

FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Thu 25 February 2016 - Part Theory

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

NAME:

MATRICOLA:

SIGNATURE:

INSTRUCTIONS - READ CAREFULLY:

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

1 Regular Expressions and Finite Automata 20%

1. Consider the following extended regular expression R , i.e., a regexp with non-standard operators, over the three-letter alphabet $\Sigma = \{ a, b, c \}$:

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

where the square brackets “[]” indicate optionality.

Answer the following questions:

- (a) By using the Berry-Sethi method (*BS*), find a deterministic finite state automaton A equivalent to expression R .
 - (b) Determine if automaton A is minimal or not, and if necessary minimize it.
 - (c) Say if expression R is ambiguous or not, and justify your answer.
 - (d) Say if language $L(R)$ is local or not, and justify your answer.
 - (e) (optional) In the case expression R is ambiguous, find an equivalent regular expression R' that is not ambiguous, and provide a brief and informal but convincing explanation of the non-ambiguity of R' .
-

Solution

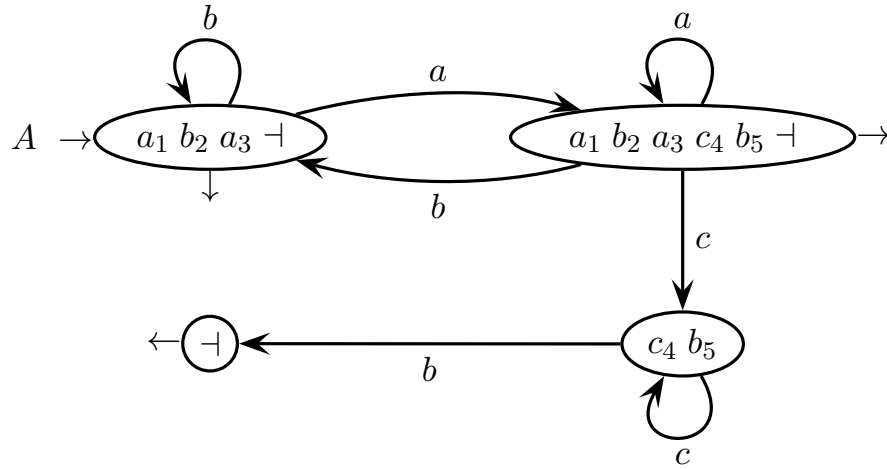
- (a) The *BS* automaton A is not difficult to obtain. Here is the numbered regular expression $R_{\#}$:

$$R_{\#} = (a_1 \mid b_2)^* [a_3 c_4^* b_5] \dashv$$

Here are the initial and follower sets of the numbered regular expression $R_{\#}$:

initials	$a_1 \ b_2 \ a_3 \ \dashv$
terminals	followers
a_1	$a_1 \ b_2 \ a_3 \ \dashv$
b_2	$a_1 \ b_2 \ a_3 \ \dashv$
a_3	$c_4 \ b_5$
c_4	$c_4 \ b_5$
b_5	\dashv

And here is the *BS* (deterministic) finite state automaton A obtained from regular expression R :



with four states, three of which are final.

- (b) By construction, the *BS* automaton A is in the clean form: all four states are useful, i.e., both reachable (from the initial state) and defined (reaching a final state). It is easy to verify that automaton A is minimal. In fact the three final states do not have the same outgoing labels, and in particular final state \dashv does not have any outgoing arc at all. Thus the four states of A are distinguishable from one another, and therefore the automaton is minimal.
- (c) The regular expression R is ambiguous, for instance due to string ab . In fact, reconsidering the numbered regular expression $R_{\#}$ used for the *BS* construction, the valid string $ab \in L(R)$ can be obtained in two different ways: $a_1 b_2$ or $a_3 b_5$. It is not difficult to get convinced that the strings of type $(a \mid b)^* ab$ are the only ambiguous ones. In fact, the subexpression $(a \mid b)^*$ of R is not ambiguous,

as actually it unambiguously generates the universal language over subalphabet $\{a, b\}$, and it can generate strings $(a \mid b)^* ab$. Instead, the optional subexpression $a c^* b$ generates only one string without any letter c , namely string ab , else it unambiguously produces strings that contain at least one letter c . Thus strings $(a \mid b)^* ab$ are the only ones that can be generated each in two ways.

- (d) Language $L(R)$ is not local. For instance, by examining the local sets of regular expression R , one sees that the three letter pairs ac , cb and ba are admissible, as well as the initial letter a and the final one b , hence the strings of type $acbacb \dots$ are unavoidable. Such substrings anyway are not generated by R : after reading a letter c , automaton A cannot read letter a any more.

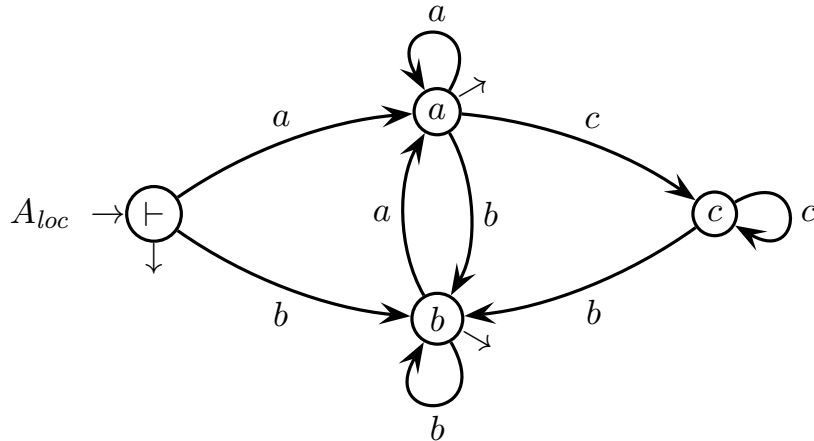
Incidentally, the BS automaton A obtained before for regular expression R is not local, as two states are entered by the same input character, which fact agrees with the rigorous conclusion obtained by analyzing R . Remember however that this fact alone does not prove that language $L(A)$ is not local.

A different approach to determine if language $L(R)$ is local or not, consists of examining certain minimal automata. Expression R has the following local sets:

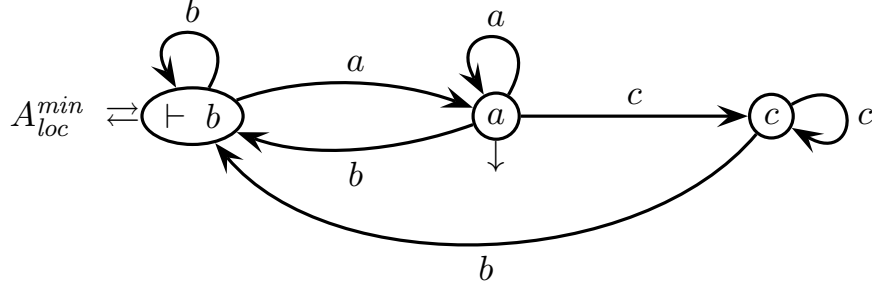
$$\begin{aligned} Ini &= \{a, b\} \\ Dig &= \{aa, ab, ba, bb, ac, cc, cb\} \\ Fin &= \{a, b\} \end{aligned}$$

which can be computed either intuitively, or by examining the initials and followers of the numbered expression $R_\#$ at point (b) and canceling all letter subscripts, or by formally computing the recursive relations explained in the textbook.

These local sets determine the following (deterministic) local automaton A_{loc} :



Since it holds $\varepsilon \in L(R)$, the initial state \vdash is also final. Automaton A_{loc} is not minimal: it is in the clean form as all its states are useful; final states \vdash and b are undistinguishable as they have the same outgoing labels and destination states, while final state a is distinguishable from them as it has the outgoing transition $a \xrightarrow{c} c$; and state c is distinguishable from all the others as it is not final. Thus states \vdash and b can be merged. Here is the minimal form A_{loc}^{min} of A_{loc} :



Now, if language $L(R)$ were local, then it should be determined by the local sets of expression R , hence the local automaton A_{loc} should be equivalent to the automaton A of $L(R)$ already obtained at point (c). It is well known that the minimal automaton is unique. Thus, since automaton A is minimal, it should be identical to the minimal local automaton A_{loc}^{min} . Yet it is evident that such minimal automata are different, as they do not have the same number of states. Therefore they are not equivalent, and in conclusion language $L(R)$ is not local. Actually, what holds is that language $L(R)$ is strictly contained in the local language $L(A_{loc})$, i.e., $L(R) \subsetneq L(A_{loc})$. In fact, it is immediate to see that automaton A_{loc} (or A_{loc}^{min}) has all the transition sequences that automaton A has, but that additionally it can read letter a after reading letter c . By the way, this is the same conclusion obtained before, by just examining a few adjacences. Notice that this alternative approach is much longer (though at all rigorous) than directly examining whether the constraints imposed by the local sets of expression R exactly model the strings generated by R itself, because it requires to compute two minimal automata. Here the approach works speedily enough as it benefits from already having found at point (c) the minimal automaton of language $L(R)$, but in other situations it might be quite slow and inefficient.

- (e) Following the previous answers, to disambiguate the original regular expression R , it suffices to let it generate strings $(a \mid b)^* a b$ in only one way. This can be done informally by disallowing the optional subexpression $a c^* b$ to generate no letter c , that is, by forcing it to generate at least one, i.e., by changing star into cross. Here is a non-ambiguous regular expression R' equivalent to R :

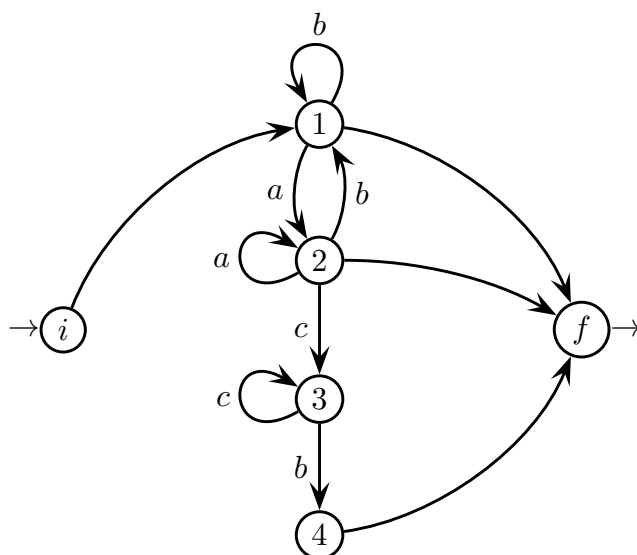
$$R' = (a \mid b)^* [a c^+ b]$$

or, only using standard operators:

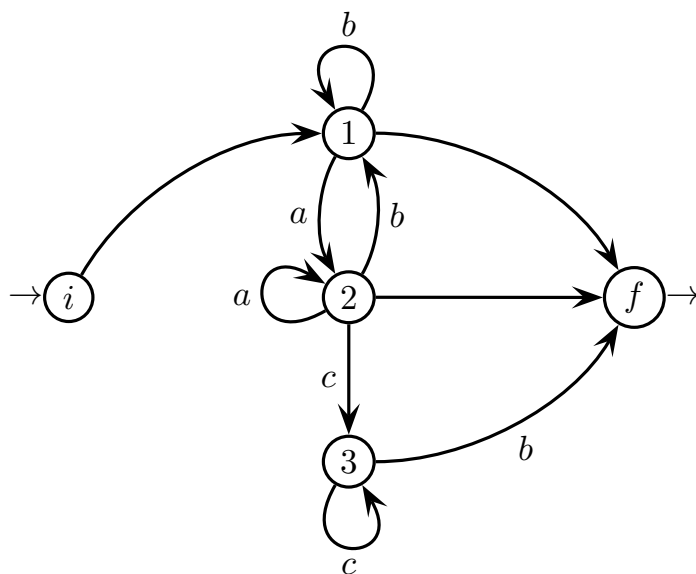
$$R' = (a \mid b)^* (a c^+ b \mid \varepsilon)$$

To widen the picture, a systematic approach for disambiguating is to resort to node elimination (Brzozowsky), starting from the deterministic automaton A . Furthermore, one should take care, at every elimination step, of applying the construction literally, or of manipulating the intermediate regular subexpressions so as to keep them unambiguous. For this reason, here such an approach is likely to take longer.

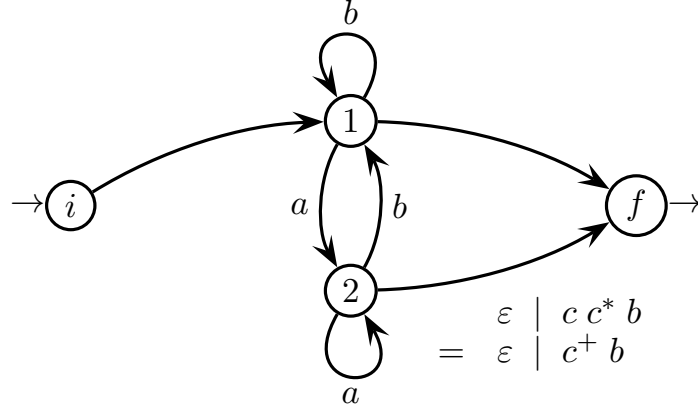
Anyway, with this in mind, here is the construction. Give automaton A unique initial and final states i and f , without ingoing and outgoing arcs, respectively:



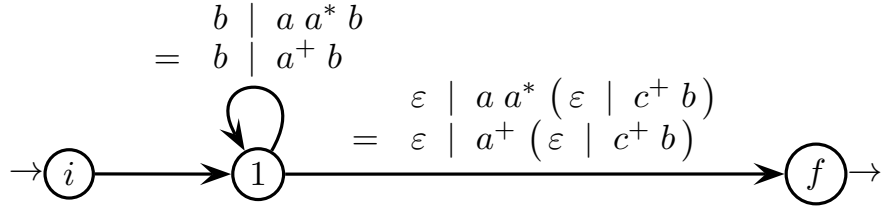
Eliminate node 4:



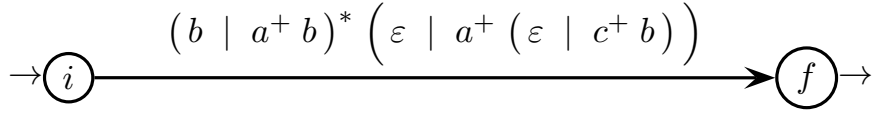
Eliminate node 3:



Eliminate node 2:



Eliminate node 1:



Thus a non-ambiguous regular expression R' equivalent to R is the following:

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

or, by unambiguously reintroducing the optionality operator (in nested form):

$$R' = (b \mid a^+ b)^* \left[a^+ [c^+ b] \right]$$

As it is apparent in both forms, it is impossible for the last letter pair ab to be generated by the final subexpression $[a^+ [c^+ b]]$, while it can still be generated by the initial subexpression $(b \mid a^+ b)^*$. This is the feature that prevents ambiguity.

Notice that the node elimination construction has been applied literally, that is, step-by-step without any shortcut or manipulation, except the surely non-ambiguous simplifications $cc^* = c^+$ and $aa^* = a^+$. Thus it grants that the resulting regexp is not ambiguous. To exclude errors, it is not difficult to prove that the above formulation of R' is equivalent to the previous ones, obtained informally:

$$\begin{aligned}
R' &= \left(\underbrace{b \mid a^+ b}_{\text{factorize } b} \right)^* \left(\varepsilon \mid \underbrace{a^+ (\varepsilon \mid c^+ b)}_{\text{distribute } a^+} \right) \\
&= \left(\left(\underbrace{\varepsilon \mid a^+}_{\text{reduce to } a^*} \right) b \right)^* \left(\underbrace{\varepsilon \mid a^+}_{\text{reduce to } a^*} \mid \underbrace{a^+}_{\text{expand as } a^* a} c^+ b \right) \\
&= (a^* b)^* \left(\underbrace{a^* \mid a^* a c^+ b}_{\text{factorize } a^*} \right) \\
&= \underbrace{(a^* b)^* a^*}_{\substack{\text{univ. lang.} \\ \text{over } a, b}} \underbrace{(\varepsilon \mid a c^+ b)}_{\text{optionality}} \\
&= (a \mid b)^* [a c^+ b] \\
&= R' \text{ as before}
\end{aligned}$$

Again, there might be other solutions, more or less different from these three.

2 Free Grammars and Pushdown Automata 20%

1. Consider the two following free languages L_1 and L_2 over the three-letter alphabet $\Sigma = \{ a, b, c \}$:

$$\begin{aligned} L_1 &= \{ a^n c^m b^{2n} \mid m > 0 \wedge n \geq 0 \} \\ L_2 &= \{ a^{2n} c^{2m} b^n \mid m > 0 \wedge n \geq 0 \} \end{aligned}$$

Answer the following questions:

- (a) Write two *BNF* grammars G_1 and G_2 , with axioms S_1 and S_2 , that generate languages L_1 and L_2 , respectively.
 - (b) Write an *ambiguous BNF* grammar G that generates the union language $L = L_1 \cup L_2$. Show that this grammar G is ambiguous, by exhibiting an ambiguous sentence s generated by G and all the syntax trees of s .
 - (c) (optional) Design a *non-ambiguous BNF* grammar G' for the same language L , and provide an adequate justification for the absence of ambiguity. Furthermore, show the unique syntax tree of G' for the same sentence s of the previous point.
-

Solution

- (a) Grammars G_1 and G_2 are the obvious ones. Here they are (axioms S_1 and S_2):

$$G_1 \left\{ \begin{array}{l} S_1 \rightarrow a S_1 b b \\ S_1 \rightarrow C_1 \\ C_1 \rightarrow c C_1 \\ C_1 \rightarrow c \end{array} \right. \quad G_2 \left\{ \begin{array}{l} S_2 \rightarrow a a S_2 b \\ S_2 \rightarrow C_2 \\ C_2 \rightarrow c c C_2 \\ C_2 \rightarrow c c \end{array} \right.$$

One could remove the copy rules, though the axiomatic rules would split and get longer. Of course, there may be other solutions, more or less simple.

- (b) The union grammar G of the two grammars G_1 and G_2 is this (axiom S):

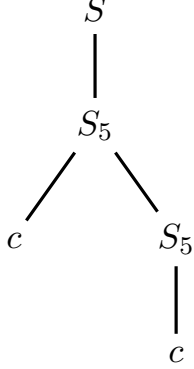
$$G \left\{ \begin{array}{l} \hline S \rightarrow S_1 \mid S_2 \\ \hline S_1 \rightarrow a S_1 b b \\ S_1 \rightarrow C_1 \\ C_1 \rightarrow c C_1 \\ C_1 \rightarrow c \\ \hline S_2 \rightarrow a a S_2 b \\ S_2 \rightarrow C_2 \\ C_2 \rightarrow c c C_2 \\ C_2 \rightarrow c c \end{array} \right. \quad \begin{array}{cc} \begin{array}{c} S \\ | \\ S_1 \\ | \\ C_1 \\ / \quad \backslash \\ c \quad C_1 \\ \quad | \\ \quad c \end{array} & \begin{array}{c} S \\ | \\ S_2 \\ | \\ C_2 \\ / \quad \backslash \\ c \quad c \end{array} \end{array}$$

Grammar G is ambiguous because the sentences of type c^{2m} , with $m > 0$, belong to both languages L_1 and L_2 . Actually they are the only shared sentences. This is a case of ambiguity due to the union of languages with a non-empty intersection, namely the (regular) sub-language $(cc)^+$. For instance, take the shortest ambiguous sentence $s = cc$, and see above the two syntax trees of s .

- (c) It is easy to design three non-ambiguous grammars G_3 , G_4 and G_5 for languages $L_3 = \{ a^n c^m b^{2n} \mid m > 0 \wedge n > 0 \}$, $L_4 = \{ a^{2n} c^{2m} b^n \mid m > 0 \wedge n > 0 \}$ and $L_5 = \{ c^n \mid n > 0 \}$, respectively. Such languages constitute a partition of the original union language $L = L_1 \cup L_2$, that is, their two-by-two intersections are empty. Thus the union of such grammars is the searched non-ambiguous grammar G' . Here are the three grammars (axiom S for all of them):

$$\begin{array}{ccc} G_3 & G_4 & G_5 \\ \left\{ \begin{array}{l} S \rightarrow a S b b \\ S \rightarrow a C b b \\ C \rightarrow c C \\ C \rightarrow c \end{array} \right. & \left\{ \begin{array}{l} S \rightarrow a a S b \\ S \rightarrow a a C b \\ C \rightarrow c c C \\ C \rightarrow c c \end{array} \right. & \left\{ \begin{array}{l} S \rightarrow c S \\ S \rightarrow c \end{array} \right. \end{array}$$

The construction of the non-ambiguous union grammar G' is as before: disjoin the nonterminal sets and unite the axiomatic rules. Here it is (axiom S), as well as the unique syntax tree of sentence $s = cc$, previously ambiguous:

$$G' \left\{ \begin{array}{l} \hline S \rightarrow S_3 \mid S_4 \mid S_5 \\ \hline S_3 \rightarrow a S_3 b b \\ S_3 \rightarrow a C_3 b b \\ C_3 \rightarrow c C_3 \\ C_3 \rightarrow c \\ \hline S_4 \rightarrow a a S_4 b \\ S_4 \rightarrow a a C_4 b \\ C_4 \rightarrow c c C_4 \\ C_4 \rightarrow c c \\ \hline S_5 \rightarrow c S_5 \\ S_5 \rightarrow c \end{array} \right.$$


Of course, since sub-grammar G_5 is right-linear, the tree of string cc is linearized. One could simplify grammar G' , since nonterminals C_3 and S_5 are equivalent (their rules are structurally identical), that is, it holds $L(C_3) = L(S_5) = c^+$. Thus for instance one can cancel the rules that expand C_3 and replace C_3 by S_5 ; see below on the left:

$$G' \left\{ \begin{array}{l} \hline S \rightarrow S_3 \mid S_4 \mid S_5 \\ \hline S_3 \rightarrow a S_3 b b \\ S_3 \rightarrow a S_5 b b \\ \hline S_4 \rightarrow a a S_4 b \\ S_4 \rightarrow a a C_4 b \\ C_4 \rightarrow c c C_4 \\ C_4 \rightarrow c c \\ \hline S_5 \rightarrow c S_5 \\ S_5 \rightarrow c \end{array} \right. \quad \left| \quad G' \left\{ \begin{array}{l} \hline S \rightarrow S_3 \mid S_4 \\ \hline S_3 \rightarrow a S_3 b b \\ S_3 \rightarrow S_5 \\ \hline S_4 \rightarrow a a S_4 b \\ S_4 \rightarrow a a C_4 b \\ C_4 \rightarrow c c C_4 \\ C_4 \rightarrow c c \\ \hline S_5 \rightarrow c S_5 \\ S_5 \rightarrow c \end{array} \right.$$

This enables another slight simplification, which is a categorization, i.e., the introduction of a copy rule, as it creates a chain of two copy rules; see above on the right. There may be other solutions, more or less simple and elegant.

2. Consider the fragment of a programming language targeted at the computation of operations on one-dimensional vectors and on scalars, i.e., integers. This language features the following syntactic structures:

- There are variables of vector (1-D) and scalar (integer) type, which are identified by uppercase and lowercase identifiers, e.g., A, B, \dots , and a, b, \dots , respectively.
- A vector can also be explicitly written as a non-empty list of scalars, enclosed in graph brackets “{” and “}”, and separated by commas “,”, e.g., $\{a, b, a, c\}$.
- There are arithmetic expressions, which can contain vector and scalar variables, with the addition operation “+”.
- There are subexpressions, enclosed in round brackets “(” and “)”.
- The associativity of the addition operator “+” is unspecified.
- There are assignments with a variable on the left side, the assignment operator “:=” in between, and an expression on the right side.
- An expression that contains a vector operand (variable name or list of scalars) may not be assigned to a scalar variable.
- The language fragment consists of a possibly empty list of assignments, and each assignment is terminated by a semicolon “;”.

Sample language fragment:

```
a := b + c;

B := a + B + c;

B := { a, b, c };

A := A + { c, d } + (c + C);
```

Answer the following questions:

- Write an extended (*EBNF*) grammar G , not ambiguous, that models the language fragment described above.
- (optional) Suppose the syntax of this language must be modified in such a way that, for instance, an expression like $A + a + b + B$ is parsed as $A + (a + b) + B$, to be able to pre-compute as many purely scalar operations as possible (for they are inexpensive in comparison to the vector operations), and in this way to save valuable computing time. Here are two more such sample assignments:

```
B := A + b + c;           // parsed as A + (b + c)

A := A + b + (c + d + B); // parsed as A + b + ((c + d) + B)
```

Write an extended (*EBNF*) grammar G' , not ambiguous, that models this modified language fragment. You can reuse the grammar G obtained before, and just show the changes to be introduced.

Solution

- (a) The key point is to isolate the purely scalar (sub)expression S_EXP , which is modeled as the vector one V_EXP , but is a sub-case thereof; the rest of the grammar is more or less standard. Notice that the assignment left member is a variable name (vector or scalar), not a list. Here is grammar G (axiom $PROG$):

$$G \left\{ \begin{array}{l} \langle PROG \rangle \rightarrow (\langle ASGN \rangle ' ; ')^* \\ \hline \langle ASGN \rangle \rightarrow \langle S_VAR \rangle ' := ' \langle S_EXP \rangle \mid \langle V_VAR \rangle ' := ' \langle V_EXP \rangle \\ \hline \langle S_EXP \rangle \rightarrow \langle S_TRM \rangle (' + ' \langle S_TRM \rangle)^* \\ \langle S_TRM \rangle \rightarrow \langle S_VAR \rangle \mid ' (' \langle S_EXP \rangle ') ' \\ \hline \langle S_VAR \rangle \rightarrow a \dots z \\ \hline \langle V_EXP \rangle \rightarrow \langle V_TRM \rangle (' + ' \langle V_TRM \rangle)^* \\ \langle V_TRM \rangle \rightarrow \langle S_VAR \rangle \mid \langle S_LST \rangle \mid \langle V_VAR \rangle \mid ' (' \langle V_EXP \rangle ') ' \\ \langle V_VAR \rangle \rightarrow A \dots Z \\ \hline \langle S_LST \rangle \rightarrow ' \{ ' \langle S_VAR \rangle (' , ' \langle S_VAR \rangle)^* ' \} ' \end{array} \right.$$

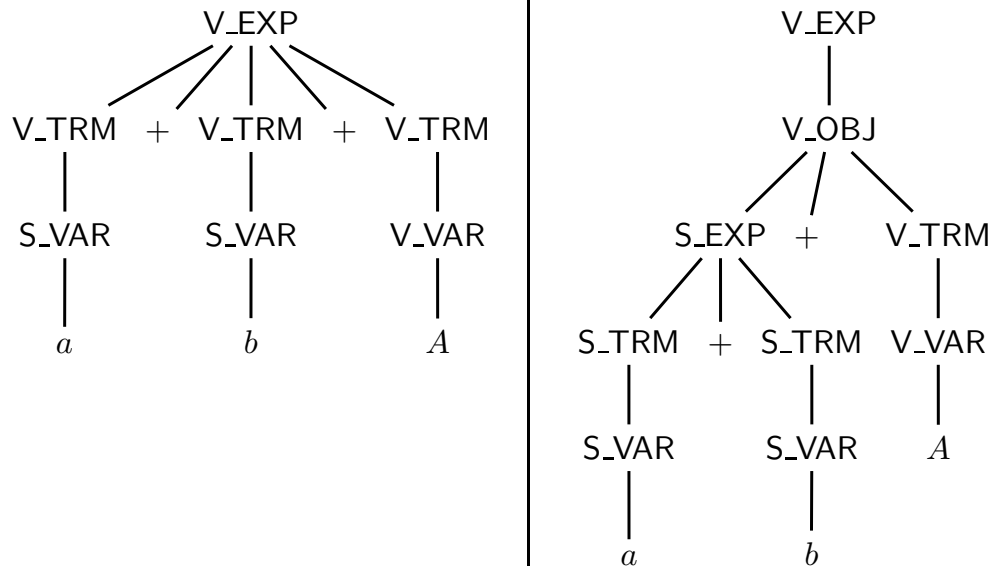
Grammar G is not ambiguous, basically being the classical *EBNF* non-ambiguous grammar of arithmetic expressions. Square brackets indicate optionality, as usual. Since *EBNF* rules are used to generate repeated sum, the associativity of the addition operation is unspecified, as permitted by the language description.

- (b) It is well known that using a rule that contains an iterative operator like star or cross leaves associativity unspecified. Since here the requirement is precisely that in a repeated addition with both scalar and vector operations, the scalar ones have to be associated (hence parsed) first (hence pre-computed), something has to be changed in the related iterative rules to model such an inner structure. Here is a possible grammar G' , which shows only the rules and nonterminals that are adjoined or modified with respect to grammar G (the rest is unchanged):

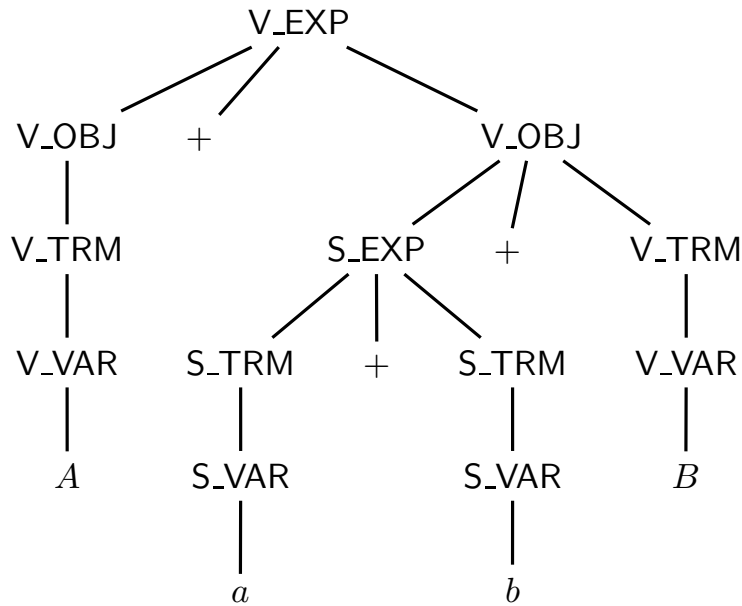
$$\begin{aligned} \langle V_EXP \rangle &\rightarrow \langle S_EXP \rangle \mid \langle V_OBJ \rangle (' + ' \langle V_OBJ \rangle)^* [' + ' \langle S_EXP \rangle] \\ \langle V_OBJ \rangle &\rightarrow [\langle S_EXP \rangle ' + '] \langle V_TRM \rangle \\ \langle V_TRM \rangle &\rightarrow \langle S_LST \rangle \mid \langle V_VAR \rangle \mid ' (' \langle V_EXP \rangle ') ' \end{aligned}$$

In practice, a new syntactic class V_OBJ is introduced between classes V_EXP and V_TRM . A vector object V_OBJ is a vector term V_TRM that can optionally have an adjoint full scalar (sub)expression S_EXP as its left addend. The subtree of S_EXP will necessarily be fully evaluated before being added to V_TRM . Consequently a vector term V_TRM may no longer be expansible into a scalar variable S_VAR as before, lest the grammar gets ambiguous, whereas now it is a vector expression V_EXP that can be wholly declassified into a scalar one S_EXP , that is, into a one-element vector. Furthermore, an adjoint scalar (sub)expression S_EXP may also occur at the end of a whole vector (sub)expression V_EXP .

What said above is the syntactic property to be granted for the compiler to generate a machine code that will eventually save computing time by reducing the number of vector operations. Consider for instance the sample expression $a + b + A$ and its syntax trees in the grammars G and G' , respectively:



Notice that parenthesizing the (sub)expression nonterminals X_EXP (with $X = V, S$) in the left tree would yield $(a + b + A)$, without any explicit indication about which one of the two additions to compute first, as the 3-term addition could be associated from left or right indifferently; whereas in the right tree it would yield $((a + b) + A)$, where clearly the scalar sum $a + b$ has to be computed first. Similarly, the G' tree of the text sample $A + a + b + B$ is the following:



In this case, parenthesizing yields $(A + (a + b) + B)$, as requested. The reader can verify by himself that also the other samples are parsed as requested. In

summary, all the scalar operations between two vector terms are (arbitrarily) attributed to the right term (if any), and computed before adding the vector terms; there is a dual solution that would attribute them to the left term.

Grammar G' could be slightly simplified by removing the new nonterminal V_OBJ : just replace it in the rule of V_EXP , which anyway gets a little longer and less readable. There may be other solutions, of *EBNF* type or not.

To better understand the impact of such a syntactic change, notice that the three addition signs that occur in grammar G' could be more precisely classified as follows:

$$\begin{array}{ll}
 \dots & \dots \\
 \langle S_EXP \rangle & \rightarrow \langle S_TRM \rangle \left(\underset{scalar}{\text{'+'}} \langle S_TRM \rangle \right)^* \\
 \dots & \dots \\
 \langle V_EXP \rangle & \rightarrow \langle S_EXP \rangle \mid \langle V_OBJ \rangle \left(\underset{vector}{\text{'+'}} \langle V_OBJ \rangle \right)^* \left[\underset{semi_vector}{\text{'+'}} \langle S_EXP \rangle \right] \\
 \langle V_OBJ \rangle & \rightarrow \left[\langle S_EXP \rangle \underset{semi_vector}{\text{'+'}} \right] \langle V_TRM \rangle \\
 \dots & \dots
 \end{array}$$

These three addition signs could be mapped by a code generation translation (possibly a syntactic one) to as many specialized machine instructions, as follows:

```

// scalar addition with two scalar addends s1 and s2
sadd s1, s2, ss      // where ss is their scalar sum

// semi-vector addition with a scalar addend s and a vector one v
svadd s, v, vs       // where vs is their vector sum

// (fully) vector addition with two vector addends v1 and v2
vadd v1, v2, vs      // where vs is their vector sum

```

Clearly these three machine operations have increasing computational costs. Grammar G' , which appends the scalar addition operation only to nonterminal S_EXP , the subtree of which will be evaluated first, substantially helps a compiler code generator identify and map such an operation to the lowest-cost machine instruction of scalar addition, and therefore satisfies the proposed requirement. For instance, a bottom-up translation driven by grammar G' would work as follows (see the previous tree):

```

A + a + b + B  ->  sadd a, b, s1; svadd s1, B, V1; vadd A, V1, V2

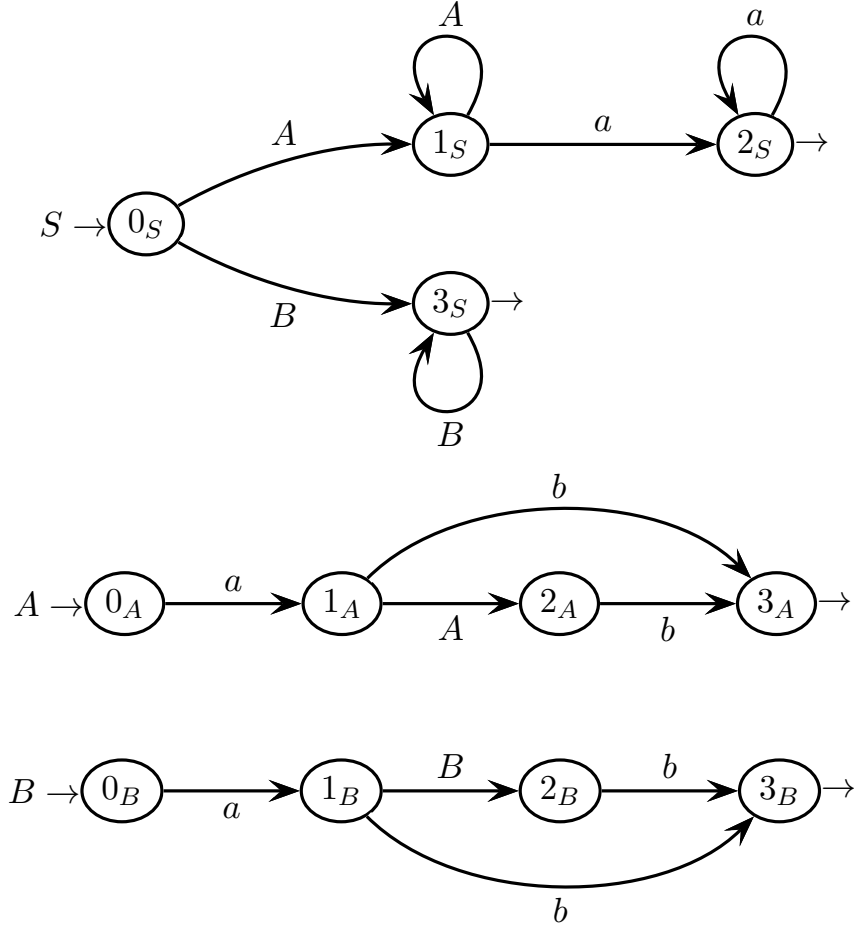
```

where arguments $s1$, $V1$ and $V2$ are one scalar and two vector registers, respectively.

This is impossible or much harder to do with grammar G . The addition sign in the rule of nonterminal V_EXP may end up with having only scalar addends, or only vector ones, or even mixed ones. Therefore a code generation translation would be unable to selectively map such an addition to the lowest-cost machine instruction available, or at least the translation should be more complex than with grammar G' , e.g., it ought to be semantic instead of purely syntactic.

3 Syntax Analysis and Parsing Methodologies 20%

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



Answer the following questions:

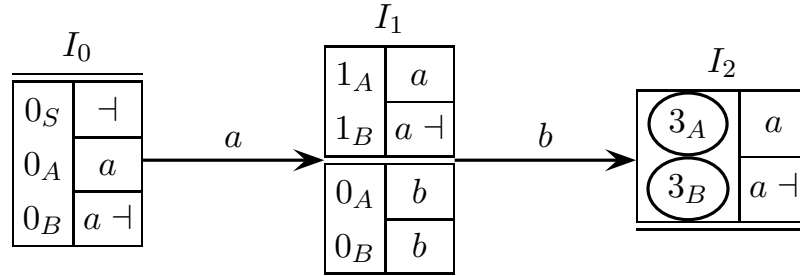
- (a) Build a small part of the pilot automaton that suffices to prove that the language is not $ELL(1)$.
- (b) Build a small part of the pilot automaton that suffices to prove that the language is not $ELR(1)$.
- (c) By using the Earley method, simulate the analysis of the string $ab a$. Furthermore, on the data structure computed by the algorithm, show whether and why the string $ab a$, or any of its prefixes, is accepted. Use the Early table prepared on the next page.
- (d) (optional) For language $L(G)$, define an $ELR(1)$ grammar G' , as simple as possible, and show that this new grammar G' is actually $ELR(1)$.

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

0	a	1	b	2	a	3
-----	-----	-----	-----	-----	-----	-----

Solution

- (a) Here is a partial pilot, corresponding to the two shifts for the shortest possible string of language $L(G)$. This pilot fragment proves that grammar G is not of type ELL (the complete pilot has many more m-states and transitions):



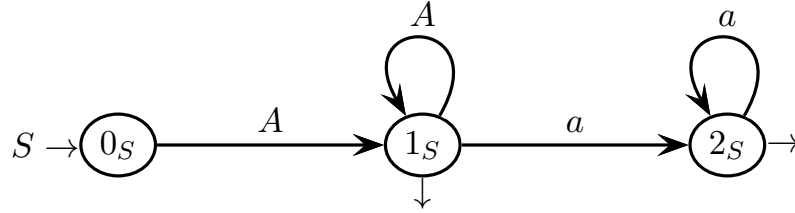
In fact both pilot transitions $I_0 \xrightarrow{a} I_1$ and $I_1 \xrightarrow{b} I_2$ are multiple (but not convergent), in particular double, because the bases of their destination m-states I_1 and I_2 contain more than one candidate (item), more precisely two per each m-state. Thus the pilot as a whole does not have the single transition property (STP), so the ELL condition fails, and grammar G is not of type $ELL(1)$.

- (b) The same pilot fragment constructed for answering question (a), also proves that grammar G is not of type ELR . In fact, since m-state I_2 contains two reduction candidates (items), namely 3_A and 3_B , with non-disjoint look-ahead (they share terminal a), the pilot has (at least) a reduce-reduce conflict, thus grammar G is not of type $ELR(1)$. Of course, the complete pilot might have more conflicts of the same type or also of other types.
- (c) Here is the complete Earley analysis of the (valid) string aba of length 3:

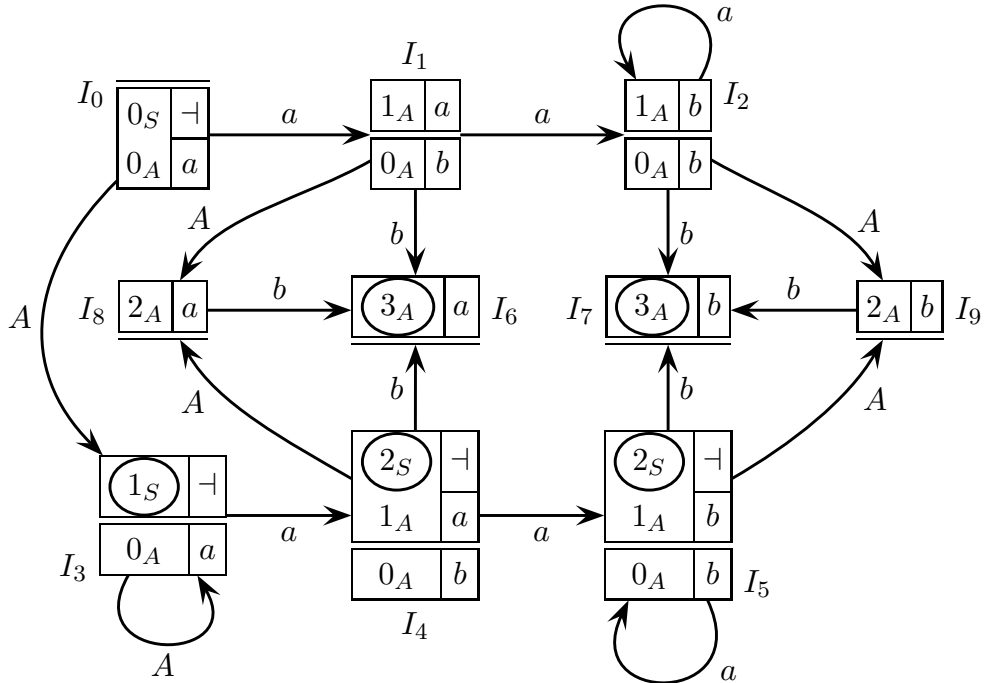
0	a	1	b	2	a	3
0_S 0		1_A 0		3_A 0		2_S 0
0_A 0		1_B 0		3_B 0		1_A 2
0_B 0		0_A 1		1_S 0		1_B 2
		0_B 1		3_S 0		0_A 3
				0_A 2		0_B 3
				0_B 2		

One can see that vector element $E(3)$ contains a final axiomatic candidate 2_S with pointer 0, so that the acceptance condition for string aba is verified. Furthermore, also vector element $E(2)$ contains a final axiomatic candidate 3_S with pointer 0, so that the acceptance condition for the prefix string ab is verified.

- (d) First notice that grammar G is not ambiguous, so that its failing to be of ELR type is not imputable to ambiguity. The conflict found at question (a), where the two machines M_A and M_B compete for reduction, suggests to analyze them more thoroughly. Then, it is immediately evident that machines M_A and M_B are identical, hence equivalent. Therefore the two languages $L(A)$ and $L(B)$ coincide. This implies that the initial paths A^+ and B^+ from state 0_S in the axiomatic machine M_S , actually recognize the same (sub)strings. As a consequence, this suggests that the axiomatic machine M_S can be simplified as follows:



It suffices to remove path B^+ and to make state 1_S final. Also machine M_B can be removed from then net. It suffices to keep machines M_S (simplified) and M_A . This new net (and grammar G') is equivalent to the original one. To fully prove that it is of type ELR , it is necessary to draw its pilot and check it. Here it is:



This new pilot does not have any conflict, therefore grammar G' is of type $ELR(1)$ and it is reasonably simple. Since it holds $L(A) = \{a^n b^n \mid n \geq 1\}$ and $L(S) = (L(A))^+ a^*$, it seems hard to find a grammar much simpler than grammar G' . Of course, there may be other comparably simple solutions.

4 Language Translation and Semantic Analysis 20%

1. It is well known that the following *EBNF* grammar G , not ambiguous (axiom S):

$$G: S \rightarrow (a S b)^*$$

generates the Dyck language with parentheses a (open) and b (closed).

Consider a natural generalization of the notion of translation grammar to the case of *Extended BNF* grammars, following the ideas underlying Regular Translation Expressions. Use the above grammar G to take inspiration for the following questions.

- (a) Write a translation grammar (or scheme) G_1 , not ambiguous, of *EBNF* type, that translates the source language $L(G)$ so that the parenthesis pairs at an even *depth* level (0, 2, 4, etc, notice that the depth index starts from 0) are transliterated onto c (open) and d (closed). The other parentheses are left unchanged by the translation. Samples:

$$\begin{aligned}\tau_1(a b a b) &= c d c d \\ \tau_1(a a b a b b) &= c a b a b d \\ \tau_1(a a a b b a b b) &= c a c d b a b d\end{aligned}$$

- (b) Write a translation grammar (or scheme) G_2 , not ambiguous, of *EBNF* type, that translates the source language $L(G)$ so that the parenthesis pairs at an even *position* (second, fourth, sixth, etc, notice that the position index starts from 1) in a list of parentheses concatenated at the same depth level are transliterated onto c (open) and d (closed). The other parentheses are left unchanged by the translation. Samples:

$$\begin{aligned}\tau_2(a b a b a b) &= a b c d a b \\ \tau_2(a a b a b a b b) &= a a b c d a b b \\ \tau_2(a a b a b a b b a a b b) &= a a b c d a b b c a b d\end{aligned}$$

- (c) (optional) Say if the translation grammar (or scheme) G_2 found at point (b) is deterministic or not, in particular if it is of type *ELL*, and justify your answer.

Solution

- (a) Here is the requested translation scheme G_1 , of type *EBNF* (because it contains a Kleene star operator) inspired to grammar G (axiom E):

$$G_1 \left\{ \begin{array}{l} E \rightarrow (a O b)^* \\ O \rightarrow (a E b)^* \end{array} \right. \quad \left| \quad \begin{array}{l} E \rightarrow (c O d)^* \\ O \rightarrow (a E b)^* \end{array} \right.$$

The source grammar G_1 (left) is structurally identical to grammar G , but the original nonterminal S that generates the nested parenthesis pairs is split into nonterminals E and O , which count the even and odd depth level parity of the nested pairs, respectively. Now the destination grammar G_1 (right) is obvious. A more insightful *EBNF* solution is the following, where star nesting takes the role of the previously split nonterminals, which are reunified (axiom S):

$$G_1: S \rightarrow \left(a \left(\underbrace{\underbrace{a S b}_{\substack{\text{odd depth} \\ 1, 3, \dots}}}_{\substack{\text{even depth } 0, 2, \dots}} \right)^* b \right)^* \quad \left| \quad S \rightarrow \left(c \left(\underbrace{\underbrace{a S b}_{\substack{\text{unchanged} \\ \text{(still } a b)}}}_{\substack{\text{translated to } c d}} \right)^* d \right)^*$$

In the source rule, the outer star concatenates parenthesis pairs at even depth level, the inner one at odd level instead, and the innermost recursion restarts the generation at the next even level. In the destination rule, the pairs at even depth level are translated as requested, while those at odd level are unchanged. This is an obvious purely *BNF* scheme equivalent to grammar G_1 (axiom E):

$$\left\{ \begin{array}{l} E \rightarrow a O b E \mid \varepsilon \\ O \rightarrow a E b O \mid \varepsilon \end{array} \right. \quad \left| \quad \begin{array}{l} E \rightarrow c O d E \mid \varepsilon \\ O \rightarrow a E b O \mid \varepsilon \end{array} \right.$$

Here star is transformed into right recursion. Anyway, this Dyck form does not satisfy the requirement of being of type *EBNF* and inspired to grammar G .

- (b) Here is the translation grammar G_2 (it could be written as a scheme, too), also of type *EBNF* inspired to G , with only one nonterminal and one rule (axiom S):

$$G_2: S \rightarrow \underbrace{\left(\frac{a}{a} S \frac{b}{b} \frac{a}{c} S \frac{b}{d} \right)^*}_{\text{twice unrolled star}} \underbrace{\left(\frac{a}{a} S \frac{b}{b} \mid \varepsilon \right)}_{\text{footer for odd cases}}$$

The source grammar G_2 is obtained from grammar G , but it is different. In fact, the star is unrolled twice to separate the concatenated parenthesis pairs in an odd (first, third, etc) or even (second, fourth, etc) position, with a footer to accomplish for the cases where there is an odd number of concatenated pairs. After so restructuring the source grammar, the destination grammar G_2 is obvious.

Other versions of grammar G_2 can be obtained by using the optionality operator:

$$G_2: S \rightarrow \left(\frac{a}{a} S \frac{b}{b} \frac{a}{c} S \frac{b}{d} \right)^* \left[\frac{a}{a} S \frac{b}{b} \right]$$

or by using a header with an (optional) footer and still unrolling twice the star:

$$G_2: S \rightarrow \left[\frac{a}{a} S \frac{b}{b} \left(\frac{a}{c} S \frac{b}{d} \frac{a}{a} S \frac{b}{b} \right)^* \left[\frac{a}{c} S \frac{b}{d} \right] \right]$$

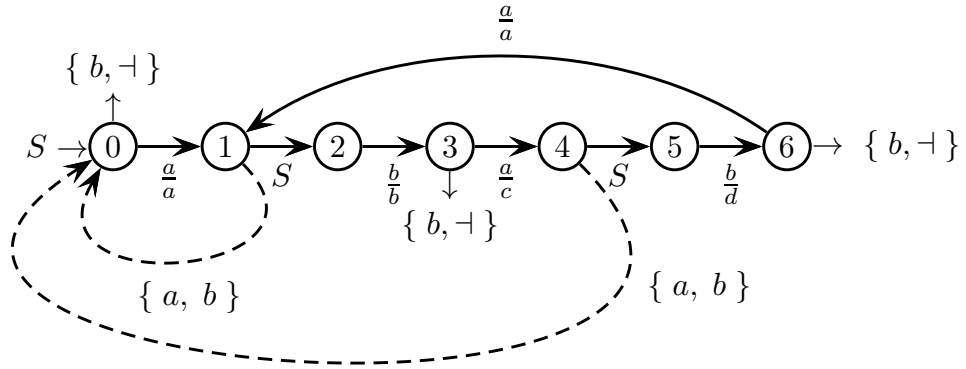
However this version is a little longer than before and uses a nested optionality.

This is an obvious purely *BNF* grammar equivalent to grammar G_2 (axiom O):

$$\begin{cases} O \rightarrow \frac{a}{a} O \frac{b}{b} E \mid \frac{\varepsilon}{\varepsilon} \\ E \rightarrow \frac{a}{c} O \frac{b}{d} O \mid \frac{\varepsilon}{\varepsilon} \end{cases}$$

Again, star is transformed into right recursion. Anyway, this Dyck form does not satisfy the requirement of being of type *EBNF* and inspired to grammar G .

- (c) It suffices to draw a machine translation net for translation grammar G_2 and verify that its source part is of type *ELL*(1), that is, deterministic. Here it is:



The guide sets on the three bifurcation points, i.e., states 0, 3 and 6, are disjoint, therefore the source net is of type *ELL*(1). Thus the *EBNF* translation grammar G_2 is deterministic and has a top-down pushdown translator.

For completeness, it is also shown a recursive-descent translator for G_2 . The main program (syntax analyzer) is the standard one, as shown in the textbook. Here it is:

```

program ELL_PARSER
  cc = next                                // initial read
  call S                                    // invoke axiom
  if cc ∈ {+} accept                       // translation success
  else reject                               // translation failure
end program

```

There is only one syntactic procedure S , which however is recursive and has a loop:

```

procedure S
  if  $cc \in \{ a \}$                                      // start in state 0
  |   write (a)                                         // output a
  |    $cc = next$                                          // goto state 1
  |   loop                                              // endless loop
  |   |   if  $cc \in \{ a, b \}$                          // enter state 1
  |   |   |   call S                                   // goto state 2
  |   |   |   if  $cc \in \{ b \}$                        // enter state 2
  |   |   |   |   write (b)                         // output b
  |   |   |   |    $cc = next$                          // goto state 3
  |   |   |   |   if  $cc \in \{ a \}$                    // enter state 3
  |   |   |   |   |   write (c)                     // output c
  |   |   |   |   |    $cc = next$                      // goto state 4
  |   |   |   |   |   if  $cc \in \{ a, b \}$            // enter state 4
  |   |   |   |   |   |   call S                   // goto state 5
  |   |   |   |   |   |   if  $cc \in \{ b \}$          // enter state 5
  |   |   |   |   |   |   |   write (d)           // output d
  |   |   |   |   |   |   |    $cc = next$            // goto state 6
  |   |   |   |   |   |   |   if  $cc \in \{ a \}$      // enter state 6
  |   |   |   |   |   |   |   |   write (a)       // output a
  |   |   |   |   |   |   |   |    $cc = next$        // goto state 1 (line loop)
  |   |   |   |   |   |   |   |   else if  $cc \in \{ b, \vdash \}$  return // return from state 6
  |   |   |   |   |   |   |   |   else error      // error in state 6
  |   |   |   |   |   |   |   |   else error      // error in state 5
  |   |   |   |   |   |   |   |   else error      // error in state 4
  |   |   |   |   |   |   |   |   else if  $cc \in \{ b, \vdash \}$  return // return from state 3
  |   |   |   |   |   |   |   |   else error      // error in state 3
  |   |   |   |   |   |   |   |   else error      // error in state 2
  |   |   |   |   |   |   |   |   else error      // error in state 1
  |   |   |   |   |   |   |   |   end
  |   |   |   |   |   |   |   |   else if  $cc \in \{ b, \vdash \}$  return // return from state 0
  |   |   |   |   |   |   |   |   else error      // error in state 0
  |   |   |   |   |   |   |   |   end procedure

```

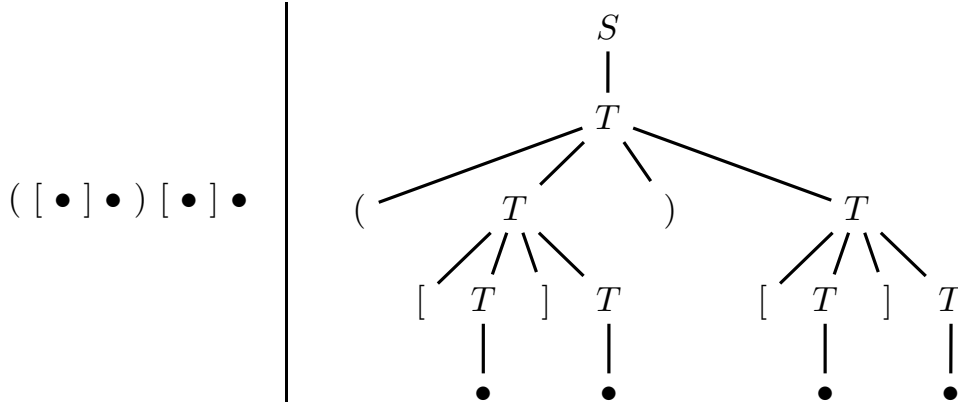
This syntactic procedure faithfully maps the machine translation net above. Comments mark the state transitions as the procedure walks the machine graph. Returns go back to the caller, while errors stop execution. Notice that an endless loop is used, which is truncated only by return or error. An alternative programming would be to use a conditional loop of some kind (while, repeat, etc). Furthermore, the procedure could be optimized by grouping the error cases into fewer ones or maybe only one, and possibly by some other known programming technique; this is left to the reader. As a result however, the mapping between procedure listing and graph topology would not be so faithfully rendered any more (though semantically equivalent).

Of course, one might wonder whether translation grammar G_2 is of type $ELR(1)$, too. For translations, differently from purely generative grammars, being ELR is not implied by being ELL , since an ELR translation has to be in the postfix form. Now, translation grammar G_2 is not in the postfix form, and it should be transformed so before being analyzed. The analysis result might be that it is ELR or not.

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

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

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



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

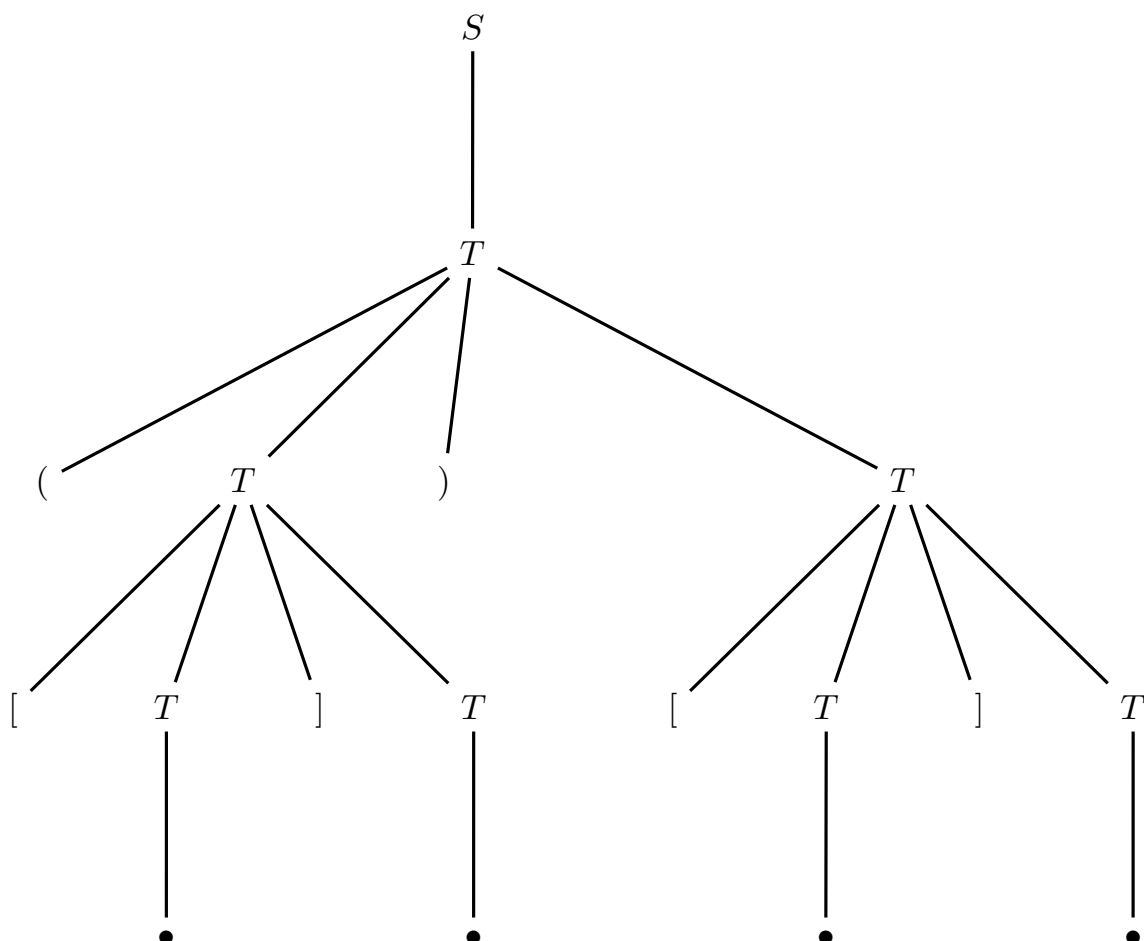
$$v = \underbrace{2+3}_{\text{first bullet}} + \underbrace{2+0}_{\text{second bullet}} + \underbrace{0+3}_{\text{third bullet}} + \underbrace{0+0}_{\text{fourth bullet}} = 10$$

Answer the following questions:

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

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

tree prepared for decoration with attributes



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

1: $S_0 \rightarrow T_1$

2: $T_0 \rightarrow (T_1) T_2$

3: $T_0 \rightarrow [T_1] T_2$

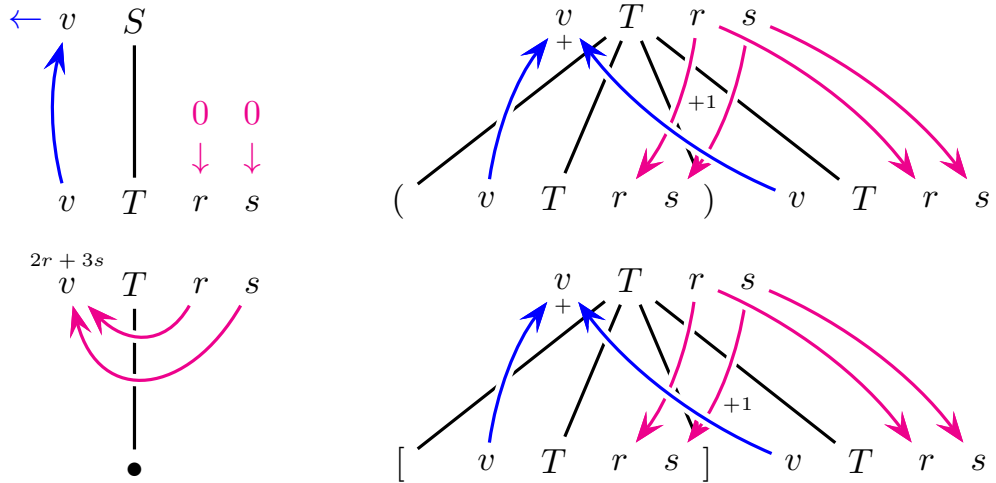
4: $T_0 \rightarrow \bullet$

Solution

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

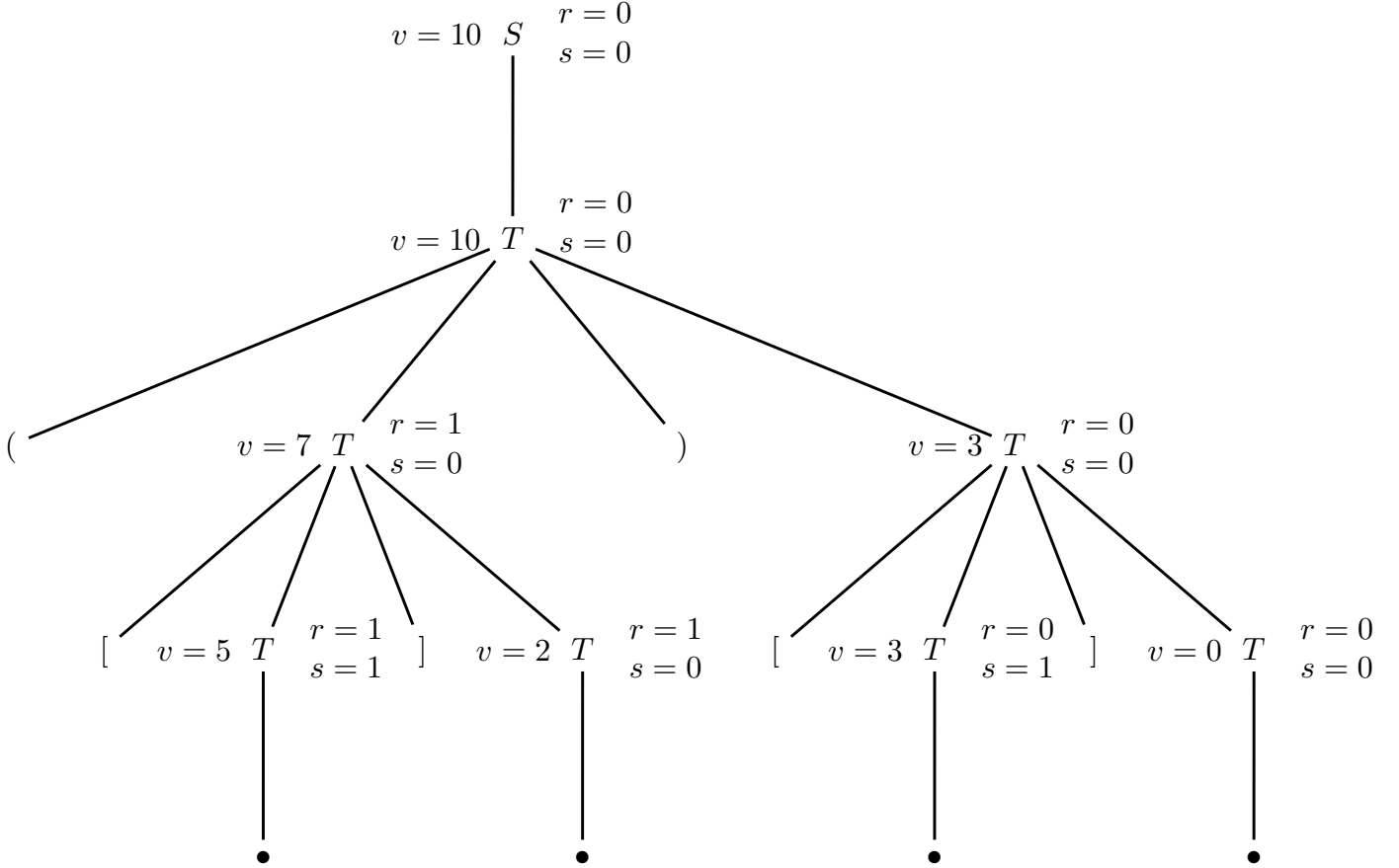
#	<i>syntax</i>	<i>semantics</i>
1:	$S_0 \rightarrow T_1$	$r_1 := 0$ $s_1 := 0$ $v_0 := v_1$
2:	$T_0 \rightarrow (T_1)T_2$	$r_1 := r_0 + 1$ $r_2 := r_0$ $s_1 := s_0$ $s_2 := s_0$ $v_0 := v_1 + v_2$
3:	$T_0 \rightarrow [T_1]T_2$	$r_1 := r_0$ $r_2 := r_0$ $s_1 := s_0 + 1$ $s_2 := s_0$ $v_0 := v_1 + v_2$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

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



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

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



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

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

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

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

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

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

<pre> program PARSER cc = next call S if cc ∈ { '⊥' } accept else reject endif end program procedure S if cc ∈ { '(', '[', '•' } call T return // scanned 1 else error // error case endif end procedure </pre>	<pre> procedure T if cc ∈ { ' ' } // 1st way: scan rule 2 cc = next if cc ∈ { '(', '[', '•' } call T if cc ∈ { ' ' } cc = next if cc ∈ { '(', '[', '•' } call T return // scanned 2 endif endif endif else if cc ∈ { '[' } // 2nd way: scan rule 3 cc = next if cc ∈ { '(', '[', '•' } call T if cc ∈ { ']' } cc = next if cc ∈ { '(', '[', '•' } call T return // scanned 3 endif endif endif else if cc ∈ { '•' } // 3rd way: scan rule 4 cc = next return // scanned 4 endif error // grouped error cases end procedure </pre>
---	--

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

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

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

program EVALUATOR

```

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

```

```

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

```

```

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

```

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