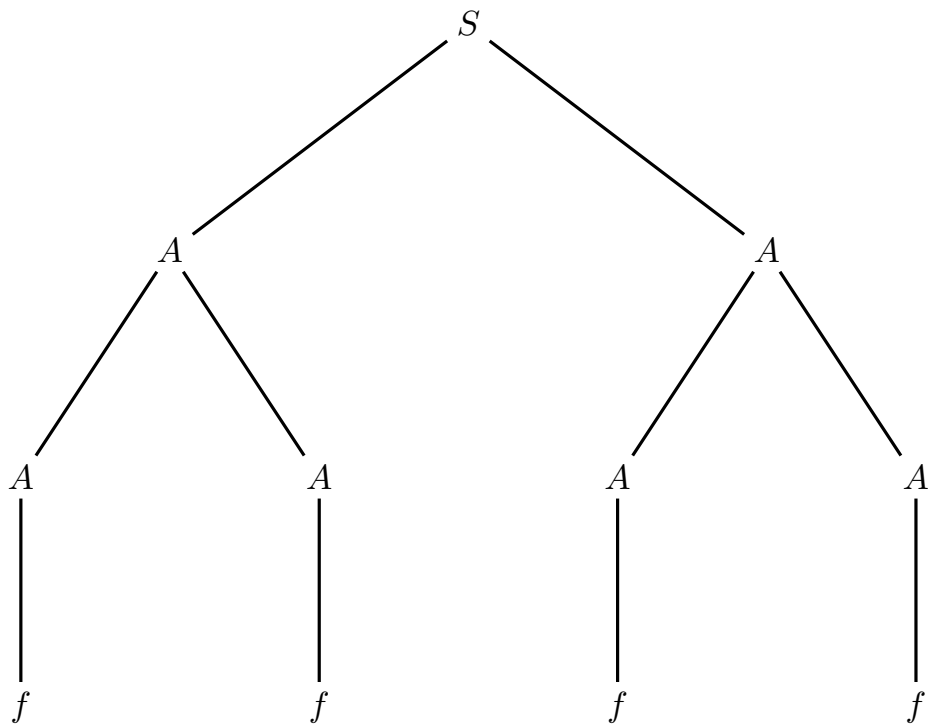
We intend to compute, in a boolean attribute *tri* of the root node of the syntax tree, the property of the tree being triangular. The attributes to use are already assigned, see the table on the next page. Other attributes are unnecessary and should not be added.

Answer the following questions:

- Write an attribute grammar (in the table prepared on the next page), based on the above syntax, that defines the attribute values. It is required to use only the attributes illustrated in the table on the next page.
- Decorate the two sample syntax trees with the appropriate attribute values. Use the two syntax trees prepared on the next page.
- (optional) Determine if the attribute grammar is of type one-sweep and if it satisfies the L condition, and reasonably justify your answers (do not use generic or tautological sentences).

attributes assigned to be used				
<i>type</i>	<i>name</i>	<i>domain</i>	<i>(non)term.</i>	<i>meaning</i>
right	<i>d</i>	integer	<i>A</i>	depth of the node from root
left	<i>nol</i>	integer	<i>S, A</i>	number of leaves of the subtree rooted at the node
left	<i>h</i>	integer	<i>S, A</i>	height of the subtree rooted at the node
left	<i>mind, maxd</i>	integer	<i>S, A</i>	minimal (respectively, maximal) depth of a leaf of the subtree rooted at the node; notice that for the left sample tree it holds $mind = maxd = 3$, while for the right one it holds $mind = 2$ and $maxd = 3$
left	<i>tri</i>	boolean	<i>S</i>	true if the tree is triangular, otherwise false

syntax trees to be decorated - question (b)



attribute grammar to write - question (a)

#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

1:	$S_0 \rightarrow A_1 A_2$	
----	---------------------------	--

2:	$A_0 \rightarrow A_1 A_2$	
----	---------------------------	--

3:	$A_0 \rightarrow A_1$	
----	-----------------------	--

4:	$A_0 \rightarrow f$	
----	---------------------	--

Solution

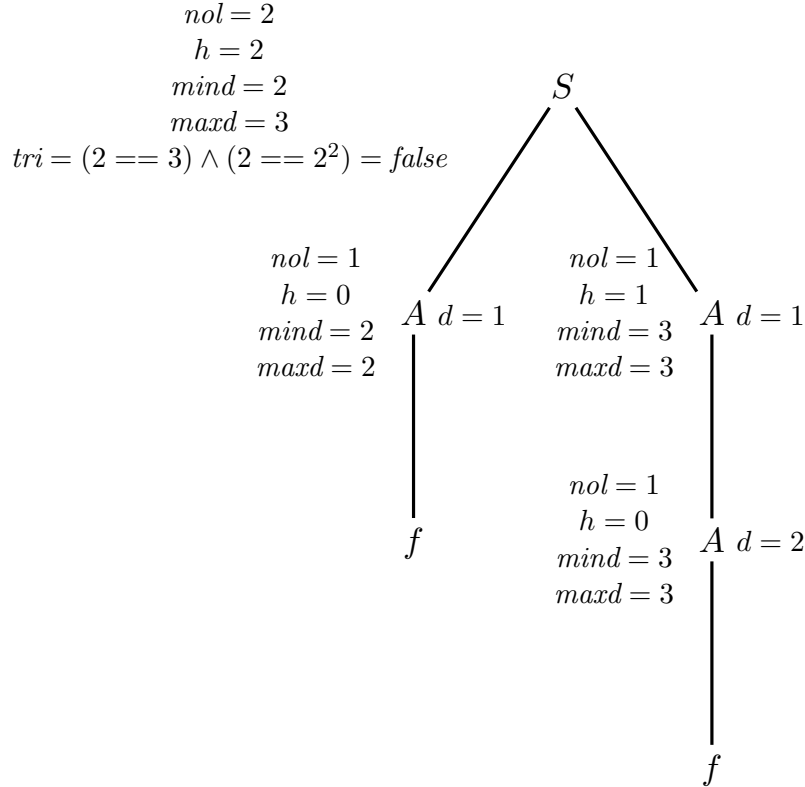
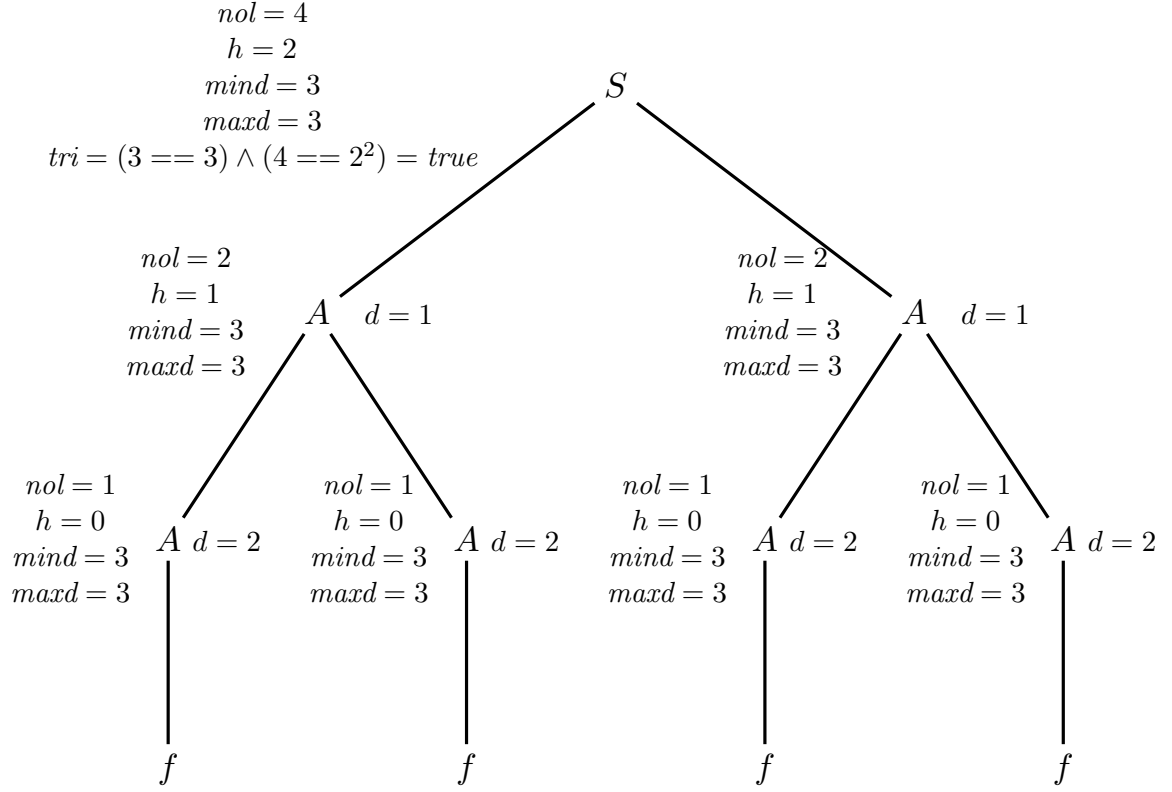
(a) Here is a working attribute grammar, which uses only the proposed attributes:

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow A_1 A_2$	$d_1, d_2 = 1$ $nol_0 = nol_1 + nol_2$ $h_0 = \max(h_1, h_2) + 1$ $mind_0 = \min(mind_1, mind_2)$ $maxd_0 = \max(maxd_1, maxd_2)$ $tri_0 = (mind_0 == maxd_0) \wedge (nol_0 == 2^{h_0})$
2:	$A_0 \rightarrow A_1 A_2$	$d_1, d_2 = d_0 + 1$ $nol_0 = nol_1 + nol_2$ $h_0 = \max(h_1, h_2) + 1$ $mind_0 = \min(mind_1, mind_2)$ $maxd_0 = \max(maxd_1, maxd_2)$
3:	$A_0 \rightarrow A_1$	$d_1 = d_0 + 1$ $nol_0 = nol_1$ $h_0 = h_1 + 1$ $mind_0 = mind_1$ $maxd_0 = maxd_1$
4:	$A_0 \rightarrow f$	$nol_0 = 1$ $h_0 = 0$ $mind_0, maxd_0 = d_0 + 1$

Attributes: d (length from root, integer, inherited) is initialized at the top and is incremented top-down; nol (number of subtree leaves, integer, synthesized) is initialized at the bottom and is accumulated bottom-up; h (max length from leaf, integer, synthesized) is initialized at the bottom and is maximized and incremented bottom-up; $mind$ and $maxd$ (min and max lengths from root to leaf, both integer and synthesized) are initialized at the bottom and are minimized and maximized bottom-up, respectively; finally, the boolean attribute tri (triangular tree, synthesized) is computed only once directly at the root node, since it is associated exclusively with the grammar axiom.

The attribute grammar G is acyclic by construction, and it is reasonably correct and equivalent to the specifications. The computation of each attribute immediately reflects its definition. Notice that a dependence between left and right attributes occurs only at the tree bottom, and that the triangularity property is computed as logical conjunction of the two conditions specified in the text.

(b) Here are the syntax trees decorated according to the semantic functions:



- (c) As already observed at point (a), the attribute grammar is acyclic and reasonably correct according to the specifications. Therefore it is computable, as already shown by simulation in a couple of different cases at point (b).

Furthermore this attribute grammar is of type one-sweep: in fact, all the attributes are synthesized, except attribute d , which is inherited and depends only on itself from top to bottom, and which contributes to the other attributes — particularly to $mind$ and $maxd$ — only at the tree bottom (i.e., in the rule 4) and nowhere else. Therefore all the attributes can be computed first top-down (attribute d) and then bottom-up (all the others).

Concerning the L condition, notice that there are not any attributes of a child node that depend on attributes of its brother nodes in the same rule. Therefore, this attribute grammar immediately satisfies the L condition, since all the sibling graphs of its rules are empty. Thus, the brother nodes in a rule can be semantically evaluated in any order, not necessarily from left to right.

Anyway, the semantic evaluation of this attribute grammar cannot be integrated with deterministic parsing, as the syntactic support is ambiguous (see the two-sided recursive rule $A \rightarrow A A$) and thus deterministic parsing is impossible.

2. Consider the two-level lists defined by the following abstract syntax (axiom A):

$$\left\{ \begin{array}{ll} 1: A \rightarrow \varepsilon & \text{--- 1-st level list} \\ 2: A \rightarrow a A \\ 3: A \rightarrow a B A \\ 4: B \rightarrow b & \text{--- 2-nd level list} \\ 5: B \rightarrow B b \end{array} \right.$$

Notice that the 2-nd level lists, i.e., the substrings of letters b , are optional.

We want to verify that the two-level list is ordered from left to right by strictly increasing length of the 2-nd level lists of elements b , if any. For instance, both the two-level lists below are syntactically valid, but:

- the two-level list $a b a a b b$ is semantically *correct*
- the two-level list $a b b a a b$ is semantically *incorrect*

Attributes permitted to use (no other attributes are allowed):

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
<i>len</i>	left	integer	B	length of a 2-nd level list, i.e., a substring of letters b
<i>lim</i>	left	integer	A	length of the <i>shortest</i> 2-nd level list in the tree rooted at A (conventionally = 0 if the tree rooted at A does not include any 2-nd level list)
<i>corr</i>	left	boolean	A	true if the two-level list is semantically correct as explained, else false
<i>err</i>	right	boolean	B	true if a 2-nd level list, i.e., a substring of letters b , violates the semantic correctness of the two-level list, else false — this attribute is for question (c)

Answer the following questions:

- Write an attribute grammar (in the table prepared on the next page), based on the above syntax and using only the first three attributes listed above (*len*, *lim* and *corr*), that computes the value of the attribute *corr* in the tree root.
- Test the attribute grammar of point (a) by decorating with the attribute values the syntax tree of the correct two-level list $a b a a b b$.
- (optional) Extend the attribute grammar of point (a) to define the value of the attribute *err* of B , so to allow for printing a diagnostic at the left end of each 2-nd level list that violates the correctness of the whole two-level list. Is the extended attribute grammar of type one-sweep ? (justify your answer)

attribute grammar to write - question (a)

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
<i>len</i>	left	integer	B	length of a 2-nd level list
<i>lim</i>	left	integer	A	length of the shortest 2-nd level list
<i>corr</i>	left	boolean	A	true if the two-level list is correct

#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

1:	$A_0 \rightarrow \varepsilon$	
----	-------------------------------	--

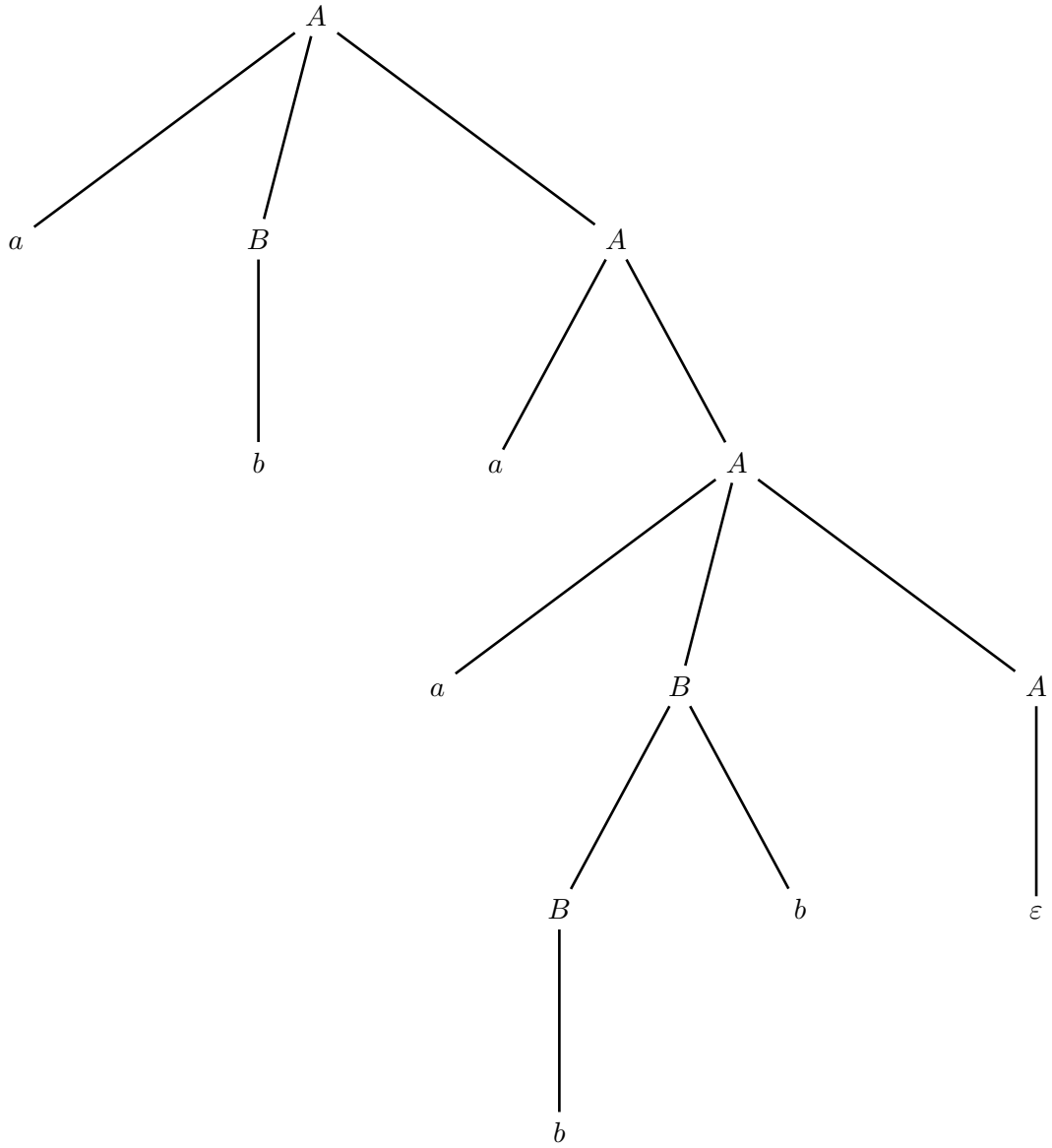
2:	$A_0 \rightarrow a A_1$	
----	-------------------------	--

3:	$A_0 \rightarrow a B_1 A_2$	
----	-----------------------------	--

4:	$B_0 \rightarrow b$	
----	---------------------	--

5:	$B_0 \rightarrow B_1 b$	
----	-------------------------	--

syntax tree to be decorated - question (b)



attribute grammar to extend - question (c)

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
the same attributes as for question (a) plus the attribute <i>err</i> below				
<i>err</i>	right	boolean	<i>B</i>	true if a 2-nd level list violates the correctness of the two-level list — for (c)

#	<i>syntax</i>	<i>semantics</i>
---	---------------	------------------

1: $A_0 \rightarrow \varepsilon$

2: $A_0 \rightarrow a A_1$

3: $A_0 \rightarrow a B_1 A_2$

4: $B_0 \rightarrow b$

5: $B_0 \rightarrow B_1 b$

Solution

(a) Here is the attribute grammar:

#	<i>syntax</i>	<i>semantics</i>
1:	$A_0 \rightarrow \varepsilon$	$lim_0 = 0$ $corr_0 = true$
2:	$A_0 \rightarrow a A_1$	$lim_0 = lim_1$ $corr_0 = corr_1$
3:	$A_0 \rightarrow a B_1 A_2$	if ($lim_2 \neq 0$) then $lim_0 = \min(len_1, lim_2)$ $corr_0 = (len_1 < lim_2) \wedge corr_2$ else $lim_0 = len_1$ $corr_0 = corr_2$ endif
4:	$B_0 \rightarrow b$	$len_0 = 1$
5:	$B_0 \rightarrow B_1 b$	$len_0 = len_1 + 1$

A different approach might extend the domain of the attribute *lim* and include a special value ∞ , so the initialization in the rule 1 could be changed into $lim_0 = \infty$, the test on *lim* in the rule 2 could be removed and the assignments therein simplified into $lim_0 = \min(len_1, lim_2)$ and $corr_0 = (len_1 < lim_2) \wedge corr_2$, that is, the **else** branch of the conditional would become useless. This way:

#	<i>syntax</i>	<i>semantics</i>
1:	$A_0 \rightarrow \varepsilon$	$lim_0 = \infty$ $corr_0 = true$
2:	$A_0 \rightarrow a A_1$	$lim_0 = lim_1$ $corr_0 = corr_1$
3:	$A_0 \rightarrow a B_1 A_2$	$lim_0 = \min(len_1, lim_2)$ $corr_0 = (len_1 < lim_2) \wedge corr_2$
4:	$B_0 \rightarrow b$	$len_0 = 1$
5:	$B_0 \rightarrow B_1 b$	$len_0 = len_1 + 1$

Of course, in this way the domain of attribute *lim* becomes: $\text{integer} \cup \{\infty\}$.

(b) See Figure 3.

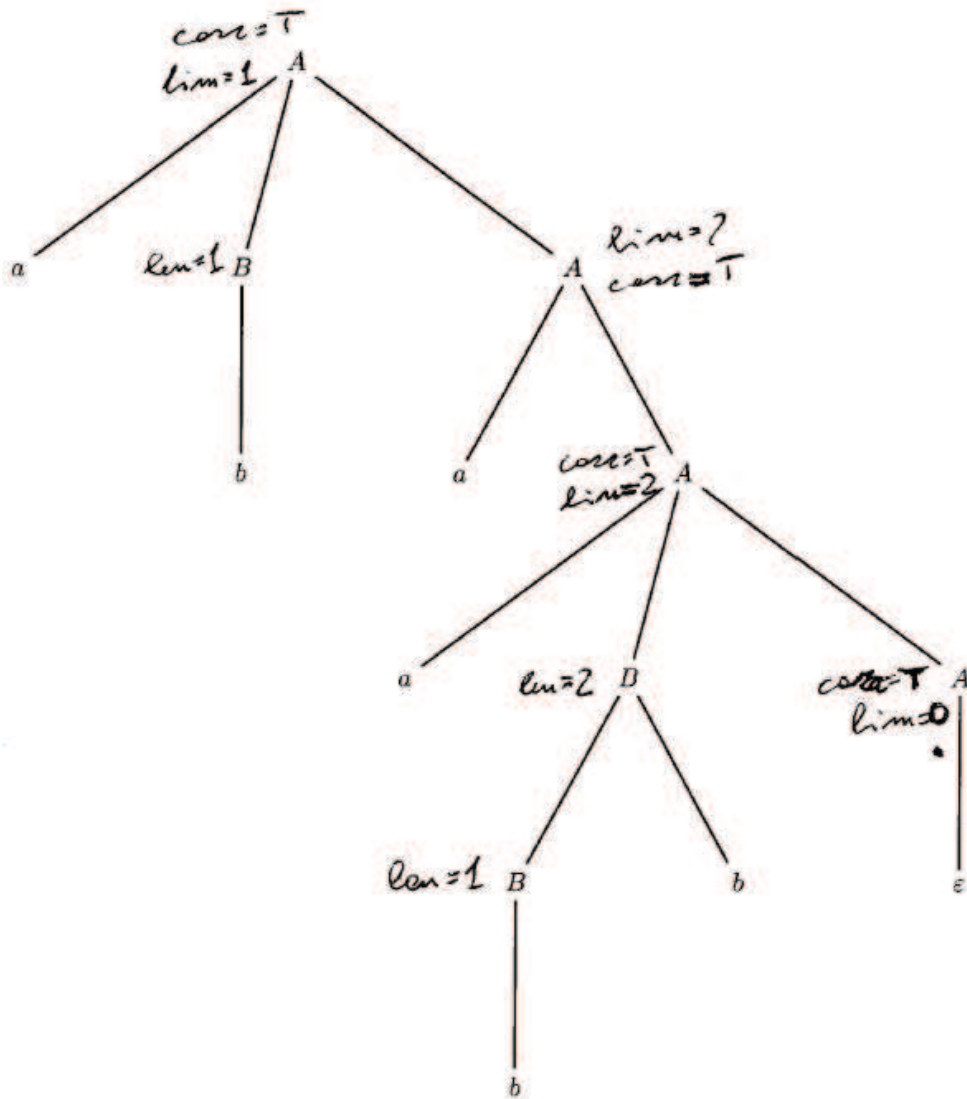


Figure 3: Decorated tree.

(c) Here is the extended attribute grammar:

#	<i>syntax</i>	<i>semantics</i>
1:	$A_0 \rightarrow \varepsilon$	$lim_0 = 0$ $corr_0 = true$
2:	$A_0 \rightarrow a A_1$	$lim_0 = lim_1$ $corr_0 = corr_1$
3:	$A_0 \rightarrow a B_1 A_2$	if ($lim_2 \neq 0$) then $err_1 = (len_1 \geq lim_2)$ $lim_0 = \min(len_1, lim_2)$ $corr_0 = (len_1 < lim_2) \wedge corr_2$ — or $corr_0 = \neg err_1 \wedge corr_2$ else $err_1 = false$ $lim_0 = len_1$ $corr_0 = corr_2$ endif
4:	$B_0 \rightarrow b$	$len_0 = 1$
5:	$B_0 \rightarrow B_1 b$	$err_1 = err_0$ $len_0 = len_1 + 1$

It might be changed as explained before.

The extended grammar is not of type one-sweep, as the inherited attribute err depends on a synthesized attribute, namely len , of the same node, namely the node B in the rule 3.

2. Consider the following abstract syntax (axiom S), over the two-letter alphabet $\{ a, b \}$:

$$\left\{ \begin{array}{l} 1: S \rightarrow a S b S \\ 2: S \rightarrow a S \\ 3: S \rightarrow S b \\ 4: S \rightarrow \varepsilon \end{array} \right.$$

A sample valid string is: $a^3 b^2$ (see also the syntax tree on the next pages).

Assume that a syntax tree is given for each string. We want to compute, in the root node of any tree, the following values:

- the depth (distance from tree root) of the *deepest* leaf node labeled a
- the depth (distance from tree root) of the *least deep* leaf node labeled b

It is assumed that, if the tree does not include any leaf node of a given kind, then the required depth value is conventionally set equal to 0.

Attributes permitted to use (no other attributes are allowed):

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
da	left	integer	S	distance, from the current node S , of the <i>deepest</i> leaf node of type a in the subtree rooted at the current node conventionally set equal to 0 if the subtree does not include any leaf node of type a
db	left	integer	S	distance, from the current node S , of the <i>least deep</i> leaf node of type b in the subtree rooted at the current node conventionally set equal to 0 if the subtree does not include any leaf node of type b

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

- Write an attribute grammar G_a , based on the above syntax and using only the attribute da , that computes in the tree root the value of the depth of the *deepest* leaf node labeled a , or 0 if the tree does not include any node labeled a .
- Decorate the tree of string $a^3 b^2$, prepared on the next page, with the values of attribute da at every non-leaf node.
- Write another attribute grammar G_b , so as to compute in the tree root the value of the depth of the *least deep* leaf node labeled b , or 0 if the tree does not include any node labeled b , based on the above syntax and using only the attribute db .
- Decorate the tree of string $a^3 b^2$, prepared on the next page, with the values of attribute db at every non-leaf node.

for exercise 4.2 see the next pages
here space for continuation if necessary – all exercises

#	<i>syntax</i>	<i>semantics of G_a – question (a)</i>
---	---------------	---

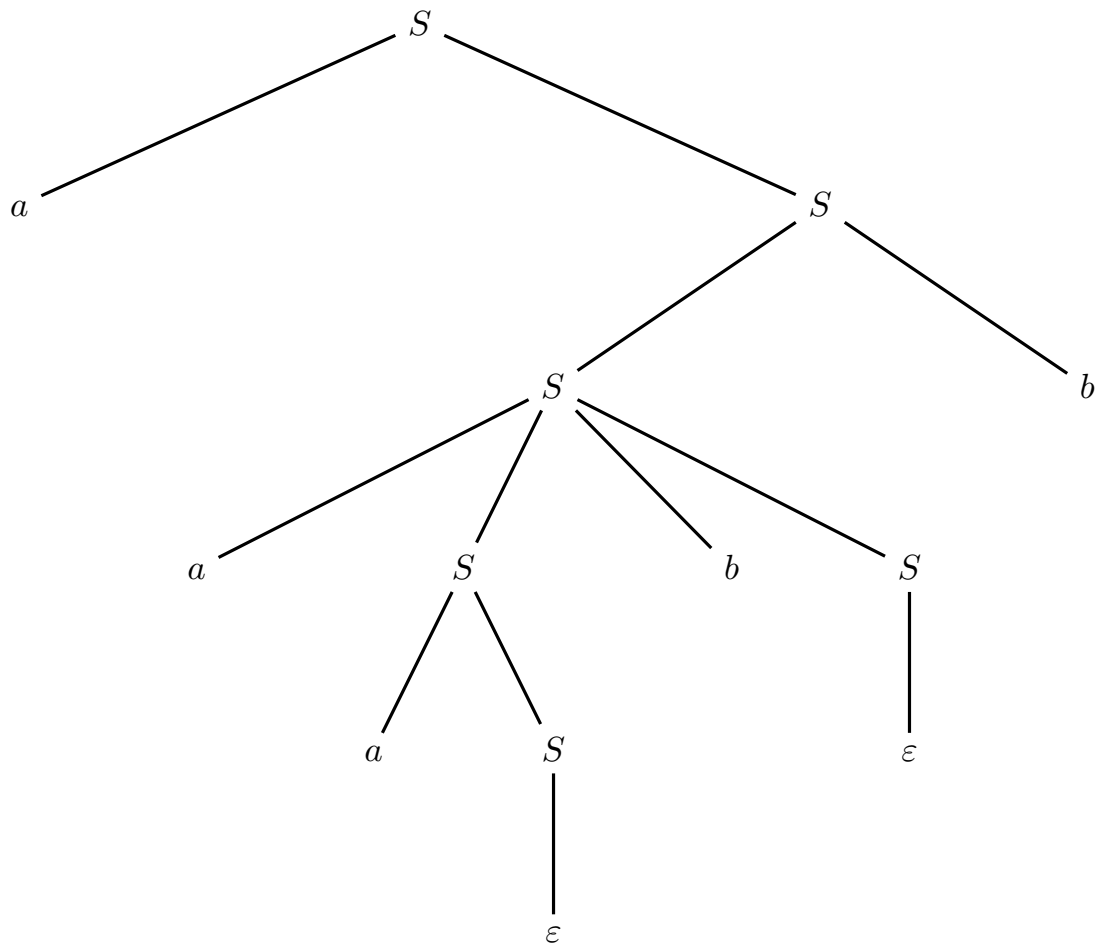
1:	$S_0 \rightarrow a S_1 b S_2$	
----	-------------------------------	--

2:	$S_0 \rightarrow a S_1$	
----	-------------------------	--

3:	$S_0 \rightarrow S_1 b$	
----	-------------------------	--

4:	$S_0 \rightarrow \varepsilon$	
----	-------------------------------	--

please decorate this tree with attribute da – question (b)



#	<i>syntax</i>	<i>semantics of G_b – question (c)</i>
---	---------------	---

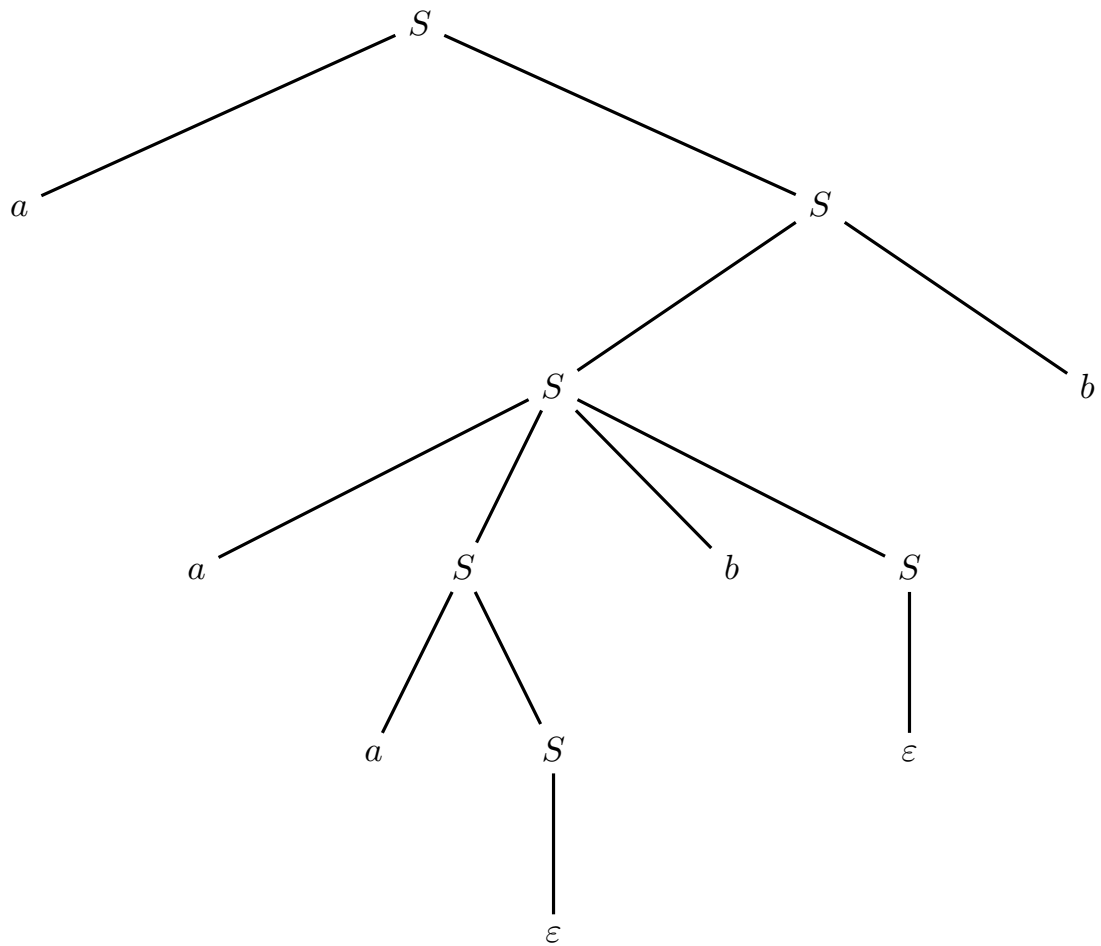
1:	$S_0 \rightarrow a S_1 b S_2$	
----	-------------------------------	--

2:	$S_0 \rightarrow a S_1$	
----	-------------------------	--

3:	$S_0 \rightarrow S_1 b$	
----	-------------------------	--

4:	$S_0 \rightarrow \varepsilon$	
----	-------------------------------	--

please decorate this tree with attribute db – question (d)



space for continuation if necessary – all exercises

Solution

(a) Here is the attribute grammar G_a , which uses only the attribute da :

#	<i>syntax</i>	<i>semantics of G_a – question (a)</i>
1:	$S_0 \rightarrow a S_1 b S_2$	if ($da_1 = 0$) and ($da_2 = 0$) then $da_0 := 1$ else if ($da_1 \geq da_2$) then $da_0 := da_1 + 1$ else $da_0 := da_2 + 1$ endif
2:	$S_0 \rightarrow a S_1$	if ($da_1 = 0$) then $da_0 := 1$ else $da_0 := da_1 + 1$ endif
3:	$S_0 \rightarrow S_1 b$	if ($da_1 = 0$) then $da_0 := 0$ else $da_0 := da_1 + 1$ endif
4:	$S_0 \rightarrow \varepsilon$	$da_0 := 0$

The attribute grammar G_a is purely synthesized. The semantic functions are coded extensively so as to be self-evident according to the specifications.

However, some coding optimization is possible. For instance, at rule 2 cancel the two-way conditional and set $da_0 = da_1 + 1$ instead, and similarly at rule 1 cancel the three-way conditional and equivalently set $da_0 = \max(da_1, da_2) + 1$. Here is the optimized attribute grammar G_a :

#	<i>syntax</i>	<i>semantics of G_a – optimized</i>
1:	$S_0 \rightarrow a S_1 b S_2$	$da_0 := \max(da_1, da_2) + 1$
2:	$S_0 \rightarrow a S_1$	$da_0 := da_1 + 1$
3:	$S_0 \rightarrow S_1 b$	if ($da_1 = 0$) then $da_0 := 0$ else $da_0 := da_1 + 1$ endif
4:	$S_0 \rightarrow \varepsilon$	$da_0 := 0$

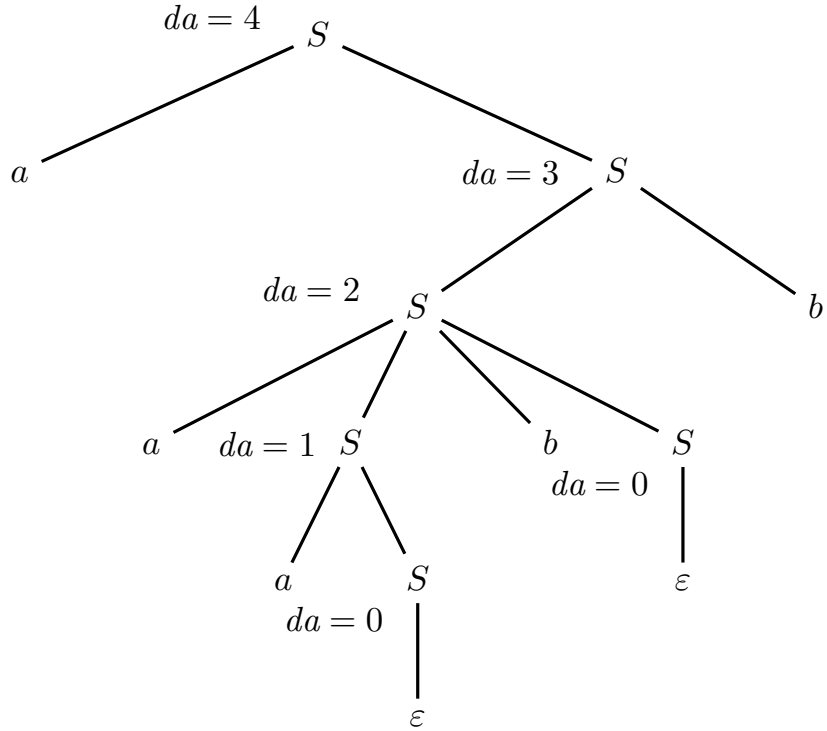
For a compact coding, one can resort to the conditional assignment of the C

language, to recode the two-way conditional in the semantic function of rule 3:

#	<i>syntax</i>	<i>semantics of G_a – recoded</i>
1:	$S_0 \rightarrow a S_1 b S_2$	$da_0 := \max(da_1, da_2) + 1$
2:	$S_0 \rightarrow a S_1$	$da_0 := da_1 + 1$
3:	$S_0 \rightarrow S_1 b$	$da_0 := (da_1 = 0) ? 0 : da_1 + 1$
4:	$S_0 \rightarrow \varepsilon$	$da_0 := 0$

There may be other coding ways or optimizations.

(b) Here is the syntax tree decorated with attribute da :



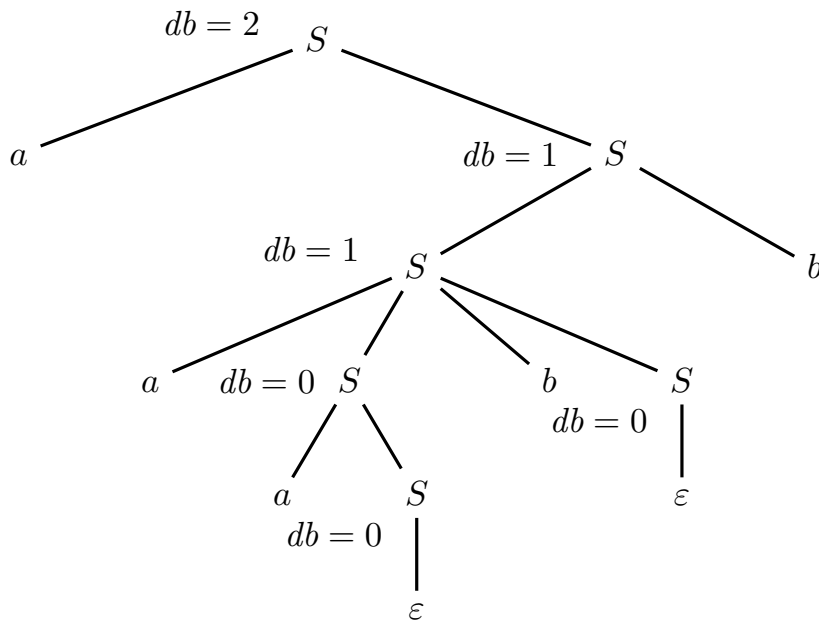
The computation on the tree is coherent with the attribute grammar G_a , and it suffices to prove the correctness of the grammar.

(c) Here is the attribute grammar G_b , which uses only the attribute db :

#	<i>syntax</i>	<i>semantics of G_b – question (c)</i>
1:	$S_0 \rightarrow a S_1 b S_2$	$db_0 := 1$
2a:	$S_0 \rightarrow a S_1$	if ($db_1 = 0$) then $db_0 := 0$ else $db_0 := db_1 + 1$ endif
2b:	equivalently to 2a	$db_0 := (db_1 = 0) ? 0 : db_1 + 1$
3:	$S_0 \rightarrow S_1 b$	$db_0 := 1$
4:	$S_0 \rightarrow \varepsilon$	$db_0 := 0$

The attribute grammar G_b is purely synthesized. The semantic resemblance of the rules 2 (version a or b) in G_b and 3 in G_a (all versions) is evident. There may be optimizations or a more compact coding, though it seems unlikely due to the extreme simplicity and compactness of the current solution.

(d) Here is the syntax tree decorated with attribute *db*:



The computation on the tree is coherent with the attribute grammar G_b , and it suffices to prove the correctness of the grammar.

For completeness, notice that both attribute grammars G_a and G_b are of type one-sweep, as they are purely synthesized. The syntactic support is however ambiguous, e.g., string $a\ b$ has three syntax trees. Therefore the computation of the attributes of both G_a and G_b cannot be integrated with syntax analysis.

2. Consider the grammar below (axiom S), which generates a (possibly empty) two-level list. Each sublist consists of letters a (possibly none) and ends with one letter b .

$$\left\{ \begin{array}{l} 1: S \rightarrow X \\ 2: X \rightarrow A X \\ 3: X \rightarrow \varepsilon \\ 4: A \rightarrow a A \\ 5: A \rightarrow b \end{array} \right.$$

A sample two-level list, with three sublists, is:

$$b a b a^2 b$$

See also the syntax tree on the next pages.

The $length \geq 0$ of a sublist is the number of elements, i.e., letters a , in the sublist.

We want to decide whether in a list the length of the sublists increases, from left to right, exactly by one, starting from an empty sublist (length 0). If this property is satisfied, a boolean attribute e assumes value *true*, else *false*. For the sample list above, the attribute e is *true*. If the whole list is empty, the attribute e is conventionally *true*.

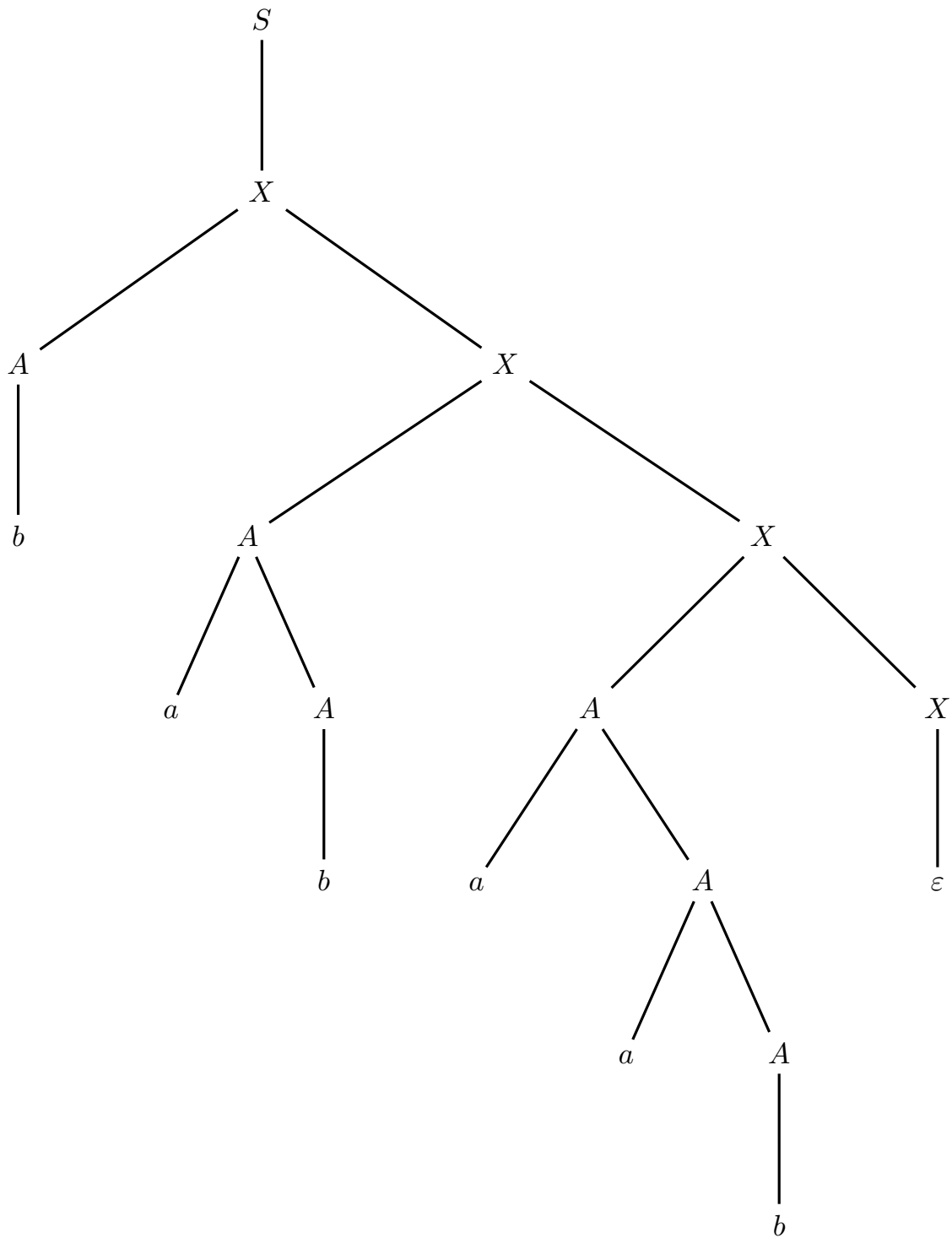
Attributes permitted to use (no other attributes are allowed):

<i>name</i>	<i>type</i>	<i>domain</i>	<i>symbol</i>	<i>meaning</i>
l	left	integer	A	$length \geq 0$ of a sublist
n	right	integer	X	required length of the sublist immediately appended to a node X
e	left	boolean	X, S	<i>true</i> if the sublists appended to a node X , or to the root S , have a length that increases by one from left to right (starting from 0), else <i>false</i>

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

- Decorate the tree of the sample string $b a b a^2 b$, prepared on the next page, with the attribute values. Pay attention to the type of each attribute (left or right).
- Write an attribute grammar, based on the syntax above and using only the permitted attributes (do not change the attribute types), that computes in the tree root the correctness attribute e for the whole list. The attribute grammar must be of type one-sweep. Please argue that your grammar is so.
- (optional) Write the procedure of the semantic analyzer (which exists as the attribute grammar must be of type one-sweep) for nonterminal X . Is it possible to integrate syntactic and semantic analysis ? Please explain your answer.

syntax tree to be decorated – question (a)



syntax *semantics* – question (b)

1: $S_0 \rightarrow X_1$

2: $X_0 \rightarrow A_1 X_2$

3: $X_0 \rightarrow \varepsilon$

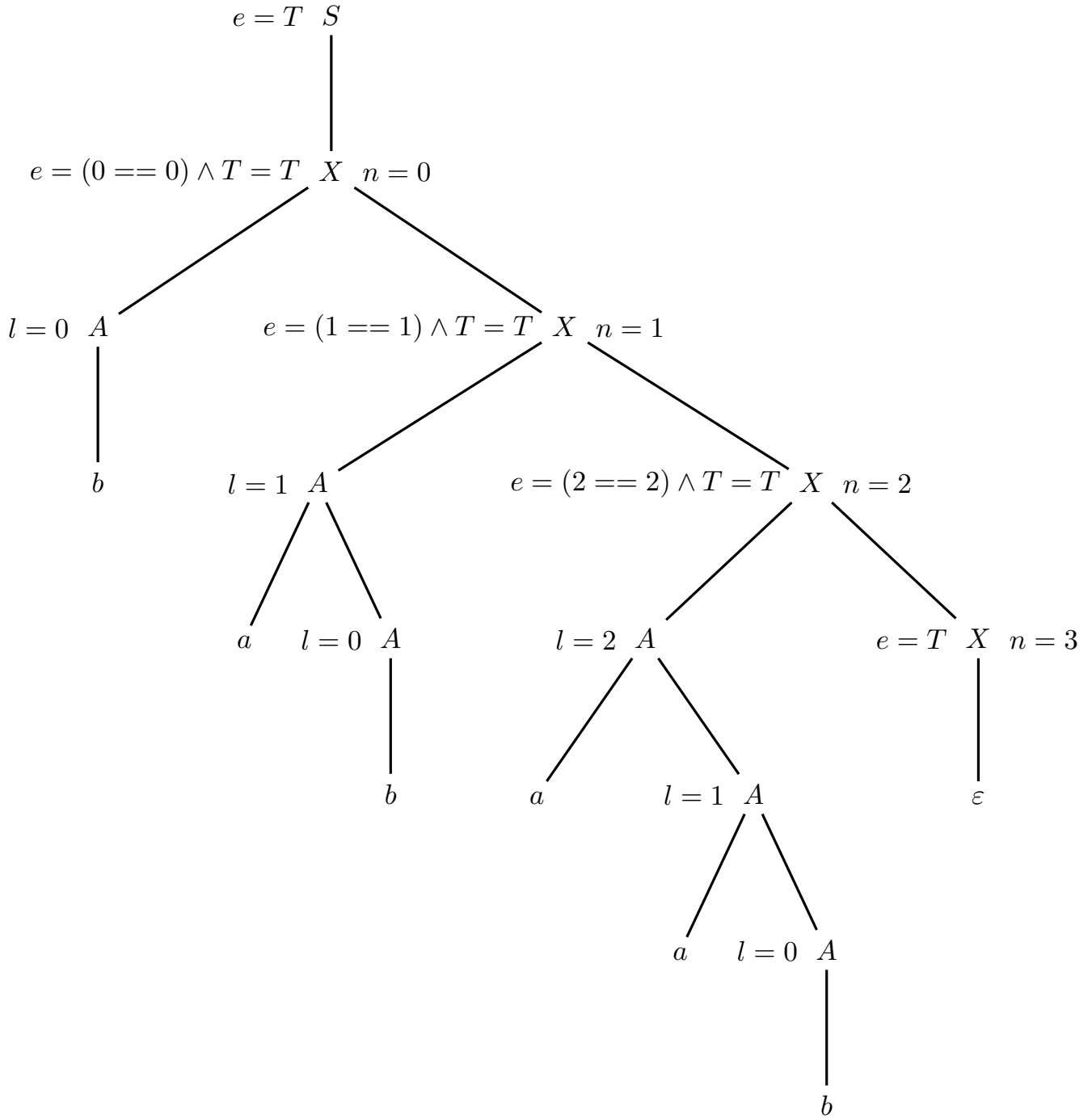
4: $A_0 \rightarrow a A_1$

5: $A_0 \rightarrow b$

semantic procedure of nonterminal X – question (c)

Solution

- (a) Here is the decorated syntax tree, computed by means of the second grammar version of point (b):



As customary, the left and the right attributes are written on the left and on the right of the tree node they refer to, respectively.

- (b) The idea is that attribute l computes the length of each sublist, that attribute n computes the expected length of each sublist for the whole list to be correct, and that attribute e evaluates the equality between the actual and expected lengths of each sublist, and computes the logical product of all such equality checks.

Attribute l is computed upwards, as $l_0 = l_1 + 1$, starting from letter b with $l_0 = 0$. Attribute n is computed downwards, as $n_2 = n_0 + 1$, starting from $n_1 = 0$ in the root. Attribute e is computed upwards as $e_0 = e_2 \wedge (l_1 == n_0)$ (and as $e_0 = e_1$ in the root), starting from $e_0 = \text{true}$ in ε . Here is the complete attribute grammar:

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow X_1$	$n_1 = 0$ $e_0 = e_1$
2:	$X_0 \rightarrow A_1 X_2$	$n_2 = n_0 + 1$ if $(l_1 == n_0)$ then $e_0 = e_2$ else $e_0 = \text{false}$ endif
3:	$X_0 \rightarrow \varepsilon$	$e_0 = \text{true}$
4:	$A_0 \rightarrow a A_1$	$l_0 = l_1 + 1$
5:	$A_0 \rightarrow b$	$l_0 = 0$

or more compactly:

#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow X_1$	$n_1 = 0$ $e_0 = e_1$
2:	$X_0 \rightarrow A_1 X_2$	$n_2 = n_0 + 1$ notation (a) $e_0 = (l_1 == n_0) \wedge e_2$ notation (b) $e_0 = (l_1 == n_0) ? e_2 : \text{false}$
3:	$X_0 \rightarrow \varepsilon$	$e_0 = \text{true}$
4:	$A_0 \rightarrow a A_1$	$l_0 = l_1 + 1$
5:	$A_0 \rightarrow b$	$l_0 = 0$

The attribute grammar is acyclic, hence it is correct. Furthermore, it is one-sweep: the right attributes depend only on right ones in their parent node, and the left attributes depend only on right ones in their own node and on left ones in their child nodes. Thus the attribute dependencies can be trivially satisfied by the one-sweep tree visit order.

In fact, attribute n (right) depends only on itself in its parent node (see rule 2), attribute l (left) depends only on itself in a child node (see rule 4), and attribute e (left) depends only on a left attribute (namely l) in a child node and on itself in its own node (see rule 2 and, limitedly, rule 1).

- (c) Yes, it is possible to integrate semantic and syntactic analysis, as the syntax is evidently of type $LL(1)$ (alternative rules have disjoint guide sets), the semantic is of type one-sweep and the right attribute depends only on itself (so there is not any need to examine the brother graph).

The semantic procedure of nonterminal X is rather simple. Here it is:

```

procedure  $X$  ( $n_0$ : in;  $e_0$ : out)
var  $l_1, n_2, e_2$            // varlocs for the attribs of the child nodes
switch rule do
  case 2:  $X \rightarrow A X$  do
    call  $A(l_1)$ 
     $n_2 = n_0 + 1$            // inherited update
    call  $X(n_2, e_2)$ 
    if ( $l_1 == n_0$ ) then  $e_0 = e_2$            // synthesized update
    else  $e_0 = false$            // synthesized update
  case 3:  $X \rightarrow \varepsilon$  do  $e_0 = false$            // synthesized update
  otherwise do error

```

For completeness, here are the other two procedures, first for nonterminal A :

```

procedure  $A$  ( $l_0$ : out)
var  $l_1$            // varlocs for the attribs of the child nodes
switch rule do
  case 3:  $A \rightarrow a A$  do
    call  $A(l_1)$ 
     $l_0 = l_1 + 1$            // synthesized update
  case 4:  $A \rightarrow b$  do  $l_0 = 0$            // synthesized initialization
  otherwise do error

```

and then for the tree root:

```

procedure  $S$  ( $e_0$ : out)
var  $n_1, e_1$            // varlocs for the attribs of the child nodes
switch rule do
  case 1:  $S \rightarrow X$  do
     $n_1 = 0$            // inherited initialization
    call  $X(n_1, e_1)$ 
     $e_0 = e_1$            // synthesized update
  otherwise do error

```

The arguments for passing the subtrees are here omitted (see the textbook). The semantic analyzer is invoked starting from the axiomatic procedure S .