FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Mon 18 JUNE 2018 - Part Theory

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

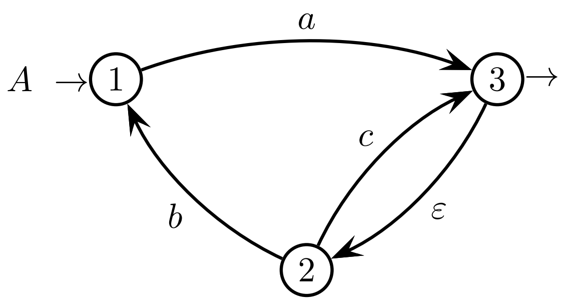
LAST + FIRST NAME:	
(capital letters please)	
MATRICOLA:	SIGNATURE:
(or PERSON CODE)	

INSTRUCTIONS - READ CAREFULLY:

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

1 Regular Expressions and Finite Automata 20%

1. Consider the nondeterministic automaton A over the three-letter alphabet $\{ a,\ b,\ c \}$:



Answer the following questions:

(a) List all the strings of length from 0 to 4 (0 and 4 included) that are accepted by automaton A. Say if automaton A is ambiguous or not and explain why.

(b) Cut the spontaneous transition of automaton A, so that after cutting the result is deterministic. Say if the result is minimal and if necessary minimize it (call it A'). Verify the correctness of A' with the short strings of point (a).

(c) Through node elimination (Brzozowsky), write a regular expression R equivalent to automaton A. Verify the correctness of R with the short strings of point (a).

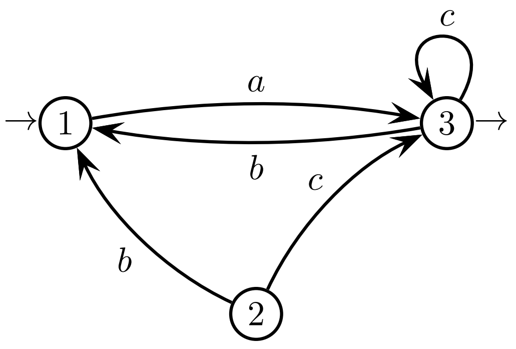
(d) Through the Berry-Sethi method, determinize automaton A (nondeterministic), and obtain a (deterministic) automaton A''. If necessary minimize automaton A'' and verify that it is identical to automaton A'.

(e) (optional) Say if the regular language $L(A)$ is local or not and explain why.

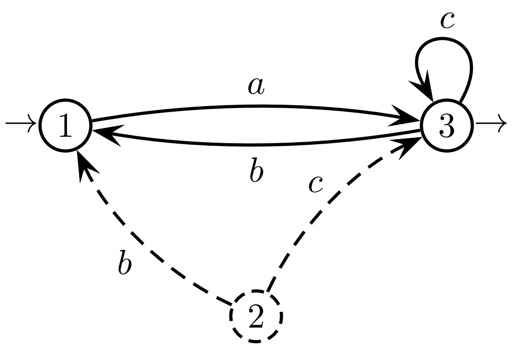
Solution

(a) Here are the short strings: 0: none; 1: a; 2: $a\,c$; 3: $a\,c\,c$ and $a\,b\,a$; 4: $a\,c\,c\,c$, $a\,c\,b\,a$ and $a\,b\,a\,c$. Automaton A is not ambiguous: clearly all the accepting paths of any length from state 1 state 3 are labeled by different strings.

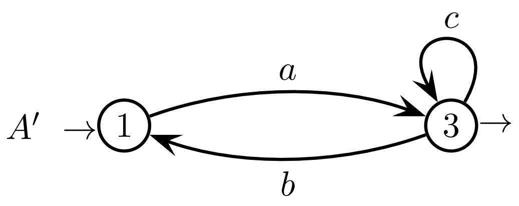
(b) Back-propagating the arcs outgoing from state 2 works:



and cleaning is necessary:

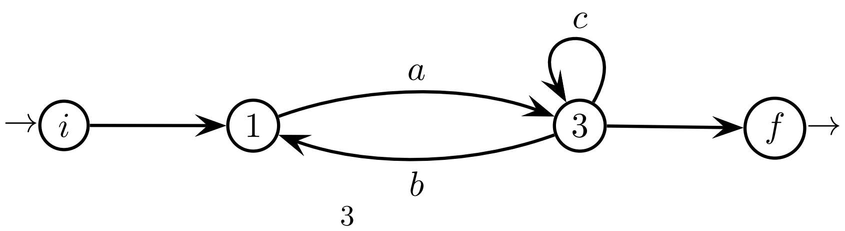


after cleaning, the result is already minimal:



since states 1 and 3 are non-final and final, hence distinguishable. The correctness verification of A' is quite immediate, for instance $1 \xrightarrow{a} 3 \xrightarrow{c} 3 \xrightarrow{b} 1 \xrightarrow{a} 3$ is an accepting computation of A'.

(c) It is simpler to start from the equivalent automaton A', as automaton A' has fewer states than automaton A does. Since the initial and final states have ingoing and outgoing arcs, automaton A' must be normalized:



By orderly eliminating nodes 1 and 3:

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

The correctness verification of R is quite immediate, for instance:

$$a \ (b\, a \ \mid \ c)^* \Rightarrow a \ (b\, a \ \mid \ c) \ (b\, a \ \mid \ c) \Rightarrow a\, c \ (b\, a \ \mid \ c) \Rightarrow a\, c\, b\, a$$

is a derivation of R.
Instead, by orderly eliminating nodes 3 and 1:

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

Of course, the two expressions R and R' are equivalent. In fact, it is well known that $(\alpha \ \mid \ \beta)^* = \alpha^* \ (\beta\, \alpha^*)^*$, thus taking $\alpha = c$ and $\beta = a\, b$ (of course the members of a union can commute), and starting from R:

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

we algebraically transform R into R'. Neither R nor R' are ambiguous, since both are obtained by node elimination starting from a deterministic automaton. Other equivalent forms are as follows (square brackets indicate optionality):

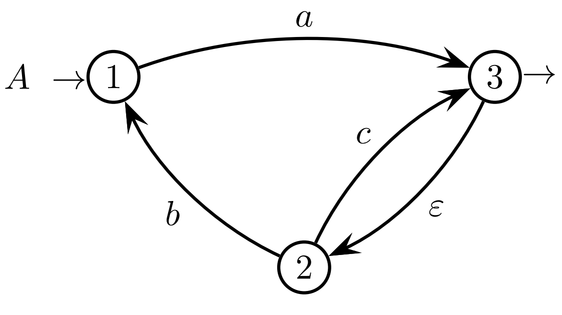
$$(a\, c^*\, b)^* \ a\, c^* \qquad a \ (c^*\, b\, a)^* \ c^*$$

$$a \left(c^* \ (b\, a)^* \ \right)^* \qquad a \left(\ (b\, a)^* \ c^* \right)^*$$

$$a \left(c^* \ [b\, a] \ \right)^* \qquad a \left(\ [c] \ (b\, a)^* \ \right)^*$$

some of which are ambiguous as they are obtained by transformations that may cause ambiguity, and possibly many more.

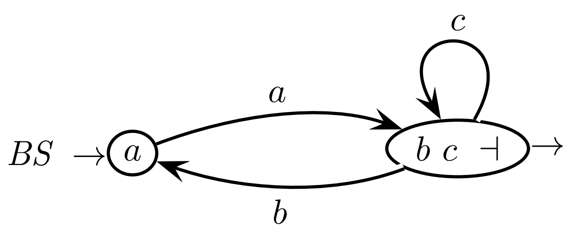
(d) Since each input letter occurs only once in automaton A, it is not necessary to number them. Thus we start from:



The initials and followers are:

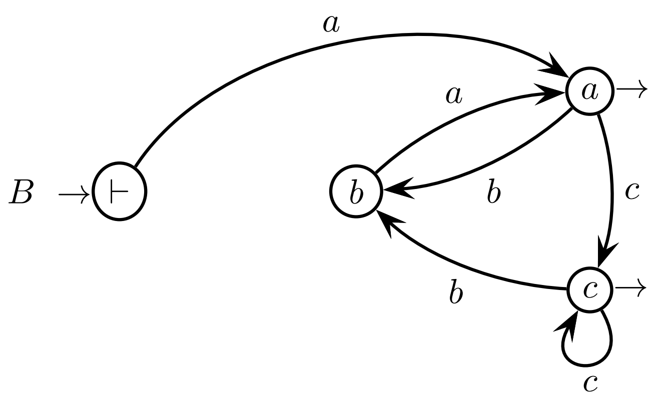
initials	a
terminals	**followers**
a	$b \ c \ \dashv$
b	a
c	$b \ c \ \dashv$

The Berry-Sethi automaton BS is:

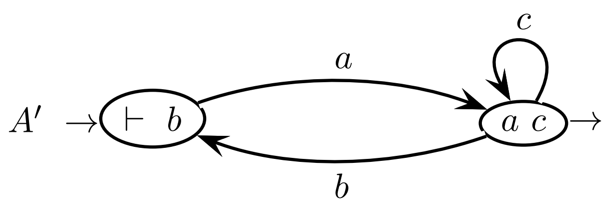


which is identical to automaton A', as expected.

(e) Language $L(A)$ is local. In fact, the deterministic automaton A' (equivalent to A) is the minimal form of a normalized local automaton, which is (call it B):



In this local normalized automaton, the final states a and c are undistinguishable, and the non-final states $\vdash$ and b are also undistinguishable. By merging each pair, automaton A' is obtained:



In conclusion, since language $L(A)$ has a normalized local automaton B, by definition it is local.

2 Free Grammars and Pushdown Automata 20%

1. Consider the context-free language L defined below, over the three-letter alphabet $\Sigma = \{ a, b, c \}$:

$$L = \left\{ y = x\, c\, x^R \mid \quad x \in \{ a, b \}^+ \text{ and } |\, y \,| = 3\, k \text{ for some } k \geq 1 \right\}$$

Examples:

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

Counterexamples:

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

Answer the following questions:

(a) Write the four smallest possible values for the length $|\, y \,|$ of the phrases of language L, and the corresponding values of $|\, x \,|$.

(b) Write a non-ambiguous *BNF* grammar G that generates language L. Solutions with a small number of nonterminal symbols and rules are preferred.

(c) Draw the syntax trees of the following two phrases of language L:

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

Solution

(a) Clearly, the four smallest length values for $|\,y\,|$ are: 3, 9, 15 and 21. The corresponding four values for $|\,x\,|$ are: 1, 4, 7 and 10. Thus reasonably the length of the phrases of language L is an odd multiple of 3. Namely:

k	1	3	5	7	...	odd integer $k \geq 1$
$\|\,y\,\|$	3	9	15	21	...	$3k$, i.e., odd multiple of 3
$\|\,x\,\|$	1	4	7	10	...	$\frac{3k-1}{2}$

More formally, it holds $2|\,x\,| + 1 = |\,y\,| = 3k$, thus $|\,x\,| = \frac{3k-1}{2}$, for $k = 1, 3, 5, 7$, etc, that is, parameter k can only take odd integers as values, as the fraction must be integer. Thus $|\,y\,| = 3k$ is an odd multiple of 3, as found before.

(b) From (a), the length of x starts from one and increases in steps of three letters. As for the rest, language L is a subset of the centred palindrome language. Thus a possible grammar is similar to the standard (non-ambiguous) centred palindrome one $S \rightarrow a\,S\,a \mid b\,S\,b \mid a\,c\,a \mid b\,c\,b$, made recursive in three steps.

Here is a working *BNF* grammar G for language L, non-ambiguous (axiom S):

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

The axiom S generates the shortest palindromes $a\,c\,a$ and $b\,c\,b$, else it enters the X, Y and Z loop, which generates longer and longer palindromes $y = x\,c\,x^R$ by increasing the length of string x in steps of three letters per loop iteration.

A quick glance shows that we can optimize, since the rule right parts of nonterminals S and Z are identical. Hence we can eliminate nonterminal Z and replace it by S. Furthermore, terminal c can be moved to the rule of X.

Thus here is an optimized *BNF* grammar G', still non-ambiguous (axiom S):

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

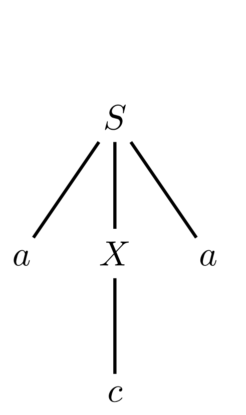
which is equivalent to grammar G.

Of course, one could even remove nonterminals X and Y by substitution, and thus obtain one long list of alternatives, as follows (axiom S):

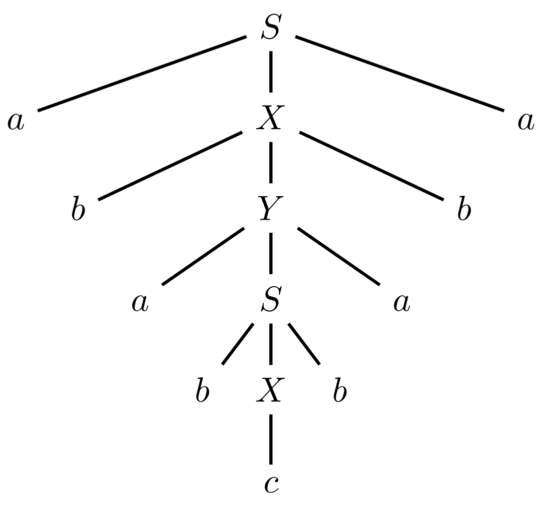
$$S \rightarrow a\,c\,a \mid b\,c\,b \mid \underbrace{a\,a\,a\,S\,a\,a\,a \mid a\,a\,b\,S\,b\,a\,a \mid \ldots \mid b\,b\,b\,S\,b\,b\,b}_{\text{a number of } 2^3 = 8 \text{ alternatives}}$$

It works, though it is not very compact and tends to flatten the tree.

(c) Here are the syntax trees of the sample valid phrases of L, with grammar G':



$|x| = 1$ and $|y| = 3$



$|x| = 4$ and $|y| = 9$

which suffice to reasonably prove the grammar correctness. The same trees can be redrawn by using the other grammar versions proposed (left to the reader).

2. Consider a language of a special kind of arithmetic expression, specified as follows:

   - there are two *binary* operators, addition `+` and multiplication `*`
   - addition and multiplication are *prefix* operators, i.e., `+ a a` and `* a a`
   - there is also a n-ary operator `max (a1, a2, ..., an)`, with $n \geq 2$, which returns the greatest of its arguments `ai` (for $1 \leq i \leq n$)
   - the maximum operator is idempotent, for instance it holds:

     `max (max (a, b), max (c, d)) = max (a, b, c, d)`

     thus for optimization a (sub)expression like the one below:

     $$\mathit{max}\big(\,\mathit{max}\,(\ldots)\,,\,\mathit{max}\,(\ldots)\,,\ldots,\mathit{max}\,(\ldots)\,\big)$$

     is considered *invalid* and *is not* in the language
   - an expression operand is a signed integer decimal number without leading zeroes, or an alphanumerical identifier possibly with dash (dashes may be consecutive), starting with a letter and not ending with a dash (letters are only lowercase)

   For any unspecified minor detail, see the samples below (all of them have a schematic operand `a`). You may also follow the usual conventions, e.g., those of the C language.

   A few valid expressions (think of how they are computed before rushing to answer):

```
+ a a

* a a

+ * a a a

+ a * a a

+ + a max (a, a) a

max (+ * a a a, max (+ a a, a))    DRAW THE SYNTAX TREE OF THIS EXPR
```

   This is a counterexample: `max (max (a, + a a), max (* a a, a))`.

   Write a non-ambiguous grammar, in general of type *EBNF*, that models the language of expressions described above. Verify the correctness of your grammar by drawing the syntax tree of the last valid sample expression.

Solution

It would be tempting to use the destination grammar of the well known infix-to-postfix translation scheme (see the textbook). Yet such a destination grammar separated from the source one is ambiguous, as it has a circular derivation $E \Rightarrow T \Rightarrow F \Rightarrow E$.

A way to obtain the required grammar, is to eliminate the circular derivation from the destination grammar mentioned before. Furthermore, remember that for prefix (and postfix) operators, associativity and precedence are uniquely determined by the operator position, and in fact parentheses are unnecessary in prefix or postfix expressions. Thus the grammar does not need to impose any kind of associativity or precedence. The grammar must be recursive, so that a sub-expression is appended as a subtree to the expression it is contained in. Said differently, the tree must model the bottom-up computation of the expression. Numbers and identifiers are standard.

Here is a preliminary still incomplete grammar, of type *EBNF* and not ambiguous (axiom EXP), which models what said above:

preliminary incomplete version

$$\left\{\begin{array}{rcl}
\langle \mathsf{EXP} \rangle & \rightarrow & (\text{‘}+\text{’} \mid \text{‘}*\text{’})\ \langle \mathsf{EXP} \rangle\ \langle \mathsf{EXP} \rangle \mid \langle \mathsf{MAX} \rangle \mid \langle \mathsf{OPR} \rangle \\
\langle \mathsf{MAX} \rangle & \rightarrow & \mathsf{max}\ \text{‘(’}\ \langle \mathsf{LST} \rangle\ \text{‘)’} \\
\langle \mathsf{LST} \rangle & \rightarrow & \langle \mathsf{EXP} \rangle\ (\text{‘,’}\ \langle \mathsf{EXP} \rangle)^{+} \\
\langle \mathsf{OPR} \rangle & \rightarrow & \langle \mathsf{NUM} \rangle \mid \langle \mathsf{ALP} \rangle \\
\langle \mathsf{NUM} \rangle & \rightarrow & (\text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \varepsilon)\ \big(\, [1 - 9]\ [0 - 9]^{*} \mid 0 \,\big) \\
\langle \mathsf{ALP} \rangle & \rightarrow & [\mathsf{a} - \mathsf{z}]\ \Big(\, (\, \langle \mathsf{AN} \rangle \mid \text{‘\_’})^{*}\ \langle \mathsf{AN} \rangle \mid \varepsilon \Big) \\
\langle \mathsf{AN} \rangle & \rightarrow & [\mathsf{a} - \mathsf{z}] \mid [0 - 9]
\end{array}\right.$$

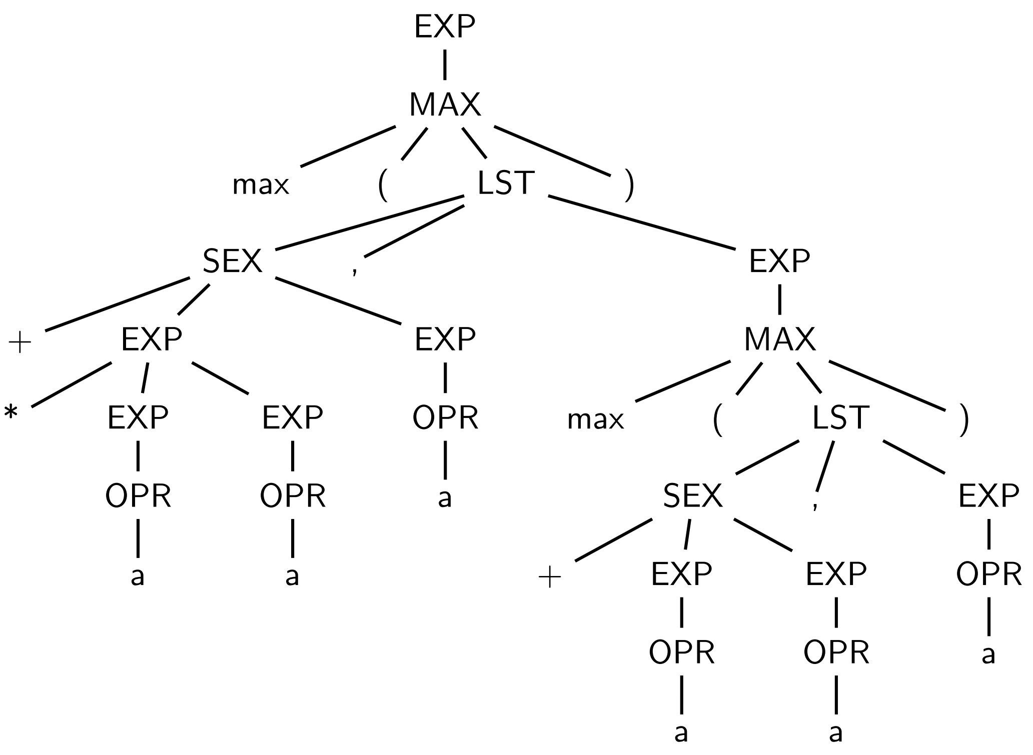
Notice that the rule of nonterminal EXP does not model any specific kind of associativity or precedence between addition and multiplication, which are dealt with at the same level. Anyway, the grammar above does not yet implement the described optimization for the operator max. This can be done by introducing a special kind of "simple" expression SEX that cannot generate an immediate operator max:

complete version

$$\left\{\begin{array}{rcl}
\langle \mathsf{EXP} \rangle & \rightarrow & (\text{‘}+\text{’} \mid \text{‘}*\text{’})\ \langle \mathsf{EXP} \rangle\ \langle \mathsf{EXP} \rangle \mid \langle \mathsf{MAX} \rangle \mid \langle \mathsf{OPR} \rangle \\
\langle \mathsf{MAX} \rangle & \rightarrow & \mathsf{max}\ \text{‘(’}\ \langle \mathsf{LST} \rangle\ \text{‘)’} \\
\langle \mathsf{LST} \rangle & \rightarrow & \langle \mathsf{SEX} \rangle\ (\text{‘,’}\ \langle \mathsf{EXP} \rangle)^{+} \mid \big(\, \langle \mathsf{MAX} \rangle\ \text{‘,’}\big)^{+}\ \langle \mathsf{SEX} \rangle\ (\text{‘,’}\ \langle \mathsf{EXP} \rangle)^{*} \\
\langle \mathsf{SEX} \rangle & \rightarrow & (\text{‘}+\text{’} \mid \text{‘}*\text{’})\ \langle \mathsf{EXP} \rangle\ \langle \mathsf{EXP} \rangle \mid \langle \mathsf{OPR} \rangle \\
\langle \mathsf{OPR} \rangle & \rightarrow & \langle \mathsf{NUM} \rangle \mid \langle \mathsf{ALP} \rangle \\
\langle \mathsf{NUM} \rangle & \rightarrow & (\text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \varepsilon)\ \big(\, [1 - 9]\ [0 - 9]^{*} \mid 0 \,\big) \\
\langle \mathsf{ALP} \rangle & \rightarrow & [\mathsf{a} - \mathsf{z}]\ \Big(\, (\, \langle \mathsf{AN} \rangle \mid \text{‘\_’})^{*}\ \langle \mathsf{AN} \rangle \mid \varepsilon \Big) \\
\langle \mathsf{AN} \rangle & \rightarrow & [\mathsf{a} - \mathsf{z}] \mid [0 - 9]
\end{array}\right.$$

The rule of nonterminal LST grants that at least one operand of an operator max is not an immediate operator max itself. The LST rule is not affected by union ambiguity, since the former member of the union may not start by max while the latter must do so instead, and the rule generates any argument list admitted by the specifications.

Here is the tree of the sample string max (+ * a a a, max (+ a a, a)):



Clearly the tree is correct. It can be used to compute the expression value from bottom to top.

This is a compacted grammar version, derived from the previous one:

$$\text{compacted version I}\quad\left\{\begin{array}{rcl}
\langle \mathsf{EXP} \rangle & \to & \langle \mathsf{SEX} \rangle \mid \langle \mathsf{MAX} \rangle \\
\langle \mathsf{SEX} \rangle & \to & (\text{‘}+\text{’} \mid \text{‘}*\text{’})\ \langle \mathsf{EXP} \rangle^2 \mid \langle \mathsf{OPR} \rangle \\
\langle \mathsf{MAX} \rangle & \to & \mathsf{max}\ \text{‘(’}\ \langle \mathsf{LST} \rangle\ \text{‘)’} \\
\langle \mathsf{LST} \rangle & \to & \langle \mathsf{SEX} \rangle\ (\text{‘,’}\ \langle \mathsf{EXP} \rangle)^+ \mid \big(\langle \mathsf{MAX} \rangle\ \text{‘,’} \big)^+\ \langle \mathsf{SEX} \rangle\ (\text{‘,’}\ \langle \mathsf{EXP} \rangle)^* \\
\langle \mathsf{OPR} \rangle & \to & \langle \mathsf{NUM} \rangle \mid \langle \mathsf{ALP} \rangle \\
\langle \mathsf{NUM} \rangle & \to & (\text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \varepsilon)\ \big([1 — 9]\ [0 — 9]^* \mid 0 \big) \\
\langle \mathsf{ALP} \rangle & \to & [\mathsf{a} — \mathsf{z}]\ \Big(\big(\langle \mathsf{AN} \rangle \mid \text{‘\_’} \big)^* \langle \mathsf{AN} \rangle \mid \varepsilon \Big) \\
\langle \mathsf{AN} \rangle & \to & [\mathsf{a} — \mathsf{z}] \mid [0 — 9]
\end{array}\right.$$

It avoids repeating the same rule right parts, namely of EXP and SEX.

We could compact even more by using the optionality operator for nonterminals NUM and ALP, and the power operator for nonterminal EXP, too:

$$\text{compacted version II}\ \left\{\begin{array}{rcl}
\langle \mathsf{EXP}\rangle & \to & \langle \mathsf{SEX}\rangle \mid \langle \mathsf{MAX}\rangle \\
\langle \mathsf{SEX}\rangle & \to & (\text{‘}+\text{’} \mid \text{‘}*\text{’})\ \langle \mathsf{EXP}\rangle^2 \mid \langle \mathsf{OPR}\rangle \\
\langle \mathsf{MAX}\rangle & \to & \mathsf{max}\,\text{‘(’}\ \langle \mathsf{LST}\rangle\ \text{‘)’} \\
\langle \mathsf{LST}\rangle & \to & \langle \mathsf{SEX}\rangle\ (\text{‘,’}\ \langle \mathsf{EXP}\rangle\ )^+ \mid (\ \langle \mathsf{MAX}\rangle\ \text{‘,’})^+\ \langle \mathsf{SEX}\rangle\ (\text{‘,’}\ \langle \mathsf{EXP}\rangle\ )^* \\
\langle \mathsf{OPR}\rangle & \to & \langle \mathsf{NUM}\rangle \mid \langle \mathsf{ALP}\rangle \\
\langle \mathsf{NUM}\rangle & \to & [\text{‘}+\text{’} \mid \text{‘}-\text{’}]\ \big(\,[1 - 9]\ [0 - 9]^* \mid 0\big) \\
\langle \mathsf{ALP}\rangle & \to & [\mathsf{a} - \mathsf{z}]\ \Big[\big(\ \langle \mathsf{AN}\rangle \mid \text{‘\_’}\big)^*\ \langle \mathsf{AN}\rangle\ \Big] \\
\langle \mathsf{AN}\rangle & \to & [\mathsf{a} - \mathsf{z}] \mid [0 - 9]
\end{array}\right.$$

Do not confuse optionality with interval sets, as both are delimited by square brackets. Last, we could rewrite rule LST and factor $\big(\text{‘,’}\ \langle \mathsf{EXP}\rangle\ \big)^*$ to the right in this way:

$$\langle \mathsf{LST}\rangle \to \Big(\ \langle \mathsf{SEX}\rangle\ \text{‘,’}\ \langle \mathsf{EXP}\rangle \mid \big(\ \langle \mathsf{MAX}\rangle\ \text{‘,’}\big)^+\ \langle \mathsf{SEX}\rangle\ \Big)\ \big(\text{‘,’}\ \langle \mathsf{EXP}\rangle\ \big)^*$$

though it does not seem to be more advantageous than the previous form. It may however help us see more clearly that this rule is not affected by union ambiguity (the rest of the grammar is very simple and clearly it is not ambiguous).

There might be more efficient optimizations for the operator max, not specified and not required in the present exercise. For instance, observe that max (a, max (a, a)) = max (a, a, a), and by generalizing the observation, we could assume the stronger specification that an operator max must not have any immediate argument max. This would lead to the following grammar (not equivalent to the previous ones):

$$\text{restricted version I}\ \left\{\begin{array}{rcl}
\langle \mathsf{EXP}\rangle & \to & \langle \mathsf{SEX}\rangle \mid \langle \mathsf{MAX}\rangle \\
\langle \mathsf{SEX}\rangle & \to & (\text{‘}+\text{’} \mid \text{‘}*\text{’})\ \langle \mathsf{EXP}\rangle^2 \mid \langle \mathsf{OPR}\rangle \\
\langle \mathsf{MAX}\rangle & \to & \mathsf{max}\,\text{‘(’}\ \langle \mathsf{LST}\rangle\ \text{‘)’} \\
\langle \mathsf{LST}\rangle & \to & \langle \mathsf{SEX}\rangle\ (\text{‘,’}\ \langle \mathsf{SEX}\rangle\ )^+ \\
\langle \mathsf{OPR}\rangle & \to & \langle \mathsf{NUM}\rangle \mid \langle \mathsf{ALP}\rangle \\
\langle \mathsf{NUM}\rangle & \to & [\text{‘}+\text{’} \mid \text{‘}-\text{’}]\ \big(\,[1 - 9]\ [0 - 9]^* \mid 0\big) \\
\langle \mathsf{ALP}\rangle & \to & [\mathsf{a} - \mathsf{z}]\ \Big[\big(\ \langle \mathsf{AN}\rangle \mid \text{‘\_’}\big)^*\ \langle \mathsf{AN}\rangle\ \Big] \\
\langle \mathsf{AN}\rangle & \to & [\mathsf{a} - \mathsf{z}] \mid [0 - 9]
\end{array}\right.$$

The grammar above grants that none of the arguments of the operator max may be an immediate operator max, whereas the original grammar just grants that at least one such argument is not an immediate max. Yet in this way we have changed the language, which now is a restriction, i.e., a subset, of the one specified in the exercise.

With a little more effort, by removing nonterminal LST, which does not have an autonomous arithmetic interpretation as operation or atomic object (it is just a list of arguments only intended for the operator max), as well as by eliminating the copy rules of nonterminal OPR and the copy rule $\langle\mathsf{EXP}\rangle \to \langle\mathsf{MAX}\rangle$ (furthermore this allows us to remove nonterminal MAX), we could even compact as follows:

$$\text{restricted version II}\;\left\{\begin{array}{rcl}
\langle\mathsf{EXP}\rangle & \to & \langle\mathsf{SEX}\rangle \mid \mathsf{max}\,\text{‘(’}\,\langle\mathsf{SEX}\rangle\,\big(\text{‘,’}\,\langle\mathsf{SEX}\rangle\,\big)^{+}\,\text{‘)’} \\
\langle\mathsf{SEX}\rangle & \to & (\text{‘+’} \mid \text{‘*’})\,\langle\mathsf{EXP}\rangle^{2} \mid \langle\mathsf{NUM}\rangle \mid \langle\mathsf{ALP}\rangle \\
\langle\mathsf{NUM}\rangle & \to & [\text{‘+’} \mid \text{‘−’}]\,\big(\,[\mathsf{1} - \mathsf{9}]\,[\mathsf{0} - \mathsf{9}]^{*} \mid \mathsf{0}\big) \\
\langle\mathsf{ALP}\rangle & \to & [\mathsf{a} - \mathsf{z}]\,\Big[\big(\,\langle\mathsf{AN}\rangle \mid \text{‘\_’}\big)^{*}\,\langle\mathsf{AN}\rangle\,\Big] \\
\langle\mathsf{AN}\rangle & \to & [\mathsf{a} - \mathsf{z}] \mid [\mathsf{0} - \mathsf{9}]
\end{array}\right.$$

still for the restricted language, anyway. This form looks very simple and clear. Only one copy rule, i.e., $\langle\mathsf{EXP}\rangle \to \langle\mathsf{SEX}\rangle$, is left, yet eliminating it, though possible, would force us to replicate its right part into three different positions of the rule of nonterminal EXP, thus causing a considerable redundancy. This is an example of how sometimes categorization may be useful and help us make a grammar more elegant.

3 Syntax Analysis and Parsing Methodologies 20%

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

$$\begin{tikzcd}
B \arrow[r] & 0_B \arrow[r, "b"] & 1_B \arrow[r, "c"] & 2_B \arrow[r] & {}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & 1_S \arrow[r, "B"] & 2_S \arrow[dr, bend left, "a"] & & & \\
S \arrow[r] & 0_S \arrow[ur, bend left, "a"] \arrow[dr, bend right, "A"'] \arrow[r] & {} & & 5_S \arrow[r, "S"] & 6_S \arrow[r] & {} \\
 & & 3_S \arrow[r, "c"'] & 4_S \arrow[ur, bend right, "b"'] & & &
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r] & 0_A \arrow[r, "a"'] & 1_A \arrow[r, "b"'] & 2_A \arrow[r] & {}
\end{tikzcd}$$

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

(a) Draw the complete pilot of grammar G and prove that grammar G is not of type *ELR* (1). Explain your answer.

(b) Modify grammar G, as little as possible (keep the same nonterminal alphabet), so as to obtain an equivalent grammar (generating the same language) that is of type *ELR* (1). Motivate your choice.

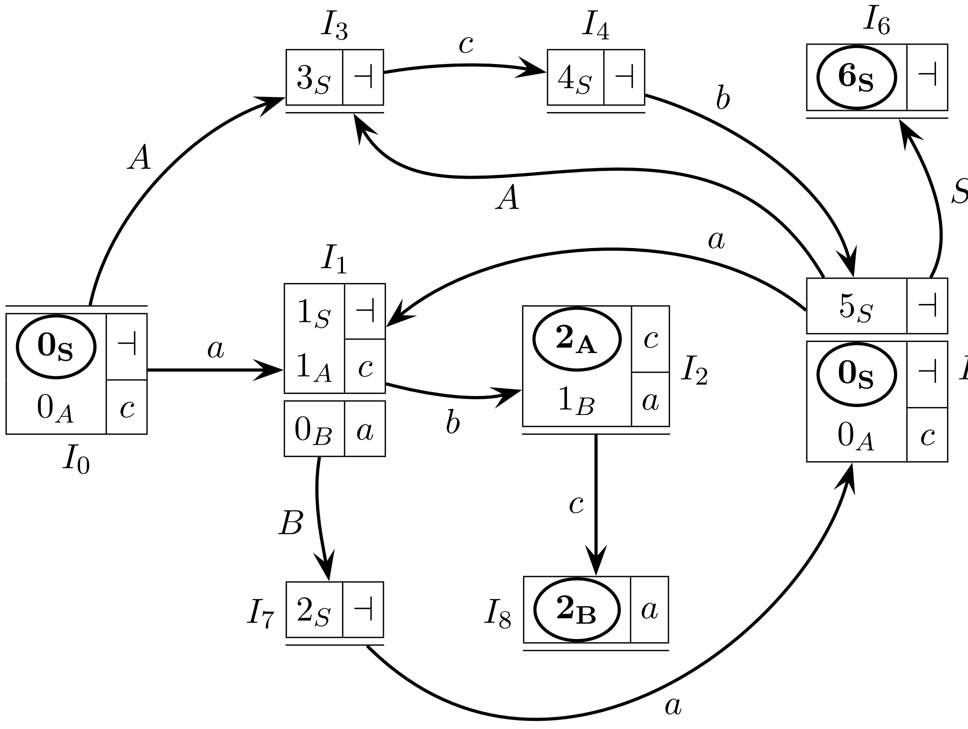
(c) Write all the guide sets on the shift arcs, call arcs and exit arrows of the machine net of grammar G, and show that grammar G is not of type *ELL* (1). Explain your answer.

(d) Determine whether grammar G is of type *ELL* (k), for some $k > 1$. Explain your answer.

(e) (optional) Which is the smallest language family, among those studied in the present course, that includes language $L(G)$? Motivate your answer.

Solution

(a) Here is the complete pilot of grammar G, with nine m-states (I_0 is initial):



The pilot has a shift-reduce conflict in the m-state I_2 on terminal c: state 1_B shifts on input c and the final state 2_A reduces on look-ahead c. This suffices to conclude that grammar G is not of type *ELR* (1).

For clarity, we complete the exam of the pilot. There are not any other conflicts of type shift-reduce or reduce-reduce. The pilot does not have the *STP*, since there are multiple transitions, for instance $I_0 \xrightarrow{a} I_1$ is a double transition, or equivalently the base of m-state I_1 contains two items; the other multiple transitions are $I_5 \xrightarrow{a} I_1$ and $I_1 \xrightarrow{b} I_2$ (both are double). Anyway, none of the multiple transitions is convergent. This excludes *a priori* the existence of convergence conflicts, too. Thus the conflict already mentioned is the only one in this pilot, and the only reason why the grammar fails to be *ELR* (1).

(b) Here is a new machine net of grammar G:

$$\begin{tikzcd}
B \arrow[r] & 0_B \arrow[r, "a"'] & 1_B \arrow[r, "b"'] & 2_B \arrow[r, "c"'] & 3_B \arrow[r] & {}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & 1_S \arrow[dr, bend left, "b"] & & & \\
S \arrow[r] & 0_S \arrow[ur, bend left, "B"] \arrow[dr, bend right, "A"'] \arrow[r] & {} & 3_S \arrow[r, "S"] & 4_S \arrow[r] & {} \\
 & & 2_S \arrow[ur, bend right, "a"'] & & &
\end{tikzcd}$$

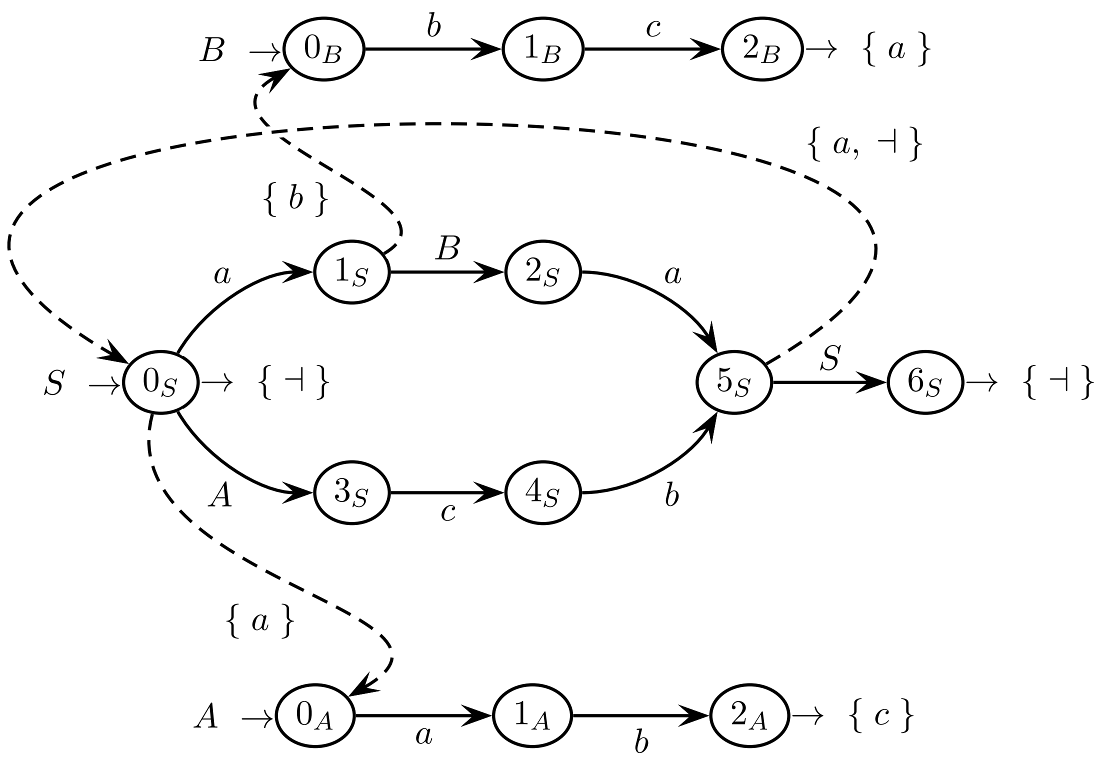
$$\begin{tikzcd}
A \arrow[r] & 0_A \arrow[r, "a"] & 1_A \arrow[r, "b"] & 2_A \arrow[r, "c"] & 3_A \arrow[r] & {}
\end{tikzcd}$$

The new machine net is obtained by postfixing the transition $3_S \xrightarrow{c} 4_S$ of G to the original machine A, by prefixing the transition $0_S \xrightarrow{a} 1_S$ of G to the original machine B, and by removing such transitions from the original machine S. Some states are removed and the state names are changed accordingly.

It is immediately evident that, since now nonterminals A and B are identical, the analyzer examines them in parallel and reduces either one without any conflict, as their look-ahead terminals are different, i.e., b and a respectively. This answers the question quickly though rigorously, and does not need to draw a new pilot.

Of course, other conflict-free net forms are possible. For instance, we might think of removing only transition $3_S \xrightarrow{c} 4_S$ from the machine of nonterminal S, and of postfixing it to that of nonterminal A with one more state. Intuitively, this also eliminates the conflict, since both nonterminals A and B would reduce in the same m-state, i.e., I_8, but with a different look-ahead, respectively a and b. The reader may wish to draw the pilot fragment that shows this behaviour.

(c) Here are the call arcs and all the guide sets of grammar G:



The guide sets on state 0_S overlap (terminal a is shared). Thus the machine net of grammar G is not of type *ELL* (1). Of course, this was expected as the net is not of type *ELR* (1) either.

Notice that the terminator $\dashv$ is included in the guide set of the call arc from state 5_S. The reason is that nonterminal S is nullable and terminator $\dashv$ occurs on the guide set of the exit arrow from state 6_S, after the shift arc associated to the call arc. The reason *is not* that the call arc is immediately followed (in 0_S) by the exit arc with $\dashv$ (an exit arrow is not a call to another machine).

(d) It is easy to see that after three shift moves on input characters a, b and c, the machine net will be either in state 2_S or in state 4_S. From these two states, the expected input character is a or b, respectively. Thus with a look-ahead distance $k = 4$, the analyzer can predict which path is the correct one to take. In conclusion, the machine net of grammar G is of type *ELL* (4).

This is not in contradiction with the machine net not being of type *ELR* (1), since the look-ahead distance is different. Of course, by increasing the look-ahead distance for the pilot, the net would turn out to be of type *ELR*, too.

(e) Machines A and B are acyclic and do not call other machines, hence their languages are finite (both consist of one string). Machine S is essentially right-recursive. By substituting the (finite) languages of A and B into machine S, and by applying the Arden identity, it soon turns out that $L(G) = (a\,b\,c\ (a \mid b))^*$, hence the language of grammar G is regular. Since for sure it is not finite, the smallest known language family that contains it is the regular one.

4 Language Translation and Semantic Analysis 20%

1. Consider a source language L of non-empty nested lists of arbitrary depth. A list element may be an atom or a sublist. An atom is schematized by a terminal "e". A sublist is enclosed between round brackets "( )", but the main list is not. Each element (atom or sublist), at any depth, has a prefix that consists of the terminal string "a" separated from the element by a colon ":", that is, "a : e" or "a : (...)". Thus the alphabet of L is { a, ':', e, '(', ')' }.

   Here is a sample two-level list, with a main list of two elements and a sublist of one:

$$\overbrace{\underbrace{\mathtt{a:(a:e)}}_{\text{sublist}}\ \mathtt{a:e}}^{\text{main list}}$$

   A *BNF* grammar G^s that generates the source language L is as follows (axiom S):

$$G^s \begin{cases} S \rightarrow S\ P \\ S \rightarrow P \\ P \rightarrow a\ :\ E \\ E \rightarrow (\ S\ ) \\ E \rightarrow e \end{cases}$$

   Please answer the following questions and keep in mind that you *may not change* the source grammar G^s:

   (a) Write a *BNF* translation scheme (or grammar) G_1 that defines a translation function τ_1 that, for each element, turns the prefix into suffix. Example:

   $$\tau_1\ (\mathtt{a:(a:e)\ \ a:e}) = \mathtt{(e:a):a\ \ e:a}$$

   Verify the correctness of scheme (or grammar) G_1 by drawing the translation tree of the example string.

   (b) Write a *BNF* translation scheme (or grammar) G_2 that defines a translation function τ_2 that, for each (sub)list, shifts the prefix of each (sub)list element and forms a chain of as many prefixes for the entire (sub)list. Example:

   $$\tau_2\ (\mathtt{a:e\ \ a:(a:e\ \ a:e)\ \ a:e}) = \mathtt{aaa:e\ \ (aa:ee)\ \ e}$$

   Verify the correctness of scheme (or grammar) G_2 by drawing the translation tree of the example string.

   (c) (optional) Say if the schemes (or grammars) G_1 and G_2 found before are deterministic of type $LL(k)$ (or $ELL(k)$), for some $k \geq 1$, and explain why.

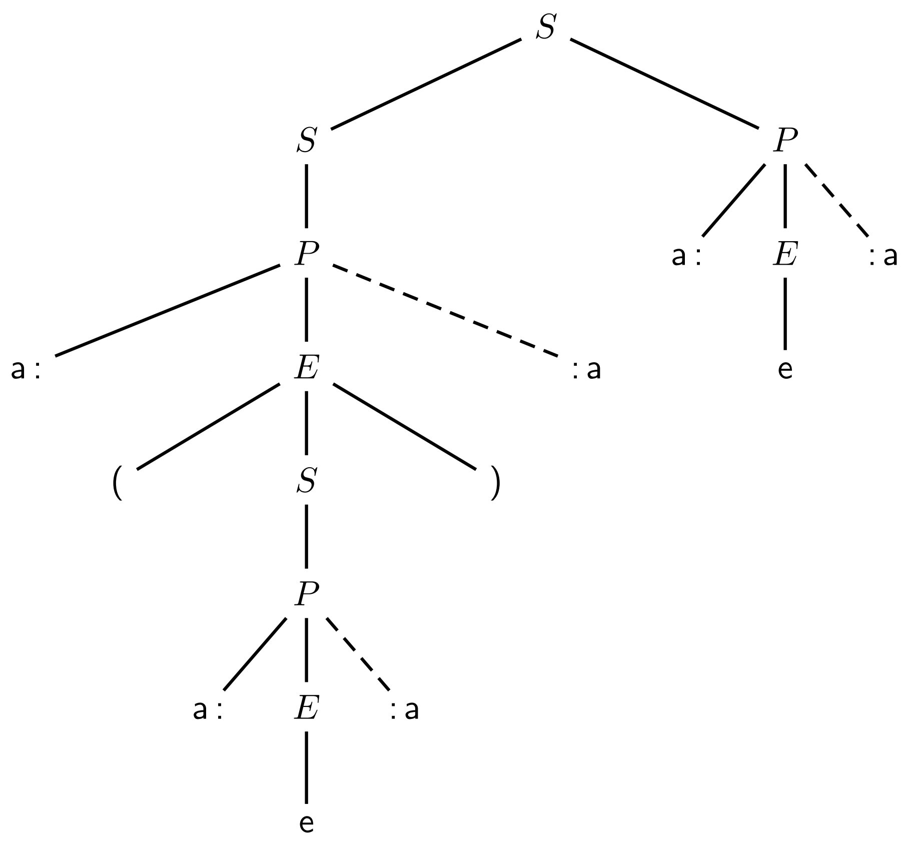
Solution

(a) Here is the destination grammar G_1^d:

$$G^s \begin{cases} S \rightarrow S\ P \\ S \rightarrow P \\ P \rightarrow a\ :\ E \\ E \rightarrow (\ S\ ) \\ E \rightarrow e \end{cases} \qquad G_1^d \begin{cases} S \rightarrow S\ P \\ S \rightarrow P \\ P \rightarrow E\ :\ a \\ E \rightarrow (\ S\ ) \\ E \rightarrow e \end{cases}$$

This is a somewhat standard prefix-to-postfix transformation, and it deserves no special comment.

Translation tree of "`a:(a:e) a:e`" into "`(e:a):a e:a`":



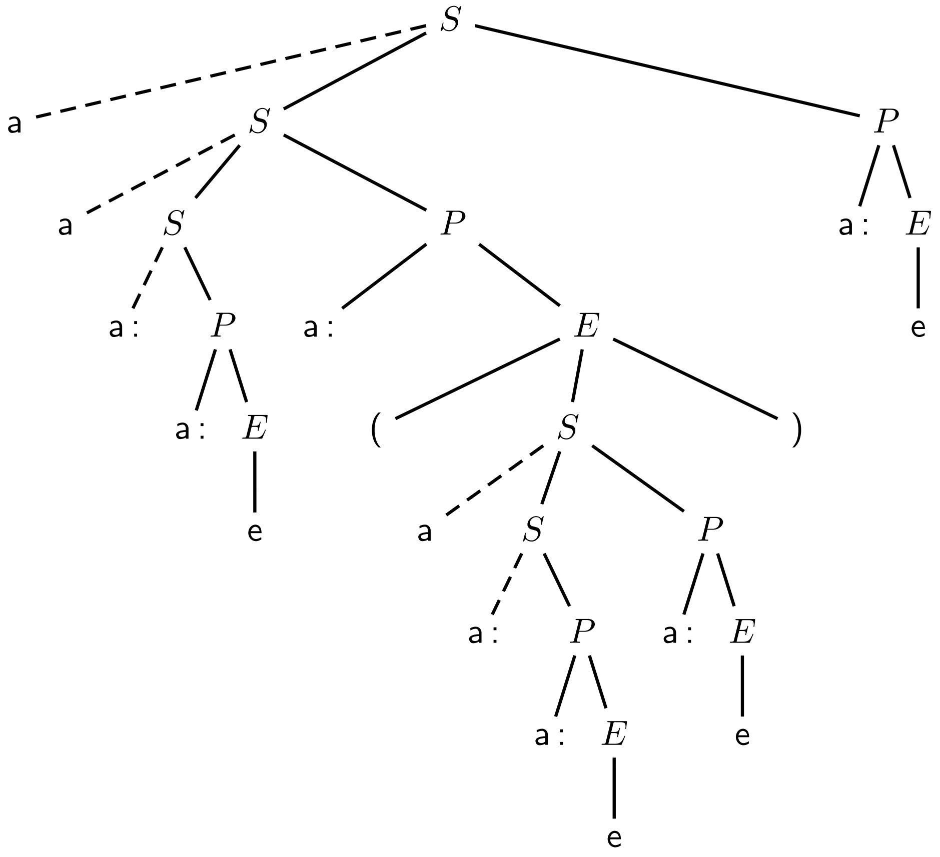
The edges of the source tree are solid. The tree edges that generate the translated prefixes are dashed (the other destination edges coincide with the source ones). The tree is clearly correct.

(b) Here is the destination grammar G_2^d:

$$G^s \begin{cases} S \rightarrow S\, P \\ S \rightarrow P \\ P \rightarrow a\ :\ E \\ E \rightarrow (\ S\ ) \\ E \rightarrow e \end{cases} \qquad\qquad G_2^d \begin{cases} S \rightarrow a\, S\, P \\ S \rightarrow a\ :\ P \\ P \rightarrow E \\ E \rightarrow (\ S\ ) \\ E \rightarrow e \end{cases}$$

The prefixes are moved from each element to the left-recursive rule that generates the element list, and to the rule ends the list (along with the separator). In this way, a list of prefixes is generated together with the element list, and the two lists are divided by the separator.

Translation tree of "`a:e a:(a:e a:e) a:e`" into "`aaa:e (aa:ee) e`":



The edges of the source tree are solid. The tree edges that generate the translated prefixes are dashed (the other destination edges coincide with the source ones). The tree is clearly correct.

(c) The source grammar G^s is left-recursive (see the axiomatic rule $S \rightarrow S\, P$), thus both schemes G_1 and G_2 are not *LL* or *ELL* for any $k \geq 1$.

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

$$\begin{cases} 1: & S \rightarrow B \\ 2: & B \rightarrow B\,B \\ 3: & B \rightarrow U\,B \\ 4: & B \rightarrow U\,U \end{cases} \qquad \begin{cases} 5: & U \rightarrow U \\ 6: & U \rightarrow B \\ 7: & U \rightarrow f \end{cases}$$

A sample tree is shown on the next page.

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

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

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

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

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

(a) Decorate the sample tree (next page) with the values of attributes *dp* and *ndp*.

(b) Write an attribute grammar, based on the syntax above and using only the permitted attributes (do not change the attribute types), that computes the required value of attribute *ndp* in the root node S of the tree. The attribute grammar must be of type one-sweep. Proceed stepwise and separately write:

   i. the semantic rules for computing attribute d into table 1

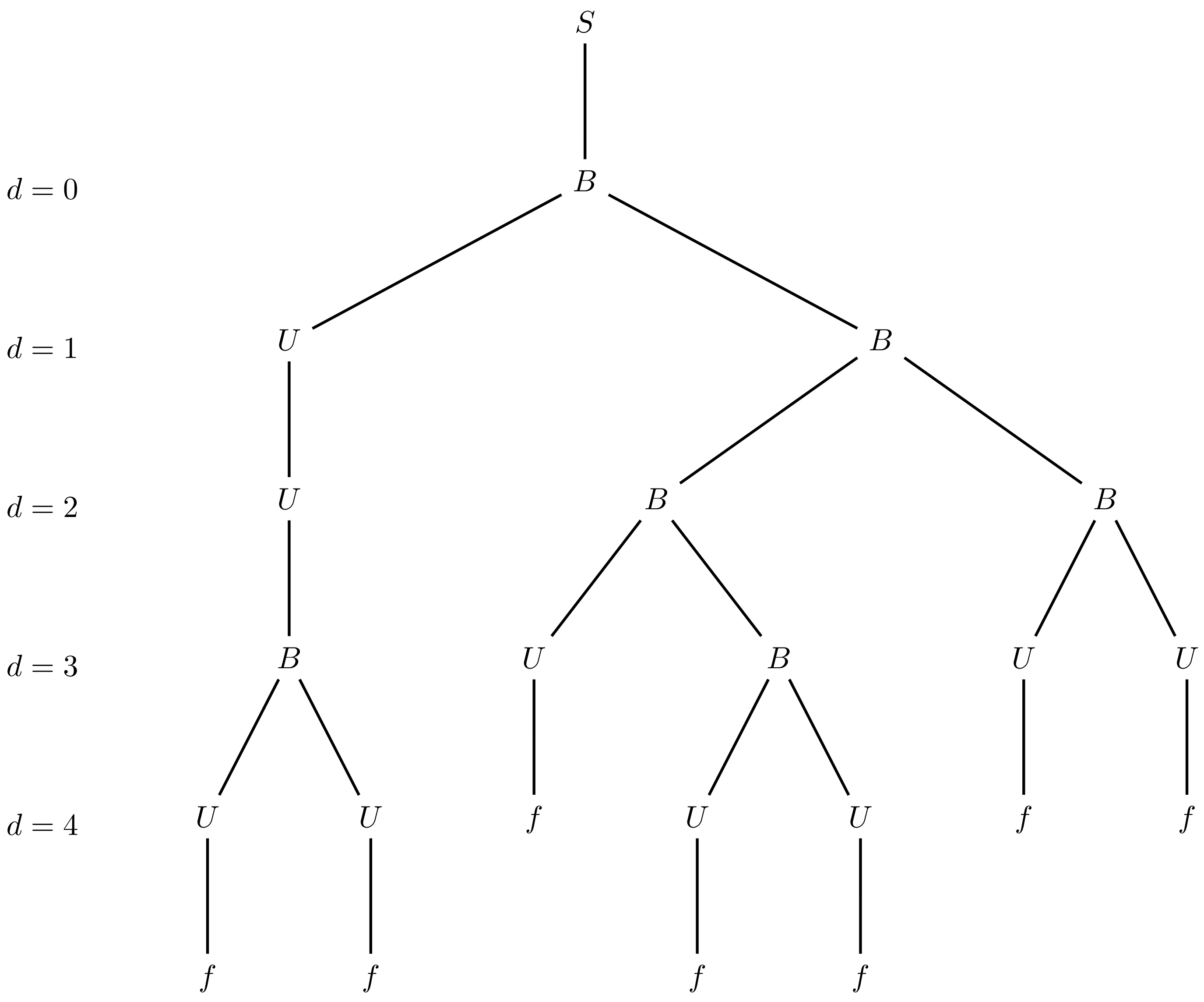
   ii. those for attribute *dp* into table 2

   iii. and those for attribute *ndp* into table 3

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

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

sample tree to decorate – question (a)



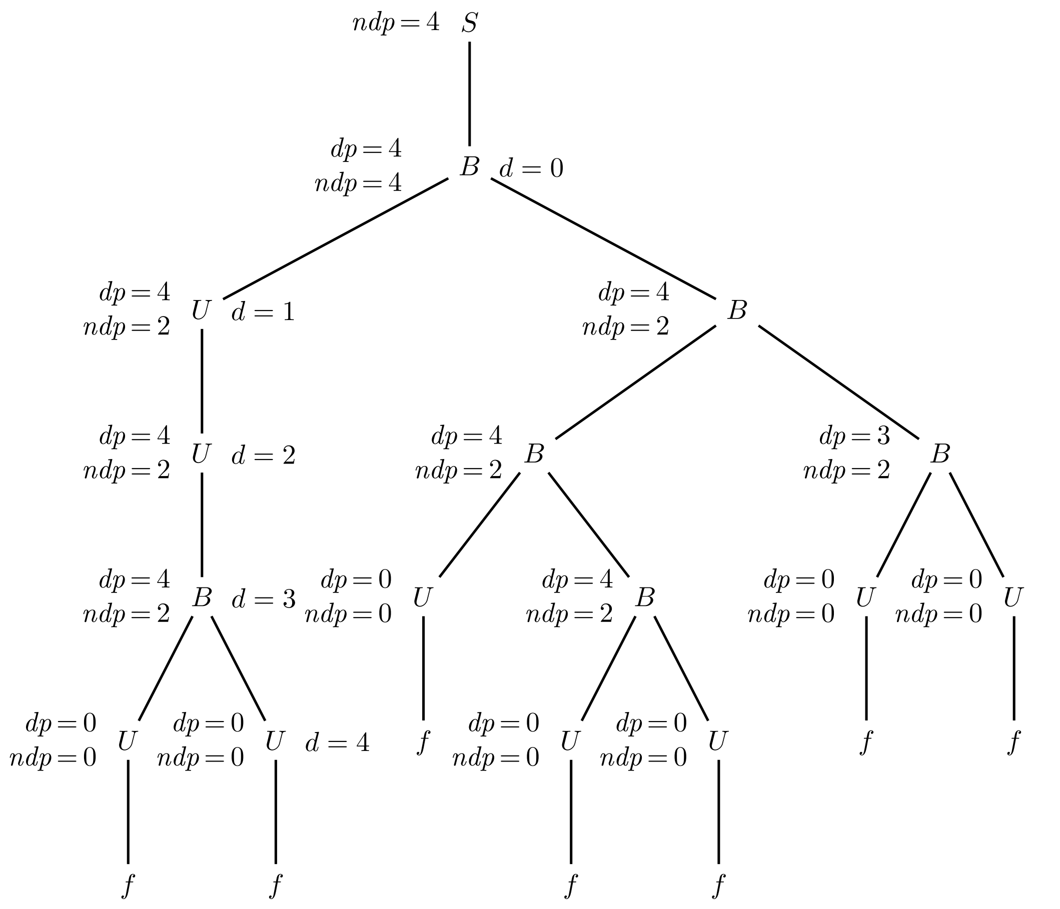
#	*syntax*	*semantics* – question (b) point (i) – TAB 1
1:	$S_0 \rightarrow B_1$	
2:	$B_0 \rightarrow B_1\ B_2$	
3:	$B_0 \rightarrow U_1\ B_2$	
4:	$B_0 \rightarrow U_1\ U_2$	
5:	$U_0 \rightarrow U_1$	
6:	$U_0 \rightarrow B_1$	
7:	$U_0 \rightarrow f$	

#	*syntax*	*semantics* – question (b) point (*ii*) – TAB 2
1:	$S_0 \rightarrow B_1$	
2:	$B_0 \rightarrow B_1\ B_2$	
3:	$B_0 \rightarrow U_1\ B_2$	
4:	$B_0 \rightarrow U_1\ U_2$	
5:	$U_0 \rightarrow U_1$	
6:	$U_0 \rightarrow B_1$	
7:	$U_0 \rightarrow f$	

#	*syntax*	*semantics* – question (b) point (*iii*) – TAB 3
1:	$S_0 \rightarrow B_1$	
2:	$B_0 \rightarrow B_1\ B_2$	
3:	$B_0 \rightarrow U_1\ B_2$	
4:	$B_0 \rightarrow U_1\ U_2$	
5:	$U_0 \rightarrow U_1$	
6:	$U_0 \rightarrow B_1$	
7:	$U_0 \rightarrow f$	

Solution

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



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

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

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

#	*syntax*	*semantics*
1:	$S_0 \rightarrow B_1$	$d_1 := 0$ $ndp_0 := ndp_1$
2:	$B_0 \rightarrow B_1\ B_2$	$d_1,\ d_2 := d_0 + 1$ $dp_0 := max(\ dp_1, dp_2\ )$ $ndp_0 :=$ **if** $(\ dp_1 == dp_2\ )$ **then** $ndp_1 + ndp_2$ **else if** $(\ dp_1 > dp_2\ )$ **then** ndp_1 **else** ndp_2 **endif**

3:	$B_0 \rightarrow U_1\ B_2$	$d_1,\ d_2 := d_0 + 1$
		$dp_0 :=$ **if** $(\ dp_1 == 0\ )$ **then** dp_2 (not d_2 here)
		else $max(\ dp_1, dp_2\ )$ **endif**
		$ndp_0 :=$ **if** $(\ dp_1 == 0\ )$ **then** ndp_2
		else if $(\ dp_1 == dp_2\ )$ **then** $ndp_1 + ndp_2$
		else if $(\ dp_1 > dp_2\ )$ **then** ndp_1
		else ndp_2 **endif**

4:	$B_0 \rightarrow U_1\ U_2$	$d_1,\ d_2 := d_0 + 1$
		$dp_0 :=$ **if** $(\ dp_1 == 0$ and $dp_2 == 0\ )$ **then** d_1 (or d_2)
		else $max(\ dp_1, dp_2\ )$ **endif**
		$ndp_0 :=$ **if** $(\ dp_1 == 0$ and $dp_2 == 0\ )$ **then** 2
		else if $(\ dp_1 == dp_2\ )$ **then** $ndp_1 + ndp_2$
		else if $(\ dp_1 > dp_2\ )$ **then** ndp_1
		else ndp_2 **endif**

5:	$U_0 \rightarrow U_1$	$d_1 := d_0 + 1$ $dp_0 :=$ **if** $(\ dp_1 == 0\ )$ **then** d_1 **else** dp_1 **endif** $ndp_0 :=$ **if** $(dp_1 == 0)$ **then** 1 **else** ndp_1 **endif**
6:	$U_0 \rightarrow B_1$	$d_1 := d_0 + 1$ $dp_0 := dp_1$ $ndp_0 := ndp_1$
7:	$U_0 \rightarrow f$	$dp_0 := 0$ $ndp_0 := 0$

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

Separately and written as a program:

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

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

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

#	*syntax*	*semantics* – question (b) point (*iii*) – TAB 3
1:	$S_0 \rightarrow B_1$	$ndp_0 := ndp_1$

2:	$B_0 \rightarrow B_1\ B_2$	**if** ($dp_1 == dp_2$) **then**
		   $ndp_0 := ndp_1 + ndp_2$
		else if ($dp_1 > dp_2$) **then**
		   $ndp_0 := ndp_1$
		else
		   $ndp_0 := ndp_2$
		endif

3:	$B_0 \rightarrow U_1\ B_2$	**if** ($dp_1 == 0$) **then**
		   $ndp_0 := ndp_2$
		else if ($dp_1 == dp_2$) **then**
		   $ndp_0 := ndp_1 + ndp_2$
		else if ($dp_1 > dp_2$) **then**
		   $ndp_0 := ndp_1$
		else
		   $ndp_0 := ndp_2$
		endif

4:	$B_0 \rightarrow U_1\ U_2$	**if** ($dp_1 == 0$ and $dp_2 == 0$) **then**
		   $ndp_0 := 2$
		else if ($dp_1 == dp_2$) **then**
		   $ndp_0 := ndp_1 + ndp_2$
		else if ($dp_1 > dp_2$) **then**
		   $ndp_0 := ndp_1$
		else
		   $ndp_0 := ndp_2$
		endif

5:	$U_0 \rightarrow U_1$	**if** ($dp_1 == 0$) **then**
		   $ndp_0 := 1$
		else
		   $ndp_0 := ndp_1$
		endif

6:	$U_0 \rightarrow B_1$	$ndp_0 := ndp_1$
7:	$U_0 \rightarrow f$	$ndp_0 := 0$

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

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

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