FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Tue 3 JULY 2018 - Part Theory

WITH SOLUTIONS - for teaching purposes here the solutions are widely commented

LAST + FIRST NAME:

(capital letters please)

MATRICOLA: SIGNATURE:

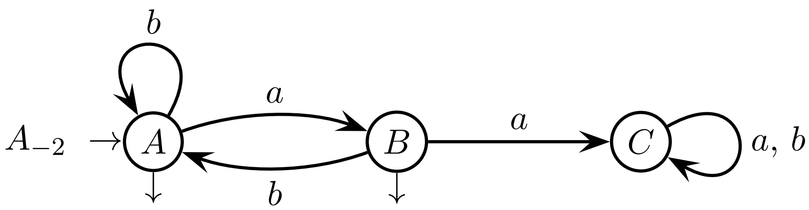
(or PERSON CODE)

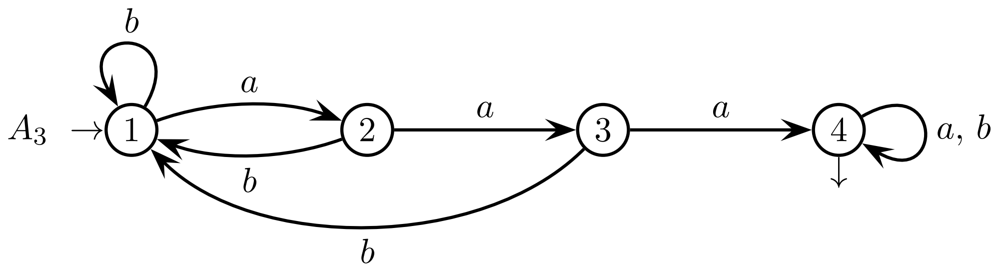
INSTRUCTIONS - READ CAREFULLY:

- 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 two automata A_{-2} and A_3 over the two-letter alphabet $\Sigma = \{ \, a, \; b \, \}$:





Language $L(A_{-2})$ includes all the strings without any occurrence of a (sub)string $a\,a$, whereas language $L(A_3)$ includes all the strings with at least one occurrence of a (sub)string $a\,a\,a$.

Answer the following questions:

(a) Transform automaton A_{-2} into a right unilinear grammar G_{-2}.

(b) Transform grammar G_{-2} into a set of language equations and solve them, thus obtaining a regular expression e_{-2} for the language $L(A_{-2})$.

(c) Apply the Berry-Sethi method to the regular expression e_{-2} obtained before and construct a deterministic automaton A'_{-2}. Verify that the automaton A'_{-2} thus obtained is identical or equivalent to automaton A_{-2} (if the two automata are not identical, argue that they are equivalent).

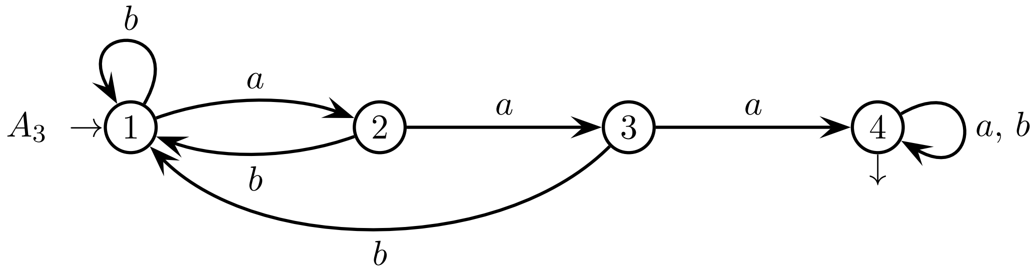
(d) (optional) Apply a variant of the cartesian product automaton construction, usually employed to construct an automaton $A_\cap$ that accepts an intersection language, to construct instead an automaton $A_\cup$ that accepts the *union language* $L_\cup = L_{-2} \cup L_3$. On the grid prepared on the next pages, which is the cartesian product of the state sets of automata A_{-2} and A_3, first identify the final states of the union automaton $A_\cup$ and mark them with an exit arrow, then draw the suitable necessary transitions. Show that:

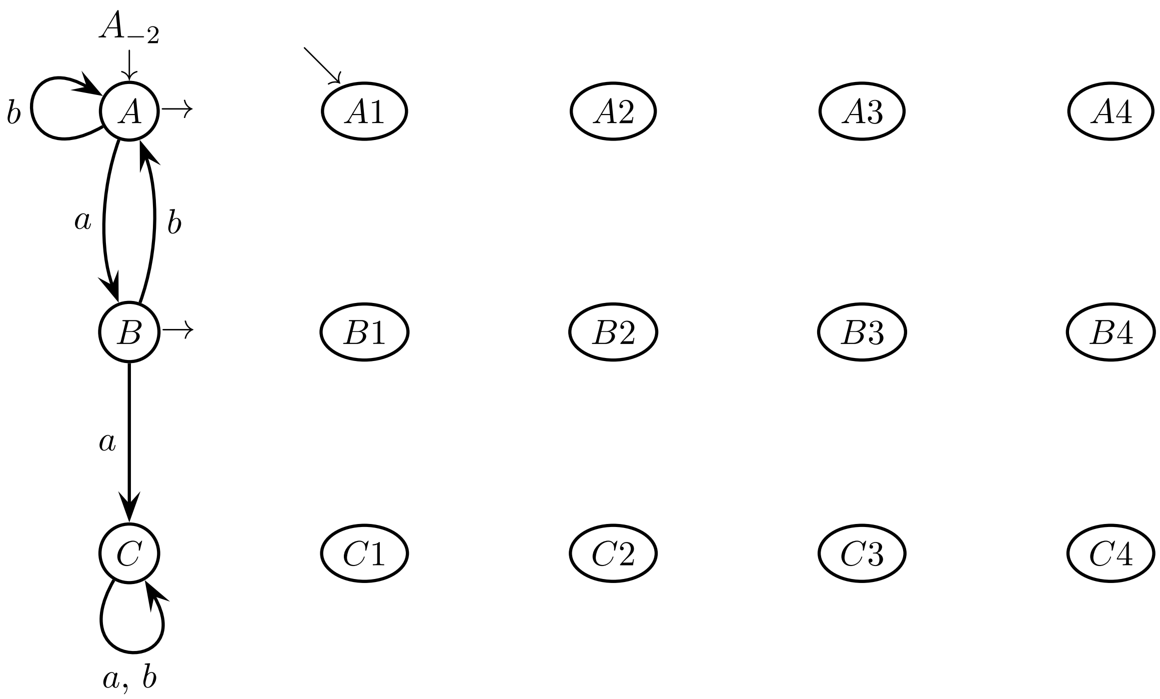
$$b\,a\,a\,b \notin L_\cup \qquad \text{and} \qquad b\,a\,a\,a\,b \in L_\cup$$

by writing the runs of $A_\cup$ that have these two strings as input.

---

grid to be used for question (d)





Solution

(a) Here is the required grammar G_{-2} (axiom A):

$$G_{-2} \begin{cases} A \rightarrow a\,B \mid b\,A \mid \varepsilon \\ B \rightarrow a\,C \mid b\,A \mid \varepsilon \\ C \rightarrow a\,C \mid b\,C \end{cases}$$

Notice that $L\,(C) = \emptyset$, because by definition $L\,(C) = \{\, x \mid \quad x \in \Sigma^* \text{ and } C \stackrel{*}{\Rightarrow} x \,\}$. Thus grammar G_{-2}, which is not clean (reduced), is equivalent to the following clean one (still axiom A):

$$G_{-2}^{clean} \begin{cases} A \rightarrow a\,B \mid b\,A \mid \varepsilon \\ B \rightarrow b\,A \mid \varepsilon \end{cases}$$

(b) The above rules can be straightforwardly transformed into the following language equations:

$$\begin{aligned} L_A &= b\,L_A \,\cup\, aL_B \cup \varepsilon \\ L_B &= b\,L_A \,\cup\, \varepsilon \end{aligned}$$

By substituting the second equation into the first one, we obtain:

$$L_A = b\,L_A \,\cup\, a\,(\,b\,L_A \,\cup\, \varepsilon\,) \,\cup\, \varepsilon$$

from which:

$$\begin{aligned} L_A &= b\,L_A \,\cup\, a\,b\,L_A \,\cup\, a \,\cup\, \varepsilon \\ &= (\,b \,\cup\, a\,b\,)\,L_A \,\cup\, a \,\cup\, \varepsilon \end{aligned}$$

By applying the Arden identity, we get:

$$L_A = (\,b \,\cup\, a\,b\,)^*\,(\,a \,\cup\, \varepsilon\,)$$

hence the requested regular expression e_{-2} is:

$$e_{-2} = (\,b \mid a\,b\,)^*\,(\,a \mid \varepsilon\,)$$

and by optimizing it with the optionality operator, it becomes:

$$e_{-2}^{opt} = \big(\,[\,a\,]\,b\,\big)^*\,[\,a\,]$$

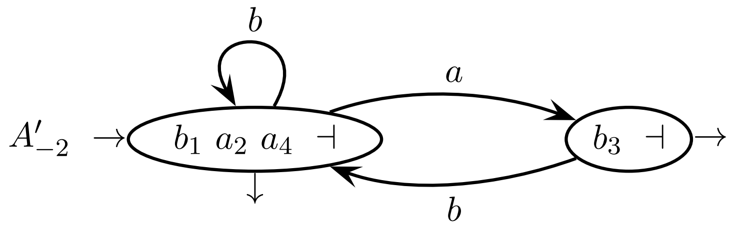
(c) Here is the application of the *BS* method to the regular expression e_{-2}:

$$e_{-2,\#} \quad = \quad (\, b_1 \;\mid\; a_2\, b_3\,)^* \, (\, a_4 \;\mid\; \varepsilon\,) \;\dashv$$

The initials and followers are:

initials	$b_1 \; a_2 \; a_4 \;\; \dashv$
generators	followers
b_1	$b_1 \; a_2 \; a_4 \;\; \dashv$
a_2	b_3
b_3	$b_1 \; a_2 \; a_4 \;\; \dashv$
a_4	$\dashv$

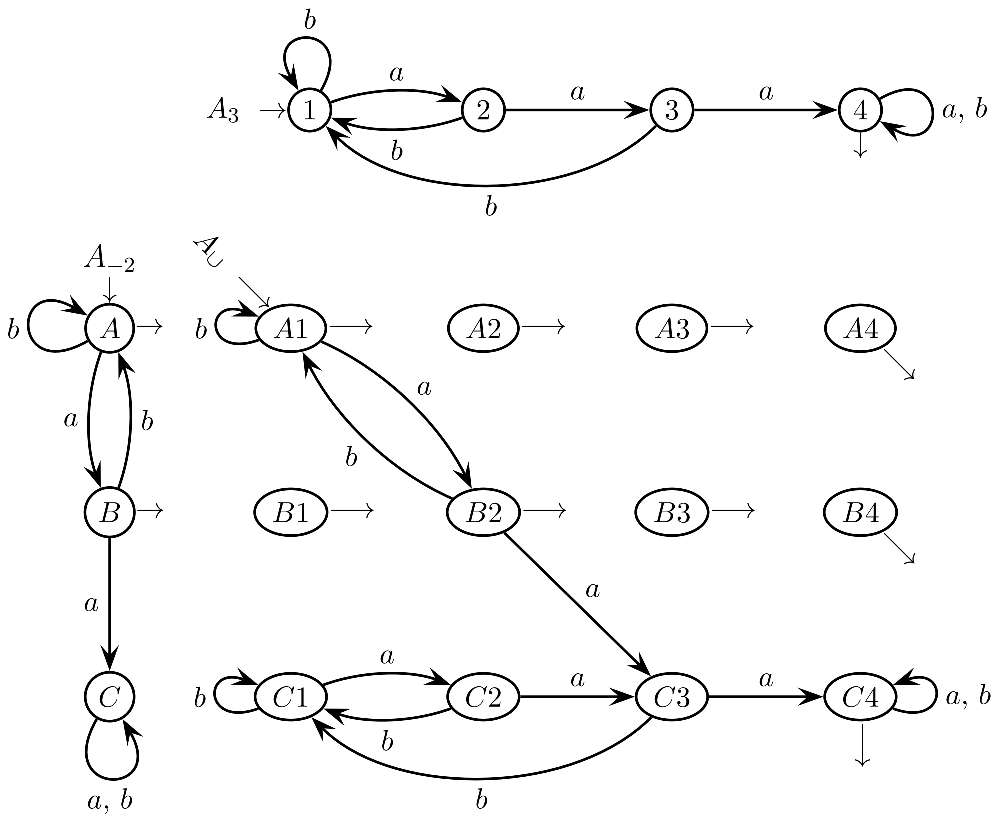
The resulting deterministic *BS* automaton A'_{-2} is:



If from automaton A_{-2} we delete state C, which is not post-accessible, that is, if we clean automaton A_{-2}, then the deterministic automaton A'_{-2} obtained through the *BS* method happens to be identical to the cleaned automaton A_{-2}.

(d) To accept the union language, the final states in the product automaton are exactly those where *at least one* of the two component states is final.

Here is the so-constructed union automaton $A_\cup$:



Several states could be removed, i.e., states $C1$, $C2$ and all the unconnected final states, and automaton $A_\cup$ put into clean form.

Here are the computations of the union automaton $A_\cup$ for the two sample strings $b\,a\,a\,b$ and $b\,a\,a\,a\,b$:

$$A1 \xrightarrow{b} A1 \xrightarrow{a} B2 \xrightarrow{a} C3 \xrightarrow{b} C1 \qquad \text{input rejected}$$

$$A1 \xrightarrow{b} A1 \xrightarrow{a} B2 \xrightarrow{a} C3 \xrightarrow{a} C4 \xrightarrow{b} C4 \qquad \text{input accepted}$$

which behave as expected.

2 Free Grammars and Pushdown Automata 20%

1. Consider the Dyck language L with two types of parenthesis pairs (open closed): $a\,b$ and $c\,d$. Thus the alphabet is $\Sigma = \{\, a,\ b,\ c,\ d \,\}$. The standard (non-ambiguous) grammar that generates language L is:

$$S \to a\,S\,b\,S \ \mid\ c\,S\,d\,S \ \mid\ \varepsilon$$

Answer the following questions:

(a) Suppose to restrict language L and to *admit* only the strings where between any matching pair $c\,d$ no characters $c\,d$ can appear (only characters $a\,b$ are allowed). The restricted language is $L_1 \subset L$.

Examples of Dyck strings that <u>*are not admitted*</u> in the language L_1:

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

These instead are *valid* strings for language L_1:

$$\varepsilon \qquad a\,b\,c\,d \qquad a\,c\,d\,b\,a\,b \qquad a\,c\,d\,c\,d\,b \qquad a\,c\,a\,b\,d\,b\,c\,d$$

Write a *BNF* grammar G_1, not ambiguous, that generates language L_1, and check grammar G_1 by drawing the syntax tree of the sample string:

$$a\,c\,a\,b\,d\,b\,c\,d$$

(b) Suppose to restrict language L and to *admit* only the strings that satisfy these two conditions:

- the string does not have any matching pair $a\,b$ or $c\,d$, at any depth level, that immediately contains more than one pair $a\,b$
- among the matching pairs at the top level of the string, there is no more than one pair $a\,b$

The restricted language is $L_2 \subset L$.

Examples of Dyck strings that <u>*are not admitted*</u> in the language L_2:

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

These instead are *valid* strings for language L_2:

$$\varepsilon \qquad a\,b\,c\,a\,b\,d \qquad c\,a\,b\,d\,a\,a\,b\,b$$

Write a *BNF* grammar G_2, not ambiguous, that generates language L_2, and check grammar G_2 by drawing the syntax tree of the sample string:

$$c\,a\,b\,d\,a\,a\,b\,b$$

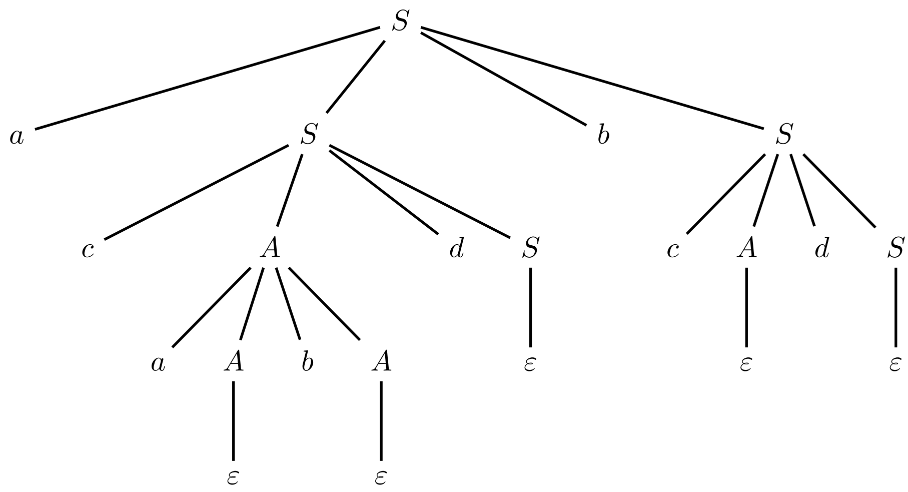
(c) (optional) Countercheck grammars G_1 and G_2 by showing that it is impossible to draw the complete trees of the counterexample strings $c\,c\,d\,d$ and $a\,b\,a\,b$, respectively. Motivate your answer.

Solution

(a) Here is a working *BNF* grammar G_1:

$$G_1 \begin{cases} S \;\rightarrow\; a\,S\,b\,S \;\mid\; c\,A\,d\,S \;\mid\; \varepsilon \\ A \;\rightarrow\; a\,A\,b\,A \;\mid\; \varepsilon \end{cases}$$

Tree of the valid string $a\,c\,a\,b\,d\,b\,c\,d$:

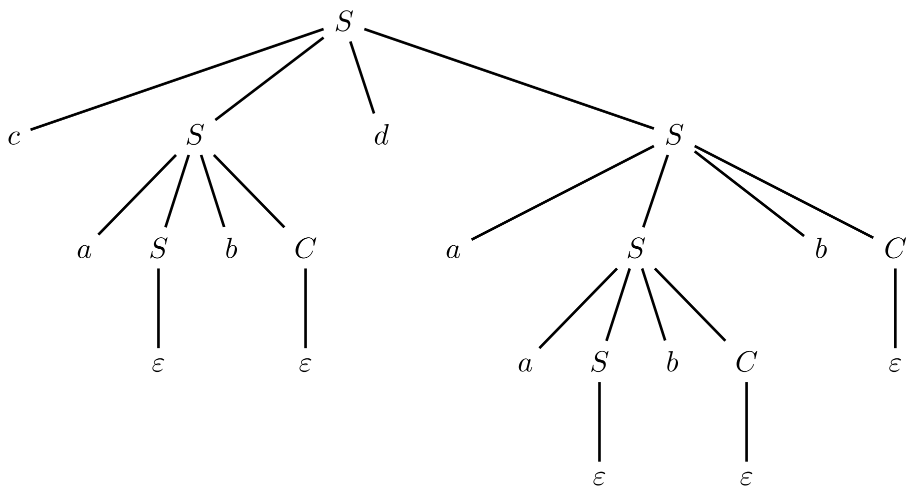


The tree is correct.

(b) Here is a working *BNF* grammar G_2:

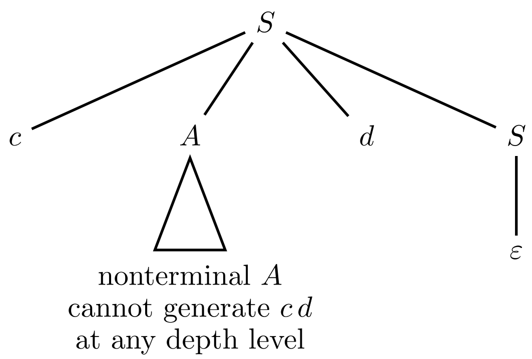
$$G_2 \begin{cases} S \;\rightarrow\; a\,S\,b\,C \;\mid\; c\,S\,d\,S \;\mid\; \varepsilon \\ C \;\rightarrow\; c\,S\,d\,C \;\mid\; \varepsilon \end{cases}$$

Tree of the valid string $c\,a\,b\,d\,a\,a\,b\,b$:

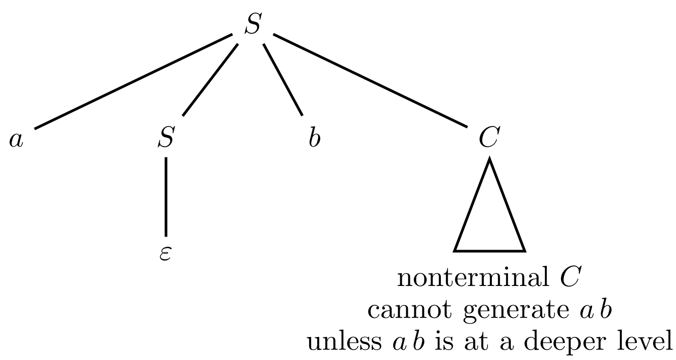


The tree is correct.

(c) Tentative tree of the invalid string $c\,c\,d\,d$ for grammar G_1:



Tentative tree of the invalid string $a\,b\,a\,b$ for grammar G_2:



Neither tree can be completed with its own grammar.

2. We introduce a language for defining, in a general and flexible way, structured data for any data intensive application. The data structure is specified as follows:

   - A language phrase consists of one *object*. An object is a non-empty list of *pairs*, separated by a comma "," and enclosed within braces "{" and "}".
   - A pair consists of a *string* and a *value*, separated by a colon ":".
   - A value is either a string or an object or an *array* or a *number* or a *boolean*.
   - An array is a possibly empty list of values, enclosed within square brackets "[" and "]", and separated by a comma ",".
   - A string consists of a possibly empty sequence of characters surrounded by double quotes, e.g., "color".
   - Individual characters should be modeled in the required grammar by the terminal symbol char. Similarly, numbers and booleans should be respectively represented by the terminals num and bool.

   Here is a sample phrase, which does not necessarily exhibit all the above constructs:

```
{
   "colors": [
      {
         "color": "red",
         "category": "hue",
         "type": "primary",
         "code": {
            "rgba": [255, 0, 0, 1],
            "hex": "#FF0"
         }
      },
      {
         "color": "green",
         "category": "hue",
         "type": "secondary",
         "code": {
            "rgba": [0, 255, 0, 1],
            "hex": "#0F0"
         }
      }
   ]
}
```

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

Solution

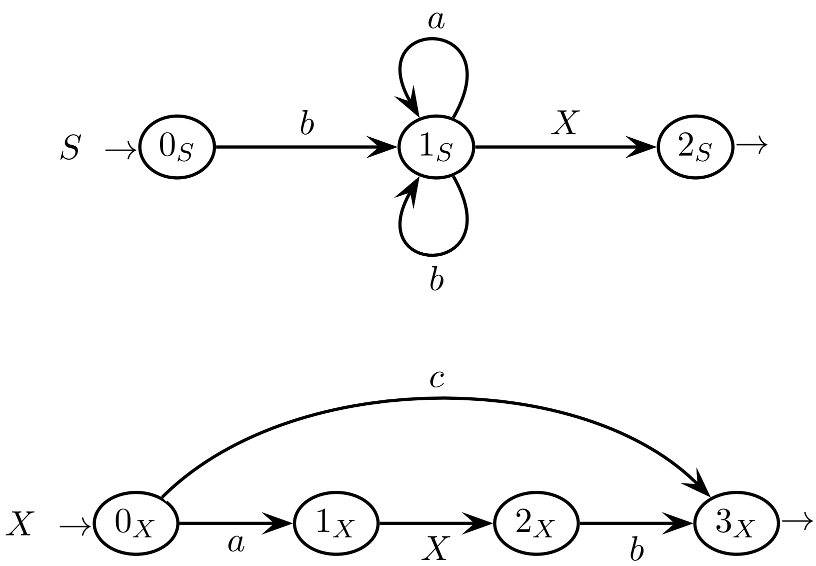
Here is a possible grammar (axiom JSON) for the sketched description language:

$$
\begin{cases}
\langle \mathsf{JSON} \rangle \rightarrow \langle \mathsf{OBJECT} \rangle \\
\langle \mathsf{OBJECT} \rangle \rightarrow \text{`\{'} \langle \mathsf{PA\_LST} \rangle \ \text{`\}'} \\
\langle \mathsf{PA\_LST} \rangle \rightarrow \langle \mathsf{PAIR} \rangle \ (\text{`,'} \ \langle \mathsf{PAIR} \rangle\ )^* \\
\langle \mathsf{PAIR} \rangle \rightarrow \langle \mathsf{STRING} \rangle \ \text{`:'} \ \langle \mathsf{VALUE} \rangle \\
\langle \mathsf{STRING} \rangle \rightarrow \text{''} \ \mathsf{char}^* \ \text{''} \\
\langle \mathsf{VALUE} \rangle \rightarrow \langle \mathsf{OBJECT} \rangle \mid \langle \mathsf{ARRAY} \rangle \mid \langle \mathsf{STRING} \rangle \mid \mathsf{num} \mid \mathsf{bool} \\
\langle \mathsf{ARRAY} \rangle \rightarrow \text{`['} \ [\langle \mathsf{VA\_LST} \rangle\,] \ \text{`]'} \\
\langle \mathsf{VA\_LST} \rangle \rightarrow \langle \mathsf{VALUE} \rangle \ (\text{`,'} \ \langle \mathsf{VALUE} \rangle\ )^*
\end{cases}
$$

Notice the different usage of the square brackets as terminals or metasymbols (for the optionality operator).

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 two-letter nonterminal alphabet $V = \{ S, X \}$ (axiom S):



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

(a) Draw the complete pilot of grammar G and prove that grammar G is of type $ELR(1)$. In particular, examine whether there are multiple transitions and convergent transitions.

(b) Draw all the guide sets on the net (shift arcs, call arcs and exit arrows), determine whether grammar G is of type $ELL(1)$ and justify your answer.

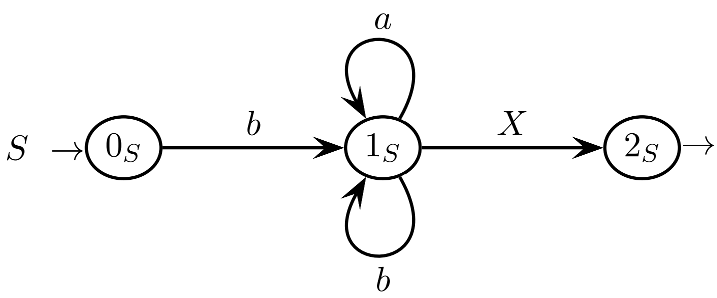
(c) Use the Earley algorithm and analyze the sample valid string:

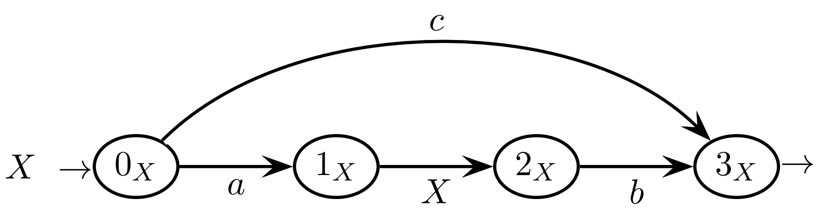
$$b\,a\,c\,b$$

Draw the syntax tree and show which paths the net machines use to recognize.

(d) (optional) Suppose that nonterminal X is nullable and modify machine M_X as little as possible. Does anything change in the $ELR(1)$ analysis ? For what reasons ? Explain what and why.

here draw the pilot of grammar G – question (a) here draw the call arcs and write all the guide sets of grammar G – question (b)





Earley vector of string $b\ a\ c\ b$ (to be filled in) – question (c)

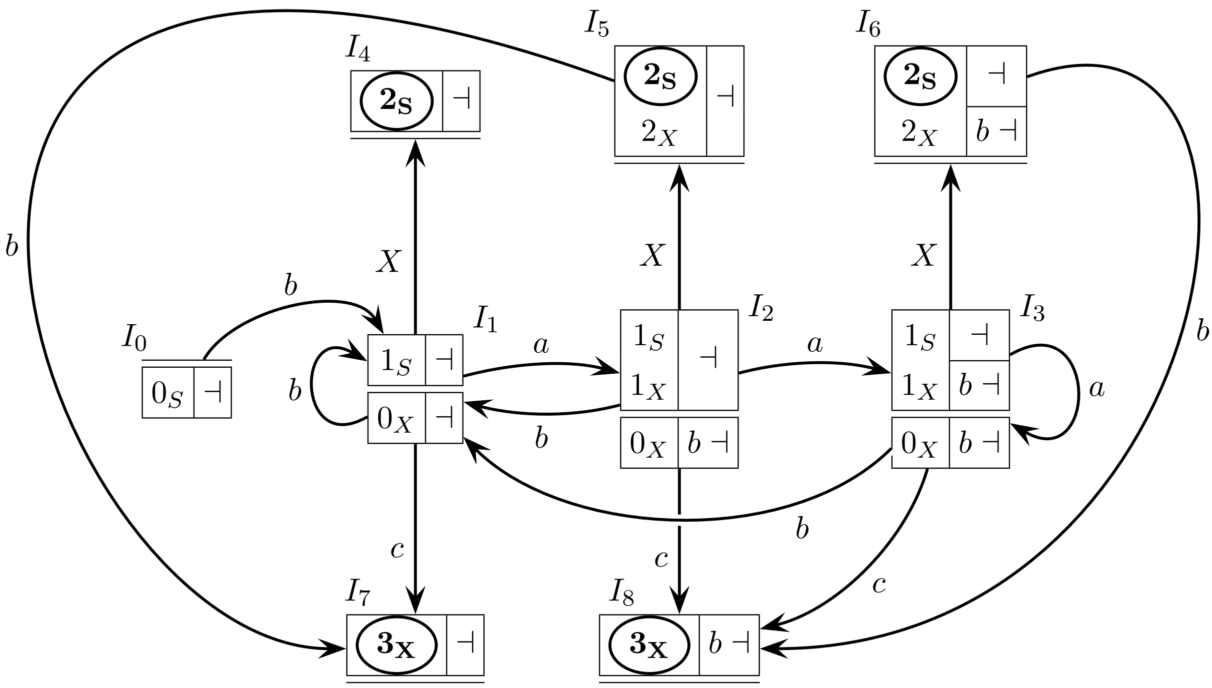
(the number of rows is not significant)

0	b	1	a	2	c	3	b	4

space for finishing question (c) and answering question (d)

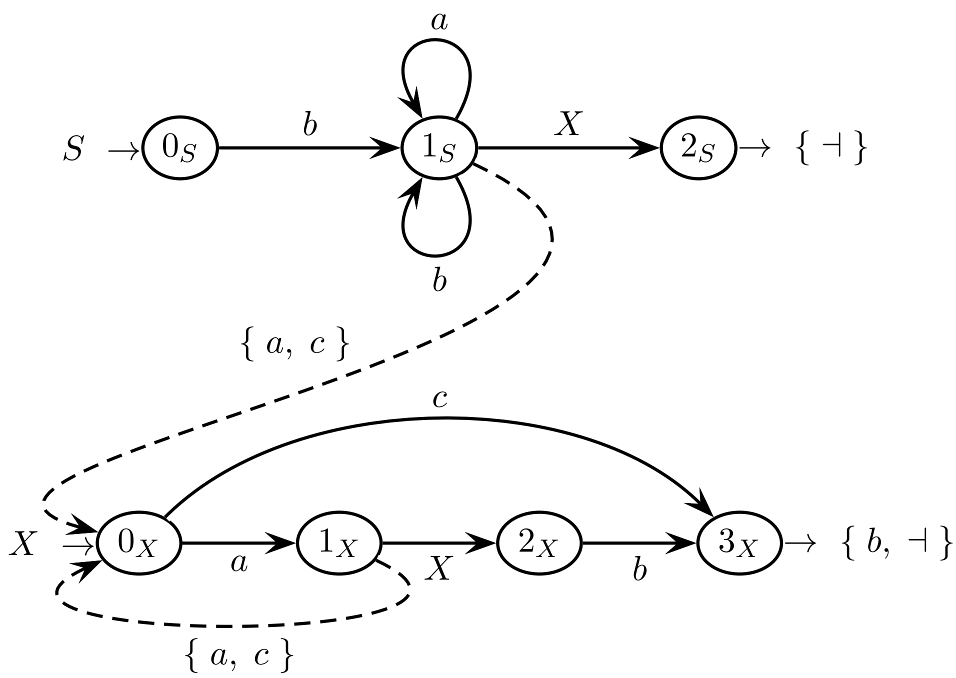
Solution

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



The pilot is conflict-free. Therefore the grammar is $ELR\,(1)$. There are five double transitions, e.g., $I_1 \xrightarrow{a} I_2$, and none of them is convergent.

(b) Guide sets (those on terminal shift arcs are obvious):

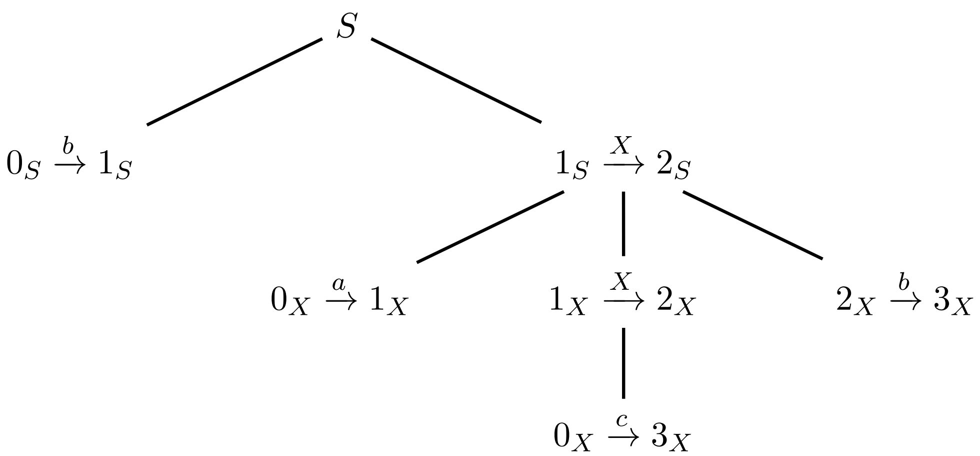


The grammar is not $ELL(1)$, as there is a conflict between a call arc and a terminal shift arc in the state 1_S on terminal a. This is the only conflict, anyway.

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

0		b	1		a	2		c	3		b	4	
0_S	0		1_S	0		1_S	0		$\mathbf{3_X}$	2		$\mathbf{3_X}$	1
			0_X	1		1_X	1		$\mathbf{2_S}$	0		$\mathbf{2_S}$	0
						0_X	2		2_X	1			

The acceptance condition is satisfied in the last Earley state. Notice that it is also satisfied in the state 3, and in fact the prefix string $b\,a\,c$ is valid as well. The tree is:



and it is the only tree, as the string is not ambiguous (nor is the grammar itself).

(d) A straightforward way to make nonterminal X nullable, is to slightly change machine M_X and make the initial state 0_X final, too. If state 0_X is nullable, in the pilot (the structure of which does not change significantly) there are shift-reduce conflicts (mark 0_X as final and see such conflicts in the m-states I_2 and I_3), thus the grammar is not $ELR\,(1)$. The structural reason is that the grammar becomes ambiguous, and all the valid strings of type $b\,(\,a \mid b\,)^*\,a^n\,b^n$ (with $n \geq 1$) are ambiguous.

4 Language Translation and Semantic Analysis 20%

1. Answer the following questions:

   (a) Consider a source language consisting of a fragment of a programming language that includes conditional and assignment instructions, as modeled by the following grammar G_1 (axiom I):

$$G_1 \begin{cases} I \rightarrow a \mid C \\ C \rightarrow \mathsf{if}\ E\ \mathsf{then}\ I\ \mathsf{else}\ I\ \mathsf{fi} \\ E \rightarrow \mathsf{ce} \end{cases}$$

   where terminals a and ce represent assignment statements and conditional expressions, respectively.

   Write a *BNF* translation scheme or grammar (without changing the source grammar) that defines the translation function τ_1 exemplified below:

$$\tau_1\ (\mathsf{if\ ce\ then\ if\ ce\ then}\ a\ \mathsf{else}\ a\ \mathsf{fi\ else}\ a\ \mathsf{fi}) =$$

$$\mathsf{if\ (ce)\ if\ (ce)}\ a\ \mathsf{other}\ a\ \mathsf{endif\ other}\ a\ \mathsf{endif}$$

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

   (b) Now consider another grammar G_2 that models a conditional instruction (nonterminal C, which is also the axiom) that includes a non-empty list of alternatives (nonterminal L):

$$G_2 \begin{cases} C \rightarrow \mathsf{if}\ E\ \mathsf{then}\ a\ \mathsf{else}\ L\ \mathsf{fi} \\ E \rightarrow \mathsf{ce} \\ L \rightarrow \mathsf{case}\ E : a\ ;\ L \\ L \rightarrow \mathsf{orelse}\ a \end{cases}$$

   Write a *BNF* translation scheme or grammar (without changing the source grammar) that defines the translation function τ_2 exemplified below:

$$\tau_2\ (\mathsf{if\ ce\ then}\ a\ \mathsf{else\ case\ ce} : a\ ;\ \mathsf{case\ ce} : a\ ;\ \mathsf{orelse}\ a\ \mathsf{fi}) =$$

$$\mathsf{if\ (ce)}\ a\ \mathsf{alt\ (ce)}\ a\ \mathsf{alt\ (ce)}\ a\ \mathsf{final}\ a\ \mathsf{endif}$$

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

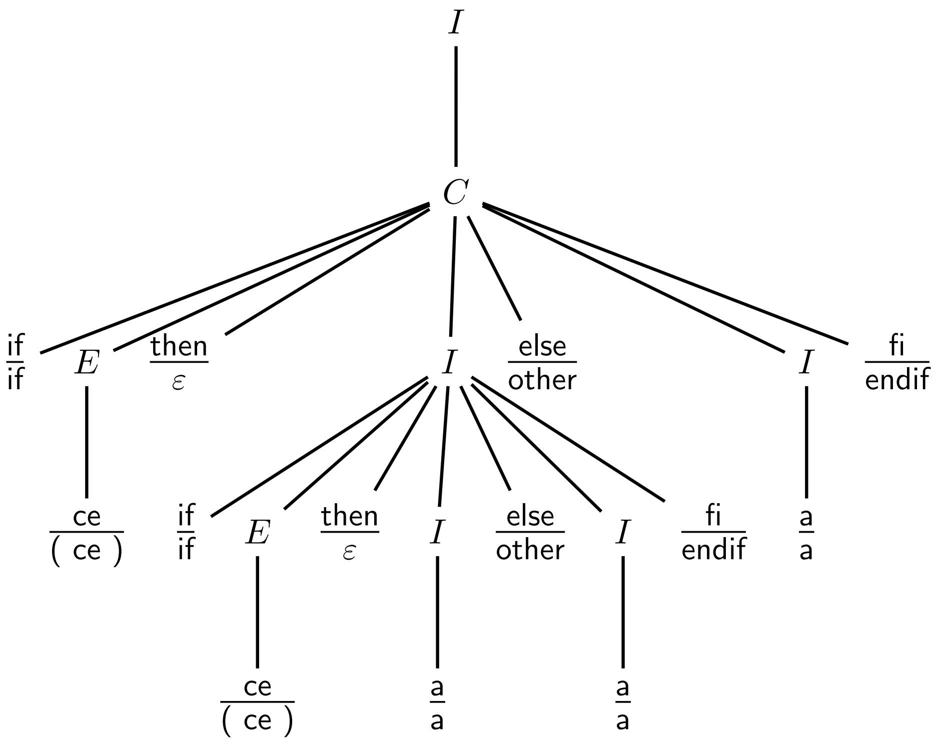
   (c) (optional) For each of the two translation schemes (or grammars) defined above, say if it can be defined by means of a finite translation model (a regular translation expression or a finite transducer). Suitably justify your answer; in particular, if a finite translation model exists, then design it.

Solution

(a) Here are the target (destination) grammar G_1^d (source G_1^s aside):

$$G_1^s \begin{cases} I \rightarrow a \mid C \\ C \rightarrow \mathsf{if}\ E\ \mathsf{then}\ I\ \mathsf{else}\ I\ \mathsf{fi} \\ E \rightarrow \mathsf{ce} \end{cases} \qquad G_1^d \begin{cases} I \rightarrow a \mid C \\ C \rightarrow \mathsf{if}\ E\ I\ \mathsf{other}\ I\ \mathsf{endif} \\ E \rightarrow \text{‘(’}\ \mathsf{ce}\ \text{‘)’} \end{cases}$$

and the translation tree of τ_1 for the sample string:

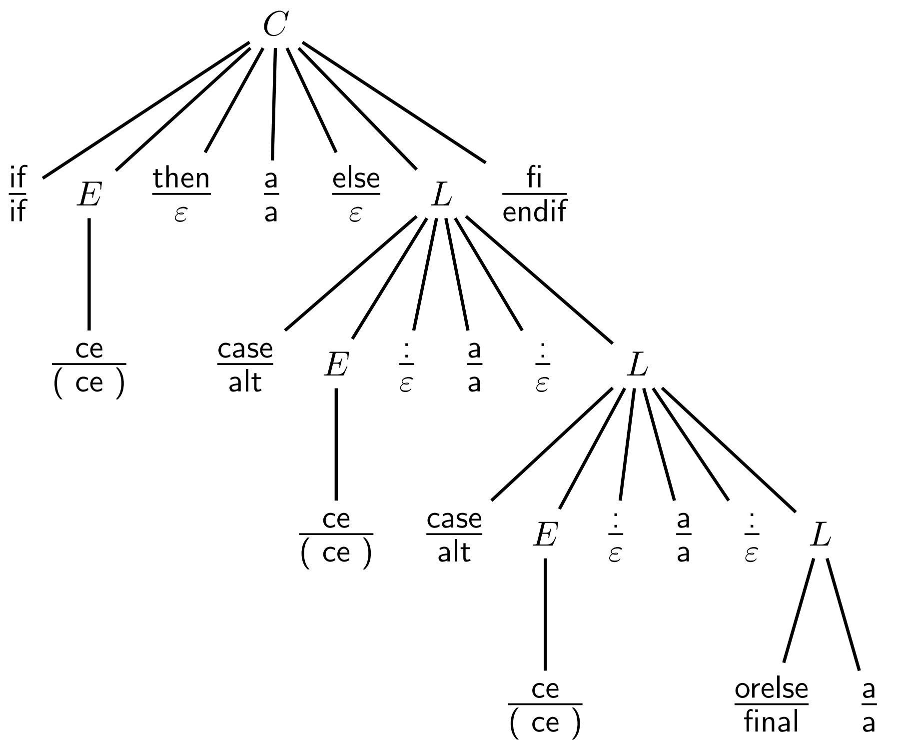


The tree is clearly correct (the keywords if, ce, etc, are considered as terminals).

(b) Here are the target (destination) grammar G_2^d (source G_2^s aside):

$$G_2^s \begin{cases} C \rightarrow \textsf{if}\ E\ \textsf{then}\ a\ \textsf{else}\ L\ \textsf{fi} \\ E \rightarrow \textsf{ce} \\ L \rightarrow \textsf{case}\ E : a\ ;\ L \\ L \rightarrow \textsf{orelse}\ a \end{cases} \qquad G_2^d \begin{cases} C \rightarrow \textsf{if}\ E\ a\ L\ \textsf{endif} \\ E \rightarrow \text{`('}\ \textsf{ce}\ \text{`)'} \\ L \rightarrow \textsf{alt}\ E\ a\ L \\ L \rightarrow \textsf{final}\ a \end{cases}$$

and the translation tree of τ_2 for the sample string:



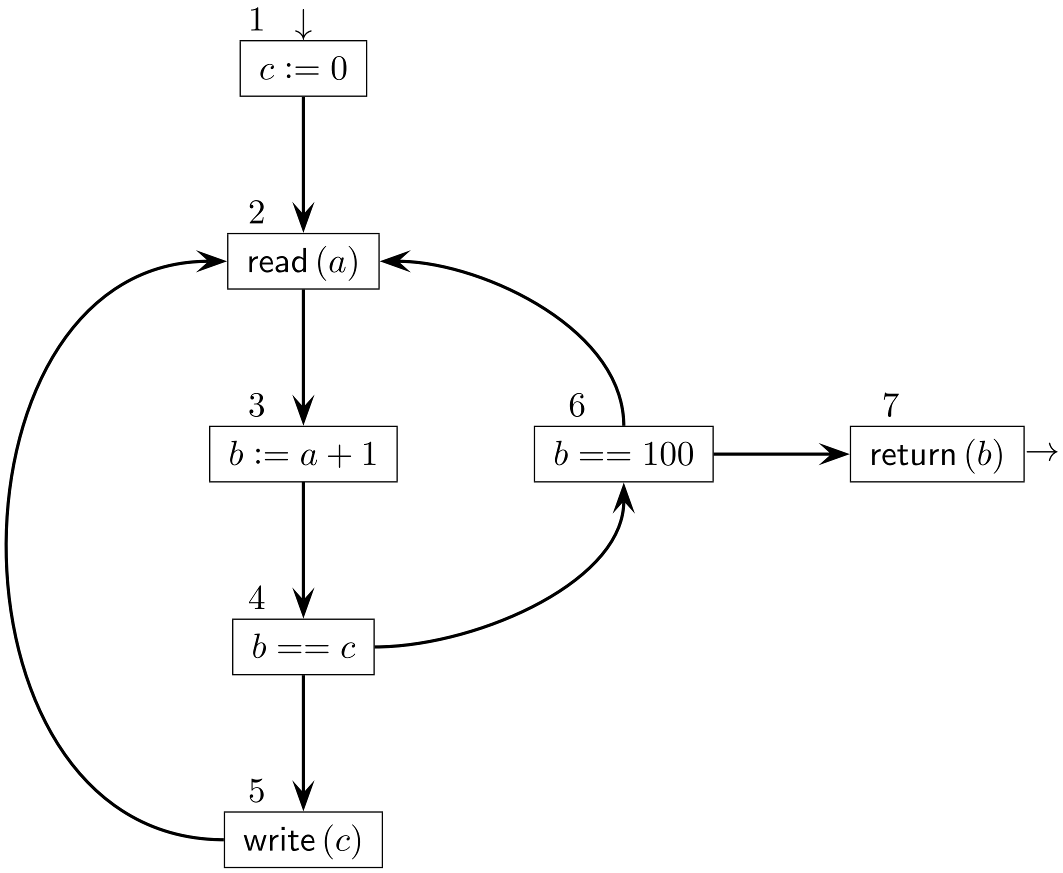
The tree is clearly correct (the keywords if, ce, etc, are considered as terminals).

(c) Grammar G_1 has an indirect self-embedding recursion on nonterminal I. This is not yet a sufficient condition for non-regularity (it is only necessary), yet it suggests that the source language may be non-regular. In fact, by examining the source language, one can see that it exhibits constructs of type $\textsf{if}^n \ldots \textsf{fi}^n$, which are not regular. Thus there is not any finite translation model for τ_1.
On the other hand, in grammar G_2 the sub-grammar for nonterminal L is right linear, and there is not any other recursion (see also the tree of question (b)). This is a sufficient condition for regularity. Thus translation τ_2 admits a finite model, for instance the following translation regular expression e_{τ_2} (some terminals are grouped):

$$e_{\tau_2} = \frac{\textsf{if ce then a else}}{\textsf{if (ce) a}} \left(\frac{\textsf{case ce : a ;}}{\textsf{alt (ce) a}} \right)^* \frac{\textsf{orelse a}}{\textsf{final a}}$$

which basically is a list, with separators, start marker and end marker. Of course it is possible to design a finite-state translator, at least non-deterministic (one might wish to examine whether a deterministic one also exists).

2. Consider the program Control Flow Graph (*CFG*) below, with seven nodes:



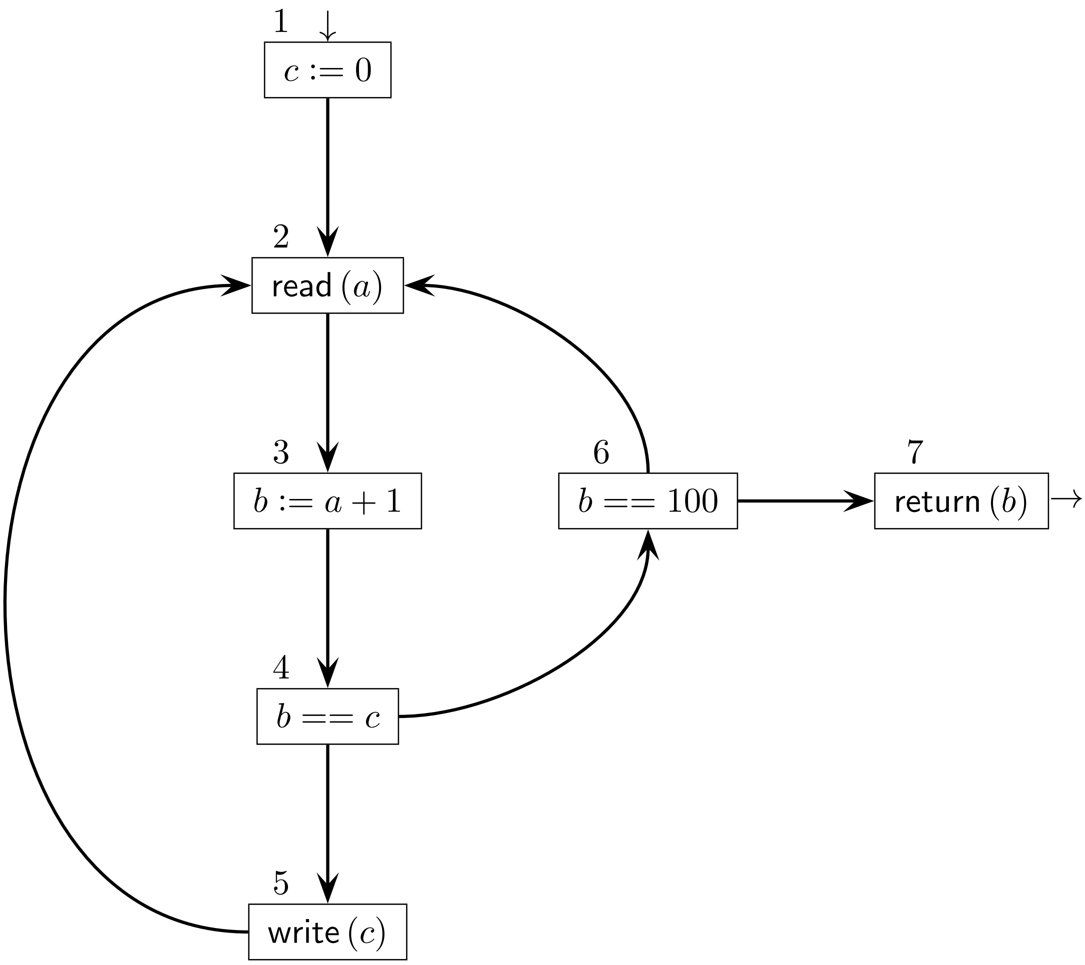
Answer the following questions (use the tables / trees / spaces on the next pages):

(a) Informally find the live variables at the input of each node of the *CFG*.

(b) Can variables a and b share the same memory cell (or processor register)? Explain why or why not.

(c) (optional) Find again the live variables at the input of each *CFG* node, through the data-flow equation method. Verify that the result is coherent with point (a).

decorate the program *CFG* with the live variables found informally – question (a)



space for answering question (b)

space for answering question (c)

tables of definitions and usages at the nodes

node	*defined*
1	
2	
3	
4	
5	
6	
7	

node	*used*
1	
2	
3	
4	
5	
6	
7	

system of data-flow equations

node	*in equations*	*out equations*
1		
2		
3		
4		
5		
6		
7		

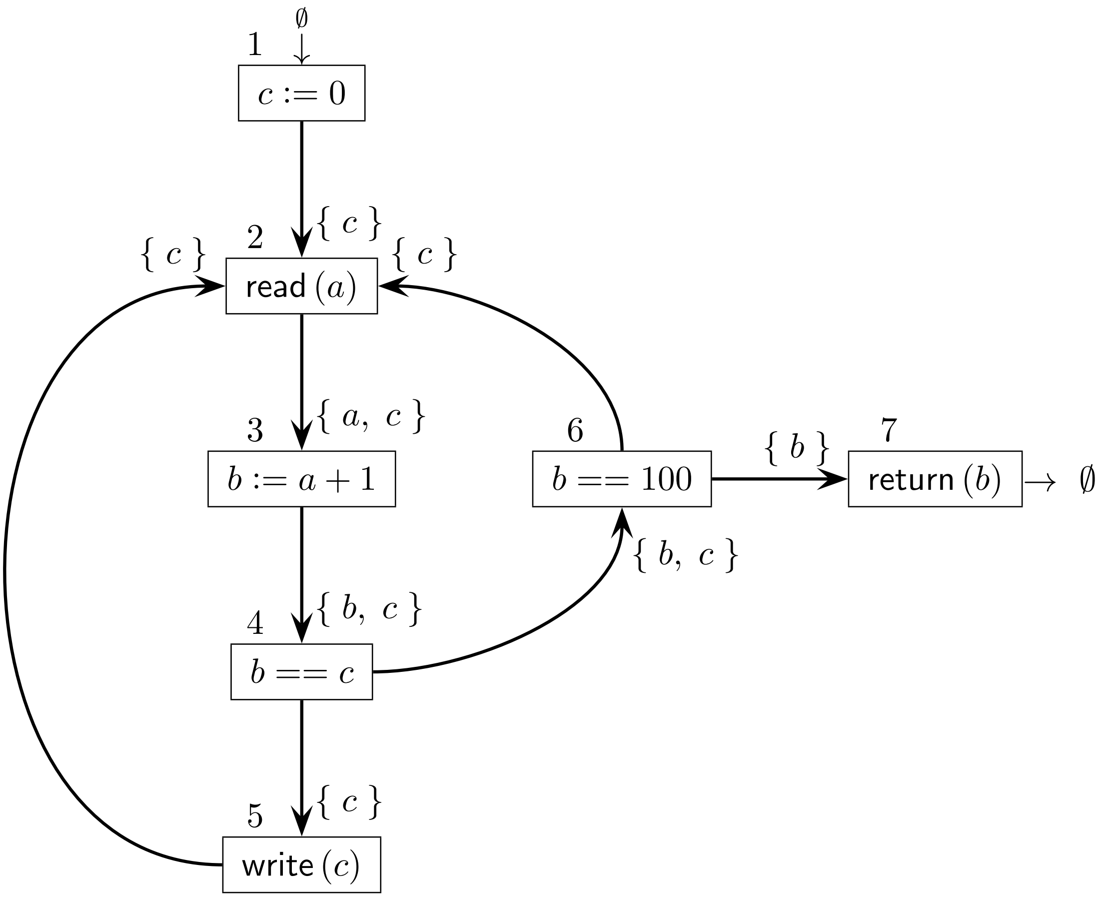
iterative solution table of the system of data-flow equations
(the number of columns is not significant and is more than sufficient to solve the equation system)

	initialization		1		2		3		4		5	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1												
2												
3												
4												
5												
6												
7												

	6		7		8		9		10		11	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1												
2												
3												
4												
5												
6												
7												

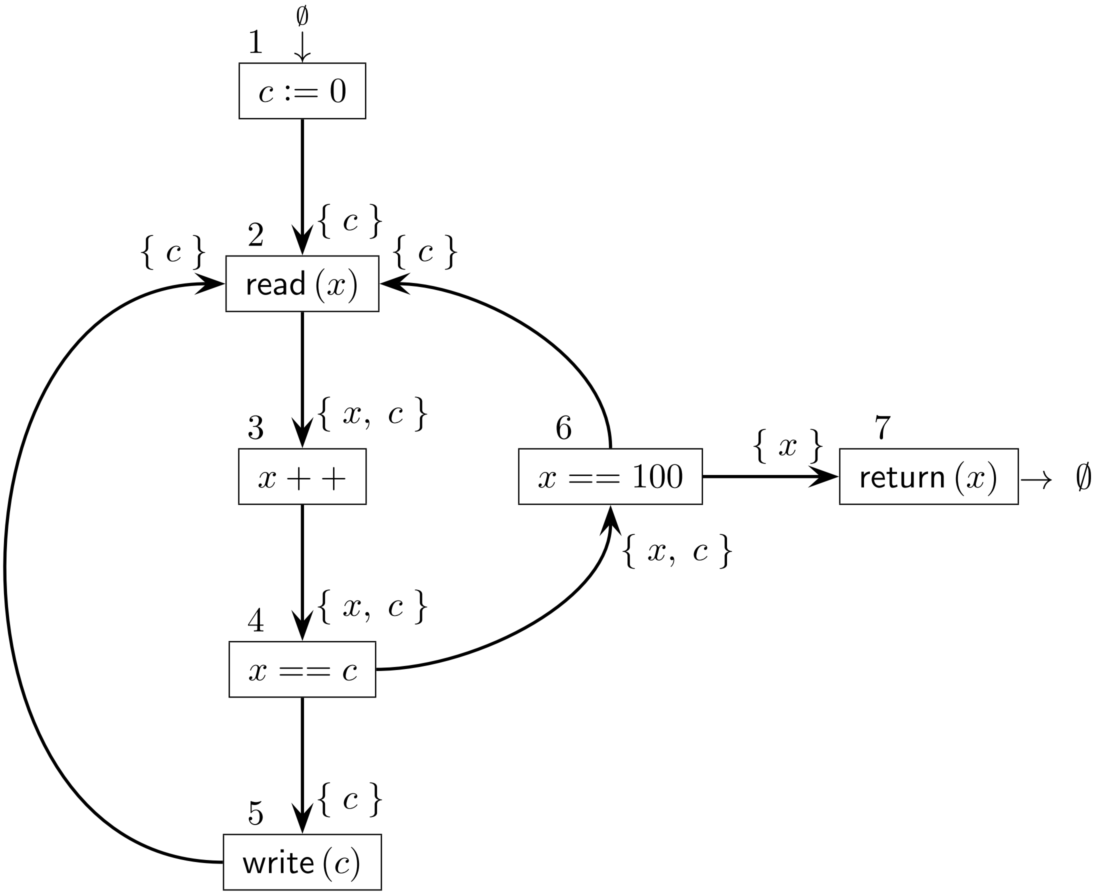
Solution

(a) Here are the live variables (at the node inputs), found informally:



At the input of node 1 (initial) no variable is live, since every variable is assigned in the initial node itself or in some successor thereof before using. At the output of node 7 (final), by definition no variable is live. The other liveness intervals are just an application of the liveness definition. For instance, variable c is live at the inputs of all the nodes in the two loops ($2-3-4-5$ and $2-3-4-6$), since it is used inside both loops (in the node 4) and is never reassigned after being initialized in the node 1 outside the loops. Variables a and b are even simpler: a is assigned in 2 and is used in 3, thus it is live only at 3; and b is assigned in 3 and is used in 4, 6 and 7, respectively 1^{st}, 2^{nd} and 3^{rd} successor of 3, thus it is live at 4, 6 and 7 themselves, i.e., on the interval $4-6-7$. No variable happens to be live at the input of all of the six nodes from 2 to 7 (included).

(b) Yes, they can. In fact, variables a and b are not simultaneously live at any node, thus they can share the same memory cell or processor register. Consequently, such variables can be unified into one variable x, and the program becomes:



which is equivalent and even structurally superposable to the original one.

(c) Here is the systematic computation of the live variables:

tables of definitions and usages at the nodes

node	*defined*
1	c
2	a
3	b
4	–
5	–
6	–
7	–

node	*used*
1	–
2	–
3	a
4	$b,\ c$
5	c
6	b
7	b

system of data-flow equations

node	*in equations*	*out equations*
1	$in(1) = out(1) - \{ c \}$	$out(1) = in(2)$
2	$in(2) = out(2) - \{ a \}$	$out(2) = in(3)$
3	$in(3) = \big(out(3) - \{ b \} \big) \cup \{ a \}$	$out(3) = in(4)$
4	$in(4) = out(4) \cup \{ b,\ c \}$	$out(4) = in(5) \cup in(6)$
5	$in(5) = out(5) \cup \{ c \}$	$out(5) = in(2)$
6	$in(6) = out(6) \cup \{ b \}$	$out(6) = in(7) \cup in(2)$
7	$in(7) = out(7) \cup \{ b \}$	$out(7) = \emptyset$

iterative solution table of the system of data-flow equations

	initializ.		1		2		3		4	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1	–	–	–	–	–	–	c	–	c	
2	–	–	a	–	$a\ c$	c	$a\ c$	c	$a\ c$	
3	–	a	$b\ c$	$a\ c$	$b\ c$	$a\ c$	$b\ c$	$a\ c$	$b\ c$	
4	–	$b\ c$	$b\ c$	$b\ c$	$b\ c$	$b\ c$	$b\ c$	$b\ c$	$b\ c$	
5	–	c	–	c	–	c	c	c	c	
6	–	b	b	b	b	b	$b\ c$	$b\ c$	$b\ c$	
7	–	b	–	b	–	b	–	b	–	

The *out* columns at steps 3 and 4 coincide, thus the iteration process converges in three steps. The *in* column at step 3 lists all the variables live at the program node inputs. The live variable latest to reach a node input is c at node 6 (compare the *in* columns at steps 2 and 3). In fact, variable c propagates backward from node 4, where it is used, to node 6, and to do so it has to cross nodes 3 and 2 in succession, which just takes three steps in total. All the other cases are faster, for instance variable b immediately reaches the three nodes where it lives (4, 6 and 7). The systematic solution is identical to the informal one of point (a).