FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Mon 9 February 2015 - Part Theory

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

NAME:

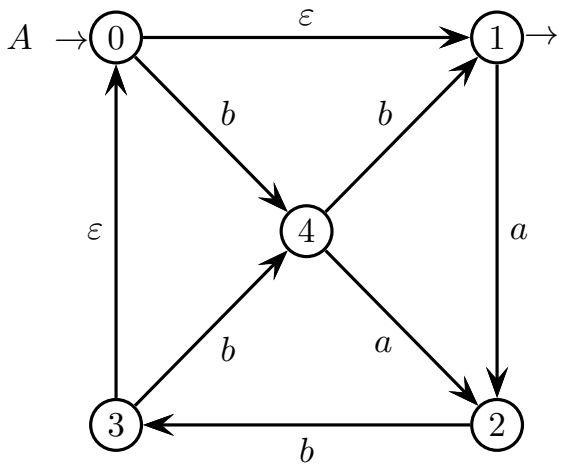
MATRICOLA: SIGNATURE:

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 also 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 nondeterministic finite state automaton A over the alphabet $\Sigma = \{a, b\}$, with spontaneous transitions (ε-transitions):



Answer the following questions:

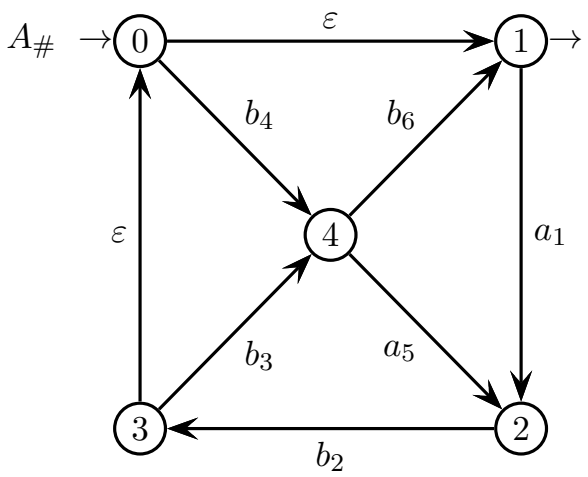
(a) By means of the Berry-Sethi method (*BSM*), find a deterministic automaton A' equivalent to A, and if necessary minimize the automaton A' and obtain the minimal automaton A'_{min}.

(b) By means of the node elimination method (Brzozowski) applied to the minimal automaton A'_{min} previously obtained, find a regular expression R equivalent to the automaton A.

(c) (optional) Cut the spontaneous transitions of automaton A, in such a way that the resulting automaton A'' is immediately deterministic (without any need to resort to *BSM* or to the subset construction after cutting).

Solution

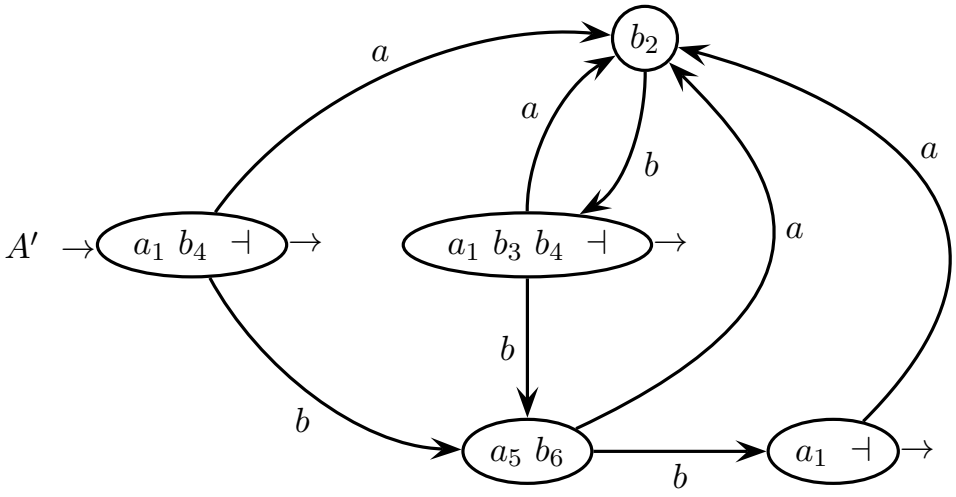
(a) Here is the Berry-Sethi automaton A', directly obtained from the automaton A. First the automaton A is numbered:



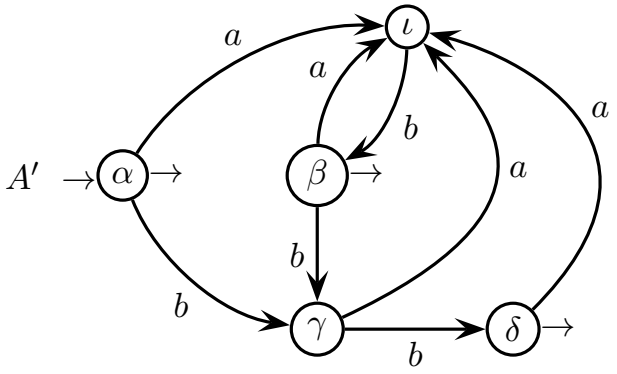
Then the initials and follows are listed:

initials	$a_1\ b_4\ \dashv$
generators	*follows*
a_1	b_2
b_2	$a_1\ b_3\ b_4\ \dashv$
b_3	$a_5\ b_6$
b_4	$a_5\ b_6$
a_5	b_2
b_6	$a_1\ \dashv$

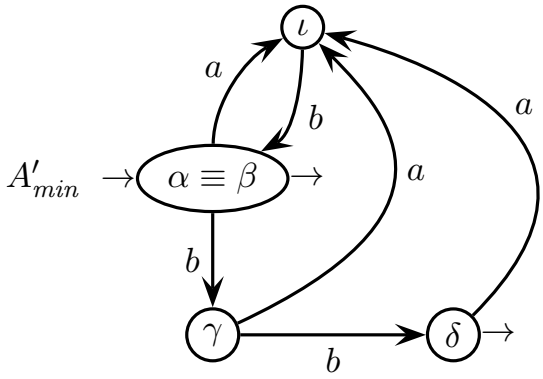
Finally the *BS* deterministic automaton A' is drawn:



It is obvious that the automaton A' is not minimal, as its final states $a_1\ b_4\ \dashv$ and $a_1\ b_3\ b_4\ \dashv$ are undistinguishable. To proceed, we rename the states as follows:

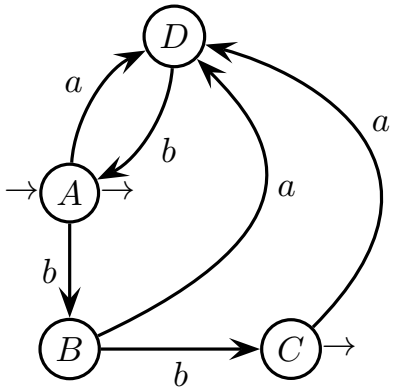


We see that $\alpha \equiv \beta$, but state δ is distinguishable from states $\alpha \equiv \beta$ because it misses a b-transition, and states γ and ι are distinguishable because state ι misses an a-transition. The minimal automaton A'_{min} is the following:



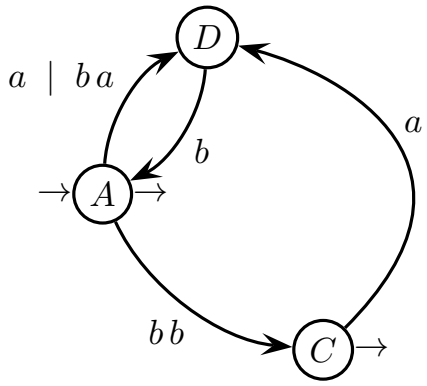
with four states.

(b) The minimal automaton A'_{min} has four states only. We proceed with some shortcut to speed up the procedure. First we rename the states of A'_{min}:

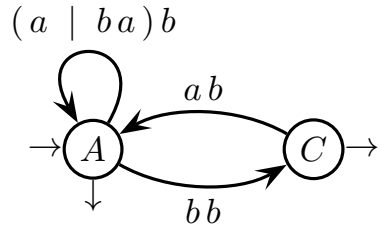


To start eliminating nodes, we should make sure to have only one initial state without ingoing arcs and only one final state without outgoing arcs. Currently this is not the case for states A (initial) and C (final), so we should adjust the graph. However, since for the moment we only want to eliminate the internal states of the graph, we can defer such an adjustment until it becomes necessary.

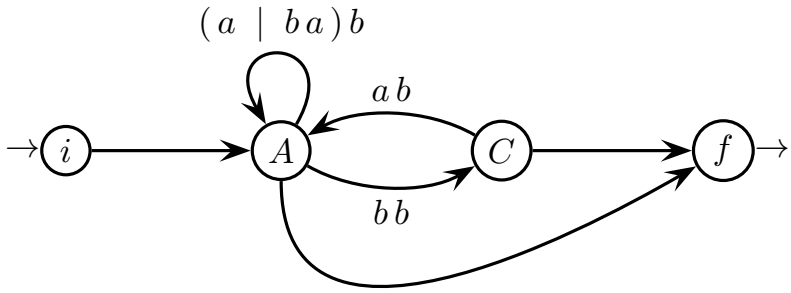
Eliminate node B:



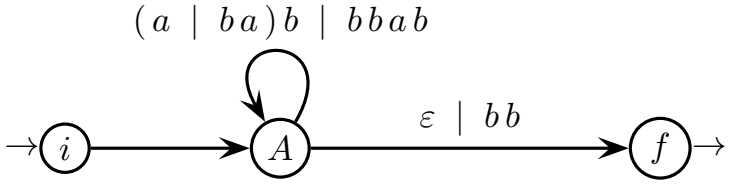
Eliminate node D:



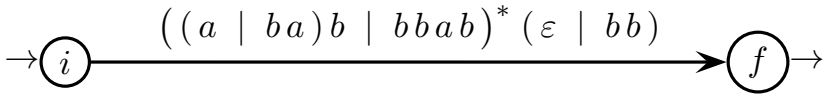
To go on, we need to have one initial state without ingoing arcs and one final state without outgoing arcs, so we have to introduce such new states now:



Eliminate node C:



Eventually, eliminate the last node A:



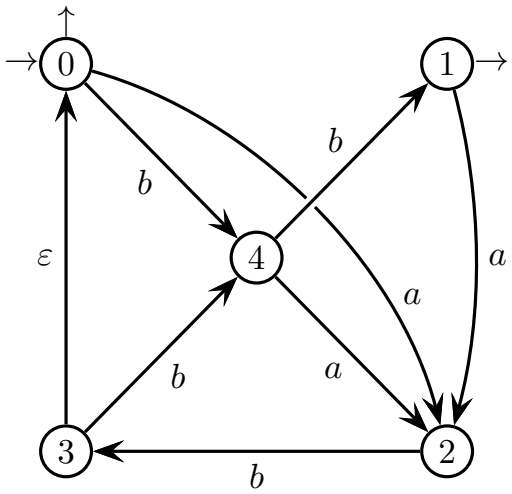
Therefore the regular expression R is the following:

$$R = \big((a \mid ba)\,b \mid bbab\big)^*(\varepsilon \mid bb)$$

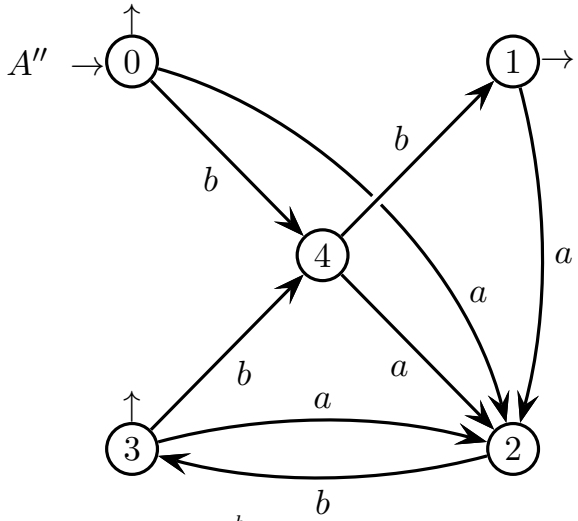
and with some simplification:

$$R = \big((\varepsilon \mid b \mid bb)\,ab\big)^*(\varepsilon \mid bb) = \big([b]_0^2\,ab\big)^*(\varepsilon \mid bb)$$

(c) Cut the arc $0 \xrightarrow{\varepsilon} 1$ by backward propagation:



Then cut the arc $3 \xrightarrow{\varepsilon} 0$ by backward propagation again:



Notice that in principle the arc $0 \xrightarrow{b} 3$ should be back-propagated onto state 3 and this would yield a new arc $3 \xrightarrow{b} 4$. However such an arc already exists, therefore introducing it again is useless. The resulting automaton A'' is deterministic. Notice that the final states 0 and 3 are undistinguishable. By unifying them, the minimal automaton A'_{min} would be reobtained.

2 Free Grammars and Pushdown Automata 20%

1. Write a non-extended (*BNF*) grammar for each of the following three languages L_1, L_2 and L_3, and draw the syntax trees of the strings listed below for each of them.

   (a) Language L_1 (axiom S):

   $$L_1 = \{\ (\ a\ \mid\ b\ )^n\, c^n \mid \quad n \geq 1\ \}$$

   where:

   $$s = a\, b\, a\, c\, c\, c$$

   (b) Language L_2 (axiom S):

   $$L_2 = \{\ a^m\, b^n\, c^p \mid \quad m, n, p \geq 1\ \wedge\ \ m + n = p\ \}$$

   where:

   $$t = a\, a\, b\, c\, c\, c$$

   (c) (optional) Language L_3 (axiom S):

   $$L_3 = \{\ a^m\, b^n\, c^p\, d^q \mid \quad m, n, p, q \geq 1\ \wedge\ m + n = p + q\ \}$$

   where:

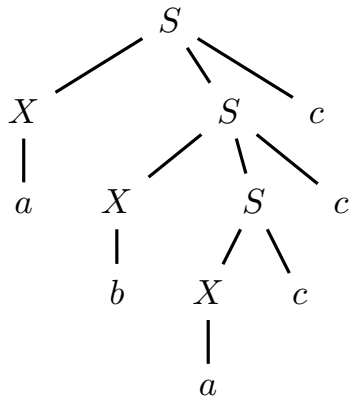
   $$v_1 = a\, a\, b\, c\, c\, d \qquad v_2 = a\, b\, b\, c\, d\, d \qquad v_3 = a\, b\, b\, c\, c\, d$$

   The grammar of language L_3 should not be ambiguous.

Solution

(a) Grammar G_1 of language L_1 and syntax tree of string $s = a\,b\,a\,c\,c\,c$:

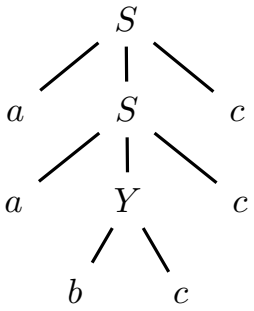
$$G_1 \begin{cases} S & \rightarrow & X\ S\ c \ \mid\ X\ c \\ X & \rightarrow & a \ \mid\ b \end{cases}$$



Notice that grammar G_1 is not ambiguous (though this was not requested), as basically it is the well known grammar of a parenthetic structure of type $a^n\,c^n$, where the left half of the string may contain n letters a and b in a free order.

(b) Grammar G_2 of language L_2 and syntax tree of string $t = a\,a\,b\,c\,c\,c$:

$$G_2 \begin{cases} S & \rightarrow & a\ S\ c \ \mid\ a\ Y\ c \\ Y & \rightarrow & b\ Y\ c \ \mid\ b\ c \end{cases}$$



Notice again that grammar G_1 is not ambiguous (though this was not requested either), as basically, like grammar G_1 before, it is the well known grammar of a parenthetic structure of type $a^n\,c^n$, where the left half of the string may contain n letters a and b with the constraint (differently from grammar G_1) that all the letters a must precede all the letters b.

(c) Grammar G_3 of language L_3:

$$G_3 \begin{cases} S & \rightarrow & a\ S\ d \ \mid\ a\ Z\ d \ \mid\ a\ X\ d \ \mid\ a\ Y\ d \\ X & \rightarrow & b\ X\ d \ \mid\ b\ Z\ d \\ Y & \rightarrow & a\ Y\ c \ \mid\ a\ Z\ c \\ Z & \rightarrow & b\ Z\ c \ \mid\ b\ c \end{cases}$$

Here is the rationale of grammar G_3 for language L_3. From the language constraint $m + n = p + q$ the following properties hold:

- if $m > q$ then $n < p \ \wedge \ m - q = p - n$, the string has the form:
  $a^q \, a^{m-q} \, b^n \, c^n \, c^{p-n} \, d^q$
  and can be derived as follows:
  $S \overset{q}{\Rightarrow} a^q \, Y \, d^q \overset{m-q}{\Rightarrow} a^q \, a^{m-q} \, Z \, c^{p-n} \, d^q \overset{n}{\Rightarrow} a^q \, a^{m-q} \, b^n \, c^n \, c^{p-n} \, d^q$
- if $m < q$ then $n > p \ \wedge \ q - m = n - p$, the string has the form:
  $a^m \, b^{n-p} \, b^p \, c^p \, d^{q-m} \, d^m$
  and can be derived as follows:
  $S \overset{m}{\Rightarrow} a^m \, X d^m \overset{n-p}{\Rightarrow} a^m \, b^{n-p} \, Z \, d^{q-m} \, d^m \overset{p}{\Rightarrow} a^m \, b^{n-p} \, b^p \, c^p \, d^{q-m} \, d^m$
- if $m = q$ then $n = p$, the string has the form:
  $a^m \, b^n \, c^p \, d^q$
  and can be derived as follows:
  $S \overset{m}{\Rightarrow} a^m \, Z \, d^q \overset{p}{\Rightarrow} a^m \, b^n \, c^p \, d^q$

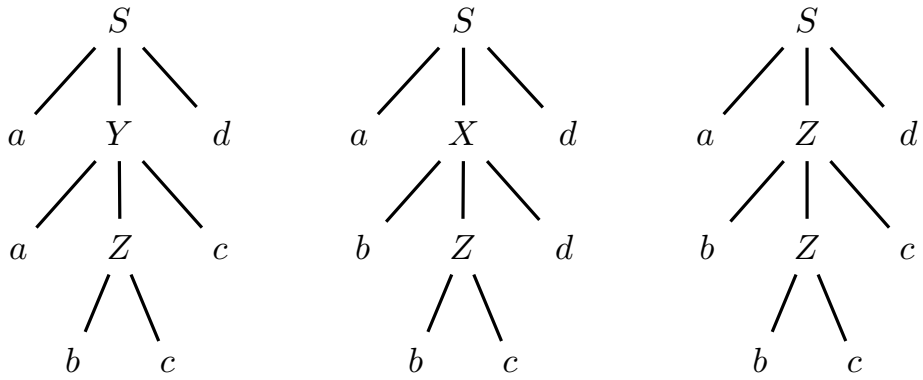
Here is a summary of the three cases:

$$m > q \quad \Rightarrow \quad n < p \quad \wedge \quad m - q = p - n$$

$$m < q \quad \Rightarrow \quad n > p \quad \wedge \quad q - m = n - p$$

$$m = q \quad \Rightarrow \quad n = p$$

Therefore grammar G_2 is not ambiguous, as it was requested. Again, it consists of self-embedding (auto-inclusive) rules. It is clear that rule $S \to a \, S \, d$ yields equal numbers of a's and d's, rule $X \to b \, X \, d$ yields equal numbers of b's and d's, rule $Y \to a \, Y \, c$ yields equal numbers of a's and c's, and rule $Z \to b \, Z \, c$ yields equal numbers of b's and c's, while the remaining rules just represent transition cases. Therefore which self-embedding rule should be used at each derivation step, is determined by the respective differences of letter numbers, as shown in the rationale before. This means there is only one syntax tree for each language string, and the grammar is not ambiguous. See also the sample trees below.
Here are the three syntax trees of the strings v_1, v_2 and v_3:

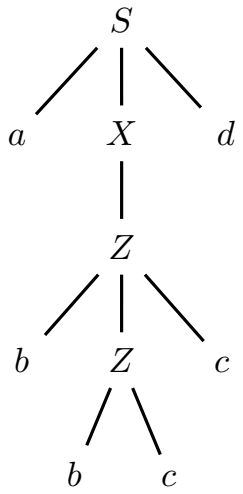


$v_1 = a \, a \, b \, c \, c \, d$ $\qquad$ $v_2 = a \, b \, b \, c \, d \, d$ $\qquad$ $v_3 = a \, b \, b \, c \, c \, d$

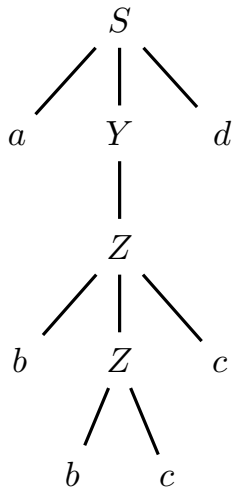
Notice that if we stand ambiguity, then grammar G_3 can be simplified as follows:

$$G'_3 \begin{cases} S \rightarrow a\ S\ d \mid a\ X\ d \mid a\ Y\ d \\ X \rightarrow b\ X\ d \mid Z \\ Y \rightarrow a\ Y\ c \mid Z \\ Z \rightarrow b\ Z\ c \mid b\ c \end{cases}$$

Not surprisingly, the ambiguous grammar G'_3 has two copy rules: $X \rightarrow Z$ and $Y \rightarrow Z$. In fact, it is well known that copy rules help to shorten a grammar, but often (though not necessarily) cause it to become ambiguous. Anyway, in the grammar G'_3 all the strings $a^m\, b^n\, c^n\, d^m$ (m, $n \geq 1$) are ambiguous (of degree two), since their derivations can pass through the nonterminals X or Y and then reach the nonterminal Z by either copy rule. For instance, with grammar G'_3 the sample string v_3 has two syntax trees, namely:



$v_3 = a\,b\,b\,c\,c\,d$



$v_3 = a\,b\,b\,c\,c\,d$

2. Consider a simplified version of the language of the first order logical formulas (predicate calculus), where a logical formula has the following syntactic features:

   - a formula starts with a list, possibly empty, of existential and universal quantifiers, respectively denoted by "E" and "U", each of which has one or more variables in its scope, separated by comma "," (so called prenex form)
   - a formula continues with a logical expression that may contain the logical operators sum "or", product "and", negation "not", the round parentheses "(" and ")" to embrace logical subexpressions, and predicates (see below)
   - the operator priority in a logical expression is as usual: logical negation (highest), logical product (middle), logical sum (lowest)
   - a predicate is a name followed by a list, delimited by round parentheses, of one or more arguments separated by comma ","
   - a predicate argument is a variable
   - a variable or a predicate name is an identifier, schematized by a terminal id

   Sample formulas:

   ```
   E x U y  pred1 (x)

   U alpha, beta E gamma  pred1 (alpha) or pred2 (beta, gamma)

   E x, y U p1, p2  (pred1 (x) or pred2 (p1, y)) and not pred2 (x, p2)
   ```

   A language phrase is a non-empty list of logical formulas as above. The formulas are separated by a semicolon ";" and the list is terminated by a dot ".".

   Answer the following questions:

   (a) Write an extended grammar (*EBNF*), not ambiguous, that reasonably models the logical language sketched above.

   (b) (optional) We wish we extended the language and admitted also non-prenex formulas, where the quantifiers may appear in the middle, like:

   ```
   U alpha E beta  pred1 (alpha) or U gamma  pred2 (beta, gamma)
   ```

   Show which rules of the previous grammar should be modified to model the extended language. The new grammar should not be ambiguous either.

Solution

(a) Here is the requested extended grammar (axiom LANG):

$$
\begin{cases}
\langle \mathsf{LANG} \rangle \;\to\; \langle \mathsf{FORM} \rangle \; (\; \text{`;'} \; \langle \mathsf{FORM} \rangle \;)^{+} \; \text{`.'} \\
\hline
\langle \mathsf{FORM} \rangle \;\to\; \langle \mathsf{QUANT} \rangle \; \langle \mathsf{EXPR} \rangle \\
\hline
\langle \mathsf{QUANT} \rangle \;\to\; (\; \langle \mathsf{UNIV} \rangle \;|\; \langle \mathsf{EXIST} \rangle \;)^{*} \\
\hline
\langle \mathsf{UNIV} \rangle \;\to\; \text{`U'} \; \langle \mathsf{VARLIST} \rangle \\
\langle \mathsf{EXIST} \rangle \;\to\; \text{`E'} \; \langle \mathsf{VARLIST} \rangle \\
\hline
\langle \mathsf{EXPR} \rangle \;\to\; \langle \mathsf{TERM} \rangle \; (\; \mathsf{or} \; \langle \mathsf{TERM} \rangle \;)^{*} \\
\langle \mathsf{TERM} \rangle \;\to\; \langle \mathsf{FACT} \rangle \; (\; \mathsf{and} \; \langle \mathsf{FACT} \rangle \;)^{*} \\
\langle \mathsf{FACT} \rangle \;\to\; [\; \mathsf{not} \;] \; \langle \mathsf{OBJ} \rangle \\
\langle \mathsf{OBJ} \rangle \;\to\; \langle \mathsf{PRED} \rangle \;|\; \text{`('} \; \langle \mathsf{EXPR} \rangle \; \text{`)'} \\
\hline
\langle \mathsf{PRED} \rangle \;\to\; \mathsf{id} \; \text{`('} \; \langle \mathsf{VARLIST} \rangle \; \text{`)'} \\
\hline
\langle \mathsf{VARLIST} \rangle \;\to\; \mathsf{id} \; (\; \text{`,'} \; \mathsf{id} \;)^{*}
\end{cases}
$$

As usual, the square brackets mean optionality. This grammar is extended and not ambiguous, because it is made of substructures and rule models that are known to be not ambiguous. The precedence of the logical operators or, and, not is correctly set as required. There may be different formulations of the grammar, or slightly different interpretations of the language structure.

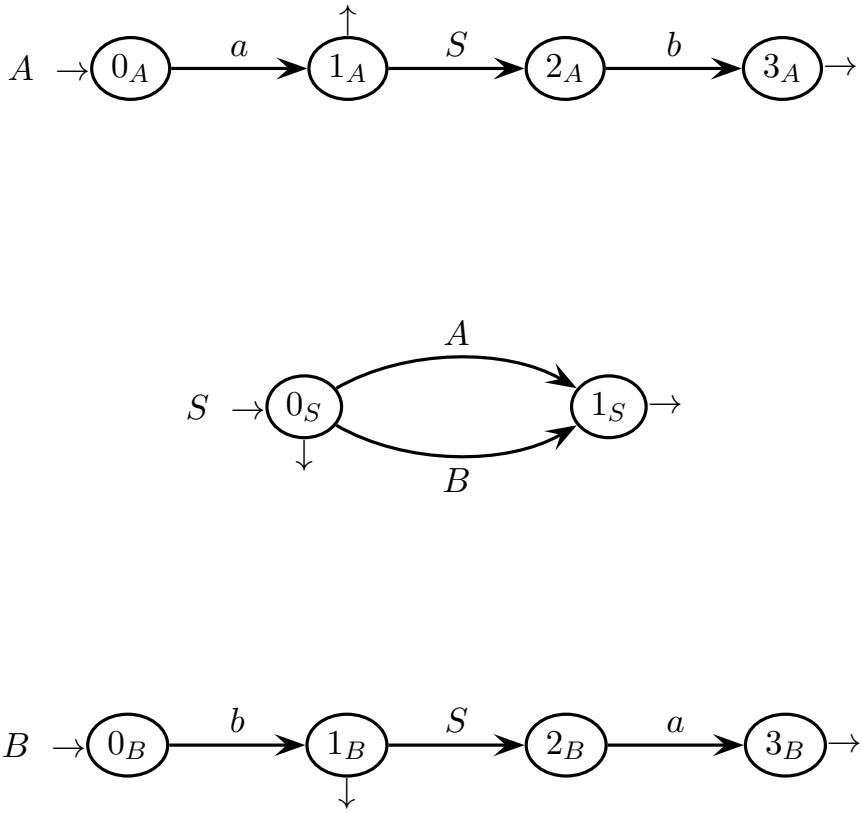
(b) It suffices to modify the rules below (all the others are left unchanged):

$$
\begin{cases}
\langle \mathsf{EXPR} \rangle \;\to\; \langle \mathsf{TERM} \rangle \; (\; \mathsf{or} \; \langle \mathsf{QUANT} \rangle \; \langle \mathsf{TERM} \rangle \;)^{*} \\
\langle \mathsf{TERM} \rangle \;\to\; \langle \mathsf{FACT} \rangle \; (\; \mathsf{and} \; \langle \mathsf{QUANT} \rangle \; \langle \mathsf{FACT} \rangle \;)^{*} \\
\langle \mathsf{FACT} \rangle \;\to\; \mathsf{not} \; \langle \mathsf{QUANT} \rangle \; \langle \mathsf{OBJ} \rangle \;|\; \langle \mathsf{OBJ} \rangle \\
\langle \mathsf{OBJ} \rangle \;\to\; \langle \mathsf{PRED} \rangle \;|\; \text{`('} \; \langle \mathsf{FORM} \rangle \; \text{`)'}
\end{cases}
$$

Here every term, factor or object may be preceded by quantifiers, except when it is located at the beginning of an expression, otherwise the modified grammar would be ambiguous as the quantifiers can also be generated by the syntactic class QUANT in the prenex position of the entire logical formula. A subexpression embraced in round brackets may have quantifiers in the prenex position as well, so that a subexpression can now be an entire logical formula.

3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the following machine net (where S is the axiom symbol):



Answer the following questions:

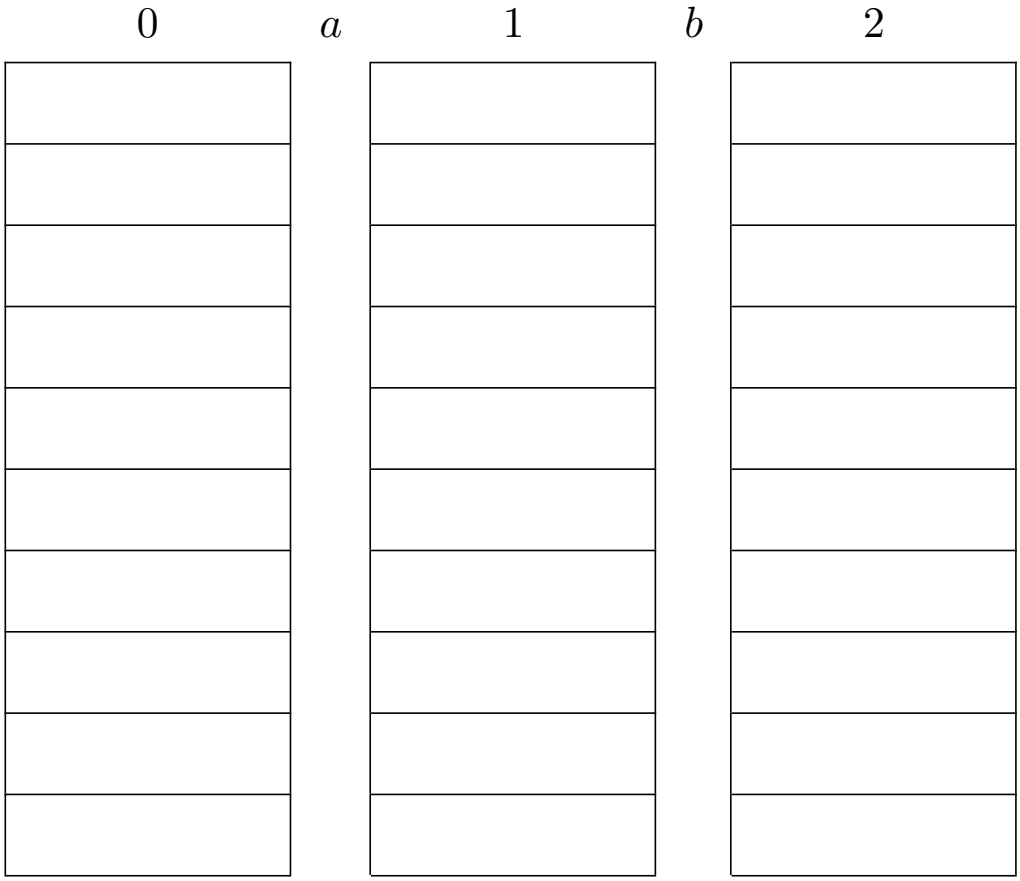
(a) Compute the guide sets of the machine net (on the call arcs and exit darts), determine if the machine net is $ELL(1)$ and provide a suitable brief explanation.

(b) Determine if the machine net is $ELR(1)$ by building the necessary portion of the pilot. If the $ELR(1)$ condition is not satisfied, list all the conflicts present in the produced (portion of the) pilot and briefly explain what conflict each of them is.

(c) Consider the valid phrase "$a\,b$" and sketch its syntax tree (or trees if the phrase is ambiguous). Then execute the Earley syntax analysis of such a phrase and indicate the items in the Earley vector that contribute to acceptance (use the vector prepared on the next page).

(d) (optional) Say if the grammar is ambiguous and briefly explain your answer.

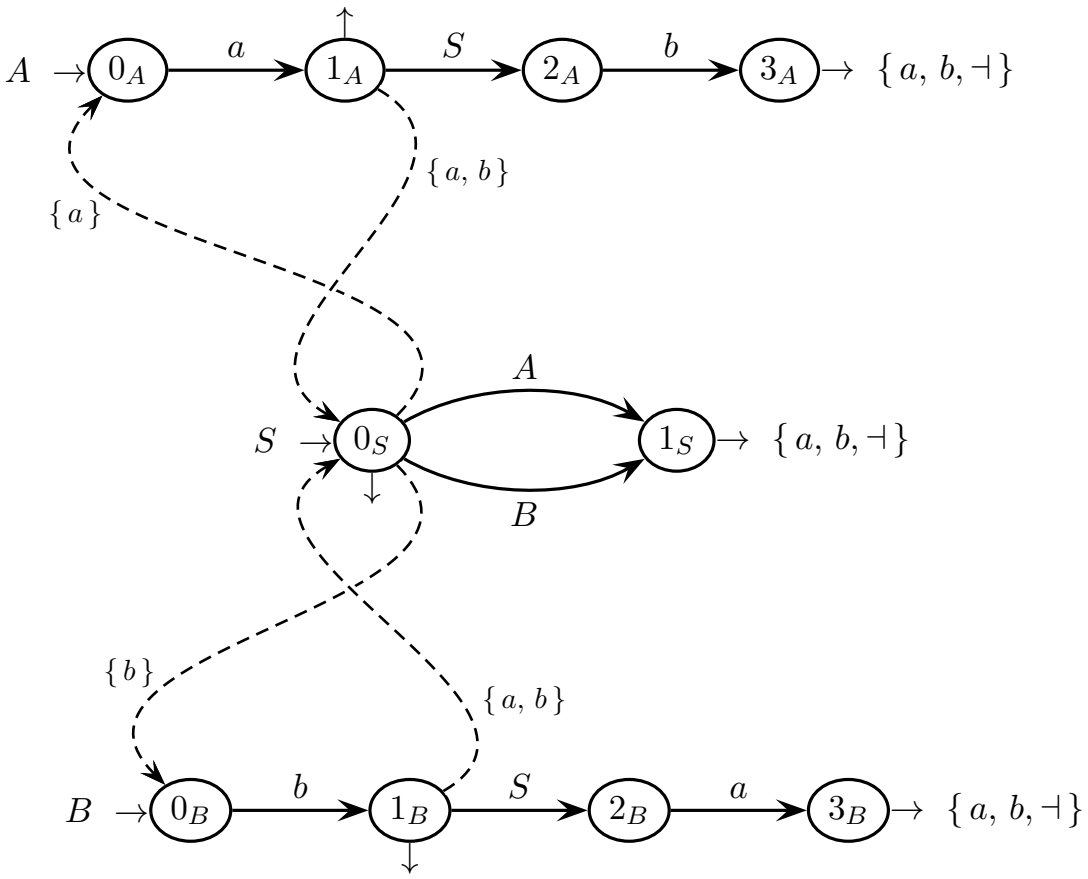
Earley vector of string $a\ b$ (to be filled)
(the number of rows is not significant)



Draw here the requested syntax tree(s)

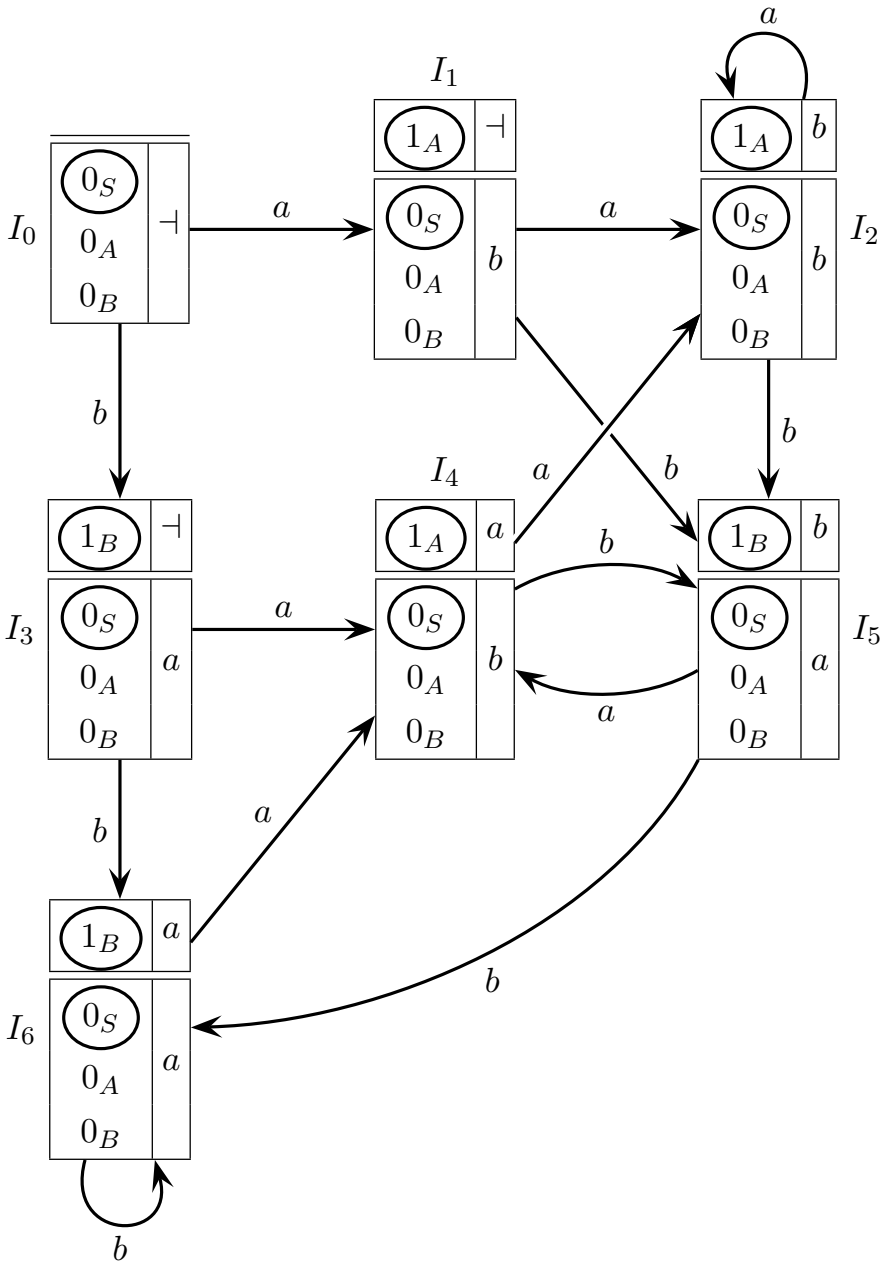
Solution

(a) Here is the *ELL* pilot of the machine net:



The prospect sets are identical on all the exit darts of each machine, thus here they are shown only once per machine. The final states 1_A and 1_B have non-disjoint guide sets (on the call arcs and exit darts), as well as the final state 0_S. Therefore the machine net (grammar) is not of type *ELL*(1).

(b) Here is a partial *ELR* pilot of the machine net:

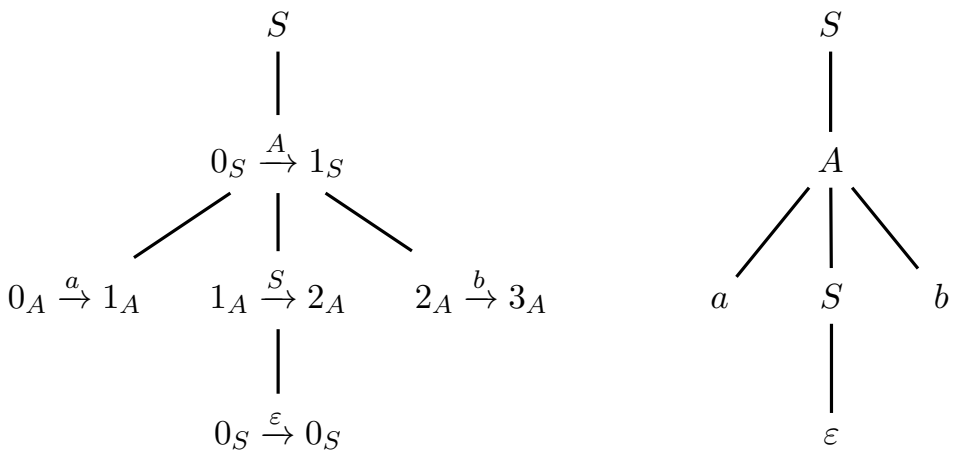


The missing m-states and arcs are related to the nonterminal shift moves on S, A and B, and here are of little interest. The partial pilot has a few shift-reduce and reduce-reduce conflicts (the reader may list them). Therefore the machine net (grammar) is not of type $ELR\,(1)$.

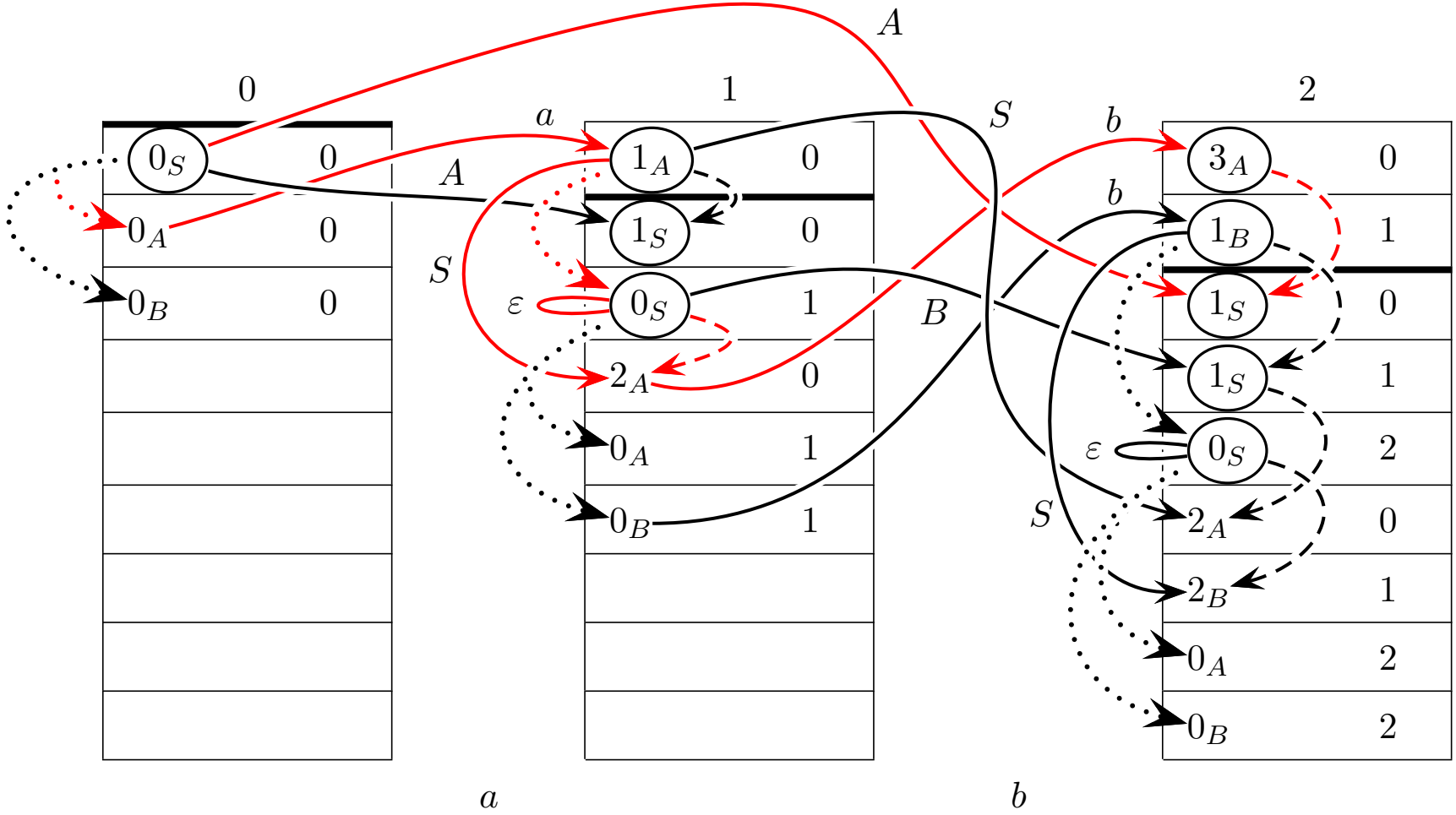
(c) Earley analysis of string $a\,b$:

0		a	1		b	2	
$\textcircled{0_S}$	0		$\textcircled{1_A}$	0		$\textcircled{3_A}$	0
0_A	0		$\textcircled{1_S}$	0		$\textcircled{1_B}$	1
0_B	0		$\textcircled{0_S}$	1		$\textcircled{1_S}$	0
			2_A	0		$\textcircled{1_S}$	1
			0_A	1		$\textcircled{0_S}$	2
			0_B	1		2_A	0
						2_B	1
						0_A	2
						0_B	2

Notice that the vector element $E\,(1)$ contains the reduction for an ε-move of the analyzer, namely the move associated to rule $S \to \varepsilon$, as the axiom S is nullable (see also the next page). Since the axiomatic final item $\left\langle \textcircled{1_S},\, 0 \right\rangle$ with pointer 0 is present in the last vector state $E\,(2)$, the string is accepted.



The tree on the left (not requested) shows the paths in the machine net. The full display of all the moves is shown on the next page, where the red coloured arrows map the syntax tree of the accepted string.


0
1
2
0_S 0
0_A 0
0_B 0
1_A 0
1_S 0
0_S 1
2_A 0
0_A 1
0_B 1
3_A 0
1_B 1
1_S 0
1_S 1
0_S 2
2_A 0
2_B 1
0_A 2
0_B 2
A
a
S
ε
B
b
a
b

(d) We see that the machines A and B are similar to those of the palindrome language and that they nest into each other by passing through machine S. The network calls machine A or B depending on which letter (a or b) is found in the left half of the generated string. We conclude that the sequence of machine calls is determined by the left half of the string in a unique way, thus every language string has only one syntax tree, therefore the grammar is not ambiguous.

So this particular grammar is not ambiguous, yet it is not deterministic. Of course, in principle there might still exist an equivalent deterministic grammar. Anyway, the language is unlikely to have a deterministic grammar of whatever type. We easily see that the right half of a language string is the reversed image of the left half, with letters a and b exchanged, whereas the central letter (still an a or b) may be missing. Thus the language is quite similar (though not identical) to the language of the palindrome strings without a centre, which is well known to be not ambiguous yet intrinsically nondeterministic. In fact, a deterministic pushdown analyzer does not have any clue on how to find the string midpoint.

4 Language Translation and Semantic Analysis 20%

1. Consider a source language of two-level lists with elements schematized by a, without separators, embraced by round parentheses "(" and ")", like for instance:

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

   We wish we had a one-valued translation τ that "flattens" such lists and reduces them to one-level only, as follows:

   $$(\; a \;\; a \;\; (\; a \;\; a \;) \;\; a \;\; (\; a \;) \;) \overset{\tau}{\mapsto} (\; a \;\; a \;\; 2\,a \;\; 2\,a \;\; a \;\; 2\,a \;)$$

   by introducing the terminal "2" to mark the elements originally at level two. Consider the following non-extended (BNF) source grammar (axiom S), which generates the source language:

   $$\begin{array}{lrcl} 1: & S & \to & (\; L_1 \;) \\ 2: & L_1 & \to & a \; L_1 \\ 3: & L_1 & \to & a \\ 4: & L_1 & \to & (\; L_2 \;) \; L_1 \\ 5: & L_1 & \to & (\; L_2 \;) \\ 6: & L_2 & \to & a \; L_2 \\ 7: & L_2 & \to & a \end{array}$$

   Answer the following questions:

   (a) Write a destination grammar associated to the above source grammar to compute the translation τ, and draw the syntax tree of the sample translation above. Say if the translation scheme for τ is deterministic and briefly explain your answer.

   (b) Consider the inverse translation τ^{-1}, say if it is deterministic and briefly explain your answer.

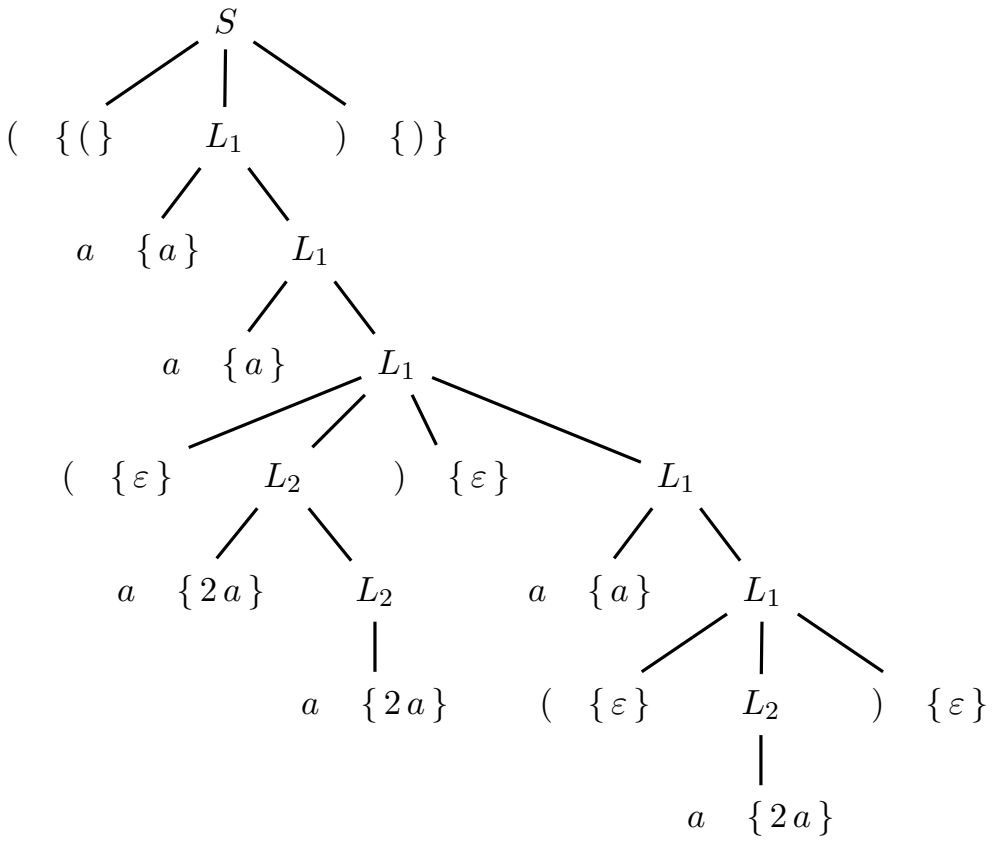
   (c) (optional) The source language (two-level lists) is well known to be regular. Write a rational translation expression R_τ that computes the translation τ.

Solution

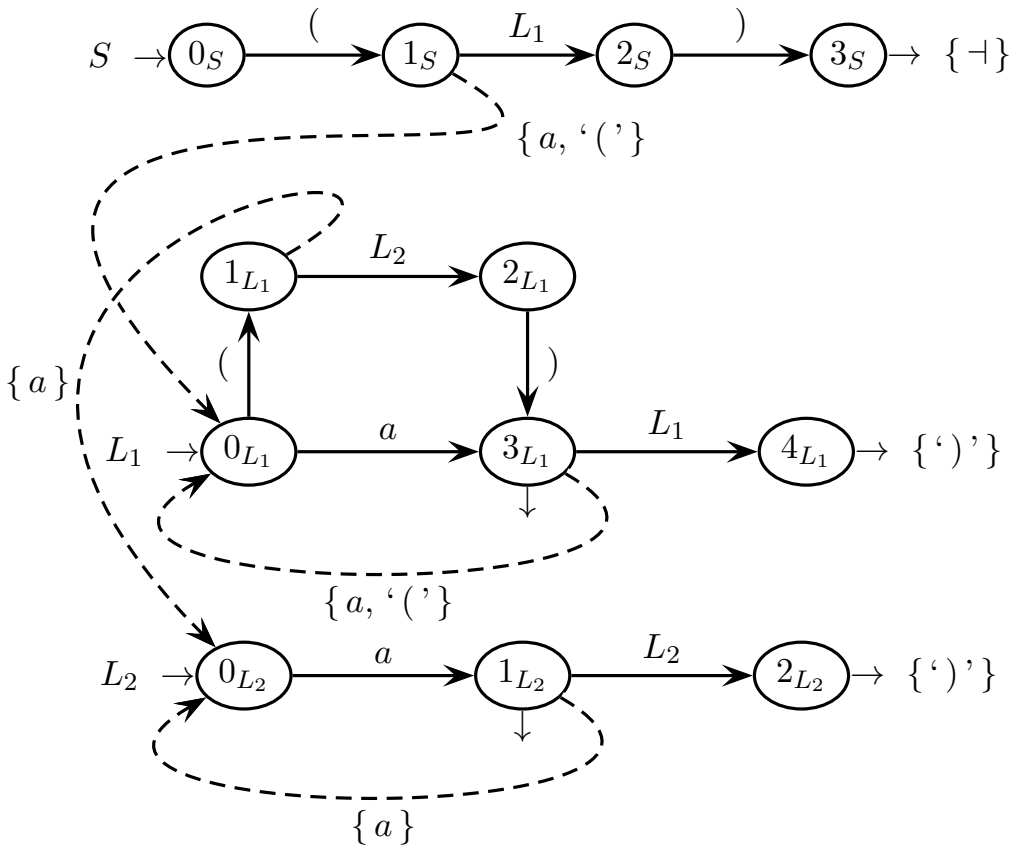
(a) Here is the destination grammar of the translation τ, which cancels the round brackets of the second-level lists and marks their elements by the terminal 2:

source grammar				*destination grammar*			
1:	S	$\to$	$(\ L_1\ )$	1:	S	$\to$	$(\ L_1\ )$
2:	L_1	$\to$	$a\ L_1$	2:	L_1	$\to$	$a\ L_1$
3:	L_1	$\to$	a	3:	L_1	$\to$	a
4:	L_1	$\to$	$(\ L_2\ )\ L_1$	4:	L_1	$\to$	$L_2\ L_1$
5:	L_1	$\to$	$(\ L_2\ )$	5:	L_1	$\to$	L_2
6:	L_2	$\to$	$a\ L_2$	6:	L_2	$\to$	$2\,a\ L_2$
7:	L_2	$\to$	a	7:	L_2	$\to$	$2\,a$

Here is the requested syntax tree (unified source and destination trees):

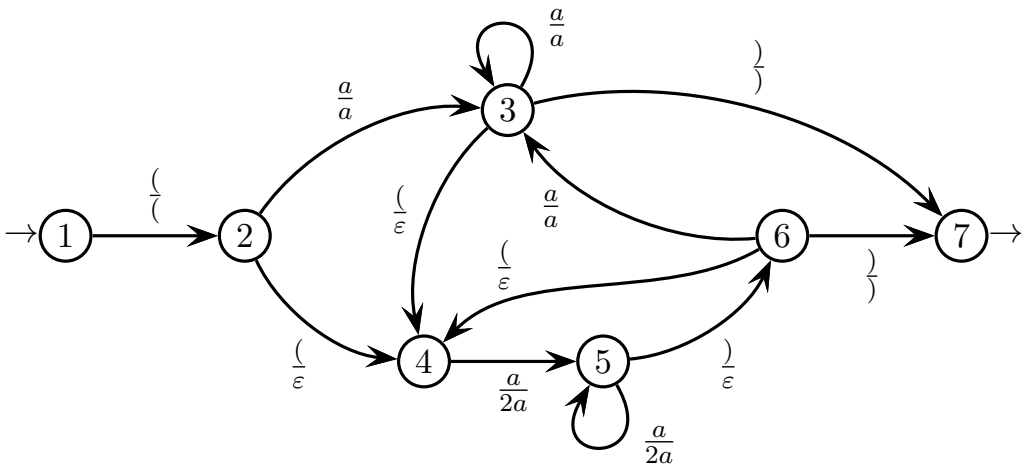


The syntactic scheme shown above is not deterministic $LL(k)$ (for any $k \geq 1$) due to the alternative source rules 4 and 5, since the nonterminal L_2 has initials of arbitrary length. However, the scheme becomes deterministic if it is transformed into a machine net and is analyzed according to the $ELL(1)$ methodology. Here is the machine net of the source grammar with the guide sets, where it helps:



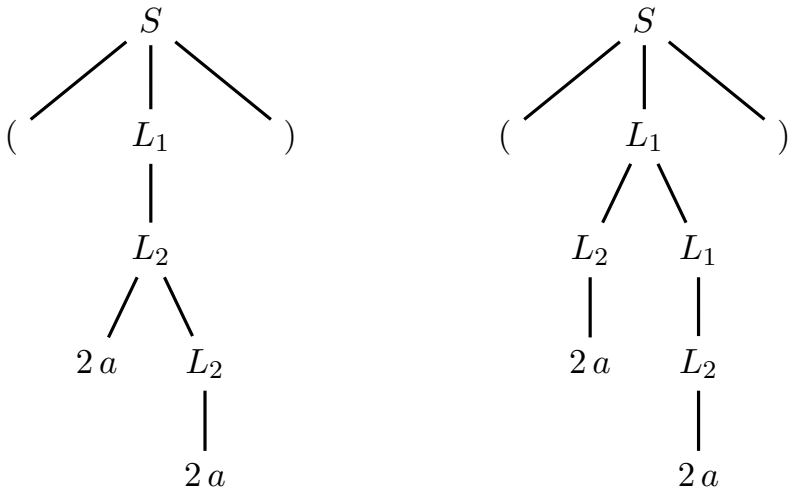
The only bifurcation points are states 3_{L_1} and 1_{L_2}, and both have disjoint guide sets. Thus the source net is $ELL(1)$, i.e., deterministic. A recursive descent syntax analyzer could be easily written, with the appropriate write actions. We conclude that the translation τ is deterministic, when it is properly analyzed.

A different approach to prove that the translation τ is deterministic, is to first notice that the source language is regular, as the recursive rules 2 and 6 are of the right-linear type, the rule 4 is only right-recursive, and the grammar does not have any indirect recursion. Said differently, the grammar does not have any recursive self-embedding (auto-inclusive) derivation. Thus there exists a rational transducer, in general nondeterministic, that computes the translation τ. But for this particular translation τ it is even possible to find a deterministic one:



It is easy to see that the underlying source recognizer is deterministic, so that the transducer itself is deterministic, similarly to what concluded before.

(b) The destination grammar is ambiguous. The reader may wish to verify that the string "(2 *a* 2 *a*)" has two trees in the destination grammar, namely:



The corresponding source trees generate the two source strings listed below. Therefore the inverse syntactic scheme for transduction τ^{-1} is not deterministic, no matter what methodology (*ELR* or *ELL*) is used to analyze it.

We could wonder if it is possible to find an equivalent deterministic scheme. However, we see soon that the inverse translation τ^{-1} is not one-valued, e.g., the translated string "(2 *a* 2 *a*)" maps back to both source strings "((*a a*))" and "((*a*) (*a*))". Therefore there may not exist any deterministic syntactic scheme for the inverse translation τ^{-1}, which is intrinsically non-deterministic.

(c) Here is a viable rational translation expression R_τ of translation τ:

$$R_\tau = \frac{\text{`('}}{\text{`('}} \left(\frac{a}{a} \mid \frac{\text{`('}}{\varepsilon} \left( \frac{a}{2\,a} \right)^{+} \frac{\text{`)'}}{\varepsilon} \right)^{+} \frac{\text{`)'}}{\text{`)'}}$$

which is simple enough to deserve no comment. Clearly the rational transducer shown at point (*a*) is equivalent to the rational translation expression R_τ.

A final observation. If we flip the numerators and denominators in the rational translation expression R_τ, we obtain the rational translation expression $R_{\tau^{-1}}$ of the inverse translation τ^{-1}. It is evident that the latter expression is ambiguous. In fact, a substring like "2 *a* 2 *a*" is generated by the (top) subexpression "$(\varepsilon\ (2\,a)^+\ \varepsilon)^+$" of $R_{\tau^{-1}}$ in two ways: either by iterating the outer cross twice and the inner cross once, or by iterating the outer cross once and the inner cross twice, i.e., like "$\varepsilon\ 2\,a\ \varepsilon\ \varepsilon\ 2\,a\ \varepsilon$" "or $\varepsilon\ 2\,a\ 2\,a\ \varepsilon$". Yet in either case the source string would contain different numbers of round brackets, or brackets displaced in different ways, as explained at point (*b*).

Similarly, if the source and destination labels on the arcs of the rational transducer shown at point (*b*) are flipped, the underlying recognizer has spontaneous transitions (ε-transitions), so it is not deterministic, and such spontaneous transitions are unlikely to be eliminable (remember that the nondeterministic and deterministic rational transducers are not equivalent, in general).

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:

$$
\begin{array}{lrcl}
1: & S & \rightarrow & F \\
2: & F & \rightarrow & F \text{ and } F \\
3: & F & \rightarrow & \text{not } F \\
4: & F & \rightarrow & \text{a}
\end{array}
$$

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:

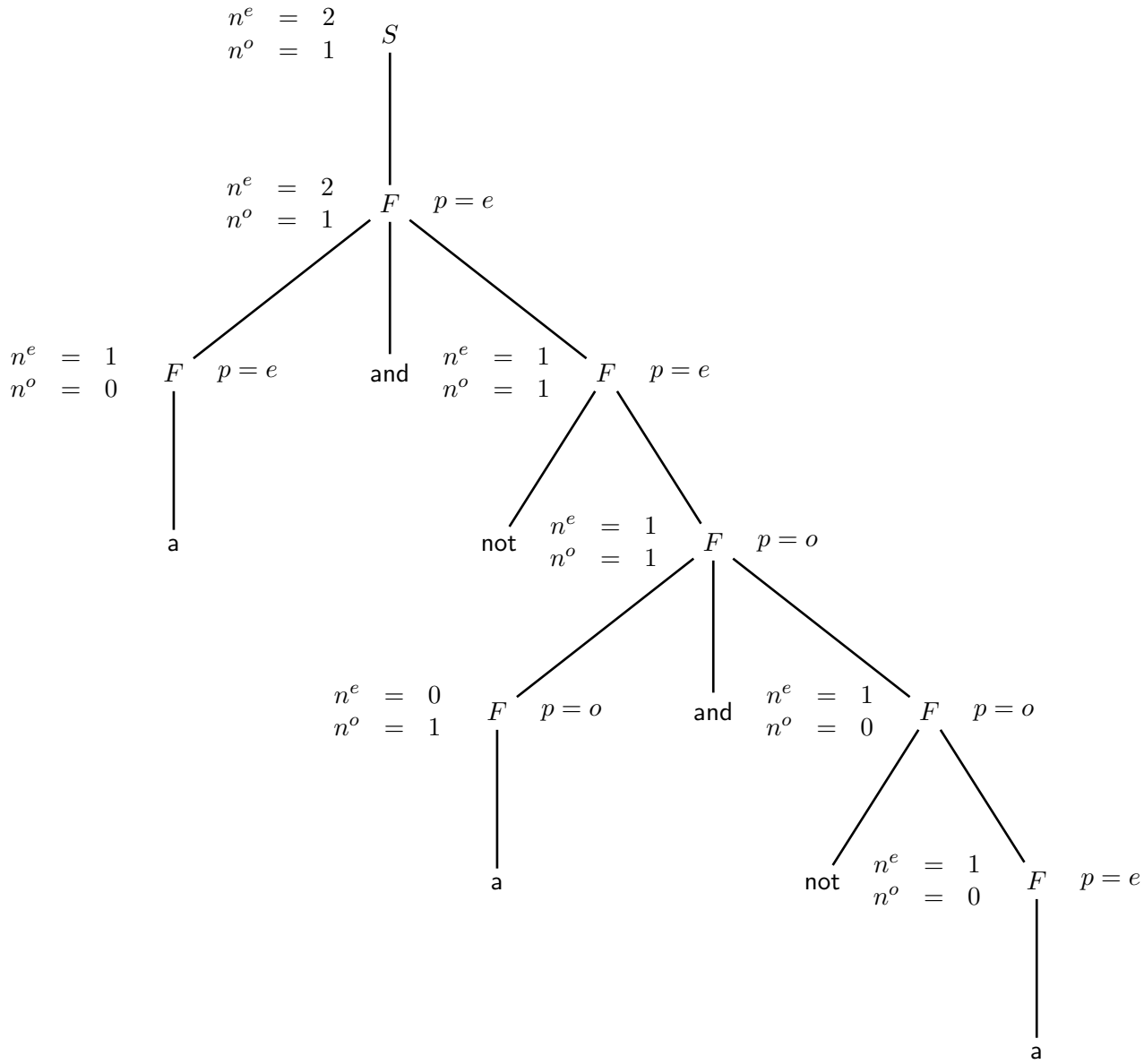
- $p \in \{ e, o \}$ is a right attribute that indicates if the associated nonterminal is in the scope of an even or odd number of negations
- n^e and n^o are two left integer attributes that count the number of letters in the scope of an even or odd number of negations, respectively.

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)

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

type	*name*	*domain*	*nonterm.*	*meaning*
left	*even*	integer ≥ 0	S ...	number of propositional letters that are in the scope of an even number of negations
left	*odd*	integer ≥ 0	S ...	number of propositional letters that are in the scope of an odd number of negations

#	*syntax*	*semantics* - question a

1: $S_0 \rightarrow F_1$

2: $F_0 \rightarrow F_1$ and F_2

3: $F_0 \rightarrow$ not F_1

4: $F_0 \rightarrow$ a

#	*syntax*	*semantics* - question b

1: $S_0 \rightarrow F_1$

2: $F_0 \rightarrow F_1$ and F_2

3: $F_0 \rightarrow$ not F_1

4: $F_0 \rightarrow$ a

Solution

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

#	*syntax*	*semantics* - question a
1:	$S_0 \rightarrow F_1$	$p_1 := e$ $n_0^o := n_1^o$ $n_0^e := n_1^e$
2:	$F_0 \rightarrow F_1$ 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:	$F_0 \rightarrow$ not F_1	$p_1 :=$ **if** $(p_0 = e)$ **then** o **else** e **endif** $n_0^o := n_1^o$ $n_0^e := n_1^e$
4:	$F_0 \rightarrow$ a	$n_0^e :=$ **if** $(p_0 = e)$ **then** 1 **else** 0 **endif** $n_0^o :=$ **if** $(p_0 = o)$ **then** 1 **else** 0 **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 $F \rightarrow F$ 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 synthetized attributes only:

#	*syntax*	*semantics* - question b
1:	$S_0 \rightarrow F_1$	$even_0 := even_1$ $odd_0 := odd_1$
2:	$F_0 \rightarrow F_1$ and F_2	$even_0 := even_1 + even_2$ $odd_0 := odd_1 + odd_2$
3:	$F_0 \rightarrow$ 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 synthethized. As before, it is not of type L either.