FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Thu 18 JANUARY 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:

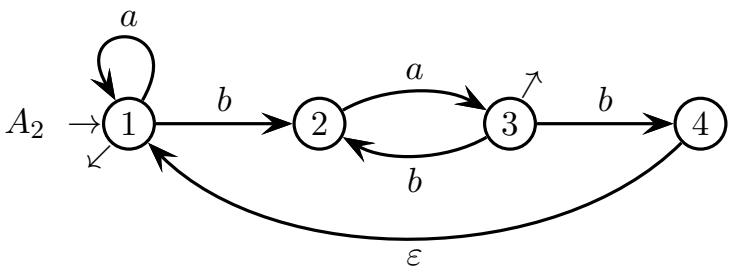
- The exam is in written form and consists of two parts:
  1. Theory (80%): Syntax and Semantics of Languages
     - regular expressions and finite automata
     - free grammars and pushdown automata
     - syntax analysis and parsing methodologies
     - language translation and semantic analysis
  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 regular expression R_1 below, over the two-letter alphabet $\{ a, b \}$:

$$R_1 = (a \mid \varepsilon)^+ \ (b\, a \mid b\, a\, b)^*$$

Consider also the non-deterministic automaton A_2 below, over the same two-letter alphabet $\{ a, b \}$, with a spontaneous transition, and with final nodes 1 and 3:



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

(a) Determine whether expression R_1 is ambiguous, and explain your answer.

(b) By first using the Berry-Sethi method and then minimizing if necessary, find a deterministic and minimal automaton A_1 equivalent to expression R_1.

(c) Determine whether the language $L(R_1)$ generated by expression R_1 is local, and explain your answer.

(d) Find an automaton, not necessarily deterministic, equivalent to automaton A_2, but without spontaneous transitions.

(e) (optional) What is the relationship between the language generated by expression R_1 and the language accepted by automaton A_2 ? Explain your answer.

---

question (a)

please build here the Berry-Sethi automaton A_1 of expression R_1 – question (b)

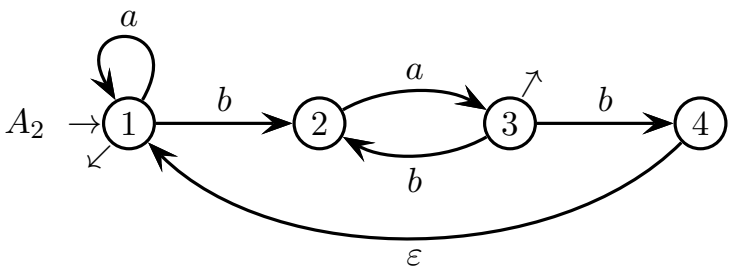
$R_{\#,1} =$

initials	
terminals	followers – the number of rows is not significant

draw here the state-transition graph of automaton A_1

please construct here the automaton without spontaneous transitions – question (d)

state-transition graph of the original automaton A_2



state-transition graph of the automaton without spontaneous transitions

rest of the exercise – questions (c), (e), and (a) if necessary

Solution

(a) Expression R_1 is unlimitedly ambiguous. For instance, string $a \in L(R_1)$ is ambiguous, as these derivations show:

$$R = (a \mid \varepsilon)^+ \ (ba \mid bab)^* \Rightarrow (a \mid \varepsilon)^+ \Rightarrow (a \mid \varepsilon)^1 = a \mid \varepsilon \Rightarrow a$$
$$R = (a \mid \varepsilon)^+ \ (ba \mid bab)^* \Rightarrow (a \mid \varepsilon)^+ \Rightarrow (a \mid \varepsilon)^2 = (a \mid \varepsilon)\,(a \mid \varepsilon) \Rightarrow a\,\varepsilon = a$$
$$R = (a \mid \varepsilon)^+ \ (ba \mid bab)^* \Rightarrow (a \mid \varepsilon)^+ \Rightarrow (a \mid \varepsilon)^2 = (a \mid \varepsilon)\,(a \mid \varepsilon) \Rightarrow \varepsilon\,a = a$$
$$R = (a \mid \varepsilon)^+ \ (ba \mid bab)^* \Rightarrow (a \mid \varepsilon)^+ \Rightarrow (a \mid \varepsilon)^3 = (a \mid \varepsilon)\,(a \mid \varepsilon)\,(a \mid \varepsilon) \Rightarrow a\,\varepsilon\,\varepsilon = a$$
$$\ldots$$

Clearly the number of derivations of string a is infinite, so expression R_1 is unlimitedly ambiguous. Notice that even the empty string $\varepsilon \in L(R_1)$ is ambiguous.

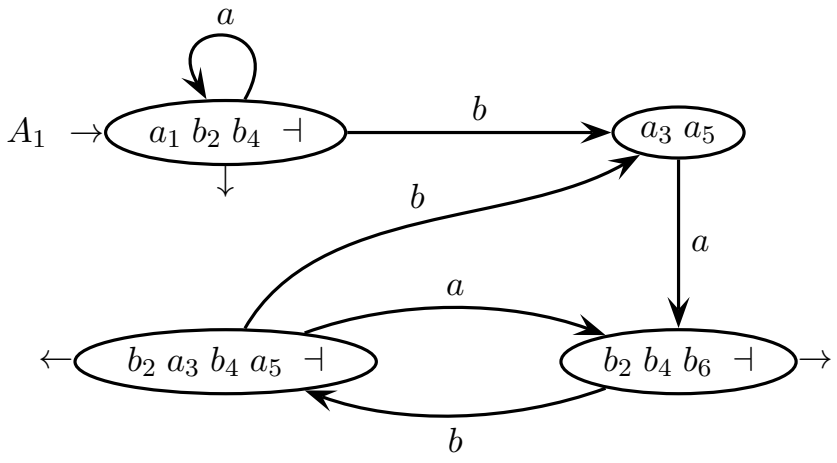
(b) The BS automaton A_1 has four states and is already minimal. Here is the construction of A_1. First the numbered expression $R_{\#,1}$:

$$R_{\#,1} = (\, a_1 \ \mid \ \varepsilon \,)^+ \ (\, b_2 \ a_3 \ \mid \ b_4 \ a_5 \ b_6 \,)^* \dashv$$

Then the sets of the initials and of the followers:

initials	$a_1,\ b_2,\ b_4, \dashv$
terminals	followers
a_1	$a_1,\ b_2,\ b_4,\ \dashv$
b_2	a_3
a_3	$b_2,\ b_4,\ \dashv$
b_4	a_5
a_5	b_6
b_6	$b_2,\ b_4,\ \dashv$

Finally, here is the state-transition graph of the deterministic automaton A_1 produced by the *BS* method, with four states:



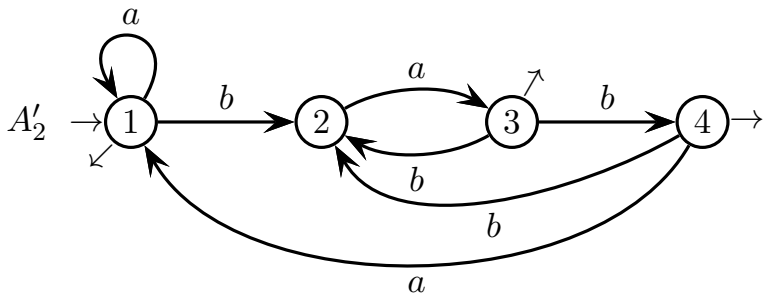
Automaton A_1 is deterministic and clean by construction. It happens to be minimal, too. In fact, the only non-final state $a_3\ a_5$ is distinguishable from the other three states, which are final. Concerning the final states: state $b_2\ b_4\ b_6\ \dashv$ does not have an outgoing arc a, while the other two states do, so it is distinguishable from them; while the two states $a_1\ b_2\ b_4\ \dashv$ and $b_2\ a_3\ b_4\ a_5\ \dashv$ go by arc a to states already known from the previous analysis to be distinguishable, thus they are distinguishable from each other as well. Therefore all the states of A_1 are distinguishable from one another, and the automaton is minimal.

(c) Language $L(R_1)$ is not local. Consider the equivalent automaton A_1: it is clean, thus all its states and arcs belong to some accepting path. Clearly the digram $a\,a$ is admissible, see the self-loop on the initial state. Yet one soon sees also that, after a letter b, there may be only one letter a, whereas the digram $a\,a$ allows an arbitrary number thereof. Thus language $L(A_1) = L(R_1)$ is not characterized by its local sets and it does not satisfy the definition of local language.

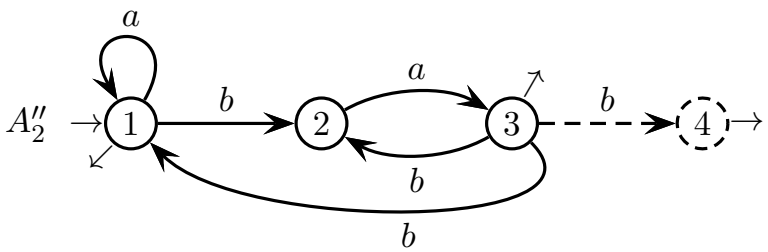
Another reasoning runs as follows. Clearly the local sets of language $L(A_1)$ are $Ini = Fin = \{\ a,\ b\ \} = \Sigma$ and $Dig = \{\ a\,a,\ a\,b,\ b\,a,\ b\,b\ \} = \Sigma^2$: just inspect automaton A_1 and find all of them. Thus, the local language characterized by such sets contains any string over the alphabet Σ, so it coincides with the universal language Σ^*. Yet, one sees at a glance that automaton A_1 does not recognize the universal language. Thus language $L(A_1) = L(R_1)$ is not local.

(d) There are two methods for cutting the spontaneous transitions: back-propagation and forward-propagation. One can try both and see their respective effect.

Back-propagation: cut the ε transition, add a transition $4 \xrightarrow{b} 2$, add a transition $4 \xrightarrow{a} 1$ and make node 4 final. Resulting automaton A'_2, still non-deterministic:

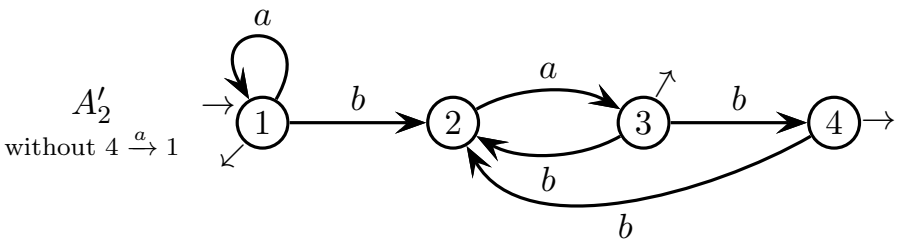


To cut the spontaneous transition, one could resort to forward-propagation as well: cut the ε transition and add a transition $3 \xrightarrow{b} 1$. Resulting automaton A''_2:



Automaton A''_2, though still non-deterministic, is even smaller than A'_2, and has only three states. In fact, state 4 (dashed) gets useless as it is unreachable from the initial state, and thus it can be eliminated.

(e) The relationship between the two languages is simple: language L_2 strictly contains language L_1. Automaton A'_2 without the added arc $4 \xrightarrow{a} 1$ (see point (d) with back-propagation) is clearly equivalent to expression R_1. Have a look at it:



One sees at a glance that it is equivalent to $a^* \ (\, b\, a \mid b\, a\, b\,)^* \equiv R_1$. Therefore it holds $L_1 \subseteq L_2$. The containment is strict, i.e., $L_1 \subset L_2$, since the now missing arc $4 \xrightarrow{a} 1$ allows automaton A'_2 to recognize more strings than expression R_1 can generate, e.g., $b\, a\, b\, a\, a$.

A similar reasoning applies to automaton A''_2, by assuming that the self-loop $1 \xrightarrow{a} 1$ is used only at the beginning and is not reused when going back from state 3 to state 1, and it leads to the same conclusion as before.

2 Free Grammars and Pushdown Automata 20%

1. Consider the following language L over the two-letter alphabet $\{ a, b \}$:

$$L = \{ a^m b^n \mid \quad m \geq n \geq 0 \quad \wedge \quad m - n \text{ is even} \}$$

Sample valid strings:

$$\varepsilon,\ a^2,\ a\,b,\ a^3\,b\ \in L$$

Sample invalid strings:

$$b,\ a^2\,b\ \notin L$$

Answer the following questions:

(a) Write an *ambiguous BNF* grammar G_1 that generates language L.

(b) Draw at least two syntax trees for string $a\ a\ a\ b$ according to grammar G_1.

(c) Write an *unambiguous BNF* grammar G_2 that generates language L.

(d) Draw the (unique) syntax tree for string $a\ a\ a\ b$ according to grammar G_2.

Solution

(a) The grammar G_1 below (axiom S) generates the language L (0 is assumed even):

$$G_1 \begin{cases} 1\colon \quad S \;\to\; a\ a\ S & \text{– exceeding even letters } a \\ 2\colon \quad S \;\to\; a\ S\ b & \text{– matching letters } a \text{ and } b \\ 3\colon \quad S \;\to\; \varepsilon & \text{– derivation end} \end{cases}$$

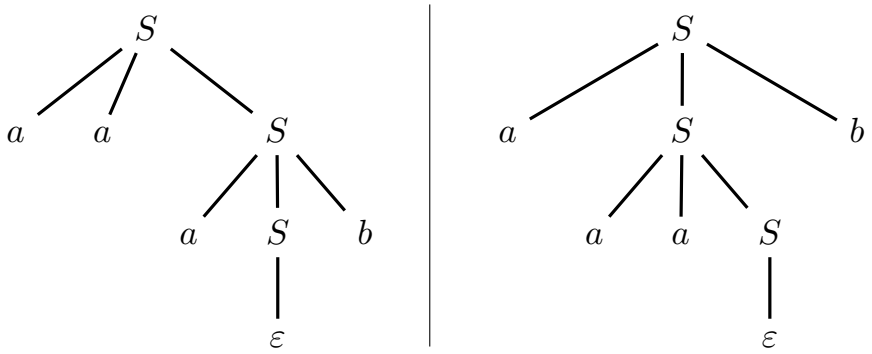
Grammar G_1 is ambiguous, as it does not impose a unique rule ordering to derive the letters (exceeding or matching) that compose the generated string.

(b) Here are the two (leftmost) derivations of the sample string $a\ a\ a\ b$, according to grammar G_1, which identify two syntax trees:

$$S \overset{1}{\Rightarrow} a\ a\ S \overset{2}{\Rightarrow} a\ a\ a\ S\ b \overset{3}{\Rightarrow} a\ a\ a\ b$$

$$S \overset{2}{\Rightarrow} a\ S\ b \overset{1}{\Rightarrow} a\ a\ a\ S\ b \overset{3}{\Rightarrow} a\ a\ a\ b$$

Here are the two syntax trees of string $a\ a\ a\ b$:



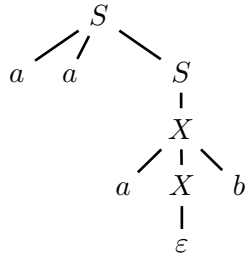
(c) The following grammar G_2 (axiom S) is equivalent to grammar G_1:

$$G_2 \begin{cases} 1\colon \quad S \;\to\; a\ a\ S & \text{– exceeding even letters } a \\ 2\colon \quad S \;\to\; X & \text{– transition to matching} \\ 3\colon \quad X \;\to\; a\ X\ b & \text{– matching letters } a \text{ and } b \\ 4\colon \quad X \;\to\; \varepsilon & \text{– derivation end} \end{cases}$$

Grammar G_2 is not ambiguous, as it imposes a rule ordering to the derivation: first the exceeding even letters a, then the matching letters a and b.

(d) Here is the one (leftmost) derivation of string $a\ a\ a\ b$, according to grammar G_2, which identifies the (unique) syntax tree (aside):

$$S \overset{1}{\Rightarrow} a\ a\ S \overset{2}{\Rightarrow} a\ a\ X \overset{3}{\Rightarrow} a\ a\ a\ X\ b \overset{4}{\Rightarrow} a\ a\ a\ b$$



2. Consider the fragment of a programming language that consists of assignments of two types of arithmetic expression, one simple and one complex, to variables. This fragmentary language is informally specified as follows:

   - Each phrase of the language is a non-empty list of assignments separated by semicolon "`;`".
   - An assignment consists of a variable name, an assignment operator and an arithmetic expression.
   - There are two assignment operators: "`=`" and "`:=`". The former wants a *simple* expression on the right, and the latter wants a *complex* one instead.
   - A *simple* arithmetic expression has at most two levels, without parentheses. It can use addition "`+`" and multiplication "`*`", of variables and constants. The associativity of addition and multiplication is unspecified. Precedence is uncommon: multiplication has lower priority and addition higher.
   - A *complex* arithmetic expression has arbitrarily many levels, with parenthesized subexpressions. It can use addition "`+`" and multiplication "`*`", of variables and constants. Addition is left-associative and multiplication is right-associative. Precedence is as usual: multiplication has higher priority and addition lower.
   - A variable identifier is alphanumerical (for simplicity use only lowercase letters), must have at least one heading letter and may freely contain underscores "`_`".
   - A constant is a decimal integer and may have a unary minus sign "`-`".

   For any unspecified minor detail, see the example below, and you may also follow the usual conventions, e.g., those of the C language. Example:

   ```
   a = b + c + 2 * d;

   e := f + g + h * 3;

   a1 = a + -5 * c * d + b;

   b34_c := a * (c + d) * -040;
   ```

   Answer the following questions:

   (a) Write a non-ambiguous grammar, in general of type *EBNF*, that models the described fragmentary language.

   (b) (optional) In order to (partially) test the behaviour of your grammar, separately draw the two syntax subtrees of the first two sample assignments. You may condense the trees by schematizing variables and constants with a terminal "`a`".

Solution

(a) Here is the grammar (axiom LANG), with a mix of *BNF* and *EBNF* rules:

⟨LANG⟩	→	(⟨ASGN⟩ ‘;’)$^+$	– non empty-list with ‘;’
⟨ASGN⟩	→	⟨S_ASG⟩ | ⟨C_ASG⟩	– assignment alternative
⟨S_ASG⟩	→	⟨VAR⟩ ‘=’ ⟨S_EXP⟩	– asgn with simple expr.
⟨C_ASG⟩	→	⟨VAR⟩ ‘:=’ ⟨C_EXP⟩	– asgn with complex expr.
⟨S_EXP⟩	→	⟨S_FCT⟩ (‘$*$’ ⟨S_FCT⟩)*	– unspec. assoc. / low pri.
⟨S_FCT⟩	→	⟨ATOM⟩ (‘+’ ⟨ATOM⟩)*	– unspec. assoc. / high pri.
⟨C_EXP⟩	→	⟨C_EXP⟩ ‘+’ ⟨C_TRM⟩ | ⟨C_TRM⟩	– left-assoc. / low pri.
⟨C_TRM⟩	→	⟨C_FCT⟩ ‘$*$’ ⟨C_TRM⟩ | ⟨C_FCT⟩	– right-assoc. / high pri.
⟨C_FCT⟩	→	‘(’ ⟨C_EXP⟩ ‘)’ | ⟨ATOM⟩	– with subexpressions
⟨ATOM⟩	→	⟨VAR⟩ | ⟨CONST⟩	– atom structure
⟨VAR⟩	→	⟨CHAR⟩ (⟨ALNUM⟩ | ‘_’)*	– variable structure
⟨CONST⟩	→	[‘−’] ⟨DIGIT⟩$^+$	– constant structure
⟨ALNUM⟩	→	⟨CHAR⟩ | ⟨DIGIT⟩	– alphanum. terminal
⟨CHAR⟩	→	[‘a’ – ‘z’]	– alphabetical terminal
⟨DIGIT⟩	→	[‘0’ – ‘9’]	– numerical terminal

The comments relate each rule to the specifications. The grammar is unambiguous, as it consists of known unambiguous substructures, e.g., list with separator, disjoint alternative, etc. Variable and constant might be refined according to some established conventions, e.g., those of the C language. However the grammar exactly models the specifications and samples without further details.

Here are a few considerations about how to model an arithmetic expression, and in particular about the associativity type of the operators, i.e., unspecified, left or right, and their relative precedence type, i.e., high or low priority:

- a regular list generated by Kleene star models unspecified associativity
- left and right recursion model left and right associativity, respectively
- hierarchical rule ordering models precedence, namely, a bottom rule has a higher priority than a top one, as an expression is evaluated bottom-up

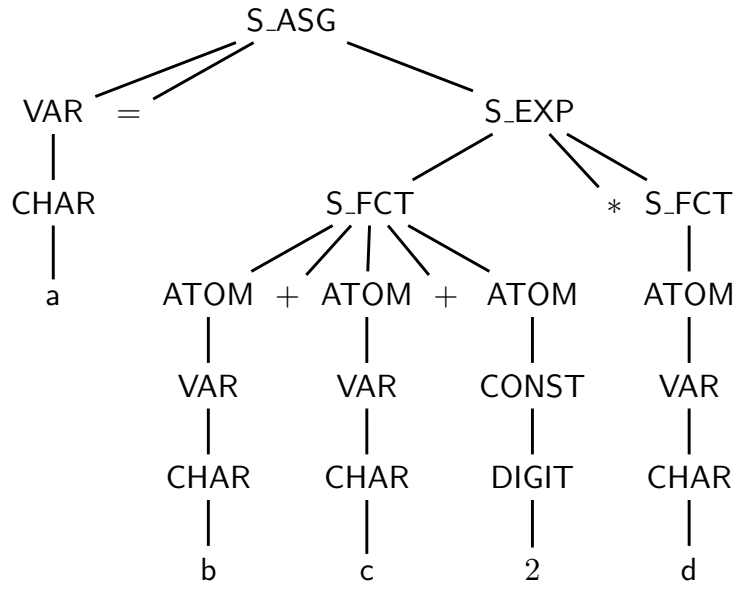
In the grammar above, simple and complex expressions are modeled accordingly. See the sample trees on the next page for more insight about the modeling of an arithmetic expression, to verify the correctness of the association order of the operators, their precedence, etc, on the two proposed assignment samples.

Of course, there may be equivalent grammars, more or less elegant. For instance, the three assignment rules above can be compacted into one as follows:

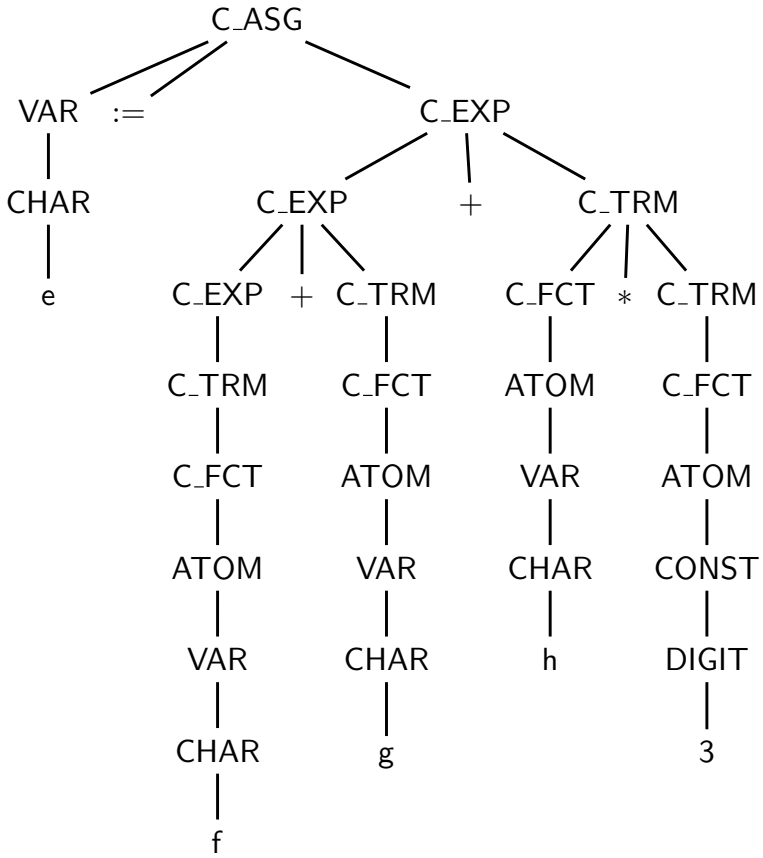
⟨ASGN⟩ → ⟨VAR⟩ (‘=’ ⟨S_EXP⟩ | ‘:=’ ⟨C_EXP⟩)

and the syntactic classes (nonterminals) S_ASG and C_ASG can be eliminated.

(b) Here are the (fully expanded) separate subtrees of the two sample assignments:



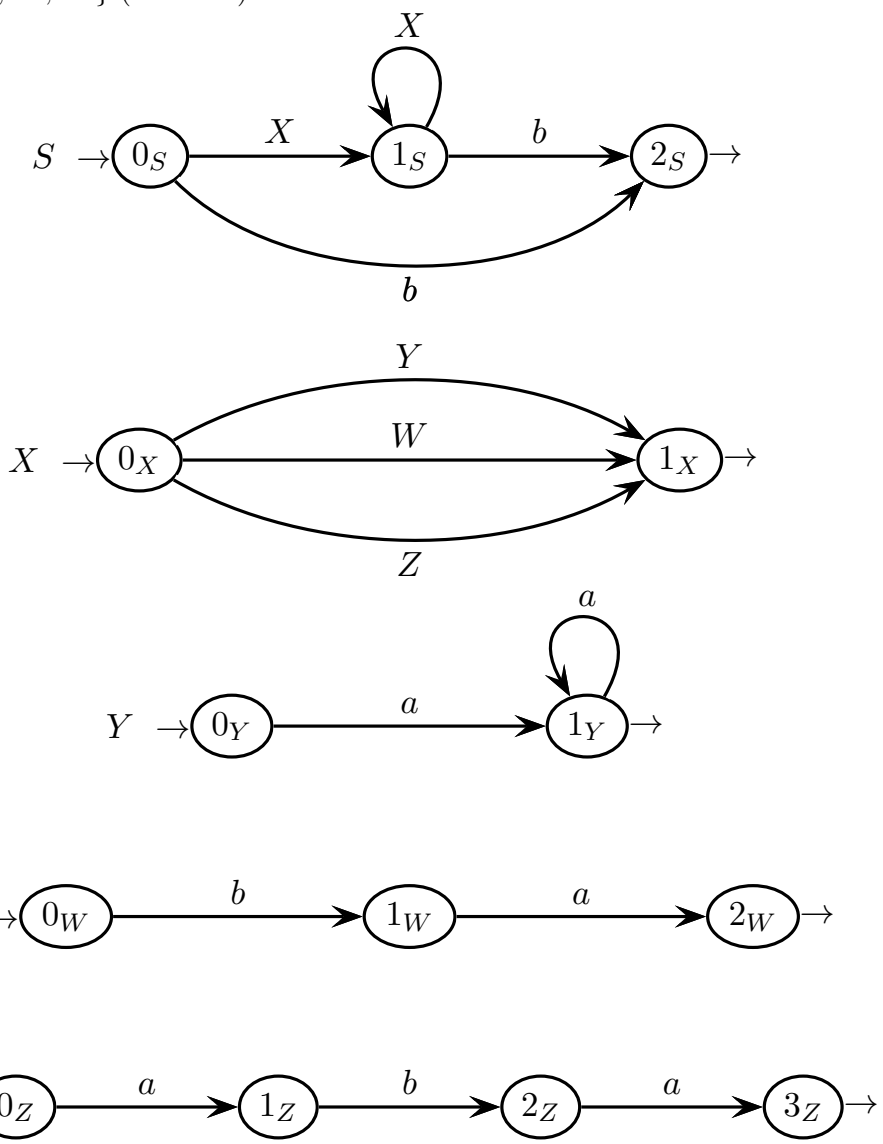
The arithmetic expression in the assignment `a = b + c + 2 * d` is evaluated bottom-up: addition evaluates before multiplication, and there is not any specific association order. To condense, assume ATOM = ' a ' and cut the branches.



The arithmetic expression in the assignment `e := f + g + h * 3` is evaluated bottom-up: addition associates on the left, multiplication associates on the right, and addition evaluates after multiplication. To condense, do as before.

3 Syntax Analysis and Parsing Methodologies 20%

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



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

(a) Draw a portion of the pilot graph sufficient to show that grammar G is not of type $ELR(1)$. Explain why grammar G is not of type $ELR(1)$.

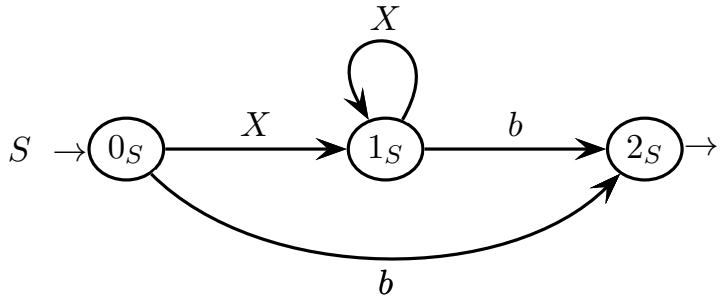
(b) Draw all the guide sets and, by using them, show that grammar G is not $ELL(1)$.

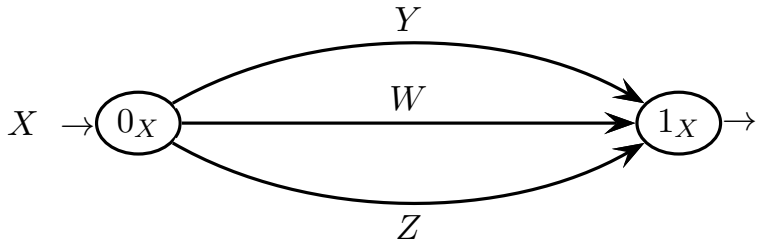
(c) By using the Earley method, analyze the string $a\,b\,a\,b$ according to grammar G and show, based on the results of the analysis, that the string is accepted.

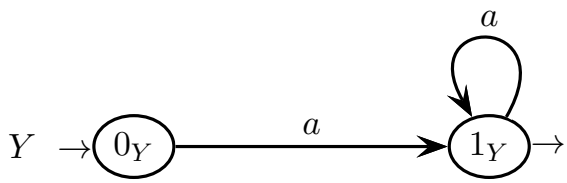
(d) (optional) Argue, as simply as possible, that the language $L(G)$ admits an $ELR(1)$ grammar and even an $ELL(1)$ grammar.

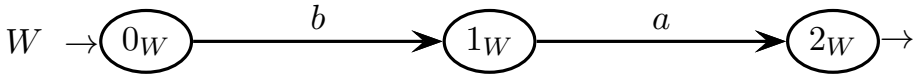
please draw here the (sufficient fragment of the) pilot graph – question (a)

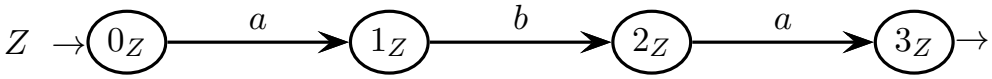
please draw here all the call arcs and write all the guide sets – question (b)



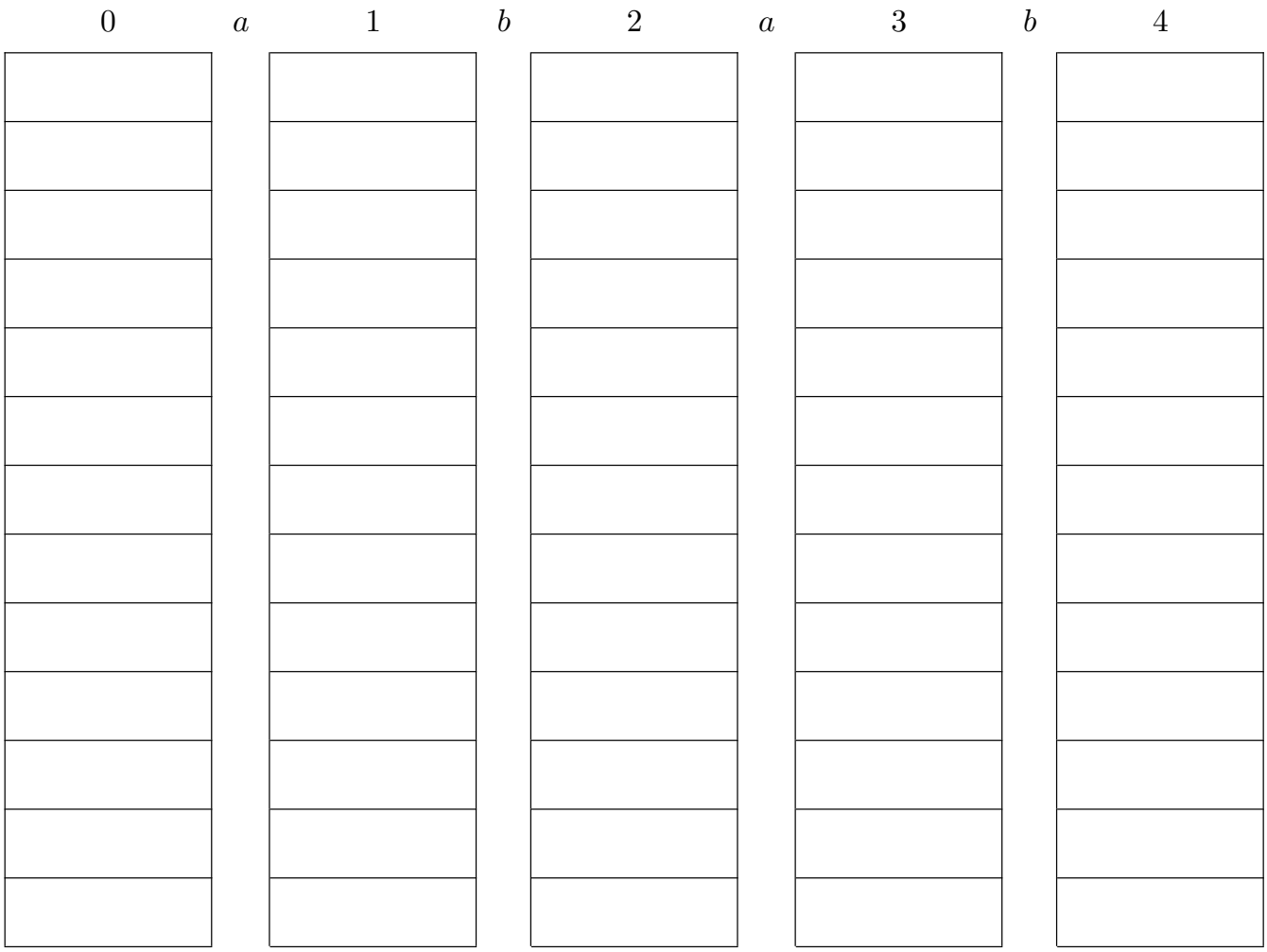








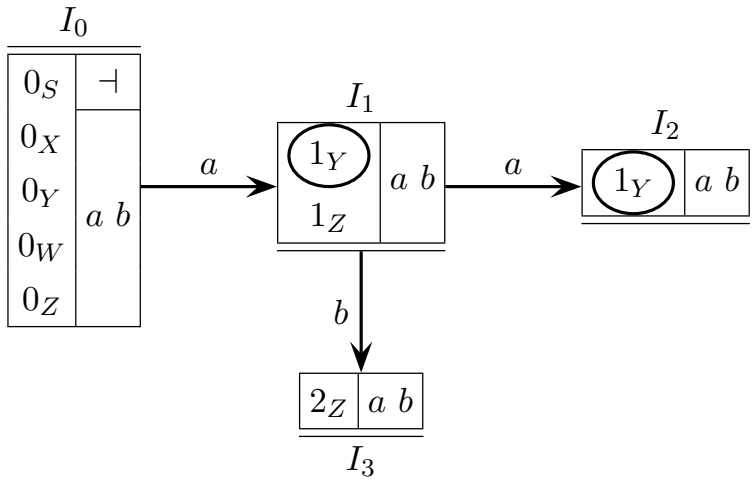
Earley vector of string $a\ b\ a\ b$ (to be filled in) – question (c)
(the number of rows is not significant)



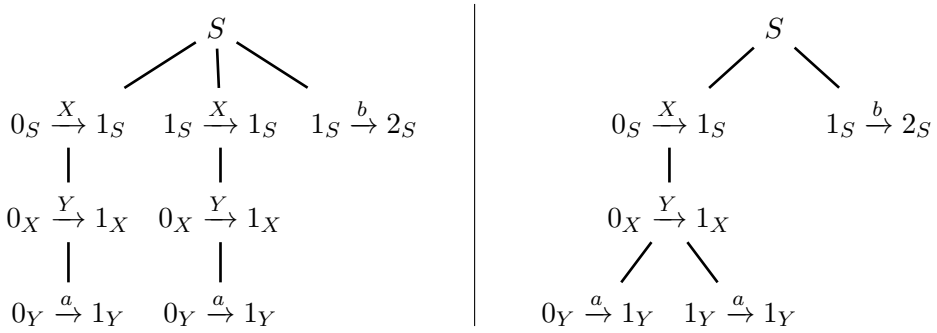
question (d)

Solution

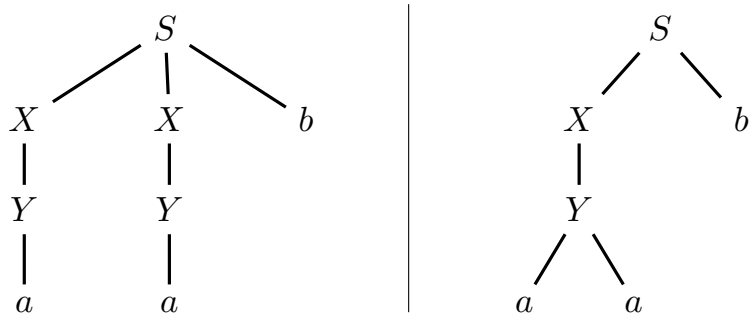
(a) Here is a pilot fragment sufficient to show that grammar G is not $ELR\,(1)$:



The complete pilot has many more m-states and transitions. However, the m-state I_1 has two shift-reduce conflicts, both on the final item $\langle\, 1_Y,\, \{\, a,\, b\,\}\,\rangle$ for terminals a and b. There may be more conflicts, involving other m-states.
For completeness, one might notice that grammar G is ambiguous. For instance, the valid string $a\ a\ b \in L\,(G)$ has these two syntax trees:

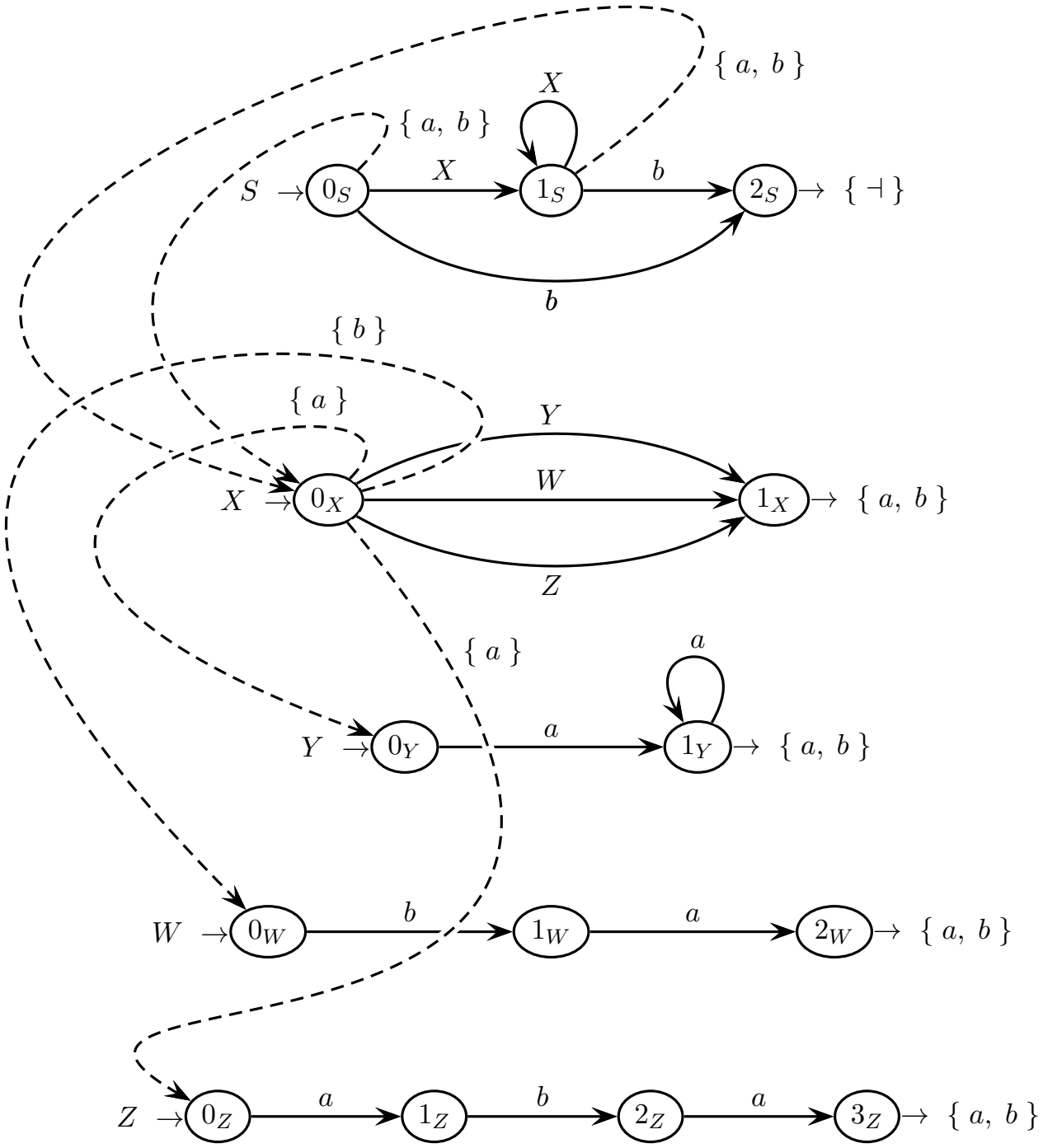


or by projecting onto the grammar symbols and canceling the net states:



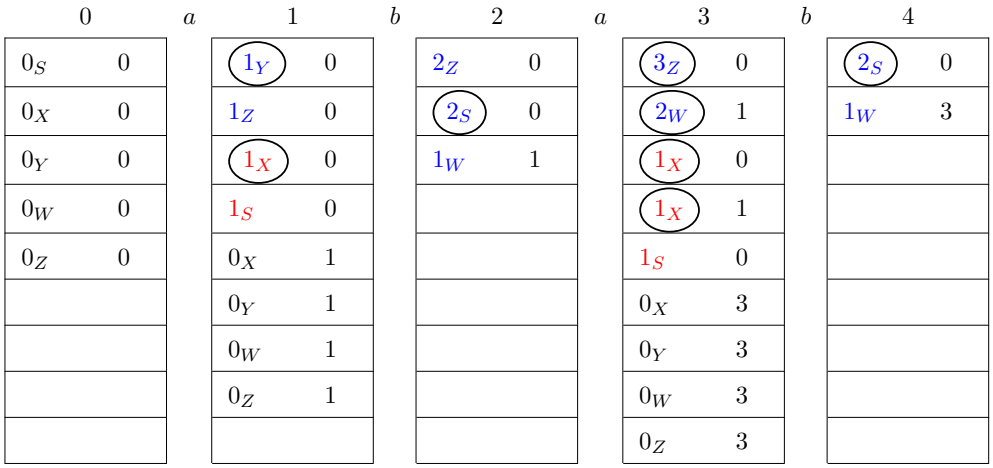
Thus grammar G is not deterministic. The valid string $a\ b\ a\ b$ is also ambiguous (see point (c)), as well as many others of increasing length. However, grammar G also generates many non-ambiguous strings, e.g., $a\ b$, $a\ b\ a\ b$, $a\ b\ a\ b\ a\ b$, etc.

(b) Here is the *PCFG* with all the guide sets (those on the terminal arcs are obvious):



There are guide set conflicts (overlaps) on states 0_S and 1_S, involving call and shift arcs for terminal b, on state 0_X, involving call arcs for terminal a, and on the final state 1_Y, involving the exit dart and the shift arc for terminal a. Thus the *PCFG* is not deterministic and therefore grammar G is not of type $ELL\,(1)$. Of course, this conclusion is coherent with grammar G being ambiguous, as already observed at point (a). See also point (c) for more on ambiguity.

(c) Here is the complete Earley vector of the valid string $a\ b\ a\ b \in L\,(G)$:

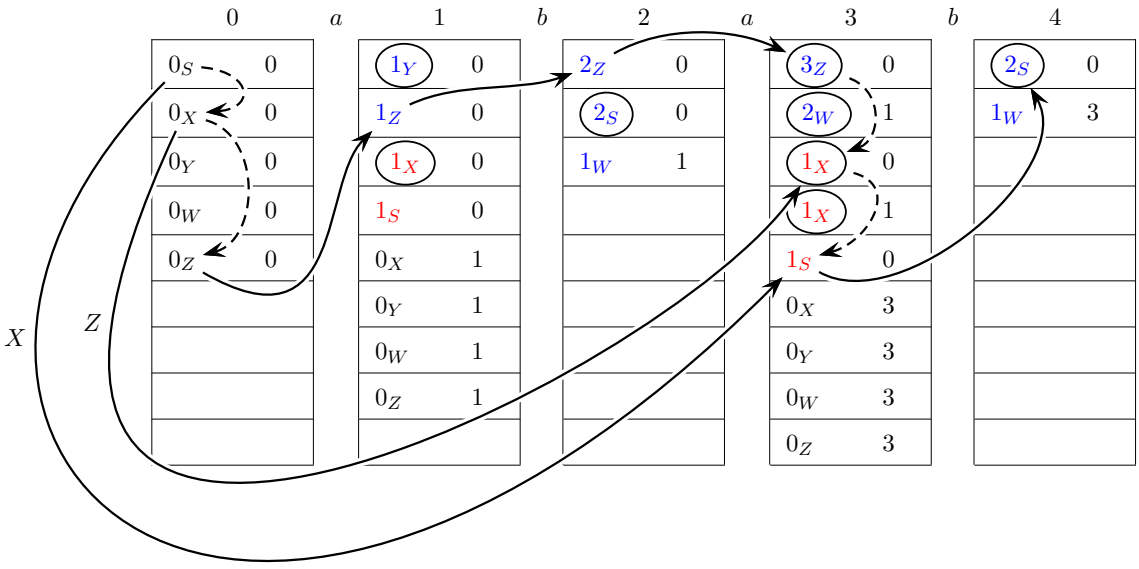


The final net states are encircled, terminal shift is coloured blue, nonterminal shift red, completion black and is clearly distinguishable as it has a net state 0.

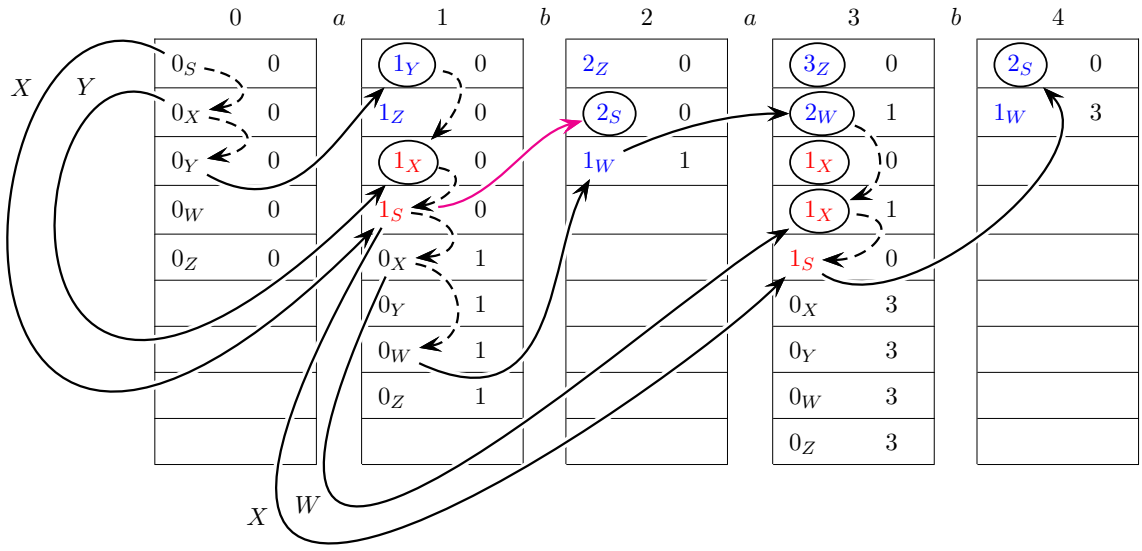
The last Earley state, numbered 4, contains the final item $\langle\, 2_S,\, 0\,\rangle$, namely an item with a final net state 2_S of the axiomatic machine S and pointer 0. Therefore string $a\ b\ a\ b$ is accepted. Of course, since the Earley state 2 contains the same item, the prefix string $a\ b$ is accepted as well, which fact is visibly correct (check it on the net by yourself). There are not any other accepted prefixes.

The reader may wish to make a tree reconstruction. String $a\ b\ a\ b$ is ambiguous and two trees can be reconstructed for it. Ambiguity stems from the item $\langle\, 1_S,\, 0\,\rangle$ in the Earley state 3 being shifted by starting from both items $\langle\, 0_S,\, 0\,\rangle$ and $\langle\, 1_S,\, 0\,\rangle$ in the Earley states 0 and 1, respectively. This happens because the Earley state 3 contains two final items $\langle\, 1_X,\, 0\,\rangle$ and $\langle\, 1_X,\, 1\,\rangle$, which both enable a different reduction and thus a different nonterminal shift to the item $\langle\, 1_S,\, 0\,\rangle$.

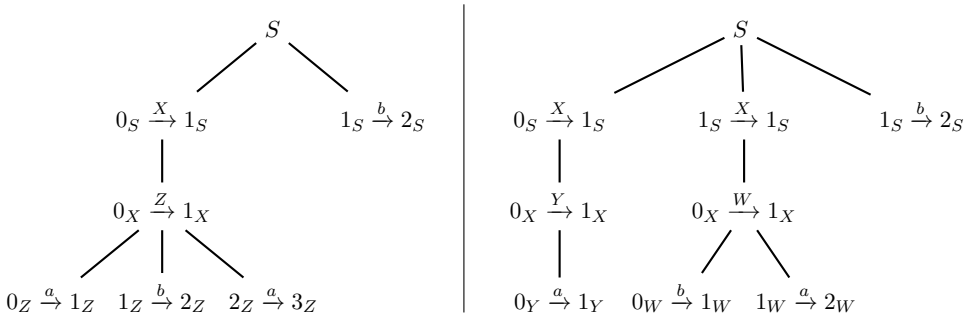
For completeness, here is a tree reconstruction, obtained by using the former final item $\langle\, 1_X,\, 0\,\rangle$ in the Earley state 3:



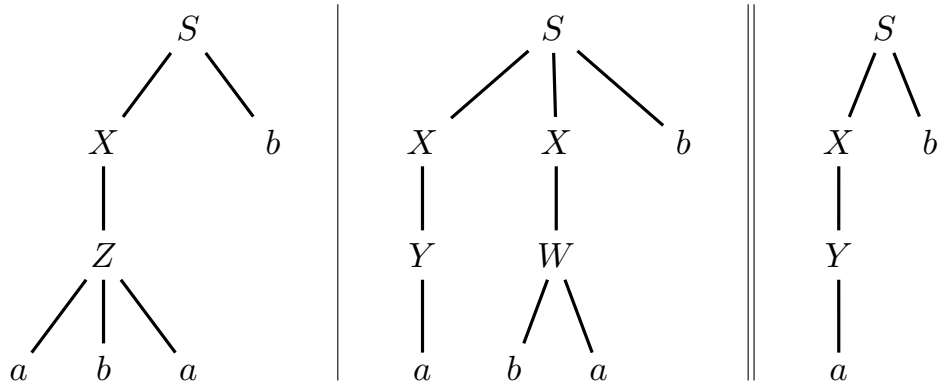
The other tree involves the latter final item $\langle\, 1_X,\, 1 \,\rangle$ in the Earley state 3:



Here are eventually the two syntax trees of the valid string $a\ b\ a\ b$:



and by projecting them onto the grammar symbols (leftmost two trees):



Instead, the accepted prefix $a\ b$ is not ambiguous and has only one syntax tree. The related projected tree is shown above (rightmost tree). Such a tree can be also traced on the Earley vector (second reconstruction), just by considering the terminal shift move $1_S \xrightarrow{b} 2_S$ to the final item $\langle\, 2_S,\, 0 \,\rangle$ in the Early state 2 (in the vector the corresponding terminal shift arc is shown coloured magenta).
The reader may wish to analyze by the Earley method also the ambiguous string $a\ a\ b$, already mentioned at point (a), and reconstruct its two syntax trees.

(d) The language $L(G)$ is regular and is generated by this regular expression R:

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

Expression R can be easily obtained by substitution, just by observing that the machine net is not recursive. Here are the substitution steps. By examining the net, it is immediate to obtain intuitively the following expressions for the regular languages recognized by each machine:

$$L(S) = X^*\,b \qquad L(X) = Y \mid W \mid Z$$

$$L(Y) = a^+ \qquad L(W) = b\,a \qquad L(Z) = a\,b\,a$$

Thus, by substitution:

$$L(X) = a^+ \mid b\,a \mid a\,b\,a$$

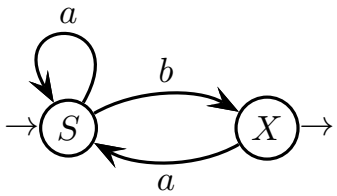
Therefore, again by substitution:

$$L(S) = \big(a^+ \mid b\,a \mid a\,b\,a \big)^*\; b = R \qquad \text{with } R = \Big(\overbrace{\underbrace{a^+}_{Y} \mid \underbrace{b\,a}_{W} \mid \underbrace{a\,b\,a}_{Z}}^{X} \Big)^*\; b$$

In conclusion, the language has a right-unilinear grammar of type $ELL(1)$, which can be obtained from any deterministic finite-state automaton equivalent to R, for instance, the minimal one. Of course, such a grammar is also of type $ELR(1)$. Just for completeness, an automaton and an $ELL(1)$ grammar equivalent to expression R can be quickly found. First, simplify expression R as follows:

$$\begin{array}{rcll}
R & = & \big(a^+ \mid b\,a \mid a\,b\,a \big)^*\; b & \text{– factorize } b\,a \text{ on the right} \\
 & = & \big(a^* \mid [a]\; b\,a \big)^*\; b & \text{– } (\alpha \mid \beta)^* = \alpha^*\;(\beta\,\alpha^*)^* \\
 & = & (a^*)^*\; \big([a]\; b\,a\; (a^*)^* \big)^*\; b & \text{– } (a^*)^* = a^* \text{ and } a\,a^* = a^+ \\
 & = & a^*\; \big([a]\; b\,a^+ \big)^*\; b & \text{– obvious simplification} \\
 & = & a^*\; \big(b\,a^+ \big)^*\; b & \text{– obvious commutation} \\
 & = & a^*\; b\; \big(a^+\; b \big)^* = R' & \text{– final result}
\end{array}$$

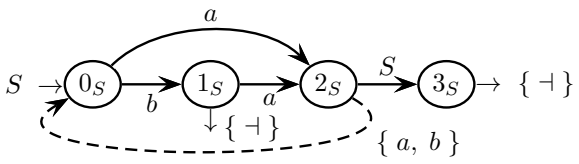
The simplified expression R' is unambiguous. A deterministic automaton intuitively derived from expression R' is as follows (and is evidently minimal):



$$\begin{cases} S \;\to\; a\,S \mid b\,X \\ X \;\to\; a\,S \mid \varepsilon \end{cases}$$

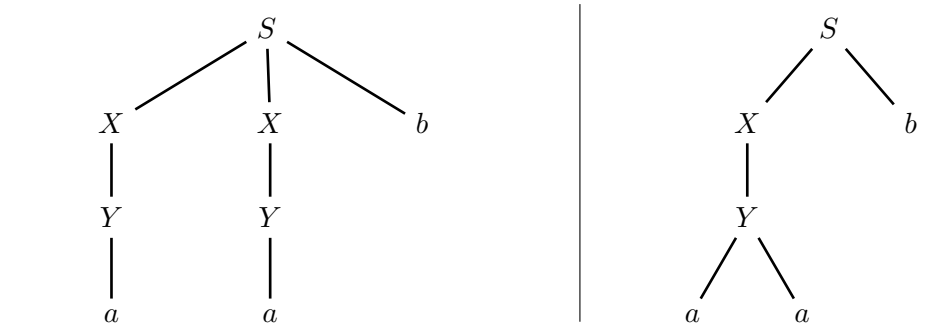
with the equivalent right-unilinear grammar (axiom S), which is *BNF* and of type $LL(1)$. By substitution, one can even obtain the simpler grammar below:

$$S \to [\,b\,]\; a\; S \mid b$$

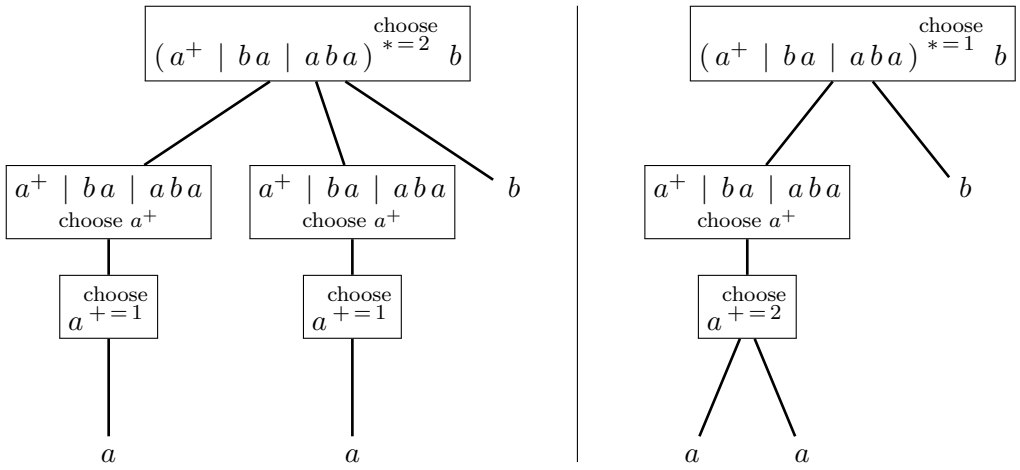


which is *EBNF* (it uses optionality) and of type $ELL(1)$ (see aside). One can find an automaton also by processing expression R differently, e.g., by Berry-Sethi.

A final observation: the regular expression R above is ambiguous, as well as the grammar it derives from. The reader may see that both the valid strings $a\ a\ b$ and $a\ b\ a\ b$ examined at points (a) and (c) are ambiguously generated by R. For instance, go back to the former one, the two trees of which are (see point (a)):



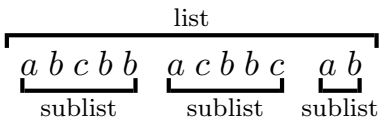
Then, the two corresponding derivation trees of the regular expression R are:



Of course, there may be other regular expressions, equivalent to expression R yet not modeled according to the structure of grammar G, and possibly unambiguous. The simplified expression R' above is just an example thereof, and for instance it could be rewritten as $[\,b\,]\ (\,a^{+}\ b\,)^{+}\ \mid\ b$.

4 Language Translation and Semantic Analysis 20%

1. Consider a source language L_s of two-level nested lists. The alphabet is $\Sigma = \{\ a,\ b,\ c\ \}$. A letter a heads each sublist, and the letters b and c are the elements of the sublists. A sublist may be empty, as well as the whole list. Example of a nested list:



A source grammar G_s that generates such a language L_s is the following (axiom S):

$$G_s \begin{cases} S \rightarrow a\ X\ S \\ S \rightarrow \varepsilon \\ X \rightarrow b\ X \\ X \rightarrow c\ X \\ X \rightarrow \varepsilon \end{cases}$$

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

(a) Consider the translation τ_1 that maps each source list of L_s to a translated one as follows: the letter a of each sublist is moved to the trailing position of the sublist, and inside each sublist all the letters b precede all the letters c. Example:

$$\tau_1\ (\ a\ b\ c\ b\ b\ \ a\ c\ b\ b\ c\ \ a\ b\ ) = b\ b\ b\ c\ a\ \ b\ b\ c\ c\ a\ \ b\ a$$

Write a translation scheme (or grammar) with source grammar G_s that computes translation τ_1. Do not change the structure of the source grammar G_s.

(b) Consider the translation τ_2 that maps each source list of L_s to a translated one as follows: first in each sublist the numbers of letters b and c are exchanged, and then from left to right the sublist includes all the letters b, the letter a and all the letters c. Example:

$$\tau_2\ (\ a\ b\ c\ b\ b\ \ a\ c\ b\ b\ c\ \ a\ b\ ) = b\ a\ c\ c\ c\ \ b\ b\ a\ c\ c\ \ a\ c$$

Write a translation scheme (or grammar) with source grammar G_s that computes translation τ_2. Do not change the structure of the source grammar G_s.

(c) Are translations τ_1 and τ_2 invertible ? Please explain your answer.

(d) (optional) Consider also the destination language of translation τ_1, i.e., the language $L_{d_1} = \tau_1\ (\ L_s\ )$, and assume that translation τ_2 has language L_{d_1} as its source language, instead of language L_s. Example:

$$\tau_2\ (\ b\ b\ b\ c\ a\ \ b\ b\ c\ c\ a\ \ b\ a\ ) = b\ a\ c\ c\ c\ \ b\ b\ a\ c\ c\ \ a\ c$$

Is translation τ_2, modified as said above, invertible? Please explain your answer.

---

write on the right the dest. gram. or the whole transl. gram. for τ_1 – question (a)

$$
\begin{cases}
S \to a\ X\ S \\
S \to \varepsilon \\
X \to b\ X \\
X \to c\ X \\
X \to \varepsilon
\end{cases}
$$

write on the right the dest. gram. or the whole transl. gram. for τ_2 – question (b)

$$
\begin{cases}
S \to a\ X\ S \\
S \to \varepsilon \\
X \to b\ X \\
X \to c\ X \\
X \to \varepsilon
\end{cases}
$$

rest of the exercise – questions (c) and (d)

Solution

(a) Scheme G_{τ_1} of translation τ_1 (axiom S) and aside the grammar version:

$$G_{\tau_1}\left\{\begin{array}{lll} S \to a\ X\ S & S \to X\ a\ S & S \to \frac{a}{\varepsilon}\ X\ \frac{\varepsilon}{a}\ S \\ S \to \varepsilon & S \to \varepsilon & S \to \frac{\varepsilon}{\varepsilon} \\ X \to b\ X & X \to b\ X & X \to \frac{b}{b}\ X \\ X \to c\ X & X \to X\ c & X \to \frac{c}{\varepsilon}\ X\ \frac{\varepsilon}{c} \\ X \to \varepsilon & X \to \varepsilon & X \to \frac{\varepsilon}{\varepsilon} \end{array}\right.$$

Each a is moved from heading to trailing and all c's are moved to the right.

(b) Scheme G_{τ_2} of translation τ_2 (axiom S) and aside the grammar version:

$$G_{\tau_2}\left\{\begin{array}{lll} S \to a\ X\ S & S \to X\ S & S \to \frac{a}{\varepsilon}\ X\ S \\ S \to \varepsilon & S \to \varepsilon & S \to \frac{\varepsilon}{\varepsilon} \\ X \to b\ X & X \to X\ c & X \to \frac{b}{\varepsilon}\ X\ \frac{\varepsilon}{c} \\ X \to c\ X & X \to b\ X & X \to \frac{c}{b}\ X \\ X \to \varepsilon & X \to a & X \to \frac{\varepsilon}{a} \end{array}\right.$$

Each letter a is moved from heading to infix, all letters b and c are exchanged and all letters c are placed on the right.

(c) Neither translation τ_1 nor translation τ_2 is invertible, i.e., one-to-one, because an original sublist of letters b and c alphabetically unordered cannot be uniquely reconstructed after alphabetically ordering (b before c). For instance:

$$\tau_1(a\,b\,b\,c) = \tau_1(a\,b\,c\,b) = b\,b\,c\,a \qquad \tau_2(a\,b\,b\,c) = \tau_2(a\,b\,c\,b) = b\,a\,c\,c$$

thus neither translation is injective, hence neither of them is invertible.

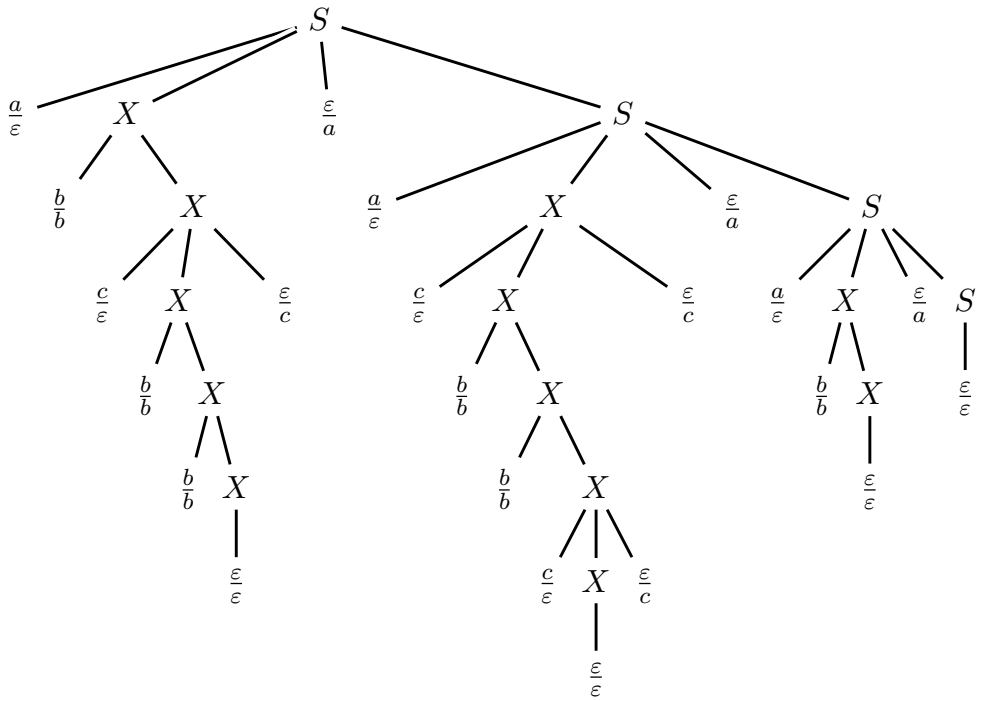
(d) The modified translation τ_2 is invertible. In fact, in the language L_{d_1} each sublist is totally ordered, thus now the source string can be uniquely reconstructed. More in detail, in the translated string each digram $c\,b$ – or $c\,a$ if the next sublist does not contain any letter b or c $\dashv$ if furthermore it is the last one – indicates where the letter a of a sublist has to be moved to, and after doing so, each sublist of L_{d_1} can be uniquely reconstructed first by exchanging the names of letters b and c, and then by moving all letters b ahead of all letters c.

For completeness, here is an inversion example of the modified translation τ_2, applied to the sample string from left to right (the letters changed at each step are in red):

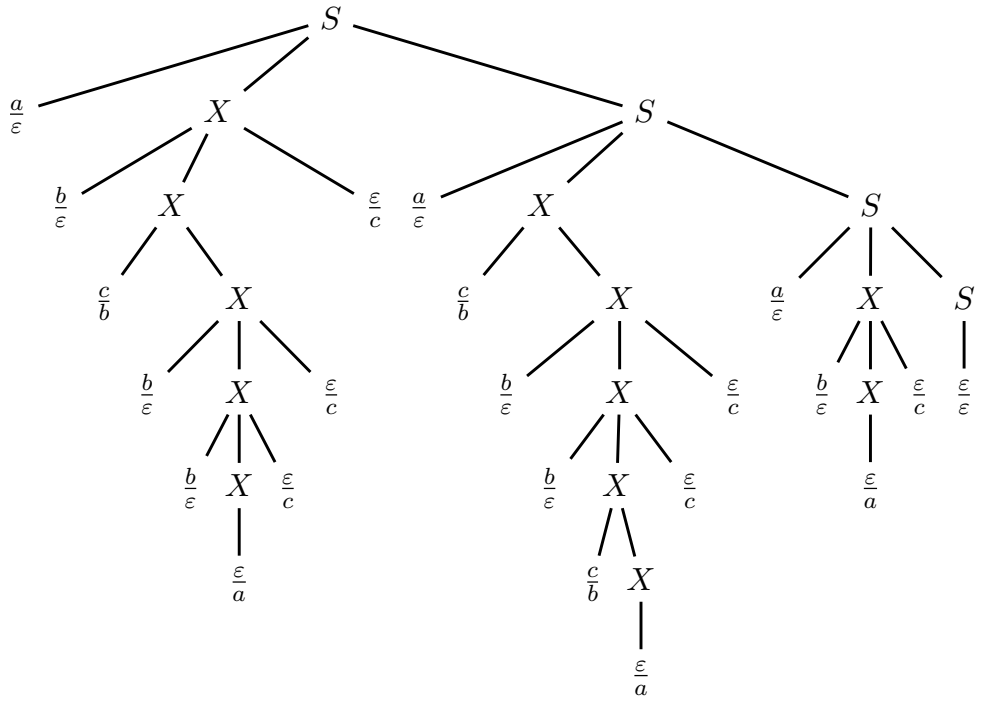
$b\,a\,c\,c\,c\ \ b\,b\,a\,c\,c\ \ a\,c\ \dashv$	– initial string $\in L_{d_2}$
$b\,c\,c\,c\,a\ \ b\,b\,a\,c\,c\ \ a\,c\ \dashv$	– letter a moved into next digram $c\,b$
$c\,b\,b\,b\,a\ \ b\,b\,a\,c\,c\ \ a\,c\ \dashv$	– letter names b and c exchanged
$b\,b\,b\,c\,a\ \ b\,b\,a\,c\,c\ \ a\,c\ \dashv$	– letters b moved ahead of letters c
$b\,b\,b\,c\,a\ \ b\,b\,c\,c\,a\ \ a\,c\ \dashv$	– letter a moved into next digram $c\,a$
$b\,b\,b\,c\,a\ \ c\,c\,b\,b\,a\ \ a\,c\ \dashv$	– letter names b and c exchanged
$b\,b\,b\,c\,a\ \ b\,b\,c\,c\,a\ \ a\,c\ \dashv$	– letters b moved ahead of letters c
$b\,b\,b\,c\,a\ \ b\,b\,c\,c\,a\ \ c\,a\ \dashv$	– letter a moved into next digram $c\,\dashv$
$b\,b\,b\,c\,a\ \ b\,b\,c\,c\,a\ \ b\,a\ \dashv$	– letter names b and c exchanged
$b\,b\,b\,c\,a\ \ b\,b\,c\,c\,a\ \ b\,a\ \dashv$	– letters b moved ahead of letters c (no change)
$b\,b\,b\,c\,a\ \ b\,b\,c\,c\,a\ \ b\,a\ \dashv$	– final string $\in L_{d_1}$

For completeness, here are the two sample syntax trees of translations τ_1 and τ_2:

$$\tau_1\ (\,a\ b\ c\ b\ b\ \ a\ c\ b\ b\ c\ \ a\ b\,) = b\ b\ b\ c\ a\ \ b\ b\ c\ c\ a\ \ b\ a$$



$$\tau_2\ (\,a\ b\ c\ b\ b\ \ a\ c\ b\ b\ c\ \ a\ b\,) = b\ a\ c\ c\ c\ \ b\ b\ a\ c\ c\ \ a\ c$$



2. Consider the following abstract syntax (axiom S), over the two-letter alphabet $\{ a, b \}$:

$$\begin{cases} 1\colon S \rightarrow a\ S\ b\ S \\ 2\colon S \rightarrow a\ S \\ 3\colon S \rightarrow S\ b \\ 4\colon S \rightarrow \varepsilon \end{cases}$$

A sample valid string is: $a^3\ b^2$ (see also the syntax tree on the next pages).

Assume that a syntax tree is given for each string. We want to compute, in the root node of any tree, the following values:

- the depth (distance from tree root) of the *deepest* leaf node labeled a
- the depth (distance from tree root) of the *least deep* leaf node labeled b

It is assumed that, if the tree does not include any leaf node of a given kind, then the required depth value is conventionally set equal to 0.

Attributes permitted to use (no other attributes are allowed):

name	*type*	*domain*	*symbol*	*meaning*
da	left	integer	S	distance, from the current node S, of the *deepest* leaf node of type a in the subtree rooted at the current node conventionally set equal to 0 if the subtree does not include any leaf node of type a
db	left	integer	S	distance, from the current node S, of the *least deep* leaf node of type b in the subtree rooted at the current node conventionally set equal to 0 if the subtree does not include any leaf node of type b

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

(a) Write an attribute grammar G_a, based on the above syntax and using only the attribute *da*, that computes in the tree root the value of the depth of the *deepest* leaf node labeled a, or 0 if the tree does not include any node labeled a.

(b) Decorate the tree of string $a^3\ b^2$, prepared on the next page, with the values of attribute *da* at every non-leaf node.

(c) Write another attribute grammar G_b, so as to compute in the tree root the value of the depth of the *least deep* leaf node labeled b, or 0 if the tree does not include any node labeled b, based on the above syntax and using only the attribute *db*.

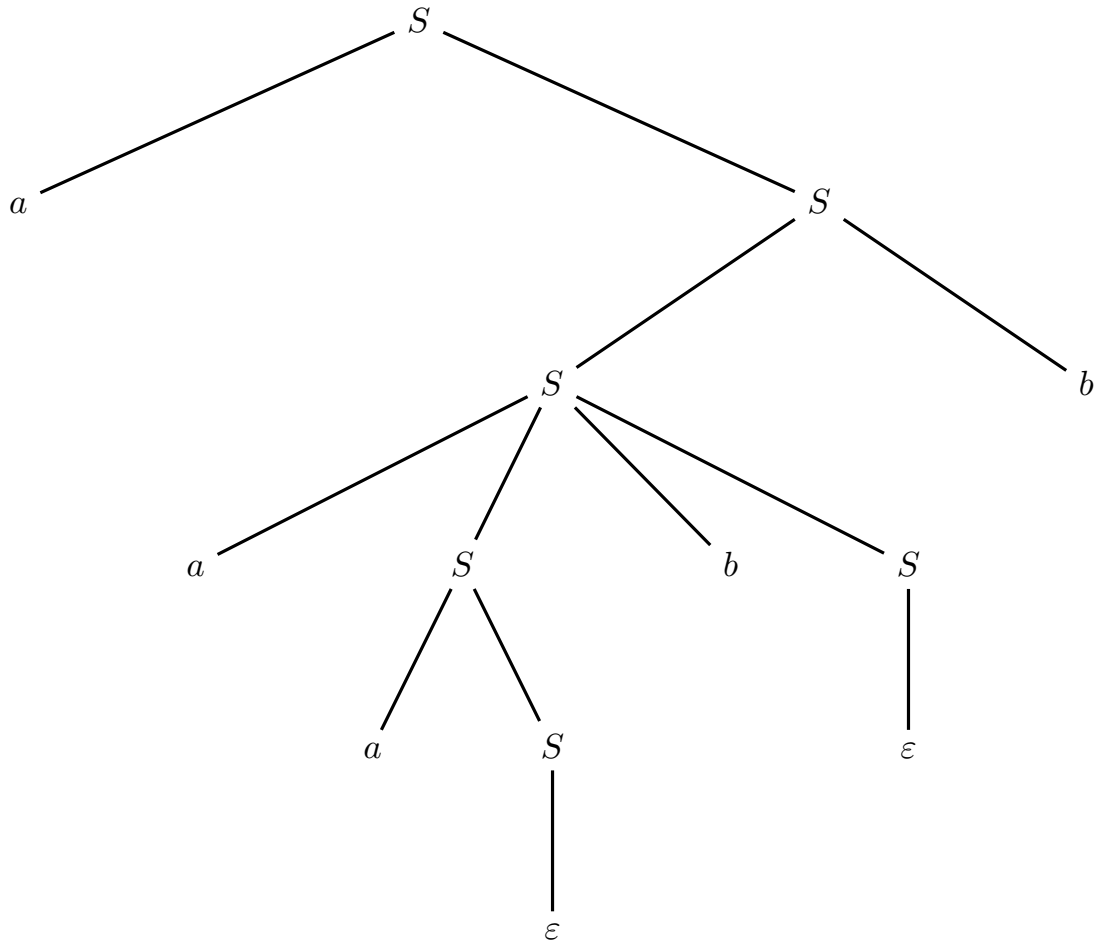
(d) Decorate the tree of string $a^3\ b^2$, prepared on the next page, with the values of attribute *db* at every non-leaf node.

for exercise 4.2 see the next pages

here space for continuation if necessary – all exercises

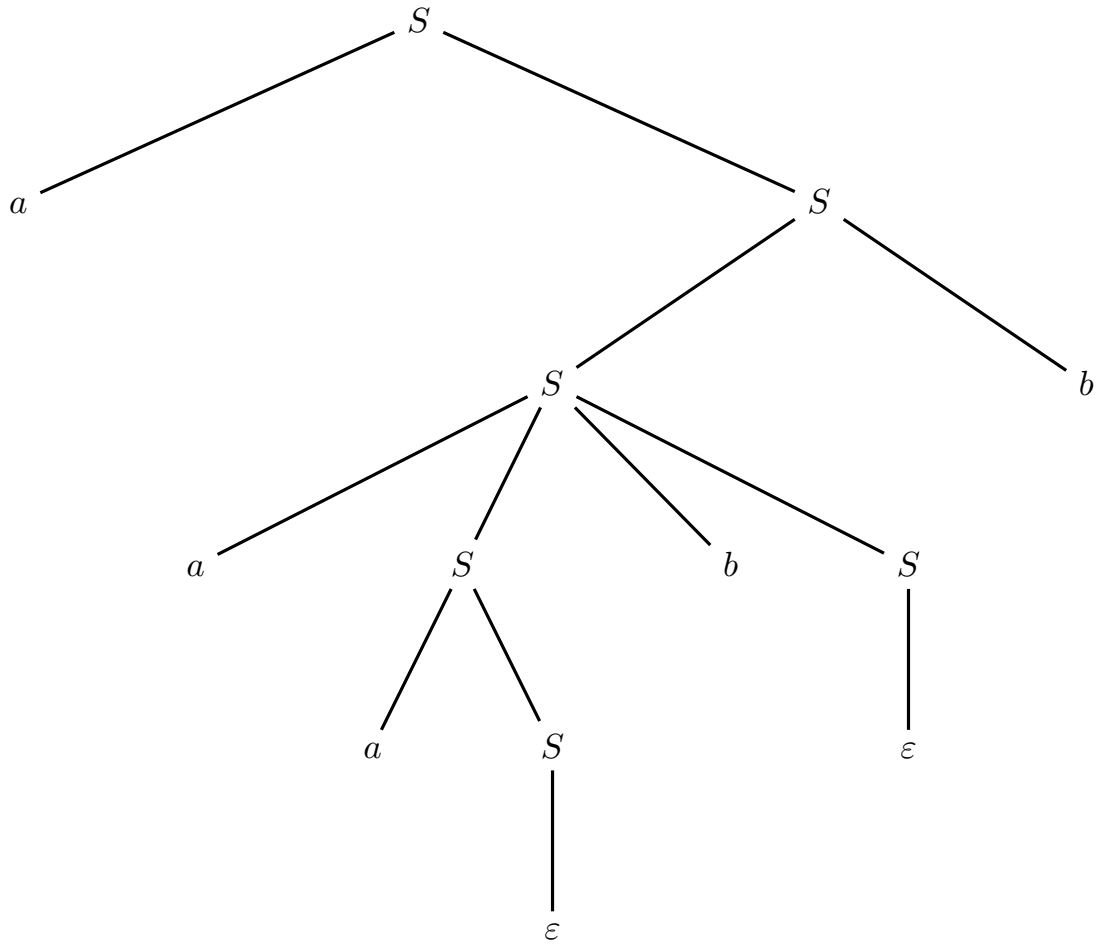
#	*syntax*	*semantics of* G_a – question (a)
1:	$S_0 \rightarrow a\ S_1\ b\ S_2$	
2:	$S_0 \rightarrow a\ S_1$	
3:	$S_0 \rightarrow S_1\ b$	
4:	$S_0 \rightarrow \varepsilon$	

please decorate this tree with attribute da – question (b)



#	*syntax*	*semantics of* G_b – question (c)
1:	$S_0 \rightarrow a\ S_1\ b\ S_2$	
2:	$S_0 \rightarrow a\ S_1$	
3:	$S_0 \rightarrow S_1\ b$	
4:	$S_0 \rightarrow \varepsilon$	

please decorate this tree with attribute db – question (d)



space for continuation if necessary – all exercises

Solution

(a) Here is the attribute grammar G_a, which uses only the attribute da:

#	*syntax*	*semantics of* G_a – question (a)

1:	$S_0 \rightarrow a\ S_1\ b\ S_2$	**if** ($da_1 = 0$) **and** ($da_2 = 0$) **then**
		  $da_0 := 1$
		else if ($da_1 \geq da_2$) **then**
		  $da_0 := da_1 + 1$
		else
		  $da_0 := da_2 + 1$
		endif

2:	$S_0 \rightarrow a\ S_1$	**if** ($da_1 = 0$) **then**
		  $da_0 := 1$
		else
		  $da_0 := da_1 + 1$
		endif

3:	$S_0 \rightarrow S_1\ b$	**if** ($da_1 = 0$) **then**
		  $da_0 := 0$
		else
		  $da_0 := da_1 + 1$
		endif

4:	$S_0 \rightarrow \varepsilon$	$da_0 := 0$

The attribute grammar G_a is purely synthesized. The semantic functions are coded extensively so as to be self-evident according to the specifications.
However, some coding optimization is possible. For instance, at rule 2 cancel the two-way conditional and set $da_0 = da_1 + 1$ instead, and similarly at rule 1 cancel the three-way conditional and equivalently set $da_0 = \max(\, da_1,\, da_2\,) + 1$. Here is the optimized attribute grammar G_a:

#	*syntax*	*semantics of* G_a – optimized
1:	$S_0 \rightarrow a\ S_1\ b\ S_2$	$da_0 := \max(\, da_1,\, da_2\,) + 1$
2:	$S_0 \rightarrow a\ S_1$	$da_0 := da_1 + 1$

3:	$S_0 \rightarrow S_1\ b$	**if** ($da_1 = 0$) **then**
		  $da_0 := 0$
		else
		  $da_0 := da_1 + 1$
		endif

4:	$S_0 \rightarrow \varepsilon$	$da_0 := 0$

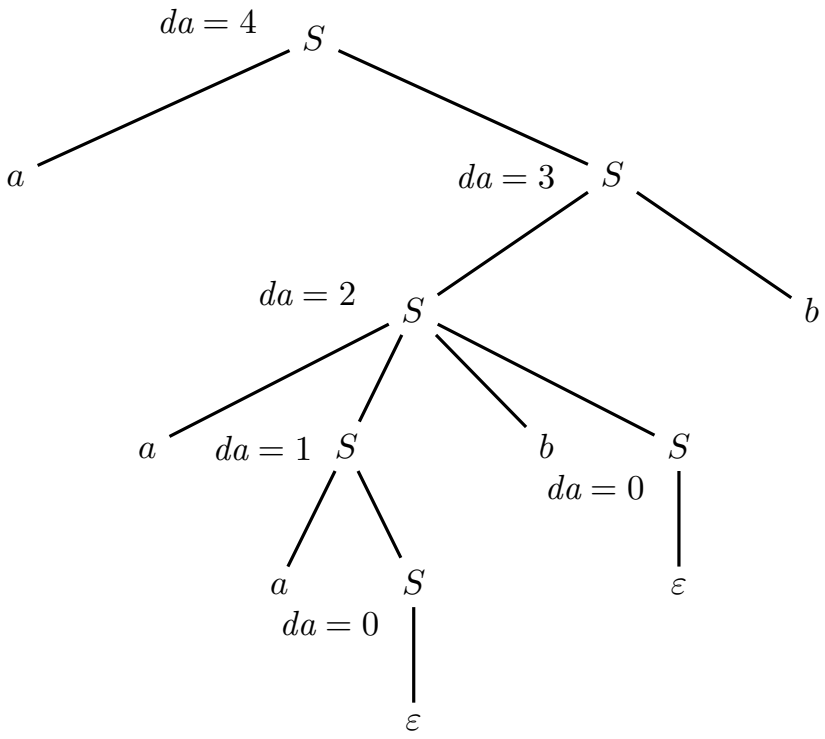
For a compact coding, one can resort to the conditional assignment of the C

language, to recode the two-way conditional in the semantic function of rule 3:

#	*syntax*	*semantics of* G_a – recoded
1:	$S_0 \rightarrow a\ S_1\ b\ S_2$	$da_0 := \max(\ da_1,\ da_2\ ) + 1$
2:	$S_0 \rightarrow a\ S_1$	$da_0 := da_1 + 1$
3:	$S_0 \rightarrow S_1\ b$	$da_0 := (\ da_1 = 0\ )\ ?\ 0\ :\ da_1 + 1$
4:	$S_0 \rightarrow \varepsilon$	$da_0 := 0$

There may be other coding ways or optimizations.

(b) Here is the syntax tree decorated with attribute da:



The computation on the tree is coherent with the attribute grammar G_a, and it suffices to prove the correctness of the grammar.

(c) Here is the attribute grammar G_b, which uses only the attribute db:

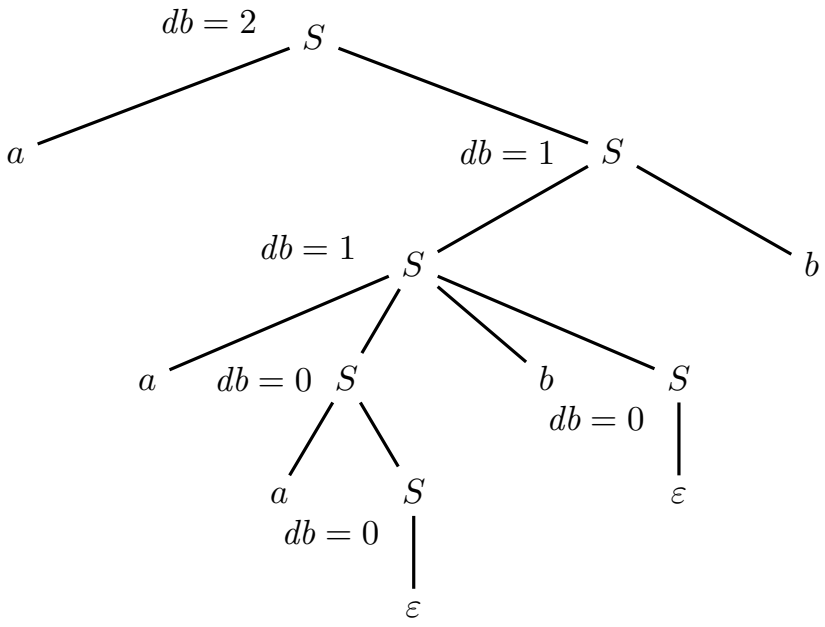
#	*syntax*	*semantics of* G_b – question (c)
1:	$S_0 \rightarrow a\ S_1\ b\ S_2$	$db_0 := 1$

2a:	$S_0 \rightarrow a\ S_1$	**if** ($db_1 = 0$) **then**
		  $db_0 := 0$
		else
		  $db_0 := db_1 + 1$
		endif

2b:	equivalently to 2a	$db_0 := (\ db_1 = 0\ )\ ?\ 0\ :\ db_1 + 1$
3:	$S_0 \rightarrow S_1\ b$	$db_0 := 1$
4:	$S_0 \rightarrow \varepsilon$	$db_0 := 0$

The attribute grammar G_b is purely synthesized. The semantic resemblance of the rules 2 (version a or b) in G_b and 3 in G_a (all versions) is evident. There may be optimizations or a more compact coding, though it seems unlikely due to the extreme simplicity and compactness of the current solution.

(d) Here is the syntax tree decorated with attribute db:



The computation on the tree is coherent with the attribute grammar G_b, and it suffices to prove the correctness of the grammar.

For completeness, notice that both attribute grammars G_a and G_b are of type one-sweep, as they are purely synthesized. The syntactic support is however ambiguous, e.g., string $a\ b$ has three syntax trees. Therefore the computation of the attributes of both G_a and G_b cannot be integrated with syntax analysis.