FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Tue 6 September 2016 - Part Theory

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

NAME (capital letters pls.):

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 regular expression R below, over the three-letter alphabet $\Sigma = \{\ a,\ b,\ c\ \}$.

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

Answer the following questions:

(a) Write all the strings x with $|\,x\,| \leq 2$ that belong to language $L\,(R)$, then say if the regular expression R is ambiguous and justify your answer.

(b) By using the Berry-Sethi method (BS), find a deterministic finite-state automaton A equivalent to the regular expression R above.

(c) By using a systematic method, find the finite-state automaton $\overline{A}$ complement of the automaton A previously found, and if necessary minimize the complement automaton $\overline{A}$.

(d) By using a systematic method, find a regular expression $\overline{R}$ equivalent to the complement automaton $\overline{A}$ previously found.

(e) (optional) Verify that the strings x of point (a) *do not belong* to the complement language $L\,(\overline{R})$ and shortly explain why.

Solution

(a) Strings b, $a\,b$, $b\,b$, $b\,c$ belong to language $L\,(R)$. Number:

$$R_{\#} = (\ a_1\ \mid\ b_2\ )^*\ b_3\ (\ b_4\ \mid\ c_5\ )^*$$

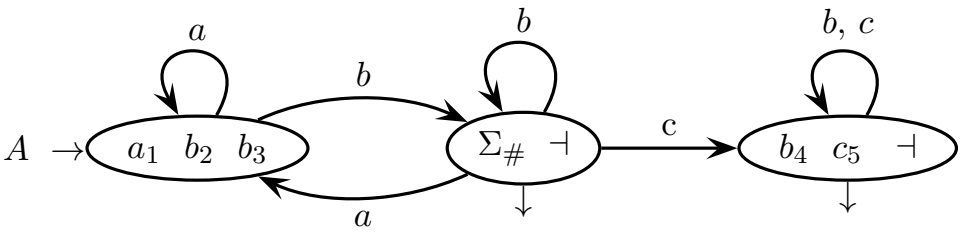
and notice that RE R is ambiguous because of $b\,b = b_2\,b_3 = b_3\,b_4$.

(b) Easy, three states. First the marked regular expression $R_{\#}$, and its two sets of initials and followers:

$$R_{\#} = (\ a_1\ \mid\ b_2\ )^*\ b_3\ (\ b_4\ \mid\ c_5\ )^*\ \dashv$$

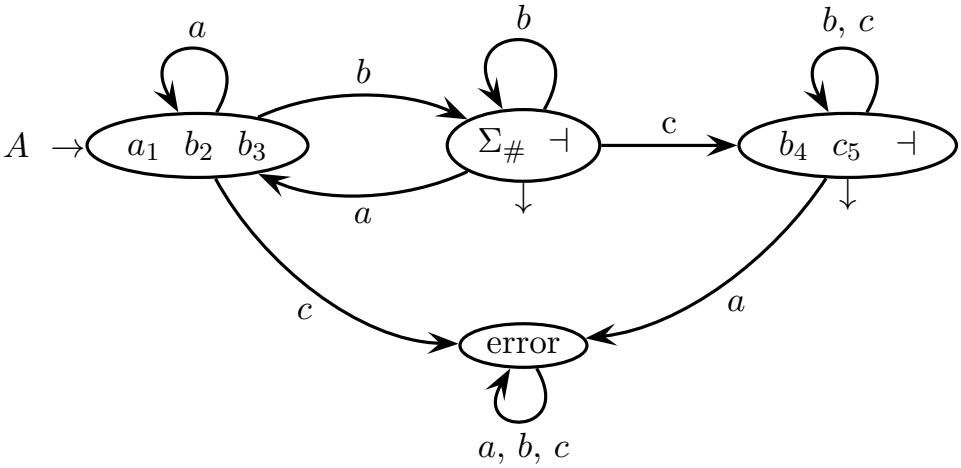
initials	a_1	b_2	b_3
terminals	followers		
a_1	a_1	b_2	b_3
b_2	a_1	b_2	b_3
b_3	b_4	c_5	$\dashv$
b_4	b_4	c_5	$\dashv$
c_5	b_4	c_5	$\dashv$

Then the BS automaton A that recognizes language $L\,(R)$:

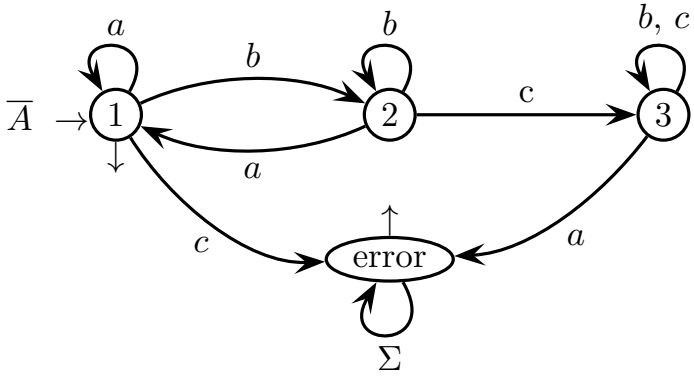


By construction, automaton A is deterministic and in clean form. It happens to be minimal, too: the two final states have different arc labels.

(c) Easy, four states, already minimal. First the completion of automaton A:

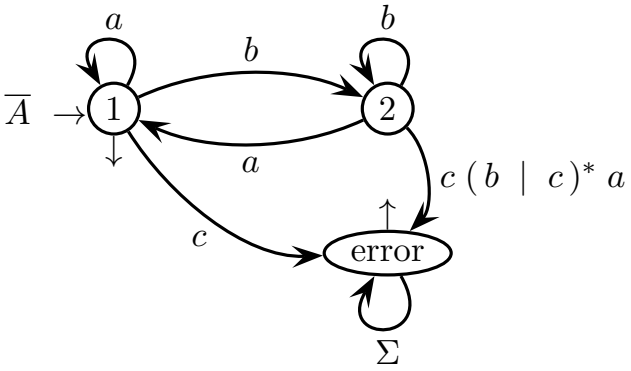


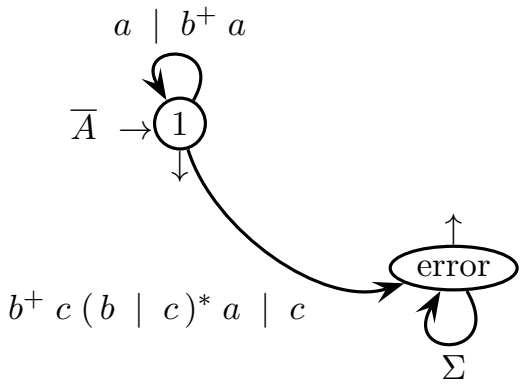
Then the complement construction (the state names are shortened):



Automaton $\overline{A}$ is deterministic and in clean form. It happens to be minimal, too: states 1 and error are distinguishable due to arc b, and consequently states 2 and 3 are distinguishable due to arc a.

(d) By node elimination one obtains (the construction is sped up):





and eventually:

$$\overline{R} = \varepsilon \mid (a \mid b^+ a)^* \left(b^+ c (b \mid c)^* a \mid c\right) \Sigma^*$$

It is easy to verify the correctness of the RE $\overline{R}$, because the characteristic of the original RE R is that it generates all and only the strings that contain at least one b and the c's (if any) follow the a's (if any) with at least one b in between, while the complement RE $\overline{R}$, expanded as follows:

$$\overline{R} = \varepsilon \mid (b^* a)^* \left(b^+ c (b \mid c)^* a\right) \Sigma^* \mid (b^* a)^* c \Sigma^*$$

corresponds to all and only the four violation cases of R: nothing, or a b followed by a c (at least) and subsequently by an a, or an a immediately followed by a c, or an initial c.

(e) To verify that the strings do not belong to the complement language $L(\overline{R})$, it suffices to check that the computation of automaton $\overline{A}$ when analyzing such strings ends in a non-final state.

- b: $1 \xrightarrow{b} 2$ and state 2 is not final
- ab: $1 \xrightarrow{a} 1 \xrightarrow{b} 2$ and state 2 is not final
- bb: $1 \xrightarrow{b} 2 \xrightarrow{b} 2$ and state 2 is not final
- bc: $1 \xrightarrow{b} 2 \xrightarrow{c} 3$ and state 3 is not final

This is perhaps the simplest way, as in a deterministic automaton the paths are clearly visible and furthermore unique. Alternatively, one might argue that the regular expression $\overline{R}$ is unable to derive such strings.

2 Free Grammars and Pushdown Automata 20%

1. Consider a three-letter alphabet $\Sigma = \{ a, b, c \}$, and the three languages L_1, L_2 and L as defined below. Answer the following questions:

   (a) Write a grammar G_1, *BNF* and unambiguous, that generates the language:

   $$L_1 = \left\{ a^i \, b^j \, c^k \mid \quad i, j, k \geq 0 \ \wedge \ k \leq i \right\}$$

   Draw the syntax tree for string $a^3 \, b \, c^2 \in L_1$.

   (b) Write a grammar G_2, *BNF* and unambiguous, that generates the language:

   $$L_2 = \left\{ a^i \, b^j \, c^k \mid \quad i, j, k \geq 0 \ \wedge \ k \leq j \right\}$$

   Draw the syntax tree for string $a \, b^3 \, c^2 \in L_2$.

   (c) Write a *BNF* grammar G that generates the language:

   $$L = \left\{ a^i \, b^j \, c^k \mid \quad i, j, k \geq 0 \ \wedge \ (k \leq i \ \vee \ k \leq j) \right\}$$

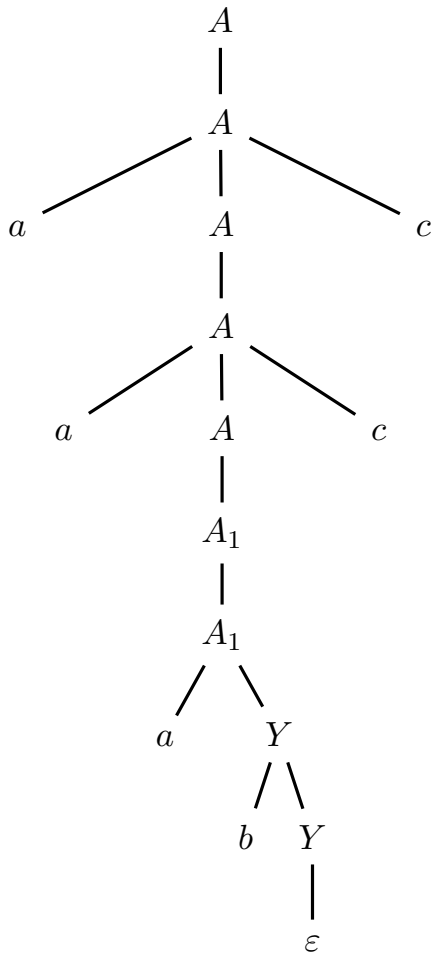
   Determine if grammar G is ambiguous and justify your answer: if it is ambiguous then provide a string with multiple syntax trees, otherwise argue that it does not admit any ambiguous string.

Solution

(a) Here is grammar G_1 (axiom A), of type *BNF*:

$$G_1 \begin{cases} A & \to & a\ A\ c\ \mid\ A_1 \\ A_1 & \to & a\ A_1\ \mid\ Y \\ Y & \to & b\ Y\ \mid\ \varepsilon \end{cases}$$

Grammar G_1 is unambiguous: the auto-inclusive rule is notoriously unambiguous, and the other rules, considered separately, are right-linear deterministic, for instance $LL(1)$, hence they are unambiguous as well.
Here is the syntax tree of the sample string $a^3\,b\,c^2$:



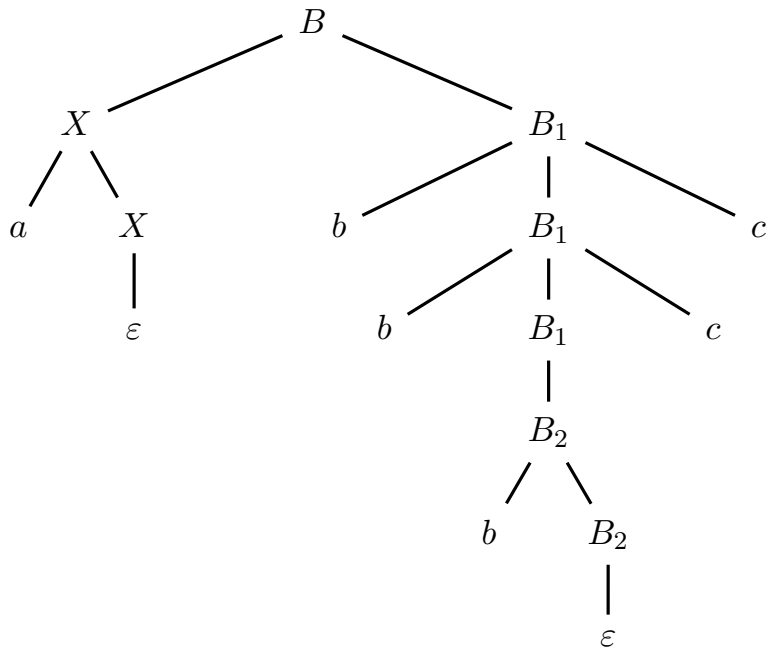
As said above, there are not any other syntax trees for this sample string.

(b) Here is grammar G_2 (axiom B), of type *BNF*:

$$G_2 \begin{cases} B & \to & X\ B_1 \\ B_1 & \to & b\ B_1\ c\ \mid\ B_2 \\ B_2 & \to & b\ B_2\ \mid\ \varepsilon \\ X & \to & a\ X\ \mid\ \varepsilon \end{cases}$$

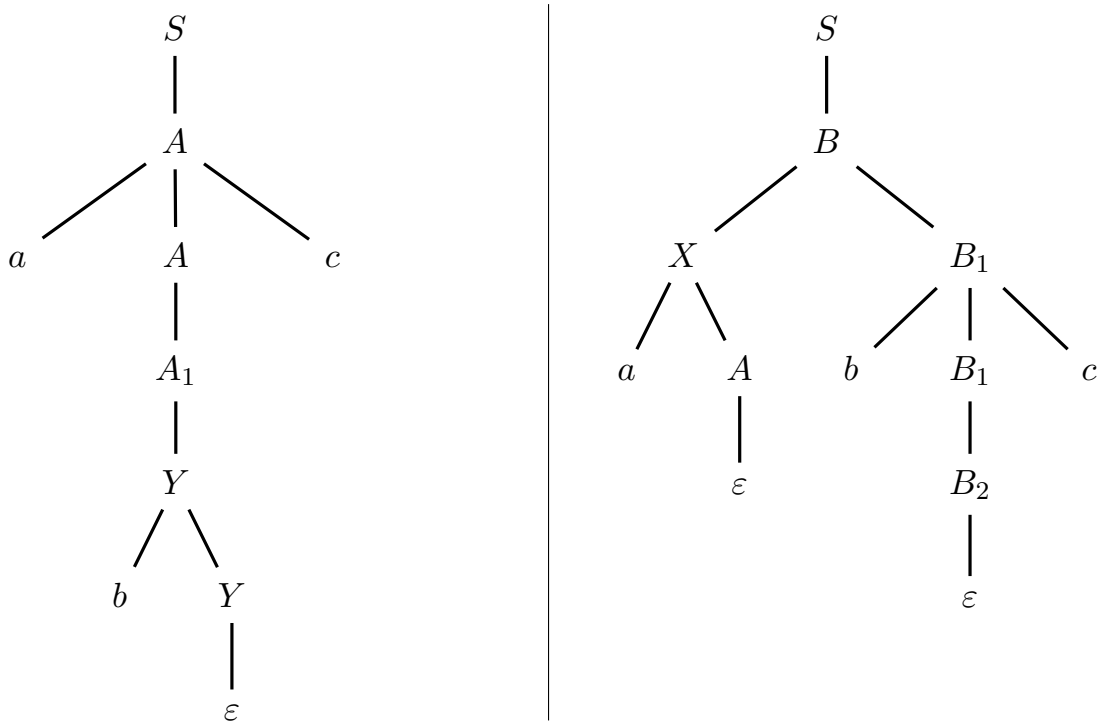
Grammar G_2 is unambiguous, for the same reasons as for grammar G_1 before: the auto-inclusive rule is notoriously unambiguous, and the other rules, considered separately, are right-linear deterministic, for instance $LL\,(1)$, hence they are unambiguous as well.

Here is the syntax tree of the sample string $a\,b^3\,c^2$:



As said above, there are not any other syntax trees for this sample string.

(c) Clearly it holds $L = L_1 \cup L_2$, therefore the grammar union of grammars G_1 and G_2 (axiom S, rules $S \to A \mid B$, etc) generates language L. The union is anyway not disjoint: every string of type $a^i\, b^j\, c^k$ such that i, j, $k \geq 0$ and $k \leq i$ and $k \leq j$, like for instance string $a\, b\, c$, is in the intersection. Thus string $a\, b\, c$ has the two syntax trees below:



By construction, string $a\, b\, c$ does not have over two syntax trees, and the same holds for any other ambiguous string generated by the union grammar, as the two united grammars are unambiguous.

2. Consider a program written in a simplified C language, which features the following syntactic structures:

   - a program consists of a declaration block followed by an execution block, both mandatory
   - the declaration block defines integer variables declared as usual in the C language
   - the execution block is a list of the following statement types:
     - assignment with a variable on the left side and an arithmetic expression on the right side
     - for loop with an initialization clause (an assignment), an iteration clause (a relational expression) and an update clause (an assignment), which are grouped in round parentheses ‘ (’ ‘) ’ and separated by semicolons ‘ ; ’ as usual in the C language, followed by an execution block
     - a nested execution block

     and each statement is followed by a semicolon ‘ ; ’
   - an arithmetic expression may contain addition operators ‘ + ’, variables, integer constants and subexpressions delimited by round parentheses ‘ (’ ‘) ’
   - a relational expression consists of a comparison operator (‘ < ’, ‘ > ’, ‘ == ’, etc) and two arithmetic expressions to be compared
   - the for loops may be nested
   - the blocks are delimited by graph parentheses ‘ { ’ ‘ } ’ and may be empty; furthermore, the execution block may contain nested blocks
   - variable names and numerical constants can be schematized by the terminals id and num, respectively, to be left unexpanded

   For any detail that may have been left unspecified one can adopt the usual conventions of the C language. Here is a sample program:

```
{ int a, b, c, d; }                        // declaration block

{                                          // execution block
  a = b + c + (a + d) + 1;                 // assignment statement
  for (a = 1; a + b < d; a = a + 1) {      // for loop
    b = b + 2;
    {                                      // nested execution block
      ...
    };
    c = d;                                 // assignment statement
  };
}
```

   Write a grammar G, *EBNF* and unambiguous, that models the sketched language.

Solution

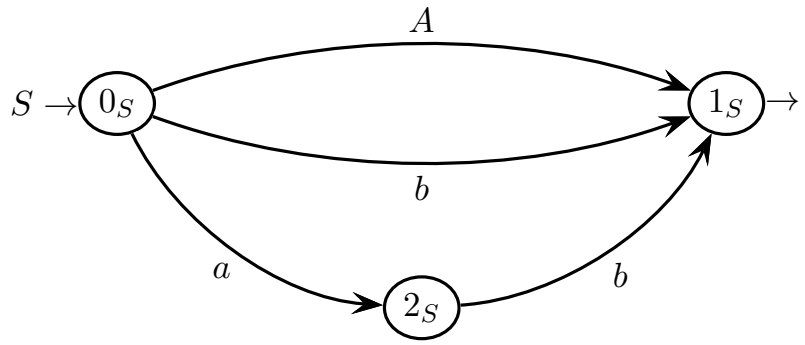
(a) Here is a reasonable grammar G for the sketched language (axiom PROG):

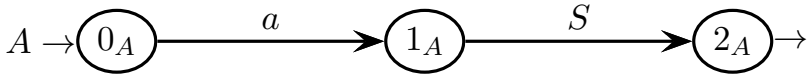
$$
G \left\{
\begin{array}{rcl}
\langle \mathsf{PROG} \rangle & \to & \langle \mathsf{DECL} \rangle \ \langle \mathsf{EXEC} \rangle \\
\hline
\langle \mathsf{DECL} \rangle & \to & \text{`}\{\text{'} \ (\ \langle \mathsf{INTD} \rangle \ \text{`};\text{'} \ )^* \ \text{`}\}\text{'} \\
\langle \mathsf{EXEC} \rangle & \to & \text{`}\{\text{'} \ (\ \langle \mathsf{STAT} \rangle \ \text{`};\text{'} \ )^* \ \text{`}\}\text{'} \\
\hline
\langle \mathsf{INTD} \rangle & \to & \mathsf{int\ id}\ (\ \text{`,'}\ \mathsf{id}\ )^* \\
\langle \mathsf{STAT} \rangle & \to & \langle \mathsf{ASGN} \rangle \mid \langle \mathsf{FORL} \rangle \mid \langle \mathsf{EXEC} \rangle \\
\hline
\langle \mathsf{ASGN} \rangle & \to & \mathsf{id}\ \text{`}=\text{'}\ \langle \mathsf{AEXPR} \rangle \\
\langle \mathsf{FORL} \rangle & \to & \mathsf{for}\ \text{`}(\text{'}\ \langle \mathsf{CLAUSES} \rangle\ \text{`})\text{'}\ \langle \mathsf{STAT} \rangle \\
\hline
\langle \mathsf{AEXPR} \rangle & \to & \langle \mathsf{TERM} \rangle\ (\ \text{`}+\text{'}\ \langle \mathsf{TERM} \rangle\ )^* \\
\langle \mathsf{CLAUSES} \rangle & \to & \langle \mathsf{ASGN} \rangle\ \text{`};\text{'}\ \langle \mathsf{CEXPR} \rangle\ \text{`};\text{'}\ \langle \mathsf{ASGN} \rangle \\
\hline
\langle \mathsf{TERM} \rangle & \to & \mathsf{id} \mid \mathsf{num} \mid \text{`}(\text{'}\ \langle \mathsf{AEXPR} \rangle\ \text{`})\text{'} \\
\langle \mathsf{CEXPR} \rangle & \to & \langle \mathsf{AEXPR} \rangle\ \langle \mathsf{CMPOP} \rangle\ \langle \mathsf{AEXPR} \rangle \\
\hline
\langle \mathsf{CMPOP} \rangle & \to & \text{`}<\text{'} \mid \text{`}>\text{'} \mid \text{`}==\text{'} \mid \ldots
\end{array}
\right.
$$

Grammar G is reasonably correct and is unambiguous by construction, as it consists of structures known to be unambiguous.

3 Syntax Analysis and Parsing Methodologies 20%

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





Answer the following questions:

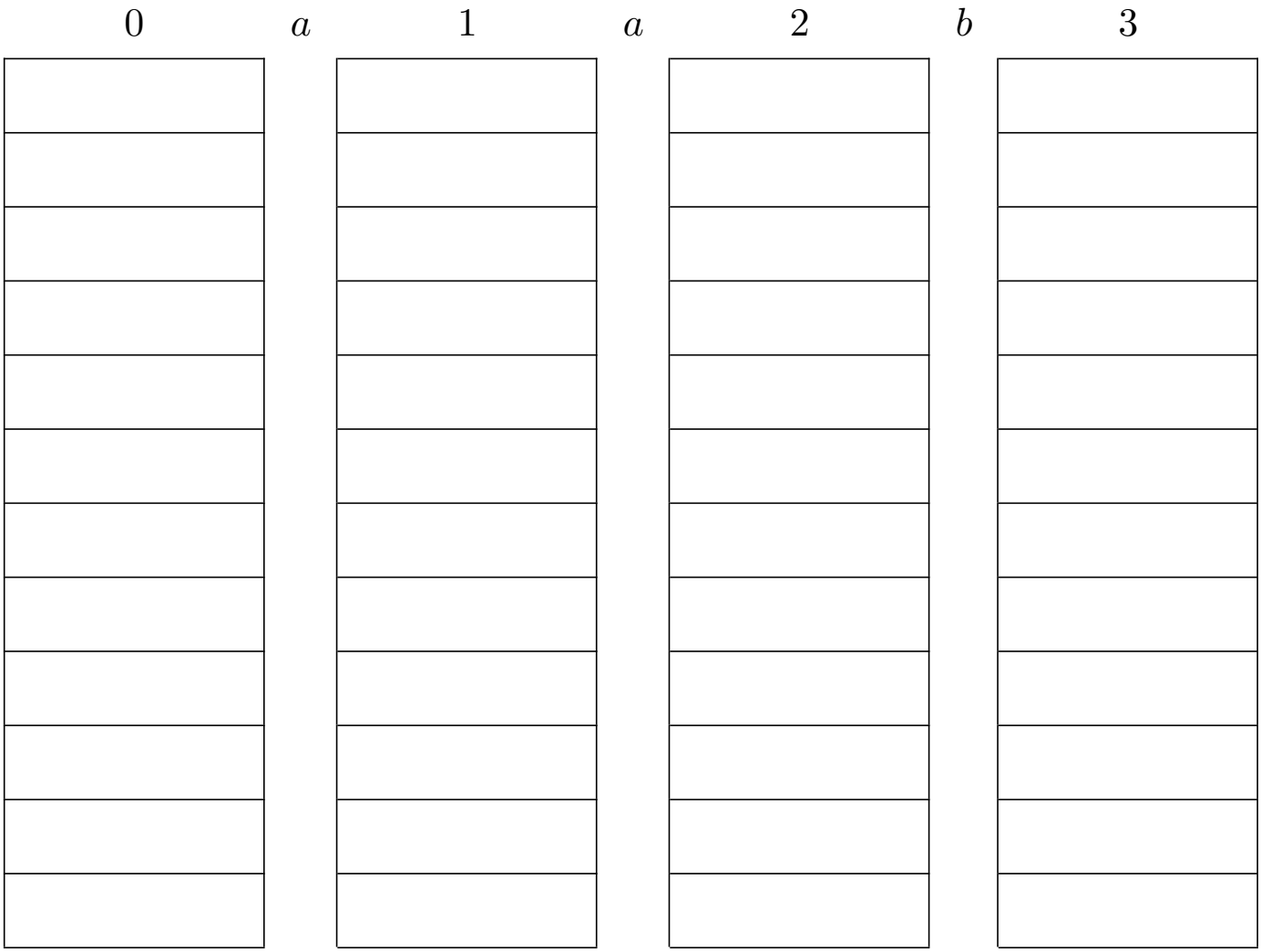
(a) Draw the syntax tree (or trees if there are two or more) of the valid string $a\,a\,b$.

(b) Draw the complete pilot of grammar G, say if grammar G is of type $ELR(1)$ and shortly justify your answer. If the grammar is not $ELR(1)$ then highlight all the conflicts in the pilot.

(c) Write all the guide sets on the arcs of the machine net (shift and call arcs, and final arrows), say if grammar G is of type $ELL(1)$, based on the guide sets, and shortly justify your answer. If you wish, you can use the figure above to add the call arcs and annotate the guide sets.

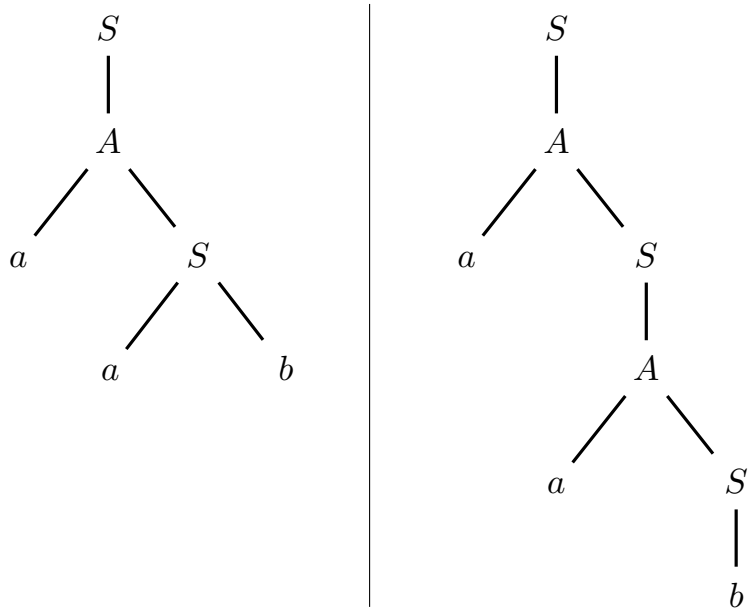
(d) (optional) Analyze the valid string $a\,a\,b$ using the Earley method; show the item/all the items that gives/give evidence of string acceptance. Use the Earley vector prepared on the next page.

Earley vector of string $a\ a\ b$ (to be filled in)
(the number of rows is not significant)



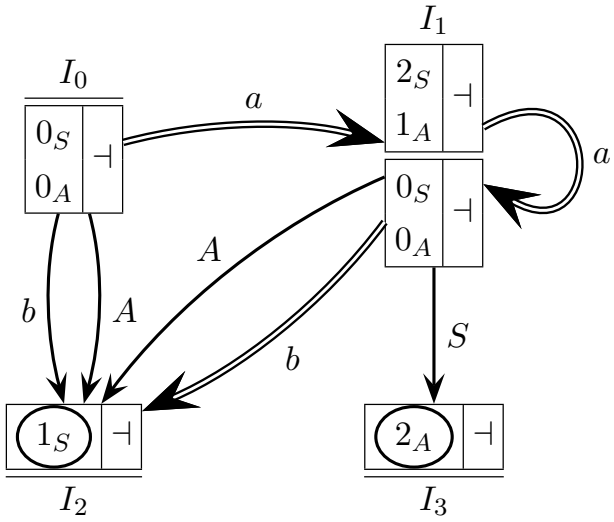
Solution

(a) The string $a\,a\,b$ is ambiguous and admits these two syntax trees:



The string does not have any other syntax tree.

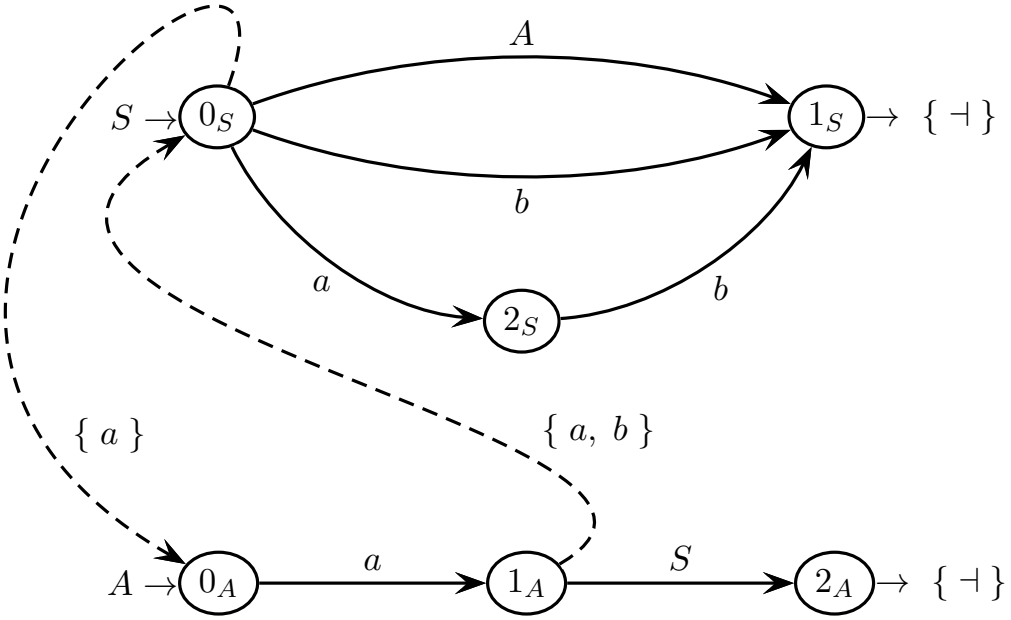
(b) Here is the requested pilot, with one conflicting convergent transition, namely the b-transition $I_1 \xrightarrow{b} I_2$ going to state 1_S (because both source states 2_S and 0_S go to the same target state 1_S and have the same look-ahead $\dashv$):



There are no other conflicts. Anyway, grammar G is not of type $ELR\,(1)$. Notice that both a-transitions are multiple as well, yet they are not convergent and consequently they are not conflicting either. Notice also that the grammar is actually right-linear, therefore the language is regular. Finally, notice that the look-ahead is always the terminator $\dashv$. This happens because the grammar is right-linear, thus every nonterminal occurs as the last symbol of a machine path,

consequently every initial item can only inherit the same look-ahead as that of the calling machine, and the inheritance chain eventually goes back to the look-ahead of the axiom, which is the terminator by definition.

(c) Here are all the guide sets of the machine net, completed with the call arcs to make the Predictive Control Flow Graph (*PCFG*):

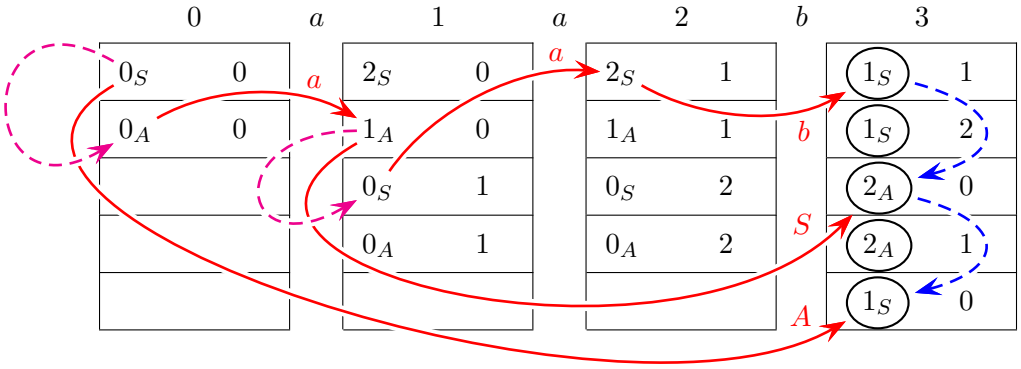


The guide sets on the terminal shift arcs are trivial and not shown. The grammar is not $ELL(1)$, because of the overlapping guide sets at the bifurcation state 0_S.

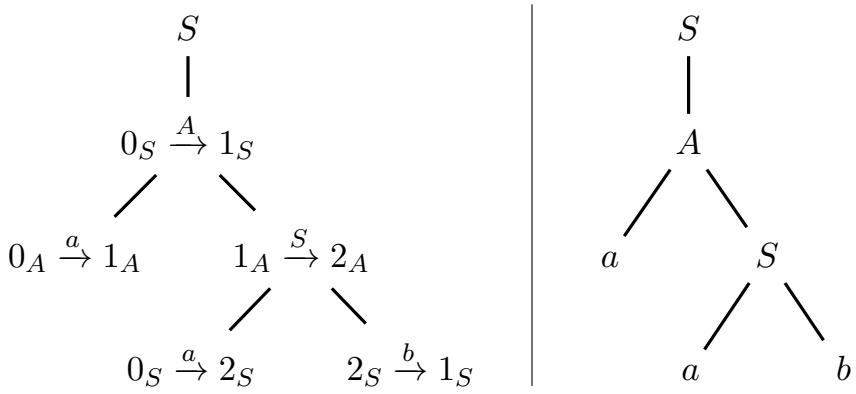
(d) Here is the Earley vector for string $a\,a\,b$:

0		a	1		a	2		b	3	
0_S	0		2_S	0		2_S	1		1_S	1
0_A	0		1_A	0		1_A	1		1_S	2
			0_S	1		0_S	2		2_A	0
			0_A	1		0_A	2		2_A	1
									1_S	0

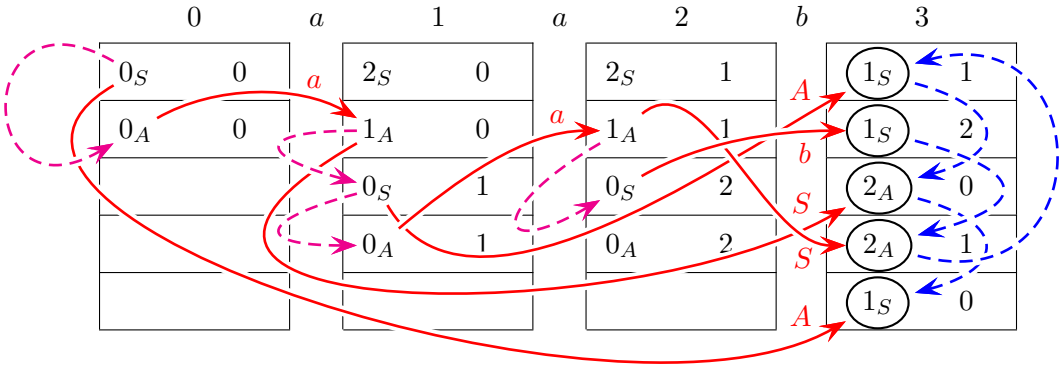
The final item $\langle\, 1_S,\, 0 \,\rangle$ in the last vector element indicates that string $a\,a\,b$ is accepted and corresponds to the two parsing trees of the string $a\,a\,b$, which is ambiguous as already known from point (a). Here is the tree construction:



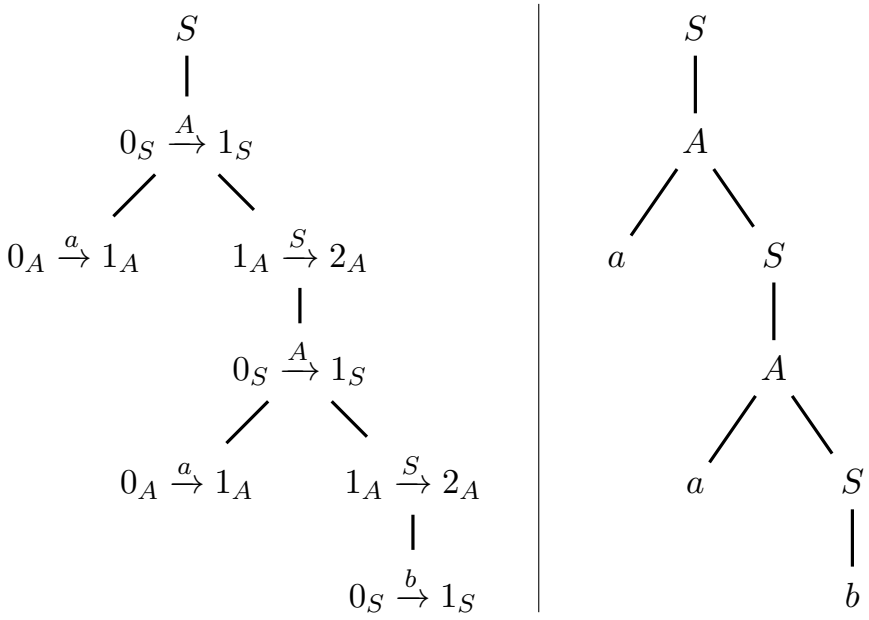
Red is for terminal and nonterminal shift, magenta is for completion and blue is for reduction. This is the left three of point (a) and is the simpler one.



These are the path tree and the standard syntax tree obtained by canceling the path information.



Red is for terminal and nonterminal shift, magenta is for completion and blue is for reduction. This is the right three of point (a) and has more nodes than the previous one.



These are the path tree and the standard syntax tree obtained by canceling the path information.

The Earley vector above does not allow to construct any other syntax tree. This formally proves that the sample string $a\,a\,b$ has exactly two syntax trees or equivalently that it is ambiguous of degree two. Of course, the grammar might still generate strings with an ambiguity degree over two and even have an unlimited ambiguity degree.

4 Language Translation and Semantic Analysis 20%

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

$$L_s = \left\{\ a^m\ b^h\ c^k \mid \quad h + k = m\ \wedge\ h,\ k \geq 0\ \wedge\ m \geq 1\ \right\}$$

The following source grammar G_s, *BNF* and unambiguous, generates the source language L_s above:

$$G_s \begin{cases} S_1 \rightarrow a\ S_1\ c \\ S_1 \rightarrow a\ c \\ S_1 \rightarrow a\ S_2\ b \\ S_2 \rightarrow a\ S_2\ b \\ S_2 \rightarrow \varepsilon \end{cases}$$

Now consider a translation $\tau \colon L_s \rightarrow \{\ a,\ c\ \}^*$ that works as follows:

$$\tau\left(a^m\ b^h\ c^k\right) = a^h\ c^h$$

Answer the following questions:

(a) Write all the source strings $x \in L_s$ with $|\,x\,| \leq 4$, and for each of them write the corresponding translation $\tau\,(x)$.

(b) Find a suitable syntactic translation scheme (or grammar) G_τ, of type *BNF* and unambiguous, based on the given source grammar G_s, that computes the translation τ.

(c) (optional) Verify that the scheme (or grammar) G_τ is correct, by means of the sample (source, translation) string pairs $\big(x,\ \tau\,(x)\big)$ found at point (a). You can write the derivation or draw the syntax trees of each pair to be verified.

Solution

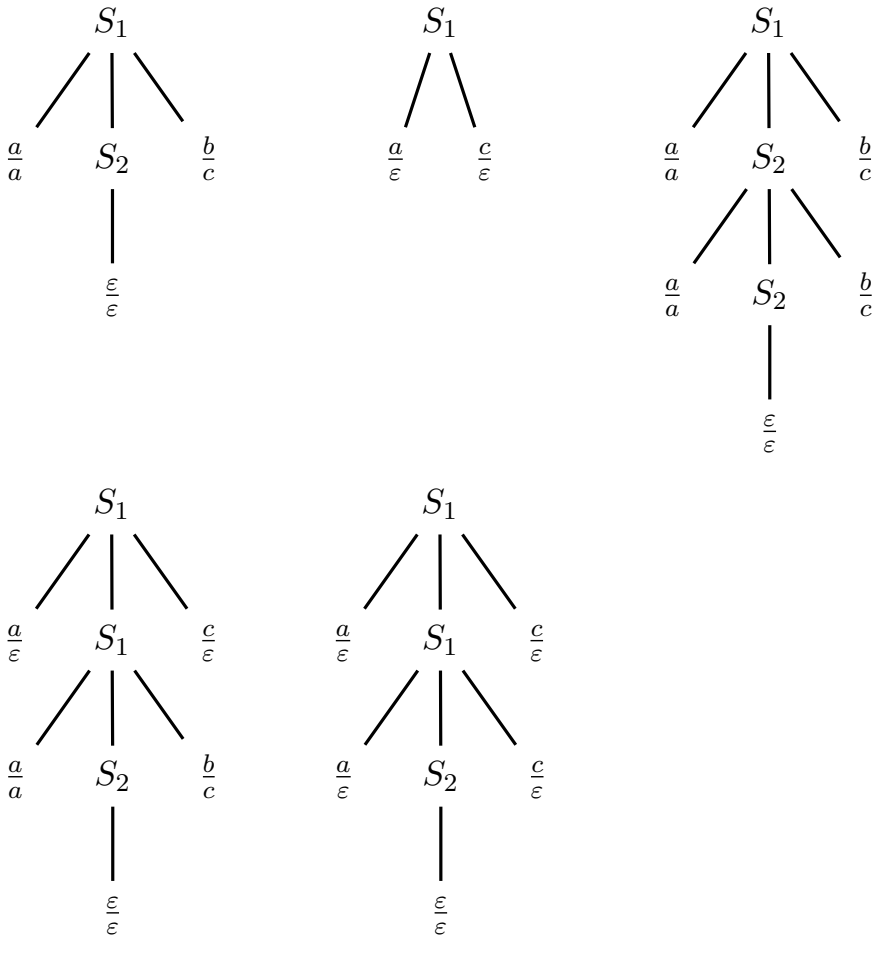
(a) Here are the requested source / target string pairs of length ≤ 4, five in total:

$$\begin{aligned} \tau\,(a\,b) &= a\,c \\ \tau\,(a\,c) &= \varepsilon \\ \tau\,(a\,a\,b\,b) &= a\,a\,c\,c \\ \tau\,(a\,a\,b\,c) &= a\,c \\ \tau\,(a\,a\,c\,c) &= \varepsilon \end{aligned}$$

(b) Here is a working translation scheme (and translation grammar) G_τ (axiom S_1):

$$G_\tau \left\{ \begin{array}{lll} S_1 \to a\ S_1\ c & S_1 \to S_1 & S_1 \to \frac{a}{\varepsilon}\ S_1\ \frac{c}{\varepsilon} \\ S_1 \to a\ c & S_1 \to \varepsilon & S_1 \to \frac{a}{\varepsilon}\ \frac{c}{\varepsilon} \\ S_1 \to a\ S_2\ b & S_1 \to a\ S_2\ c & S_1 \to \frac{a}{a}\ S_2\ \frac{b}{c} \\ \hline S_2 \to a\ S_2\ b & S_2 \to a\ S_2\ c & S_2 \to \frac{a}{a}\ S_2\ \frac{b}{c} \\ S_2 \to \varepsilon & S_2 \to \varepsilon & S_2 \to \frac{\varepsilon}{\varepsilon} \end{array} \right.$$

(c) Here are the syntax trees of the above five pairs (source and target trees unified):



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):

   $$\begin{cases} 1\colon\ S \ \rightarrow\ L \\ 2\colon\ L \ \rightarrow\ i\ L \\ 3\colon\ L \ \rightarrow\ i \end{cases}$$

   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.

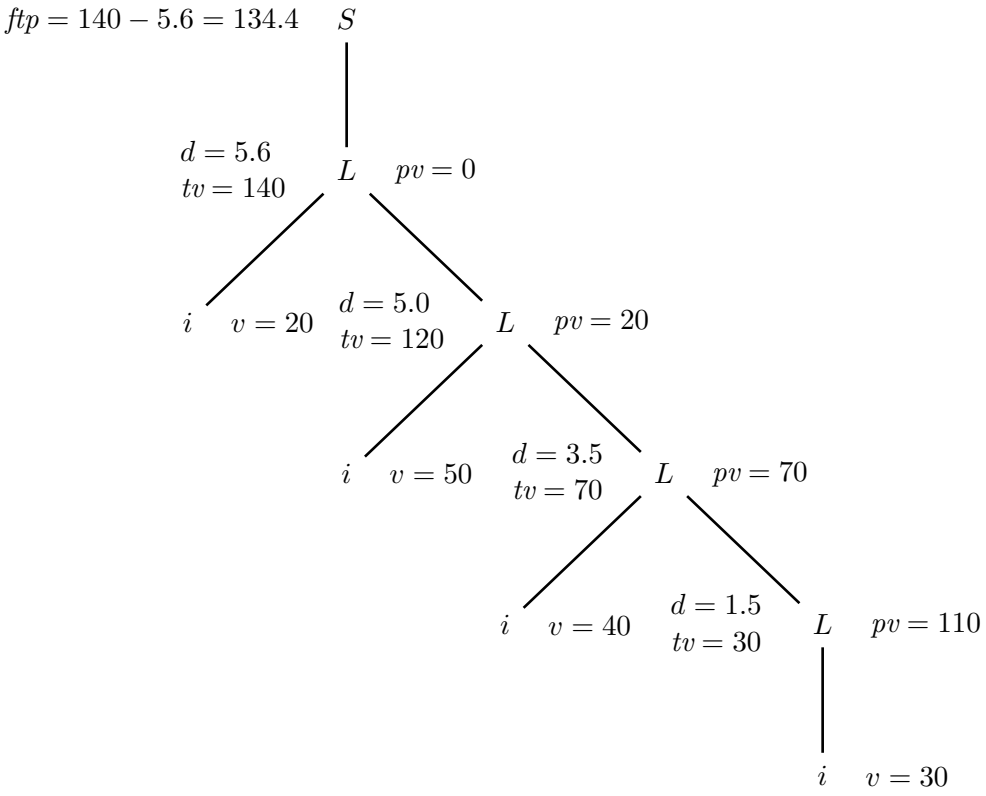
   The attributes to use are already assigned. See the table below. Other attributes are unnecessary and should not be added.

   attributes assigned to be used

type	*name*	*domain*	*(non)term.*	*meaning*
right	pv	real	L	*previous value*: total value of the items occurring in the previous list positions
left	d	real	L	*discount*: discount for the items in the subtree rooted in L
left	tv	real	L	*total value*: total value of the items in the subtree rooted in L
left	*ftp*	real	S	*final total price* = total value of the items in the whole tree − overall discount
right	v	real	i	*value*: 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.

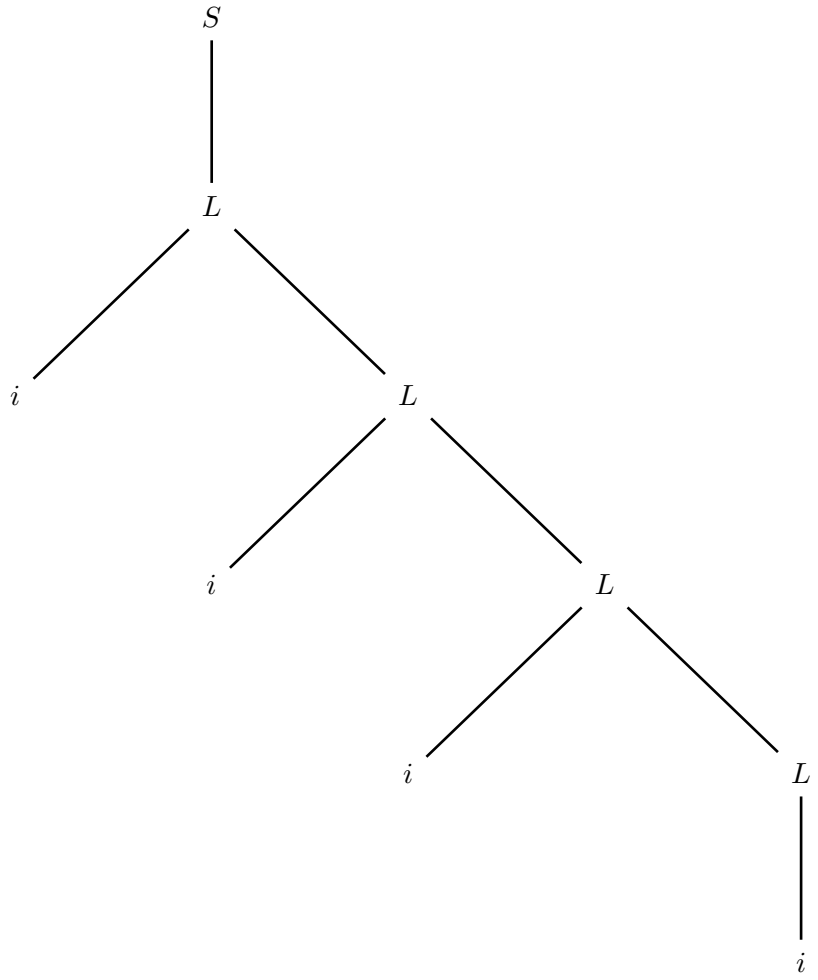
(c) Determine if the attribute grammar is of type one-sweep and if it satisfies the L condition, and reasonably justify your answers.

(d) (optional) Write the semantic evaluation procedure for the nonterminal L.

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

#	*syntax*	*semantics*
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:

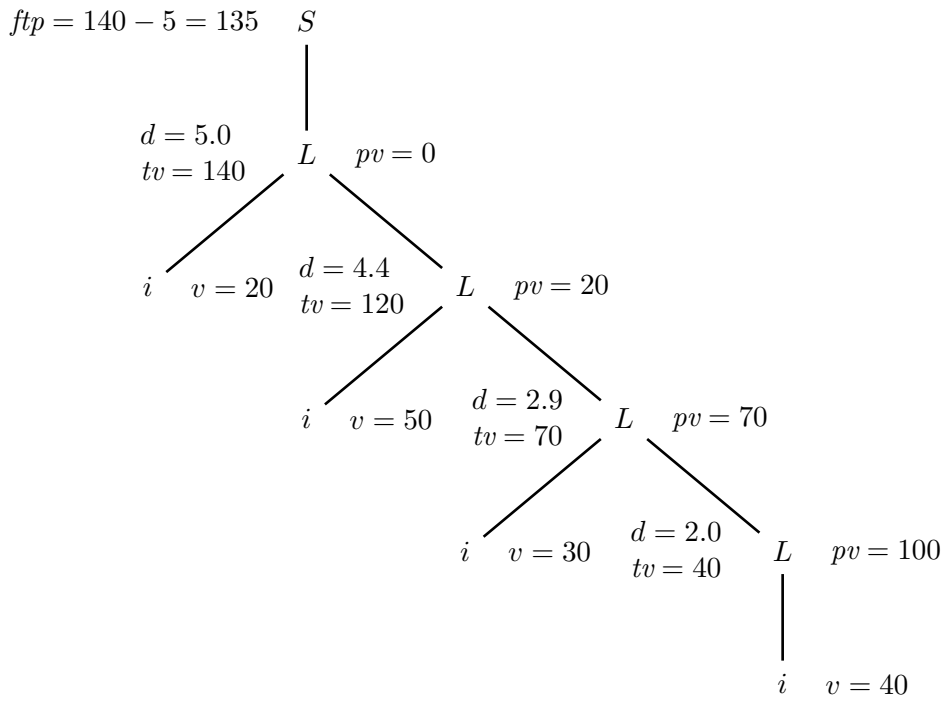
#	*syntax*	*semantics*
1:	$S_0 \rightarrow L_1$	$pv_1 = 0$ $ftp_0 = tv_1 - d_1$

2:	$L_0 \rightarrow i_1\ L_2$	$pv_2 = pv_0 + v_1$
		if $(pv_0 + v_1 \leq 100)$ then
		$\quad d_0 = d_2 + v_1 \times 0.03$
		else
		$\quad d_0 = d_2 + v_1 \times 0.05$
		endif
		$tv_0 = tv_2 + v_1$

3:	$L_0 \rightarrow i_1$	if $(pv_0 + v_1 \leq 100)$ then
		$\quad d_0 = v_1 \times 0.03$
		else
		$\quad d_0 = v_1 \times 0.05$
		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 donwards, 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 $\{\ i\,i\ \}$ and $\{\ i\ \dashv\ \}$, 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 sematic 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 pv0: real; out d0, tv0: real )
    var pv2, d2, tv2: real    - - local vars for updating the attributes
    - - checks look-ahead (k = 2)
    case ⟨ cc1, cc2 ⟩ of          - - syntax of type LL(2)
    ' i i ' : begin               - - (recursive) rule L → i L
                  next            - - computes v1 then shifts ⟨ cc1, cc2 ⟩
                  - - updates right attributes to pass downwards
                  pv2 = pv0 + v1
                  call L ( pv2; d2, tv2 )   - - recursive invocation of L
                  - - updates left attributes to pass upwards
                  if ( pv0 + v1 ≤ 100 ) then
                      d0 = d2 + v1 × 0.03
                  else
                      d0 = d2 + v1 × 0.05
                  end if
                  tv0 = tv2 + v1
              end
    ' i ⊣ ' : begin               - - (terminal) rule L → i
                  next            - - computes v1 then shifts ⟨ cc1, cc2 ⟩
                  - - updates left attributes to pass upwards
                  if ( pv0 + v1 ≤ 100 ) then
                      d0 = v1 × 0.03
                  else
                      d0 = v1 × 0.05
                  end if
                  tv0 = v1
              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:

```
procedure S ( out ftp0 : real )
    var pv1, d1, tv1 : real
    - - checks look-ahead (k = 1)
    if ( cc1 == ' i ' ) then
        - - updates right attributes
        pv1 = 0
        call L ( pv1; d1, tv1 )
        - - updates left attributes
        ftp0 = tv1 − d1
    else
        error    - - token ≠ i
    end if
end procedure


procedure next ( out v0 : real )
    - - computes term. attrib.
    v0 = value of cc1
    - - shifts window
    cc1 = cc2
    if ( cc2 ≠ ' ⊣ ' ) read ( cc2 )
end procedure
```

```
program EVALUATOR
    var ftp0 : real     - - translation
    - - initializes window
    read ( cc2 )
    cc1 = cc2
    if ( cc2 ≠ ' ⊣ ' ) read ( cc2 )
    - - launches analysis
    call S ( ftp0 )
    - - verifies acceptance
    if (⟨ cc1, cc2 ⟩ == ' ⊣⊣ ') then
        accept and output ftp0
    else
        reject (no output)
    end if
end program
```

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.