FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 19 July 2017 - 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%

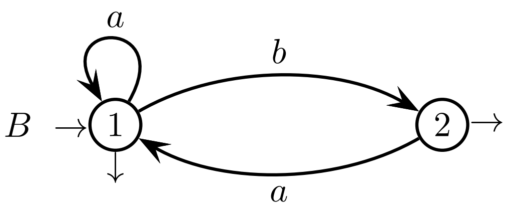
1. Consider the two-letter alphabet $\Sigma = \{ \, a, \; b \, \}$ and answer the following questions:

   (a) By using the Berry-Sethi method, design a deterministic finite-state automaton A that accepts the language generated by the regular expression R_A below:

   $$R_A = (\, a \; b \,)^* \; (\, b \; a \; a \,)^*$$

   (b) Check if the automaton A designed at point (a) is minimal, and minimize it if necessary.

   (c) Consider the finite-state automaton B shown below, with initial state 1, and two final states 1 and 2:

   By using the *BMC* method (node elimination), write a regular expression R_B equivalent to the above automaton B.

   (d) From the automaton B of point (c) write, in a systematic way, an equivalent right-linear grammar and a system of linear language equations in the unknowns L_1 and L_2, which respectively represent the languages associated to the states 1 and 2 of B.

   Then solve the equation system and write a regular expression R_1 for L_1 (make sure that expression R_1 is equivalent to the expression R_B of point (c)).

   (e) (optional) By using a systematic method, design a deterministic finite-state automaton the accepts the intersection language $L\,(R_A) \; \cap \; L\,(B)$.

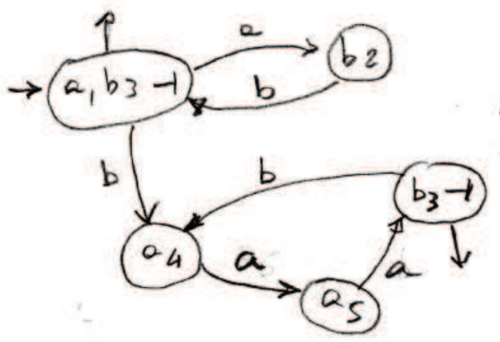
Solution

(a) Deterministic finite-state automaton A:

$e = (a_1 b_2)^* (b_3 a_4 a_5)^* \dashv$

$\text{Ini} = a_1\, b_3 \dashv$

x	Fol(x)
a_1	b_2
b_2	$a_1\, b_3 \dashv$
b_3	a_4
a_4	a_5
a_5	$b_3 \dashv$

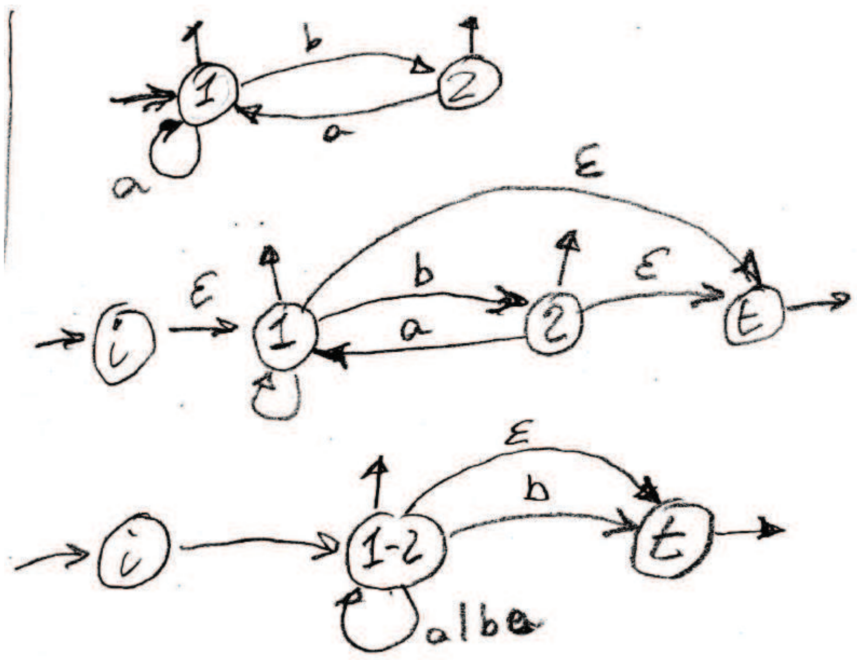


Automaton A
ans
part (a)

Figure 1: Ex.1.1.a - Berry-Sethi construction

(b) The automaton is indeed minimal. Final and non-final states are distinguishable. The two final states are distinguishable because they have outgoing arcs with different labels; the same holds for state pairs (b_2, a_4) and (b_2, a_5). State a_5 is distinguishable from a_4 because its next state is final.

(c) Regular expression R_B:



$$((a|(ba))^* b) \mid ((a|(ba))\varepsilon)$$

$$(a|(ba))^* (b|\varepsilon)$$

Figure 2: Ex.1.1.c - application of the BMC method

(d) Right-linear grammar and a system of linear language equations:

$$\begin{cases} S_1 \to \varepsilon \\ S_1 \to aS_1 \\ S_1 \to bS_2 \\ S_2 \to \varepsilon \\ S_2 \to aS_1 \end{cases} \qquad \begin{cases} L_1 = aL_1 \cup \varepsilon \cup bL_2 \\ L_2 = aL_1 \cup \varepsilon \end{cases}$$

$$\begin{aligned} L_1 &= aL_1 \cup \varepsilon \cup b(aL_1 \cup \varepsilon) \\ &= aL_1 \cup \varepsilon \cup baL_1 \cup b \\ L_1 &= (a \cup ba)L_1 \cup b \cup \varepsilon \\ L_1 &= (a|(ba))^*(b|\varepsilon) \end{aligned}$$

Figure 3: Ex.1.1.d - linear language equations

(e) Deterministic finite-state automaton for the intersection language:

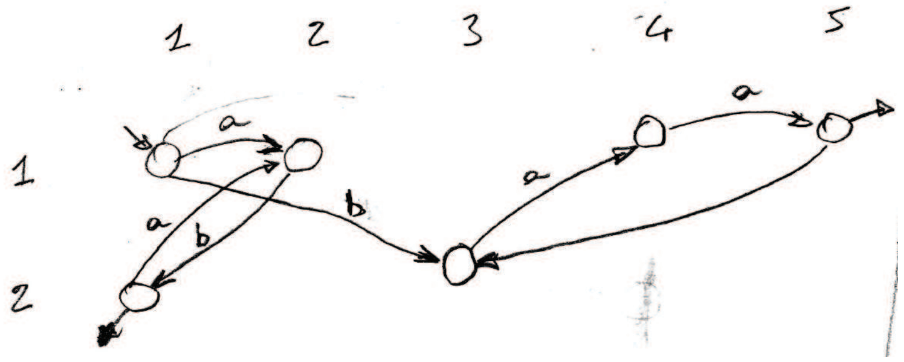


Figure 4: Ex.1.1.e - product automaton

2 Free Grammars and Pushdown Automata 20%

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

$$L_1 = \left\{ a^{h_1}\ b^{k_1} \mid \quad k_1 \geq h_1 \geq 0 \right\}$$

$$L_2 = \left\{ b^{k_2}\ a^{h_2} \mid \quad k_2 > h_2 \geq 1 \right\}$$

   Answer the following questions:

   (a) Write two *BNF* grammars G_1 and G_2 (no matter if they are ambiguous) that generate languages L_1 and L_2, respectively, and test such grammars by drawing the syntax trees of the valid strings $a\,b\,b \in L_1$ and $b\,b\,a \in L_2$.

   (b) Consider the concatenation language $L_3 = L_1 \cdot L_2$, write a description of L_3 as a set of strings (in the style of the above definitions of L_1 and L_2), and write a *BNF* grammar G_3 that generates language L_3, by using as components the grammars G_1 and G_2 found at point (a).

   (c) (optional) Say if the grammar G_3 found at point (b) is ambiguous or not. If it is ambiguous, say if the language L_3 is inherently ambiguous or not, and justify your answer.

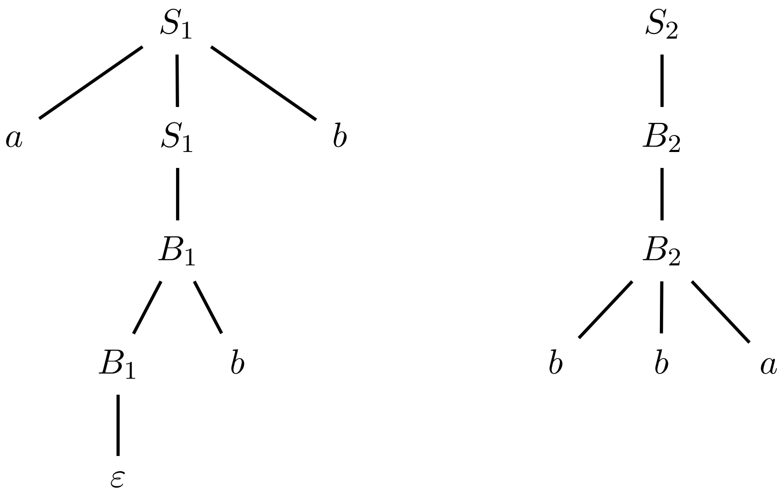
Solution

(a) Standard grammar G_1 (axiom S_1), non-ambiguous. Simple variation for grammar G_2 (axiom S_2), also non-ambiguous. Here they are:

$$G_1 \begin{cases} S_1 & \rightarrow & a\ S_1\ b \mid B_1 \\ B_1 & \rightarrow & B_1\ b \mid \varepsilon \end{cases}$$

$$G_2 \begin{cases} S_2 & \rightarrow & b\ S_2\ a \mid B_2 \\ B_2 & \rightarrow & b\ B_2 \mid b\ b\ a \end{cases}$$

Other solutions are possible. Here are the two sample syntax trees:



(b) Here is a reasonable description of language L_3, in the style of a set of strings:

$$L_3 = \left\{ a^{h_1}\ b^{k_1+k_2}\ a^{h_2} \mid \quad k_1 \geq h_1 \geq 0, k_2 > h_2 \geq 1 \right\}$$

A viable and easy grammar G_3 can be the standard concatenation of grammars G_1 and G_2: unite G_1 and G_2 (disjoint nonterminals), and add to their rules an axiomatic rule $S_3 \rightarrow S_1\ S_2$ (axiom S_3); see the textbook for details.

The standard concatenation grammar G_3 is ambiguous, as it satisfies the concatenation ambiguity condition (see textbook for details). For instance, the concatenation $a\ b^4\ a$ of the two sample strings below is ambiguous, because:

$$\underbrace{a\ b\ b}_{G_1} \cdot \underbrace{b\ b\ a}_{G_2} \in L\left(G_3\right) = L_3$$

yet also:

$$\underbrace{a\ b}_{G_1} \cdot \underbrace{b\ b\ b\ a}_{G_2} \in L\left(G_3\right) = L_3$$

that is, the string $a\ b^4\ a \in L\left(G_3\right) = L_3$ can be factored in two ways as a pair of strings that can be generated by grammars G_1 and G_2, respectively. An even shorter ambiguous string is $b^3\ a \in L_3$, which can be factored as $\varepsilon \cdot b^3\ a$ and $b \cdot b^2\ a$. Actually grammar G_3 generates infinitely many ambiguous strings.

(c) Language L_3 is not inherently ambiguous. Indeed, it is generated ambiguously by the standard concatenation grammar G_3 shown above, yet it can also be generated by a different grammar G'_3, which is not ambiguous at all. This new grammar G'_3 produces the letters b additional to the total number of letters a (there has to be at least one additional b) either when generating the leftmost letters a or when generating the rightmost letters a, yet not in both cases. Here is the latter choice for grammar G'_3 (axiom S_3), basically self-evident:

$$
G'_3 \left\{
\begin{array}{lcll}
S_3 & \rightarrow & S_1 \; S_2 & \text{— concatenation axiom} \\
\hline
S_1 & \rightarrow & a \; S_1 \; b \;\mid\; \varepsilon & \text{— modified grammar } G_1 \\
\hline
S_2 & \rightarrow & b \; S_2 \; a \;\mid\; B_2 & \multirow{2}{*}{\text{— original grammar } G_2} \\
B_2 & \rightarrow & b \; B_2 \;\mid\; b \; b \; a &
\end{array}
\right.
$$

Essentially, grammar G'_3 is still a concatenation grammar, but the component G_1 is modified to generate a number of letters b equal to (and not greater than) the number of (leftmost) letters a. Grammar G'_3 can be easily tested with the two ambiguous strings proposed above, which here are no longer ambiguous.

2. Consider a fragment of a programming language that allows a user to define arithmetic expressions by using the well known "let *var* in *expr*" construct, where the value of a nested expression *expr* is determined by that of the variables defined in the *var* clause of the enclosing expressions. A language phrase consists of a single expression introduced by the keyword let. Below are the phrase composition rules.

   - The *var* clauses contain one or more variable definitions, separated by a comma ", "; each definition uses the operator " = " with a variable name in the left part and an arithmetic expression in the right part.
   - Arithmetic expressions can contain arithmetic subexpressions of the type *if-then-else-endif*, the condition of which is composed of a relational operator (namely, " == " for equality, " != " for inequality, and " > " and " $\geq$ " with the usual meaning) applied to two arithmetic expressions.
   - Arithmetic expressions can contain the usual operators of addition " + ", subtraction " − " (possibly unary), multiplication " $*$ " and division " / ". The operator precedences are as usual, and the *if-then-else-endif* subexpressions have the highest precedence. Notice that parenthesized subexpressions are allowed.
   - The atomic subexpressions are identifiers and numbers, represented by the terminals id and num, respectively.

   Further detailed information about the lexical and syntactic structure of the programming language can be inferred from the example below:

```
let (x = 3, y = 7) in

   y - let (z = 9 - y) in

      x + z * if (z * 2 == 17)

      then 4 * (y + x)

      else

         let (alpha = -z / 2) in

            alpha - y / x

         endlet

      endif

   endlet

endlet
```

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

Solution

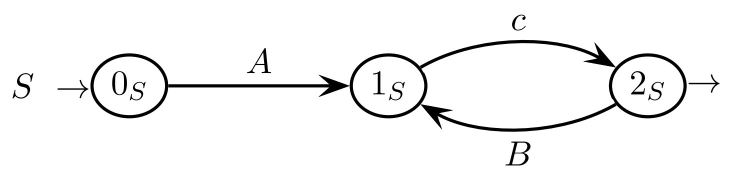
Here is a possible grammar G (axiom EXP) for the sketched programming language:

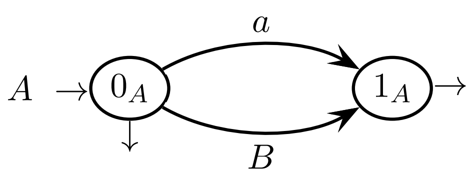
$$
\left\{
\begin{array}{rcl}
\langle \mathsf{EXP} \rangle & \rightarrow & \langle \mathsf{L\_EXP} \rangle \mid \langle \mathsf{C\_EXP} \rangle \mid \langle \mathsf{A\_EXP} \rangle \\
\hline
\langle \mathsf{L\_EXP} \rangle & \rightarrow & \mathsf{let}\ `(\text{'}\ \langle \mathsf{V\_DEC} \rangle\ `)\text{'}\ \mathsf{in}\ \langle \mathsf{EXP} \rangle\ \mathsf{endlet} \\
\langle \mathsf{V\_DEC} \rangle & \rightarrow & \langle \mathsf{ASGN} \rangle \left(`,\text{'}\ \langle \mathsf{ASGN} \rangle \right)^* \\
\langle \mathsf{ASGN} \rangle & \rightarrow & \mathsf{id} = \langle \mathsf{EXP} \rangle \\
\hline
\langle \mathsf{A\_EXP} \rangle & \rightarrow & \left[`-\text{'} \right] \langle \mathsf{TERM} \rangle \left( \left( `+\text{'} \mid `-\text{'} \right) \langle \mathsf{TERM} \rangle \right)^* \\
\langle \mathsf{TERM} \rangle & \rightarrow & \langle \mathsf{FACT} \rangle \left(\left(`*\text{'} \mid `/\text{'} \right) \langle \mathsf{FACT} \rangle \right)^* \\
\langle \mathsf{FACT} \rangle & \rightarrow & \mathsf{num} \mid \mathsf{id} \mid \langle \mathsf{L\_EXP} \rangle \mid \langle \mathsf{C\_EXP} \rangle \mid `(\text{'}\ \langle \mathsf{EXP} \rangle\ `)\text{'} \\
\hline
\langle \mathsf{C\_EXP} \rangle & \rightarrow & \mathsf{if}\ `(\text{'}\ \langle \mathsf{CND} \rangle\ `)\text{'}\ \mathsf{then}\ \langle \mathsf{EXP} \rangle \left[\mathsf{else}\ \langle \mathsf{EXP} \rangle \right] \mathsf{endif} \\
\langle \mathsf{CND} \rangle & \rightarrow & \langle \mathsf{EXP} \rangle\ \langle \mathsf{R\_OP} \rangle\ \langle \mathsf{EXP} \rangle \\
\langle \mathsf{R\_OP} \rangle & \rightarrow & `==\text{'} \mid `!=\text{'} \mid `>=\text{'} \mid `>\text{'}
\end{array}
\right.
$$

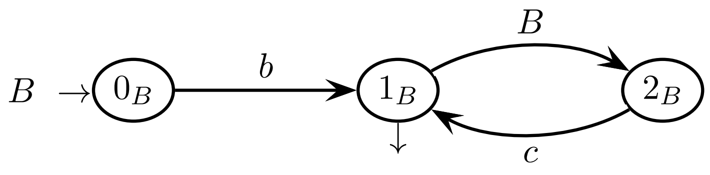
This *EBNF* grammar is not ambiguous, as it is a composition of non-ambiguous structures. The example shows that a condition and the assignment clause(s) in a *let* construct are always enclosed in round brackets, whereas the text is silent about this issue. Here we choose to abide to the example, thus the grammar above models such a language trait. However, these brackets are unnecessary for correctly interpreting the expression, and they could be eliminated. Furthermore, while an arithmetic factor may be a parenthesized subexpression of any type, i.e., *let*, *if* or arithmetic, the subexpressions of type *let* or *if* may be not parenthesized (as the example shows), because their pairs of initial and final keywords (*if*-*endif* and *let*-*endlet* respectively) clearly delimit them and so make parentheses unnecessary (though permitted). Instead, parentheses are necessary for expanding a factor into an arithmetic subexpression.

3 Syntax Analysis and Parsing Methodologies 20%

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







Answer the following questions:

(a) Draw the complete pilot of grammar G, say if grammar G is of type $ELR(1)$, and shortly justify your answer (highlight all the conflicts, if any).

(b) Write the necessary guide sets on all the arcs of the machine net (call arcs and exit arrows), say if grammar G is of type $ELL(1)$, based on the guide sets, and shortly justify your answer (highlight all the conflicts, if any).

(c) Independently of whether grammar G is of type $ELL(1)$, it has at least one machine without conflicting guide sets. Write the recursive descent syntactic procedure of such a machine (if there are two or more, choose one at will).

(d) (optional) Examine the language $L(G)$ and determine if it can be parsed with a recognizer type simpler than a pushdown automaton (of whatever kind).

please here draw the call arcs and write all the guide sets

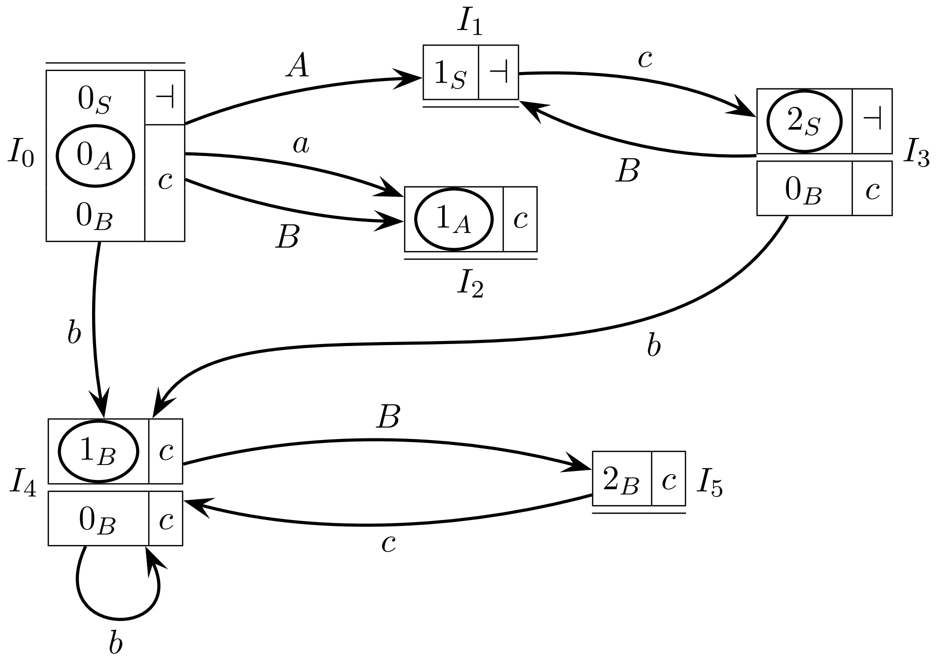
$$\begin{tikzcd}
S \arrow[r] & 0_S \arrow[r, "A"] & 1_S \arrow[r, bend left, "c"] & 2_S \arrow[l, bend left, "B"] \arrow[r] & {}
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r] & 0_A \arrow[r, bend left, "a"] \arrow[r, bend right, "B"'] \arrow[d] & 1_A \arrow[r] & {} \\
 & {} & &
\end{tikzcd}$$

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

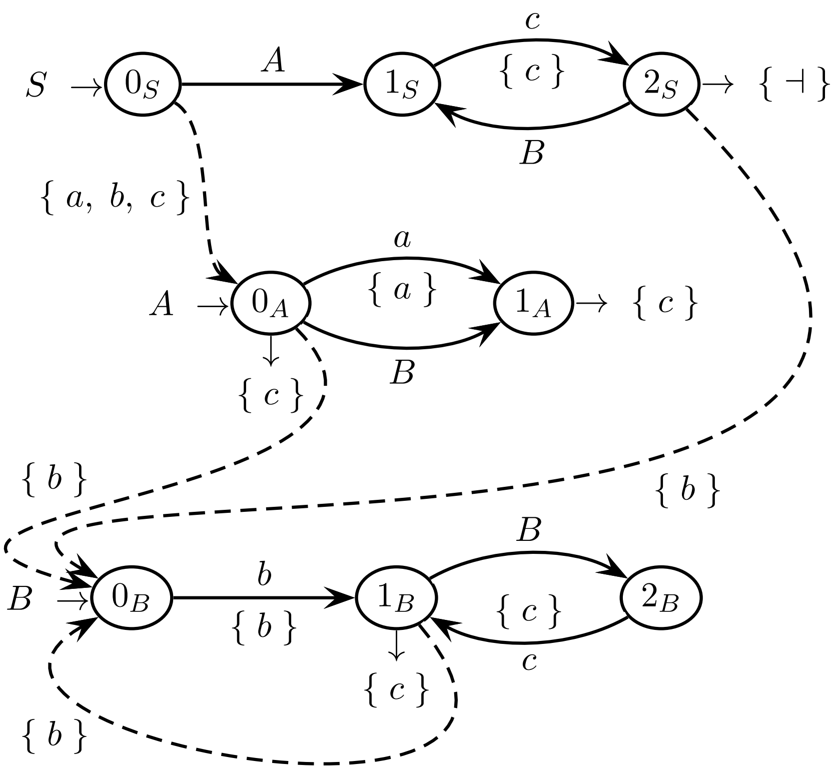
Solution

(a) Here is the complete pilot:



There are not any conflicts. Grammar G is $ELR(1)$.

(b) Here is the *PCFG*:



There are not any conflicts between guide sets. Grammar G is *ELL* (1).

(c) TO BE DONE, all procedures are deterministic and their encoding is easy

(d) The language is not regular, essentially due to the auto-inclusive machine of nonterminal B:

$$L(B) = b^{n+1} c^{n} \quad n \geq 0$$
$$L(A) = \varepsilon \mid a \mid b^{n+1} c^{n} \qquad L(S) = L(A)\, c\, \big(L(B)\, c\big)^{*}$$

Figure 5: the language is not regular

4 Language Translation and Semantic Analysis 20%

1. The *BNF* source grammar G_S below (axiom S), over the two-letter terminal alphabet $\{\ a,\ b\ \}$:

$$G_S \begin{cases} S & \rightarrow & a\ a\ b \\ S & \rightarrow & a\ a\ a\ S\ b\ b \end{cases}$$

generates the source language L_S below:

$$L_S = \left\{\ a^i\ b^j \ \mid \quad 2i = 3j + 1\ \right\}$$

The language L_S includes, among others, the following three strings:

$$a^2\, b \qquad a^5\, b^3 \qquad a^8\, b^5$$

Please answer the following questions:

(a) Write a target (or destination) grammar G_T associated to the source grammar G_S, without changing G_S, that generates the following target language L_T:

$$L_T = \left\{\ a^i\ c\ b^j \ \mid \quad 3i = 2j + 2\ \right\}$$

according to the four translation examples below:

$$a^2\, b \mapsto a^2\, c\, b^2 \quad a^5\, b^3 \mapsto a^4\, c\, b^5 \quad a^8\, b^5 \mapsto a^6\, c\, b^8 \quad a^{11}\, b^7 \mapsto a^8\, c\, b^{11}$$

(b) Write the syntax trees of the source and target strings for the two translation examples below:

$$a^2\, b \mapsto a^2\, c\, b^2 \qquad a^5\, b^3 \mapsto a^4\, c\, b^5$$

(c) (optional) Can the translation defined by the scheme above be computed deterministically? Provide an adequate justification to your answer.

Solution

(a) Here is the target grammar:

$$G_T \begin{cases} S & \rightarrow & a\,a\,c\,b\,b \\ S & \rightarrow & a\,a\,S\,b\,b\,b \end{cases}$$

(b) Some are shown at point (a)

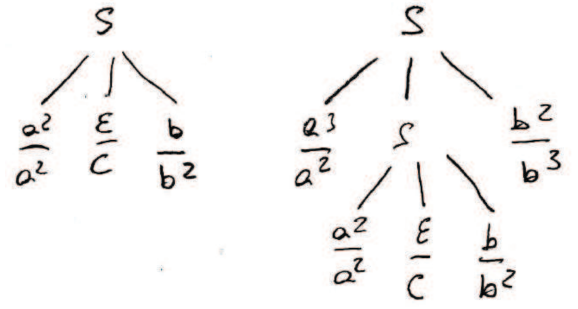
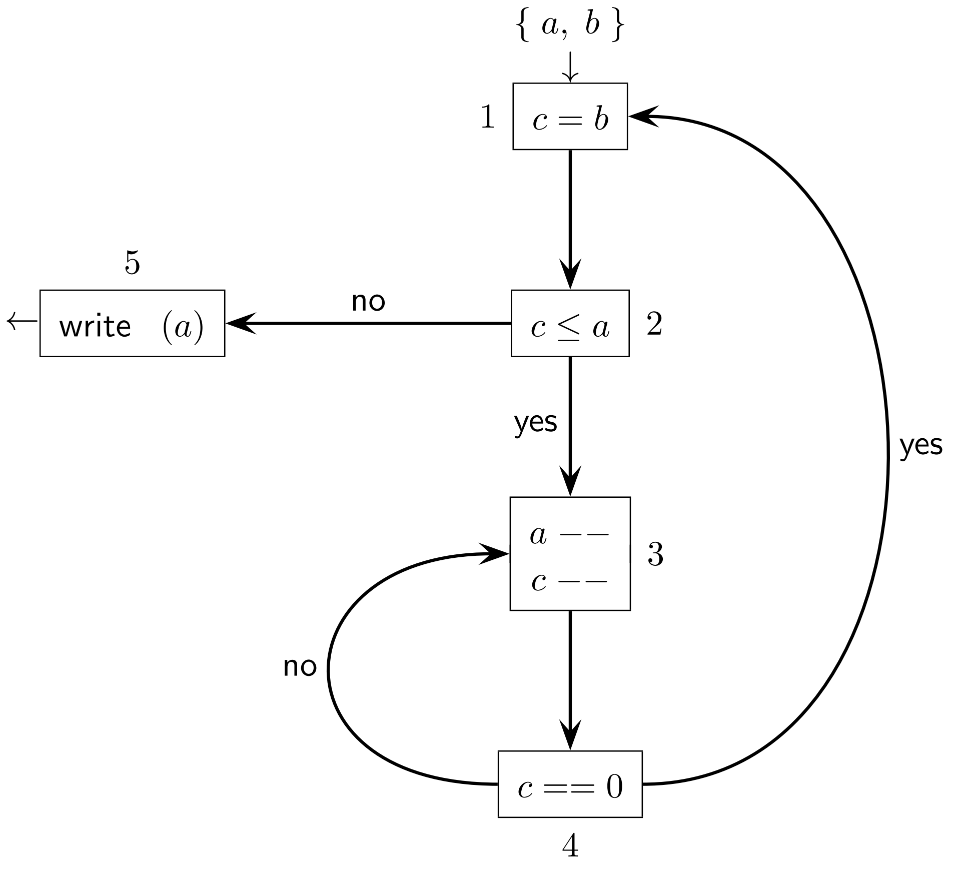


Figure 6: Ex.4.1.b - trees for the translations

(c) The source grammar is deterministic $ELL(3)$, hence the translation can be computed by a top-down syntax analyzer with additional write actions.

2. Consider the following Control Flow Graph (*CFG*) of a program, with some nodes that contain multiple assignments (input node 1 and output node 5):



This program has two input parameters a and b, and one variable c. All of them are integers, with $a \geq 0$ and $b \geq 1$.

Answer the following questions:

(a) Find a regular expression R, over the node name alphabet, that describes the execution traces of the program, by ignoring any semantic restriction.

(b) Write the system of flow equations for the *reaching definitions* of the *CFG*, and iteratively solve the equation system (use the tables and the figure prepared on the next pages).

(c) (optional) Determine the final value of a written by the program as a function of the initial values of a and b. Based on this function, refine the regular expression R found at point (a) into a language expression, over the node name alphabet, that exactly defines the program execution traces. Is the language of the execution traces still regular ? Adequately justify your answer.

table of definitions and suppressions at the nodes
for the **REACHING DEFINITIONS**

node	*defined*
1	
2	
3	
4	
5	

node	*suppressed*
1	
2	
3	
4	
5	

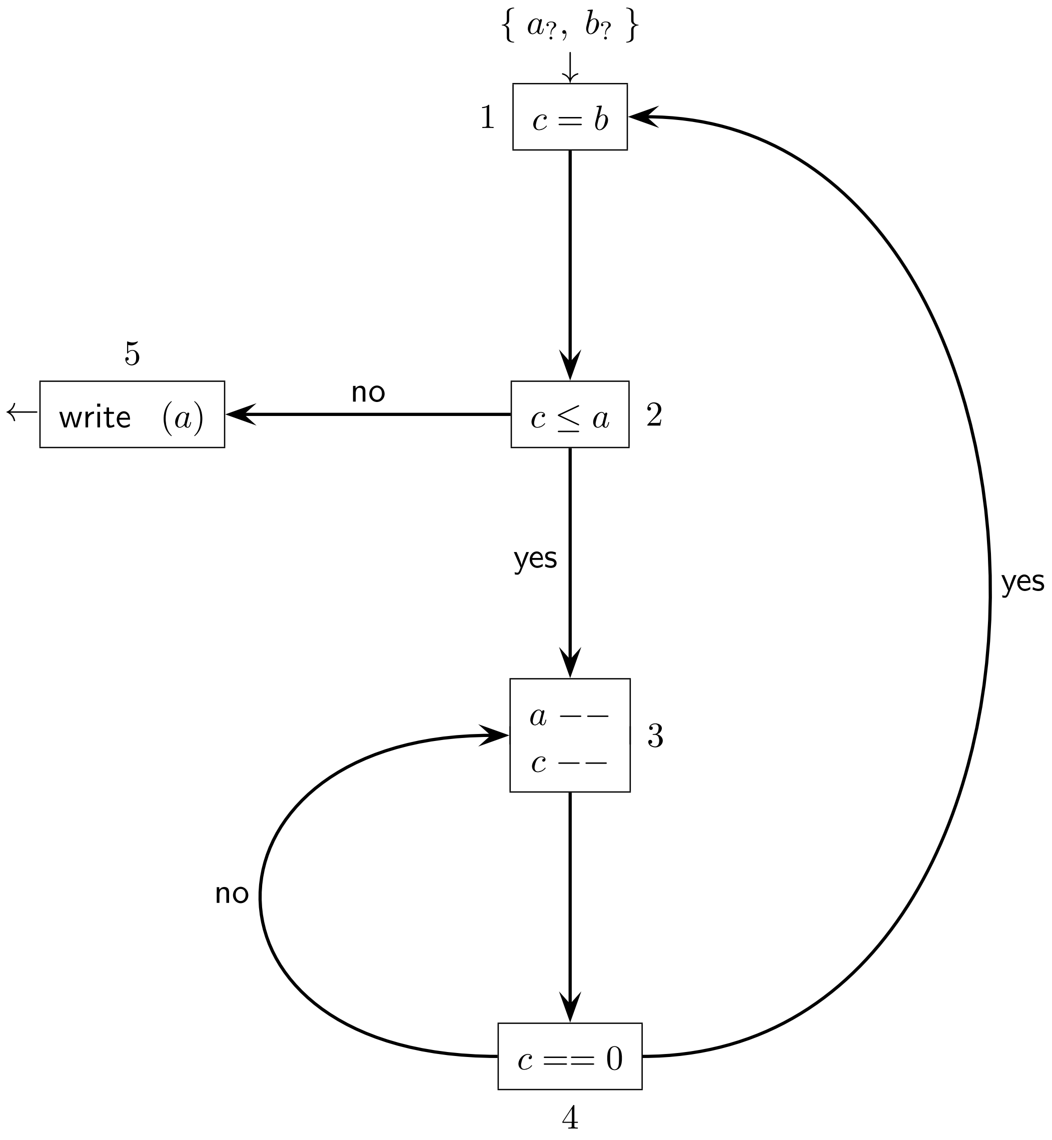
system of data-flow equations for the **REACHING DEFINITIONS**

node	*in equations*	*out equations*
1		
2		
3		
4		
5		

iterative solution table of the system of data-flow equations (**REACHING DEF.S**)
(the number of columns is not significant)

	initialization		1		2		3		4		5		6	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1														
2														
3														
4														
5														

write here the **REACHING DEFINITIONS** computed



Solution

(a) Here is the regular expression R of the execution trace language of the *CFG*:

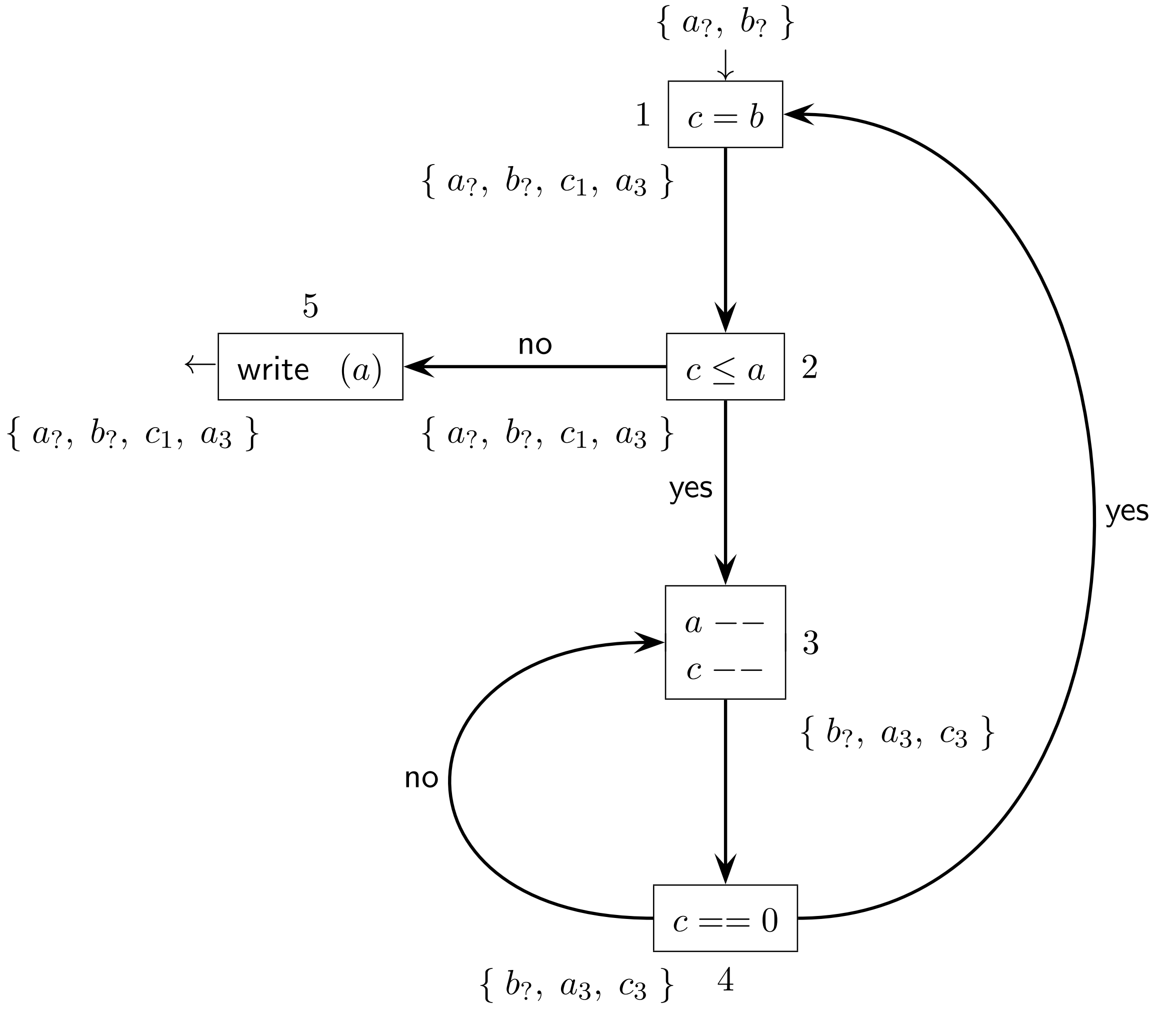
$$R = 1\ 2\ \Big(\big(3\ 4\big)^{+} 1\ 2\Big)^{*}\ 5$$

(b) In total there are the following definitions: $a_?$, $b_?$, c_1, a_3 and c_3. Here are the definitions and the suppressions at each node of the *CFG*:

node	*defined*
1	c_1
2	
3	$a_3\ c_3$
4	
5	

node	*suppressed*
1	c_3
2	
3	$a_?\ c_1$
4	
5	

We omit the equations and their iterative solution, at all systematic. Here is the solution, with the definitions that reach the output of each node of the *CFG*:



(c) The program computes the remainder r of a divided by b: $r = a \mod b \geq 0$. In fact, the final value taken by a and written is an integer $r \geq 0$, related to the initial values of a and b as $a = q \times b + r$ with $q \geq 0$ and $0 \leq r < b$, namely the remainder. The reader can verify how the remainder r is correctly (though inefficiently) computed: if initially $a \geq b$, then the inner loop $(3, 4)$ subtracts b from a by repeatedly decrementing a, the outer loop $(1, 2, \text{inner loop})$ iterates this subtraction for q times until a gets lower than b, and at the exit clearly the residual value of a is the remainder r; else (i.e., initially $a < b$) the program exits immediately with the initial value of a, which is already the remainder r.

Notice that the Kleene star and cross operators are related to the initial values of the input parameters a and b as follows:

$$R = 1\,2\,\Big(\big(3\,4\big)^{b}\,1\,2\Big)^{a \div b}\,5 \qquad \text{i.e., posing } + = b \geq 1 \text{ and } * = a \div b \geq 0$$

where $a \div b \geq 0$ is the quotient of the integer division of a by b. In logical terms:

$$\forall\, a,\, b,\, q \geq 0,\, 1,\, 0 \qquad q = a \div b \;\Rightarrow\; R = 1\,2\,\Big(\big(3\,4\big)^{b}\,1\,2\Big)^{q}\,5$$

Now, for any values of the input parameter $b \geq 1$ and of the quotient $q \geq 0$, which is the final value of a, we can always find an initial value for the input parameter $a \geq 0$ such that $q = a \div b$. Therefore all the execution traces generated by the regular expression R are semantically possible, that is, semantics is irrelevant for the execution trace language. The description provided by R is exact, and the execution trace language of the *CFG* is actually regular.