FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 8 July 2015 - Part Theory

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

NAME:

MATRICOLA: SIGNATURE:

INSTRUCTIONS - READ CAREFULLY:

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

1 Regular Expressions and Finite Automata 20%

1. Consider the following regular expressions R_1 and R_2, over the two-letter alphabet $\Sigma = \{ a, b \}$:

$$R_1 = (a\ a \mid b)^* \qquad\qquad R_2 = \Sigma^*\ a\ b\ \Sigma^*$$

Answer the following questions:

(a) Draw the two deterministic automata A_1 and A_2 equivalent to the two regular expressions R_1 and R_2, by using the Berry-Sethi method at least for A_2.

(b) Check if the two automata A_1 and A_2 are minimal, and if necessary minimize them.

(c) Find a deterministic automaton A_3 equivalent to the regular expression R_3 below, extended with the intersection operator and the complement one ($\neg$):

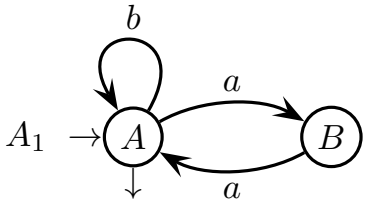
$$R_3 = R_1 \cap \neg R_2$$

and if necessary clean the automaton A_3 by removing the useless states.

(d) (optional) Find, in a systematic way and in any case justifying your answer, a regular expression R_4 that is equivalent to the extended regular expression R_3 and that contains only the standard operators of concatenation, union and star (or cross).

Solution

(a) Automaton A_1 is really straightforward:

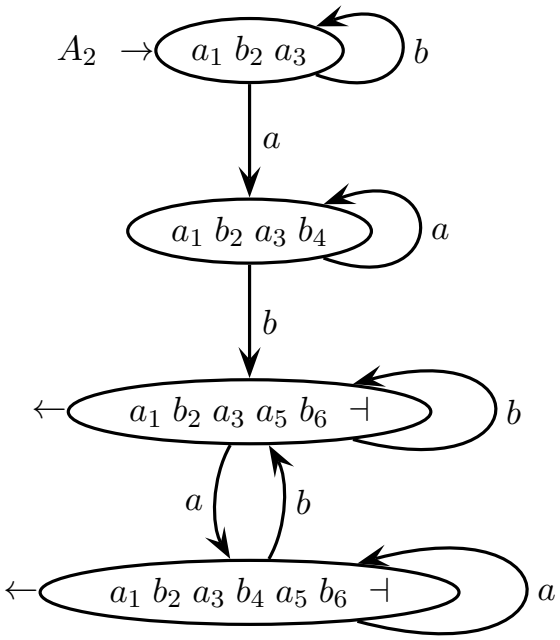


and is deterministic. Berry-Sethi for automaton A_2:

$$R_{2\#} = (\, a_1 \;\mid\; b_2 \,)^*\; a_3\; b_4\; (\, a_5 \;\mid\; b_6 \,)^*$$

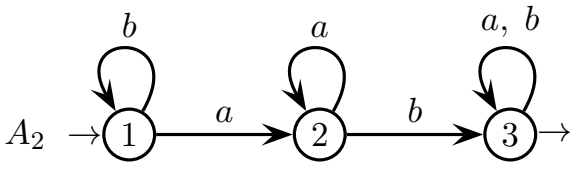
Initials and followers:

initials	$a_1\ b_2\ a_3$
generators	*followers*
a_1	$a_1\ b_2\ a_3$
b_2	$a_1\ b_2\ a_3$
a_3	b_4
b_4	$a_5\ b_6\ \dashv$
a_5	$a_5\ b_6\ \dashv$
b_6	$a_5\ b_6\ \dashv$



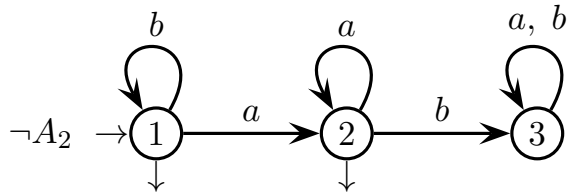
Automaton A_2 is deterministic, of course.

(b) Automaton A_1 is minimal, since the two states are final and non-final, respectively. Automaton A_2 is not minimal: the two final states have the same outgoing labels and the corresponding arcs have the same destination states. So the final states are undistinguishable and can be grouped. The two non-final states are distinguishable by the input letter b, which sends them to a non-final and a to final state. Minimal form of A_2:

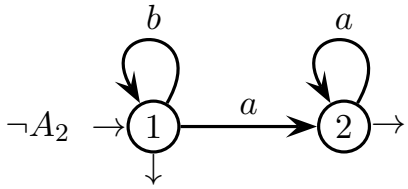


where the final state 3 groups the two original final states.

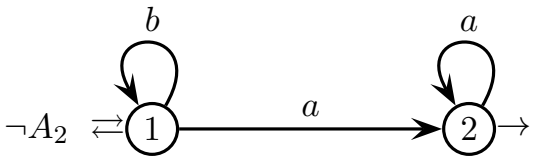
(c) Notice that automaton A_2 is deterministic and already in the complete form (no need to introduce the error state). So it can be immediately complemented by switching the final and non-final states, as follows:

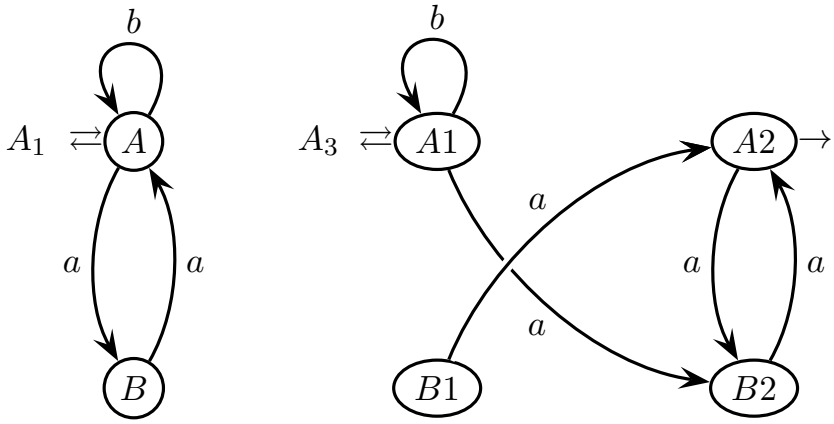


Then it can be cleaned by removing state 3, which has become not post-accessible:

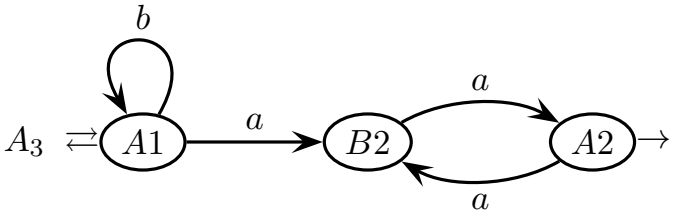


The automaton $\neg A_2$ in clean form is also deterministic and evidently minimal. Now the product automaton A_3 can be constructed:



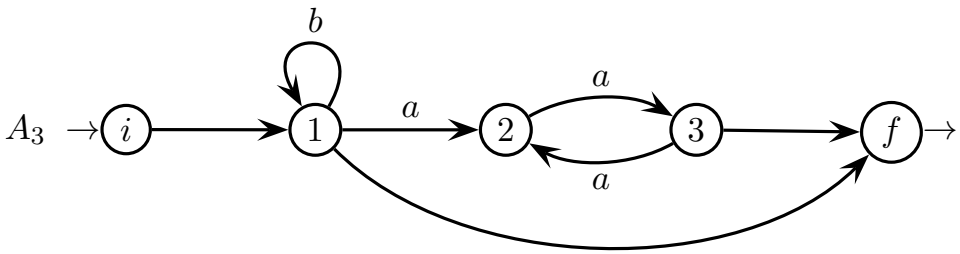


Automaton A_3 is deterministic, of course. State $B1$ is not accessible and can be removed. Clean form of A_3:

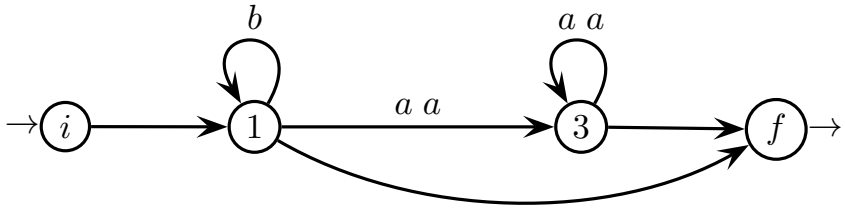


Automaton A_3 in clean form is also evidently minimal.

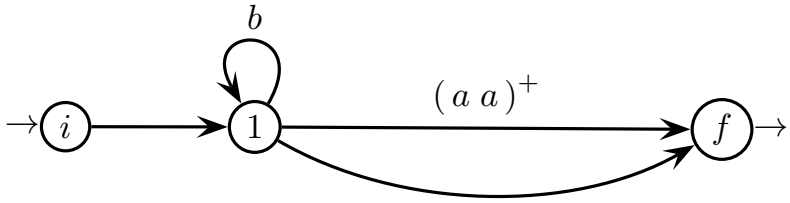
(d) We can proceed by node elimination (Brzozowski) from automaton A_3. Unique initial and final states:



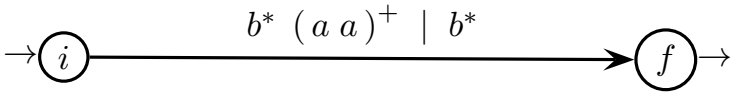
Eliminate node 2:



Eliminate node 3:



Eliminate node 1:



The standard regular expression R_4 equivalent to R_3 is the following:

$$R_4 = b^* \ (\, a\ a \,)^+ \ \mid \ b^* = b^* \ \Big(\ (\, a\ a \,)^+ \ \mid \ \varepsilon \Big) = b^* \ (\, a\ a \,)^*$$

Of course, different formulations may exist, as standard as the one above.

2 Free Grammars and Pushdown Automata 20%

1. Answer the following questions:

   (a) Write a *BNF* grammar G_1 for the language L_1 below:

   $$L_1 = \{\ a^i\ b^j\ c^k \mid \quad i \neq j\ \} \qquad \text{with } i,\ j,\ k \geq 0$$

   Write also a similar *BNF* grammar G_2 for the language L_2 below:

   $$L_2 = \{\ a^i\ b^j\ c^k \mid \quad j \neq k\ \} \qquad \text{with } i,\ j,\ k \geq 0$$

   (b) Based on the grammar G_1 of language L_1, draw the syntax trees of the two strings:

   $$a\,a\,b\,c \qquad \text{and} \qquad a\,b\,b$$

   (c) (optional) Considering that the language $L = \{\ a^n\ b^n\ c^n \mid \quad n \geq 1\ \}$ is *not* context-free (i.e., $L \notin CF$), use the fact that $L_1 \in CF$ and $L_2 \in CF$ (as proved by your answer to question (a) above) to argue that the family of context-free languages CF is *not* closed under set complement.

Solution

(a) Here is a *BNF* grammar G_1 (axiom S) of language L_1, commented aside:

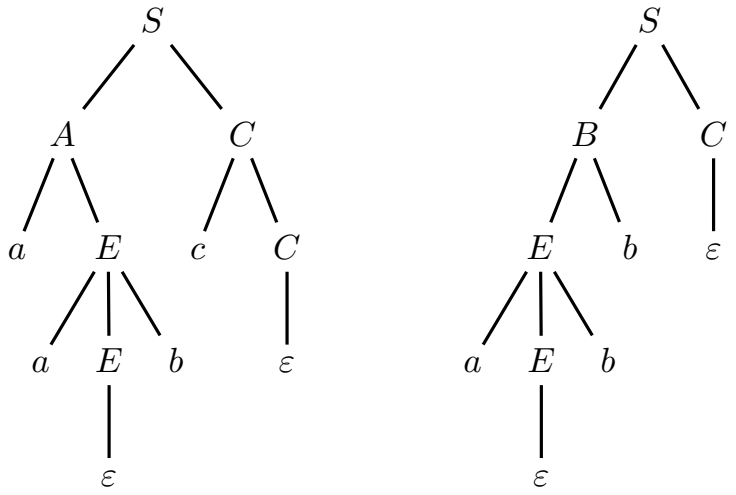
$$G_1 \begin{cases} S \;\to\; A\,C \;\mid\; B\,C & //L\,(S) = a^+\,a^n\,b^n\,c^* \;\cup\; a^n\,b^n\,b^+\,c^* \\ A \;\to\; a\,A \;\mid\; a\,E & //L\,(A) = a^+\,a^n\,b^n \\ B \;\to\; B\,b \;\mid\; E\,b & //L\,(B) = a^n\,b^n\,b^+ \\ C \;\to\; c\,C \;\mid\; \varepsilon & //L\,(C) = c^* \\ E \;\to\; a\,E\,b \;\mid\; \varepsilon & //L\,(E) = a^n\,b^n \end{cases}$$

with $n \geq 0$. Ratio: A = excess ≥ 1 of a's, B = excess ≥ 1 of b's, E = equal number ≥ 0 of a's and b's, C = any number ≥ 0 of c's.
Exploit the same idea as for grammar G_1. Here is a *BNF* grammar G_2 (axiom S) of language L_2:

$$G_2 \begin{cases} S \;\to\; A\,B \;\mid\; A\,C \\ A \;\to\; a\,A \;\mid\; \varepsilon \\ B \;\to\; b\,B \;\mid\; b\,E \\ C \;\to\; C\,c \;\mid\; E\,c \\ E \;\to\; b\,E\,c \;\mid\; \varepsilon \end{cases}$$

The ratio is of the same kind as before, swapping the roles of letters a and c.
The explanation of the various nonterminals reasonably proves that both grammars G_1 and G_2 are not ambiguous, though this was not requested.

(b) Here are the syntax trees of grammar G_1 for the strings $a\,a\,b\,c$ and $a\,b\,b$:



which reasonably demonstrate the correctness of grammar G_1 (similarly for G_2).

(c) Notice that the complement language $\neg L$ is equal to the union of these four component languages, namely:

$$\neg\, L = \{\, \varepsilon \,\} \;\cup\; L_1 \;\cup\; L_2 \;\cup\; \neg\, (\, a^*\; b^*\; c^* \,)$$

The first component provides the empty string, the second and third ones provide the strings of type $a^i\, b^j\, c^k$ where at least one of the exponents i, j and k is different from the others, and the fourth one provides the strings where the letters a, b and c are not alphabetically ordered. It is fairly evident that their union makes the complement $\neg\, L$ of the language L with three equal exponents.

These four component languages are all context-free, and actually the first and fourth ones are even regular (the fourth one is the complement of a regular language, so it is regular as well). Hence the complement language $\neg\, L$ is context-free as well, because the *CF* family is closed under set union. But if the family *CF* were closed under set complement, then it would also hold $L \in CF$, which is not the case as recalled before, since L is the language with three equal exponents. Whence the *CF* family is not closed under set complement.

We give a few more working grammars, for completeness. Here is a different *BNF* grammar for language L_1:

$$G_1' \begin{cases} S & \to & E\; C \\ E & \to & a\; E\; b \;\mid\; A \;\mid\; B \\ A & \to & a\; A \;\mid\; a \\ B & \to & B\; b \;\mid\; b \\ C & \to & c\; C \;\mid\; \varepsilon \end{cases}$$

which "pumps" the exceeding letters a or b from inside of the structure $a^n\, b^n$, instead of concatenating them on the left or right thereof, respectively. A similar one can be written for language L_2.

If it were permitted to use an *EBNF* form, then the grammar of language L_1 could be shortened in this way:

$$G_1'' \begin{cases} S & \to & (\, a^+\; E \;\mid\; E\; b^+ \,)\; c^* \\ E & \to & a\; E\; b \;\mid\; \varepsilon \end{cases}$$

or

$$G_1''' \begin{cases} S & \to & E\; c^* \\ E & \to & a\; E\; b \;\mid\; a^+ \;\mid\; b^+ \end{cases}$$

depending on the mechanism chosen for adding the exceeding letters a or b, and similarly for language L_2.

2. Consider a data description language that can model the structure of an alphabetical text. The text is modeled by breaking it into substrings, which can be arranged into lists or arrays, and can be possibly given a name. Such a language has these features:

   - A text is modeled as a (possibly empty) list of elements.
   - A list is enclosed in graph brackets "{" and "}", and the elements are separated by a comma ",".
   - An element has a name, which is an identifier schematized by the terminal id, followed by a colon ":" and then by a value.
   - A value may be a string, an array or, recursively, a (possibly empty) list.
   - A string is a non-empty sequence of letters (uppercase and lowercase), decimal digits and dots ".".
   - An array is a (non-empty) list of (unnamed) values; the array is enclosed in square brackets "[" and "]", and the values are separated by a semicolon ";".

   Example: the following text

   ```
   xy.zAabc..0121alphaAXO
   ```

   is modeled by the data description language as:

   ```
   {
     one : xy.zA,
     two : [
       a; bc..0; 12
     ],
     next : {
       inner : [
         1; alpha
       ]
     },
     last : AXO
   }
   ```

   Write a grammar, in the extended form (*EBNF*) and not ambiguous, that generates such a data description language.

Solution

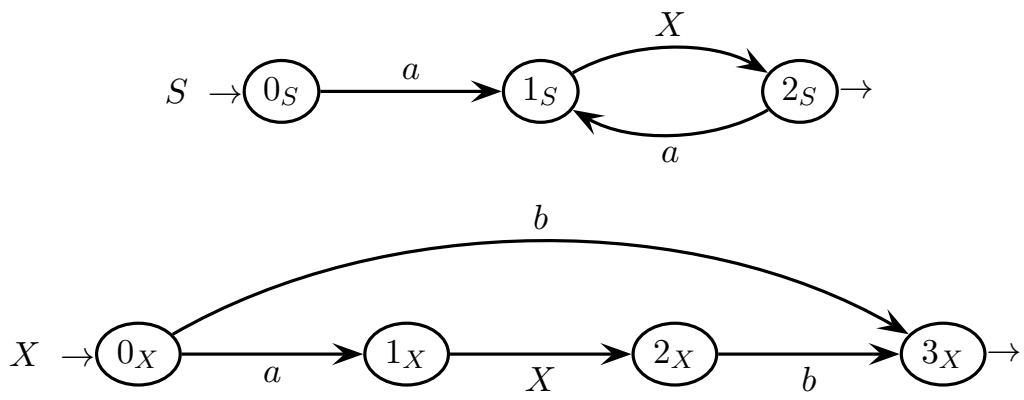
Here is the *EBNF* grammar requested, not ambiguous (axiom TEXT):

$$
\begin{aligned}
\langle \mathsf{TEXT} \rangle &\rightarrow \text{`\{'} \; [\, \langle \mathsf{ELEM\_LST} \rangle \,] \; \text{`\}'} \\
\langle \mathsf{ELEM\_LST} \rangle &\rightarrow \langle \mathsf{ELEMENT} \rangle \; \left(\text{`,'} \; \langle \mathsf{ELEMENT} \rangle \right)^{*} \\
\langle \mathsf{ELEMENT} \rangle &\rightarrow \mathsf{id} \; \text{`:'} \langle \mathsf{VALUE} \rangle \\
\langle \mathsf{VALUE} \rangle &\rightarrow \langle \mathsf{STRING} \rangle \mid \langle \mathsf{ARRAY} \rangle \mid \langle \mathsf{TEXT} \rangle \\
\langle \mathsf{STRING} \rangle &\rightarrow \left((a \ldots z) \mid (A \ldots Z) \mid (0 \ldots 9) \mid \text{`.'} \right)^{+} \\
\langle \mathsf{ARRAY} \rangle &\rightarrow \text{`[' } \langle \mathsf{VAL\_LST} \rangle \text{ `]'} \\
\langle \mathsf{VAL\_LST} \rangle &\rightarrow \langle \mathsf{VALUE} \rangle \; \left(\text{`;'} \; \langle \mathsf{VALUE} \rangle \right)^{*}
\end{aligned}
$$

Do not confuse between the square brackets [and] when used as metasymbols to mean optionality, and when used as terminal symbols to mean array delimiters; in the latter usage they are enclosed in apices, as customary to clearly specify the terminals. A notation like $(a \ldots z)$ is a set containing the elements of an alphabetical (or numerical) interval. Lists may be empty, while arrays may not. The terminal id schematizes a whole identifier and, as said before, it is left unexpanded. The grammar is evidently recursive, as lists and arrays may be nested into each other at an arbitrary depth. It is not ambiguous, as reasonably proved by being designed using only unambiguous constructs. Different formulations of this grammar may exist, of course.

3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the grammar G below, modeled as a net of recursive machines (axiom S):



Answer the following questions:

(a) Draw the complete *ELR* pilot of grammar G, say if the grammar is of $ELR(1)$ type and justify your answer.

(b) Examine the *ELR* pilot of grammar G, say if the grammar is of $ELL(1)$ type and justify your answer. Furthermore, compute all the guide sets (on all the arcs and the exit arrows).

(c) Simulate the *ELR* (shift-reduce) analysis of the sample string below:

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

which belongs to the language $L(G)$. Use the table prepared on the next pages.

(d) (optional) Simulate the *ELL* (predictive) analysis of the sample string below:

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

which belongs to the language $L(G)$. Use the table prepared on the next pages.

---

simulation table for the *ELR* analysis to be completed
(the number of rows is not significant - I_0, I_1, ... are the pilot macro-states)

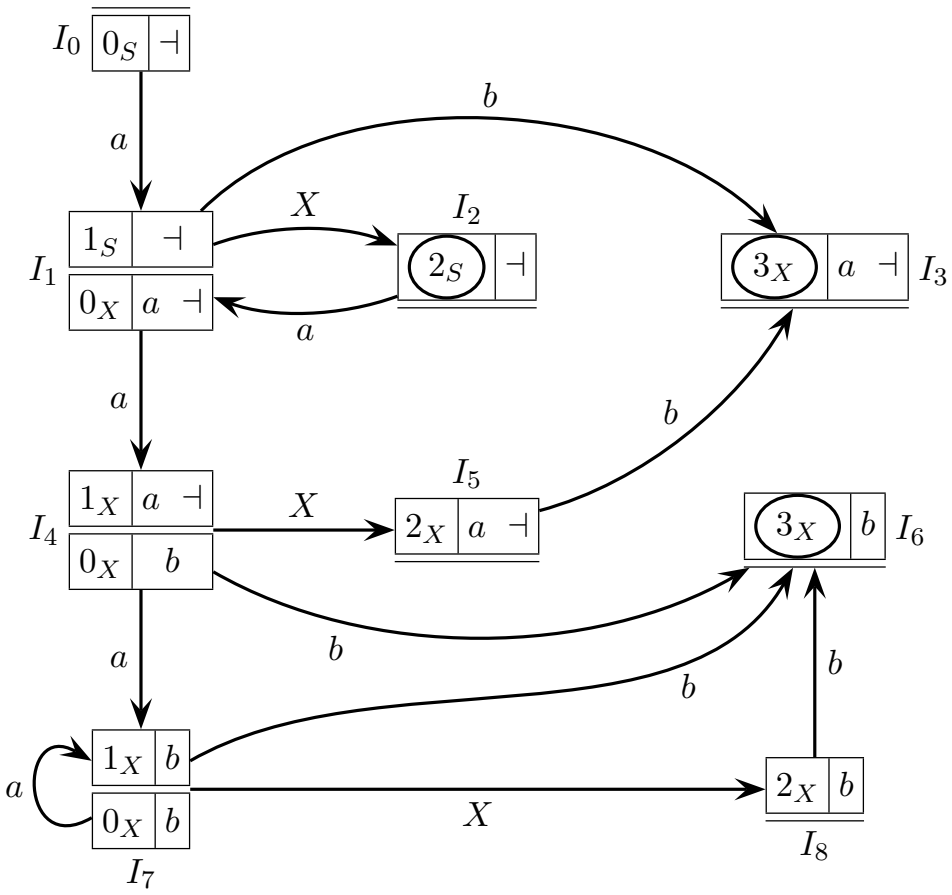
stack	*input*	*move executed*
I_0	$a\ a\ b\ b\ a\ b \dashv$	initial configuration
$I_0\ a\ I_1$	$a\ b\ b\ a\ b \dashv$	shifted a

simulation table for the *ELL* analysis to be completed
(the number of rows is not significant)

stack of machine states	*input*	*move executed*
0_S	$a\ a\ b\ b\ a\ b \dashv$	initial configuration
1_S	$a\ b\ b\ a\ b \dashv$	scanned a

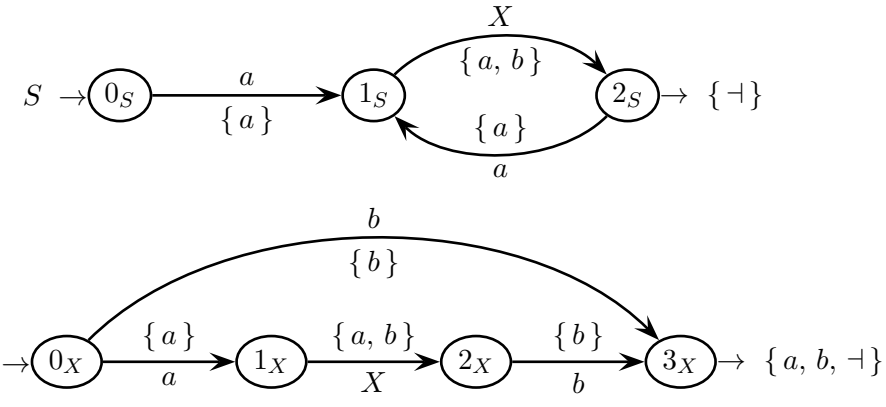
Solution

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



There are not any *ELR* conflicts, hence grammar G is of type $ELR\,(1)$.

(b) Condition $ELL\,(1)$ is satisfied: the pilot is of type $ELR\,(1)$, the base of every m-state contains at most one candidate and so the pilot necessarily has the *STP*, and the grammar is not left-recursive. Hence grammar G is of type $ELL\,(1)$. Here are all the guide sets:



Of course, the guide sets at the bifurcation points (i.e., 2_S and 0_X) are disjoint.

(c) Here is the simulation of the bottom-up pushdown recognizer (*ELR*). The terminator is shown, as it may appear in the reduction look-ahead sets. For simplicity the item identifiers are not shown. Remember that the stack contents consist of interleaved m-states and grammar symbols (nonterminals and terminals):

stack of m-states and symbols	*input*	*move executed*
I_0	$a\ a\ b\ b\ a\ b \dashv$	initial configuration
$I_0\ a\ I_1$	$a\ b\ b\ a\ b \dashv$	shifted a
$I_0\ a\ I_1\ a\ I_4$	$b\ b\ a\ b \dashv$	shifted a
$I_0\ a\ I_1\ a\ I_4\ b\ I_6$	$b\ a\ b \dashv$	shifted b
$I_0\ a\ I_1\ a\ I_4$	$b\ a\ b \dashv$	reduced $b \rightsquigarrow X$
$I_0\ a\ I_1\ a\ I_4\ X\ I_5$	$b\ a\ b \dashv$	shifted X
$I_0\ a\ I_1\ a\ I_4\ X\ I_5\ b\ I_3$	$a\ b \dashv$	shifted b
$I_0\ a\ I_1$	$a\ b \dashv$	reduced $a\ X\ b \rightsquigarrow X$
$I_0\ a\ I_1\ X\ I_2$	$a\ b \dashv$	shifted X
$I_0\ a\ I_1\ X\ I_2\ a\ I_1$	$b \dashv$	shifted a
$I_0\ a\ I_1\ X\ I_2\ a\ I_1\ b\ I_3$	$\dashv$	shifted b
$I_0\ a\ I_1\ X\ I_2\ a\ I_1$	$\dashv$	reduced $b \rightsquigarrow X$
$I_0\ a\ I_1\ X\ I_2\ a\ I_1\ X\ I_2$	$\dashv$	shifted X
I_0	$\dashv$	reduced $a\ X\ a\ X \rightsquigarrow S$
only the initial m-state	empty tape	reduction to axiom S
ELR acceptance condition true - the string belongs to the language		

Notice that the last reduction $a\ X\ a\ X \rightsquigarrow S$ rolls back twice a loop in the pilot. In principle, the item pointers (not shown here) in the stack m-states should be used to clearly identify the reduction handle. Here the situation is quite simple however, and a little intuition suffices even without the pointer mechanism.

(d) Here is the simulation of the top-down pushdown recognizer (*ELL*). The terminator is shown, as it may appear in the guide sets. Of course, here the stack alphabet consists simply of the machine states.

stack of machine states	*input*	*move executed*
0_S	$a\ a\ b\ b\ a\ b\ \dashv$	initial configuration
1_S	$a\ b\ b\ a\ b\ \dashv$	scanned a
$2_S\ 0_X$	$a\ b\ b\ a\ b\ \dashv$	called X
$2_S\ 1_X$	$b\ b\ a\ b\ \dashv$	scanned a
$2_S\ 2_X\ 0_X$	$b\ b\ a\ b\ \dashv$	called X
$2_S\ 2_X\ 3_X$	$b\ a\ b\ \dashv$	scanned b
$2_S\ 2_X$	$b\ a\ b\ \dashv$	returned from X
$2_S\ 3_X$	$a\ b\ \dashv$	scanned b
2_S	$a\ b\ \dashv$	returned from X
1_S	$b\ \dashv$	scanned a
$2_S\ 0_X$	$b\ \dashv$	called X
$2_S\ 3_X$	$\dashv$	scanned b
2_S	$\dashv$	returned from X
only an axiom final state	empty tape	
ELL acceptance condition true - the string belongs to the language		

4 Language Translation and Semantic Analysis 20%

1. Consider a language L of strings that are two-level lists (non-empty). An atomic element of a list is schematized by the terminal e. Any two consecutive elements (atomic or not) are separated by a comma ",". The lists at the second level are enclosed in round brackets. Here are a few sample lists:

   $$e \qquad e,\, e \qquad (\,e\,) \qquad e,\, (\,e,\, e\,) \qquad e,\, e,\, (\,e,\, e\,),\, e \qquad (\,e\,),\, (\,e\,)$$

   Answer the following questions:

   (a) Consider a syntactic translation τ_1 of the language L, which flattens the list by replacing every second-level list with a terminal placeholder "$\perp$". For instance:

   $$\tau_1\big(\,e,\, e,\, (\,e,\, e\,),\, e\,\big) = e,\, e,\, \perp,\, e$$

   Write a syntactic translation scheme equivalent to such a translation τ_1. Use the source grammar G_s below, which generates the language L, and complete it with the destination grammar G_d.

   (b) (optional) Consider a syntactic translation τ_2 of the language L, which swaps the first-level and second-level elements. For instance:

   $$\tau_2\big(\,e,\, e,\, e,\, (\,e,\, e\,),\, e\,\big) = (\,e,\, e,\, e\,),\, e,\, e,\, (\,e\,)$$

   Write a syntactic translation scheme equivalent to such a translation τ_2. If necessary, you can change the source grammar.

---

source G_s for (a) (axiom L_1) *destination G_d for (a)*

$L_1 \;\to\; e\;,\; L_1$

$L_1 \;\to\; e$

$L_1 \;\to\; (\;L_2\;)\;,\; L_1$

$L_1 \;\to\; (\;L_2\;)$

$L_2 \;\to\; e\;,\; L_2$

$L_2 \;\to\; e$

Solution

(a) Source and destination grammars G_s and G_d (axiom L_1):

source grammar G_s	*destination grammar* G_d
$L_1 \rightarrow e\ ,\ L_1$	$L_1 \rightarrow e\ ,\ L_1$
$L_1 \rightarrow e$	$L_1 \rightarrow e$
$L_1 \rightarrow (\ L_2\ )\ ,\ L_1$	$L_1 \rightarrow L_2\ ,\ L_1$
$L_1 \rightarrow (\ L_2\ )$	$L_1 \rightarrow L_2$
$L_2 \rightarrow e\ ,\ L_2$	$L_2 \rightarrow L_2$
$L_2 \rightarrow e$	$L_2 \rightarrow \perp$

Here the idea is that a placeholder $\perp$ replaces the last element e of a sublist, while all the other elements e (if any) of the sublist, as well as their delimiters (round brackets) and separators (commas) are canceled. Of course, the two grammars G_s and G_d have paired rules with the same nonterminal structure, and the rules of each pair differ only as for the terminals.

The destination grammar G_d has a few alternative formulations, which differ only as for the exact point where to put a placeholder $\perp$. Here are two of them, called G'_d and G''_d. The source grammar is unchanged. Only the last four rules are shown, the first two are the same as in the previous solution:

source gram. G_s	*dest. gram.* G'_d	*dest. gram.* G''_d
...	...	...
$L_1 \rightarrow (\ L_2\ )\ ,\ L_1$	$L_1 \rightarrow \perp\ L_2\ ,\ L_1$	$L_1 \rightarrow L_2\ \perp\ ,\ L_1$
$L_1 \rightarrow (\ L_2\ )$	$L_1 \rightarrow \perp\ L_2$	$L_1 \rightarrow L_2\ \perp$
$L_2 \rightarrow e\ ,\ L_2$	$L_2 \rightarrow L_2$	$L_2 \rightarrow L_2$
$L_2 \rightarrow e$	$L_2 \rightarrow \varepsilon$	$L_2 \rightarrow \varepsilon$

In the previous destination grammar G_d, a placeholder $\perp$ replaces the last element e of a sublist, while in the two new destination grammars G'_d and G''_d it replaces the open or the closed round bracket of a sublist, respectively; all the other elements of a sublist, as well as their remaining delimiters and all their separators, are canceled. Of course, both grammars G'_d and G''_d have the same nonterminal structure as grammar G_s.

(b) Moving the round brackets from L_2 to L_1 clearly yields a grammar that generates lists with arbitrarily many levels. Thus it is necessary to change the source grammar G_s, to avoid generating arbitrary-level lists. Here is a possible solution (axiom S):

source grammar G_s			*destination grammar* G_d		
S	$\rightarrow$	E	S	$\rightarrow$	$(\,E\,)$
S	$\rightarrow$	$E\,,\,(\,E\,)$	S	$\rightarrow$	$(\,E\,)\,,\,E$
S	$\rightarrow$	$E\,,\,(\,E\,)\,,\,S$	S	$\rightarrow$	$(\,E\,)\,,\,E\,,\,S$
S	$\rightarrow$	$(\,E\,)$	S	$\rightarrow$	E
S	$\rightarrow$	$(\,E\,)\,,\,S$	S	$\rightarrow$	$E\,,\,S$
E	$\rightarrow$	e	E	$\rightarrow$	e
E	$\rightarrow$	$e\,,\,E$	E	$\rightarrow$	$e\,,\,E$

Now a two-level list is generated as a sequence of pieces of type E and $(\,E\,)$, where the nonterminal E is then expanded into a (non-empty) list of atomic elements e. See for instance the sample string below (left):

$$(\underbrace{e,\,e,\,e}_{(\,E\,)})\,,\,\underbrace{e,\,e}_{E}\,,\,(\underbrace{e}_{(\,E\,)}) \qquad \Rightarrow \qquad \underbrace{e,\,e,\,e}_{E}\,,\,(\underbrace{e,\,e}_{(\,E\,)})\,,\,\underbrace{e}_{E}$$

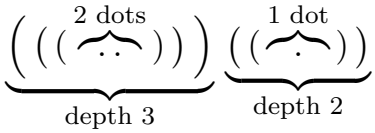
Swapping the first-level and second-level lists is thus immediate (see above right). Notice that the source grammar is not ambiguous, thanks to the fact of being designed so that a derivation may not contain two consecutive symbols E (there is always at least one sublist $(\,E\,)$ in between). This ensures that the translation is one-valued. Of course, there may be different solutions.

It may be interesting to see an extended (*EBNF*) form of the above solution:

source grammar G'_s			*destination grammar* G'_d		
S	$\rightarrow$	$E\,\Big[\,,\,(\,E\,)\,\big[\,,\,S\,\big]\,\Big]$	S	$\rightarrow$	$(\,E\,)\,\Big[\,,\,E\,\big[\,,\,S\,\big]\,\Big]$
S	$\rightarrow$	$(\,E\,)\,\big[\,,\,S\,\big]$	S	$\rightarrow$	$E\,\big[\,,\,S\,\big]$
E	$\rightarrow$	$e\,\big[\,,\,E\,\big]$	E	$\rightarrow$	$e\,\big[\,,\,E\,\big]$

which is heavily based on the optionality operator $[\,]$. Of course, also here the source and destination nonterminal structures must be identical.

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



encodes the number $3^2 + 2^1 = 9 + 2 = 11$, while the list $\big(\,(\,.\,.\,.\,.\,.\,)\,\big)\,\Big(\,\big(\,(\,(\,.\,.\,.\,)\,)\,\big)\,\Big)$ 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):

$$\begin{cases} L & \to & E\ L \ \mid\ E \\ E & \to & (\ E\ ) \ \mid\ (\ P\ ) \\ P & \to & .\ P \ \mid\ . \end{cases}$$

(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 $\Big(\,\big(\,(\,.\,.\,)\,\big)\,\Big)\,\big(\,(\,.\,)\,\big)$, 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)

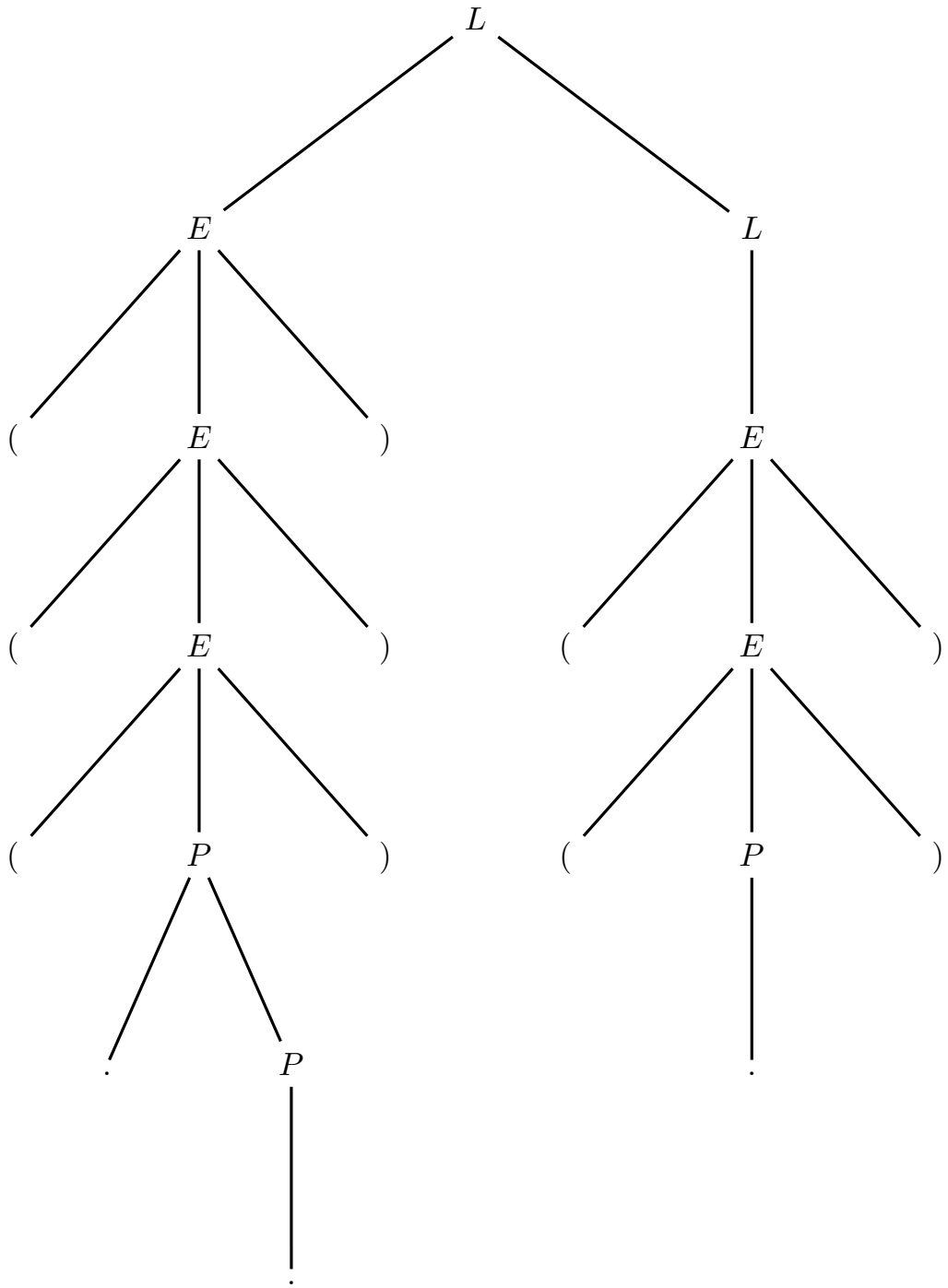
type	*name*	*domain*	*nonterm.*	*meaning*
	d	integer		nesting level of each sublist
	n	integer		number of dots in each sublist
	v	integer		numerical value encoded

attributes to be added, if any

type	*name*	*domain*	*nonterm.*	*meaning*

#	*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)

type	*name*	*domain*	*nonterm.*	*meaning*
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.

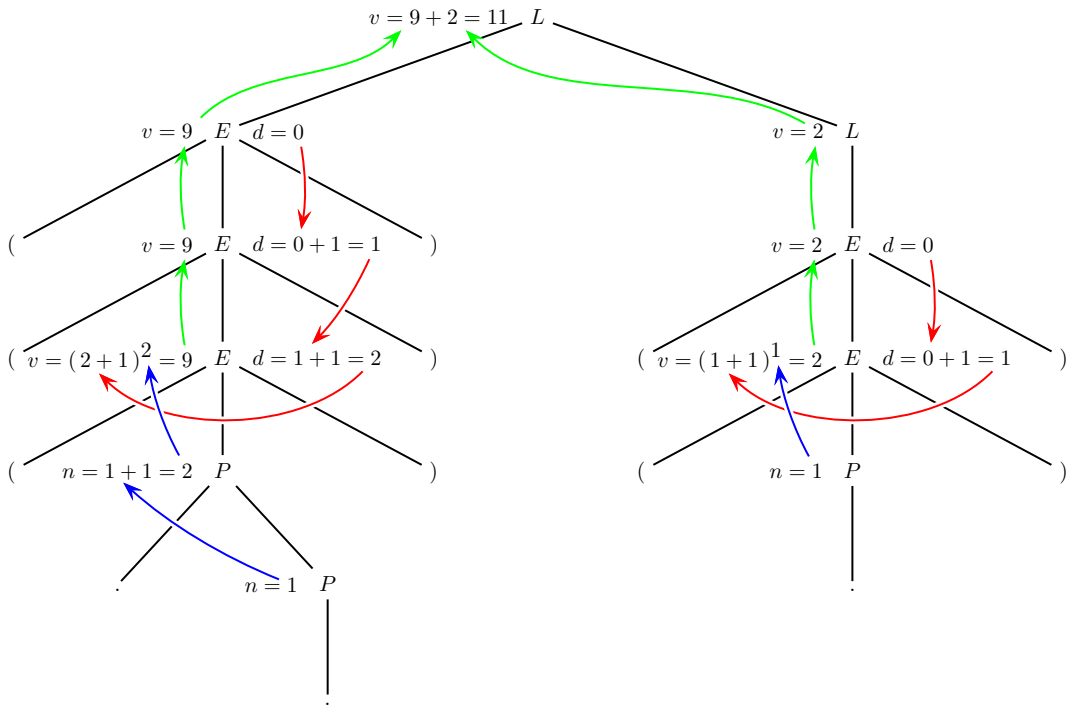
Semantic functions (axiom L_0):

#	*syntax*	*semantics*	*comment*
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: $E \to (\ P\ )$, 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)

type	*name*	*domain*	*nonterm.*	*meaning*
left	d	integer	E	reverse 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 innermost 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):

#	*syntax*	*semantics*	*comment*
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.