FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Fri 14 JUNE 2019 - 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 regular expressions e_1 and e_2 below:

$$e_1 = \big((a \mid b)\,(c \mid d) \big)^+$$

$$e_2 = \big((a \mid b)\,(b \mid c) \big)^+$$

over the four-letter alphabet $\Sigma = \{ a, b, c, d \}$.

Answer the following questions:

(a) By using the Berry-Sethi (BS) method, obtain two deterministic automata A_1 and A_2 equivalent to the two above regular expressions e_1 and e_2, respectively. For each of the obtained automata A_1 and A_2, say if it is minimal (explain why yes or no) and minimize it, if necessary.

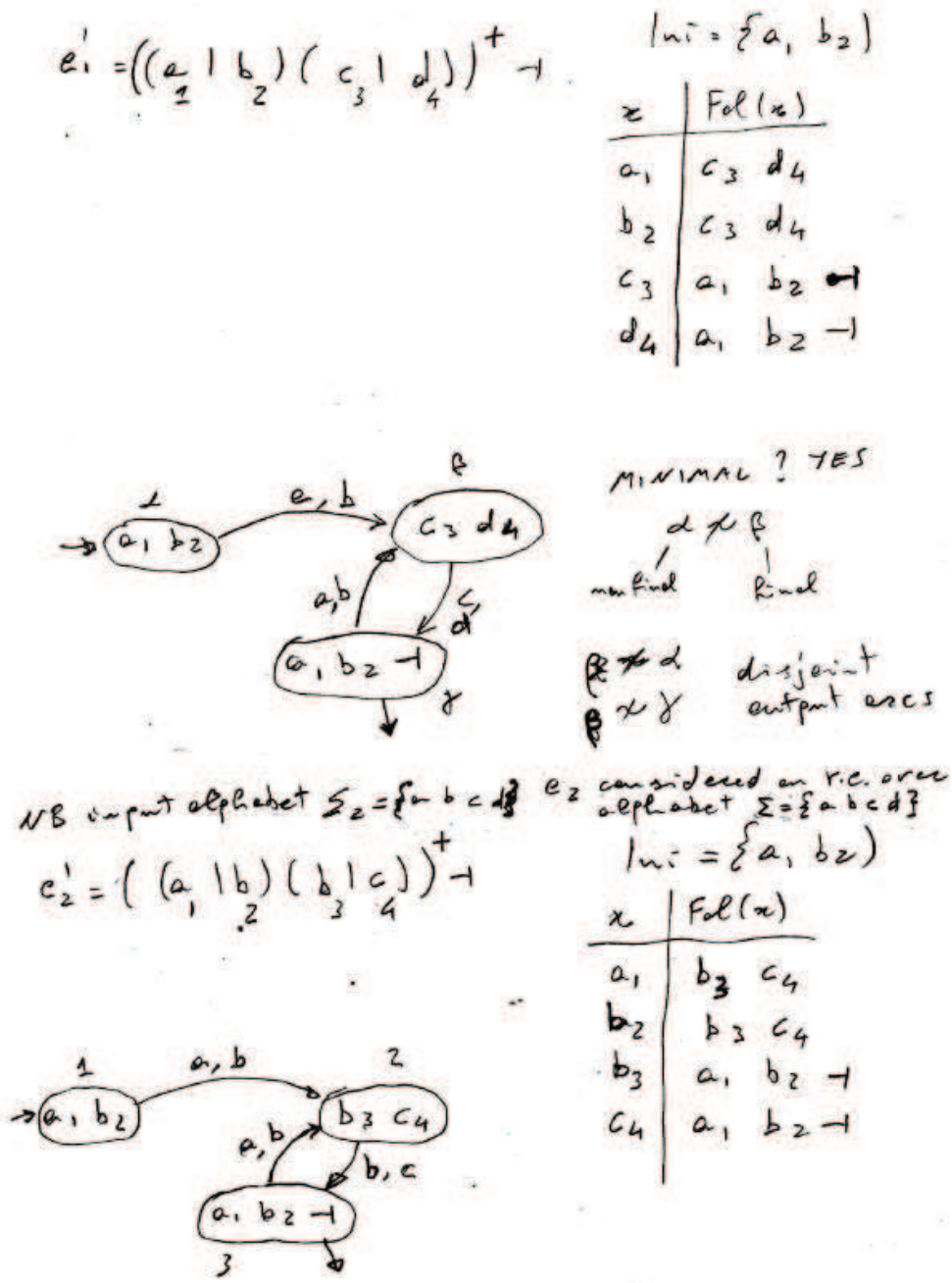
(b) Starting from automata A_1 and A_2, build the product automaton $A_\cap$ that accepts the intersection language $L(e_1) \cap L(e_2)$.

(c) Starting from the product automaton $A_\cap$ and by using the node elimination method (BMC), derive a regular expression $e_\cap$ equivalent to $A_\cap$.

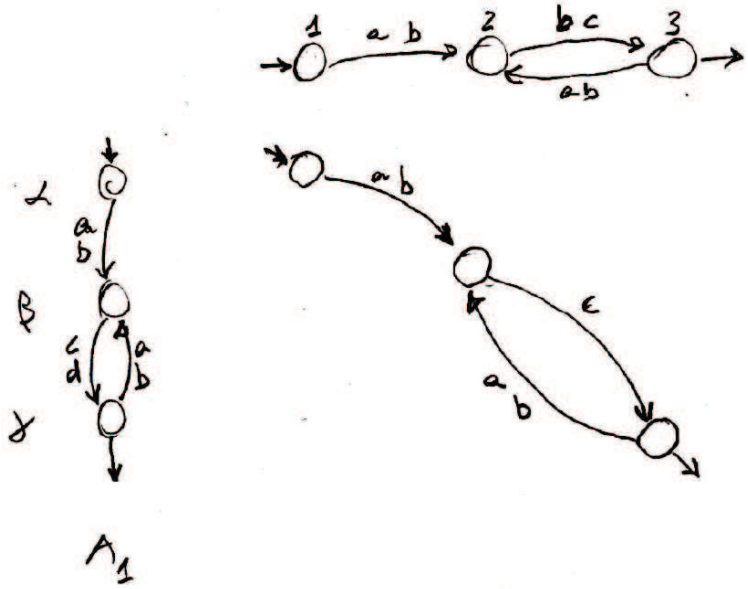
(d) (optional) By using a systematic method, obtain from the product automaton $A_\cap$ an equivalent nullable unilinear grammar G. Then, from this grammar G, derive a system of language equations, and solve the system to obtain a regular expression e_G for the language generated by grammar G.

Solution

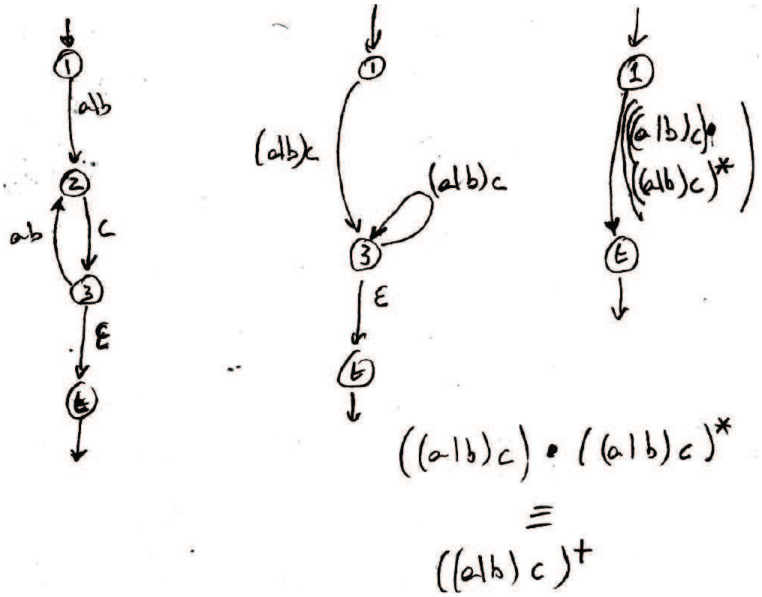
(a) Deterministic automata A_1 and A_2:



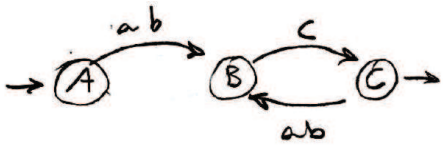
(b) Product automaton $A_\cap$:



(c) Regular expression $e_\cap$:



(d) Language equations:



The automaton is translated into the following nullable unilinear grammar:

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

which in turn is translated into the set of language equations below:

i. $L_A = a\,L_B \cup b\,L_B$
ii. $L_B = c\,L_C$
iii. $L_C = a\,L_B \cup b\,L_B \cup \{\,\varepsilon\,\}$

Substitute eq. (*ii*) into (*iii*) and obtain:

$$L_C = a\,c\,L_C \cup b\,c\,L_C \cup \{\,\varepsilon\,\} = (\,a\,c \mid b\,c\,)\,L_C \cup \{\,\varepsilon\,\}$$

from which, by the Arden identity:

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

From this and eq. (*ii*):

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

and, by substitution into eq. (*i*):

$$L_A = a\,L_B \cup b\,L_B = (\,a \mid b\,)\,L_B = (\,a \mid b\,)\,c\,\big((\,a \mid b\,)\,c\big)^*$$

which can be simply written as:

$$L_A = \big((\,a \mid b\,)\,c\big)^+$$

2 Free Grammars and Pushdown Automata 20%

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

$$L = \left\{\ a^h\ b^k \mid \quad 0 \leq h \leq k \leq 2h\ \right\}$$

Valid strings: ε, $a\,b$, $a\,b^2$, $a^2\,b^2$, $a^2\,b^3$, $a^2\,b^4$

Invalid strings: a, b, $b\,a$, $a^2\,b$, $a^2\,b^5$

Answer the following questions:

(a) Write a *BNF* grammar G, no matter if ambiguous, that generates language L. Test grammar G: draw the syntax trees of the valid strings above, and argue that those of the invalid strings are impossible.

(b) Say if the grammar G found before is ambiguous or not, and justify your answer. In the case grammar G is ambiguous, write an equivalent non-ambiguous *BNF* grammar G'.

(c) Consider the complement language $\overline{L}$ and write a description of $\overline{L}$ as a string set (or a union of string sets), similarly to language L.

(d) (optional) Argue whether the complement language $\overline{L}$ found before is context-free or not. In the case language $\overline{L}$ is context-free, write a *BNF* grammar $\overline{G}$ (preferably non-ambiguous) that generates it.

Solution

(a) Grammar G (ambiguous) (axiom S):

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

Trees: TBD.

(b) Grammar G is ambiguous as in a derivation (or tree) rules 2 and 3 can be commuted without changing the generated string (or tree frontier).
Grammar G' (non-ambiguous) (axiom S):

$$G' \begin{cases} S \to \varepsilon \\ S \to a\ S\ b \\ S \to Z \\ Z \to a\ b\ b \\ Z \to a\ Z\ b\ b \end{cases}$$

Grammar G' is non-ambiguous: the rule commutation is impossible; notice that nonterminal Z is non-nullable, to avoid ambiguity for ε.

(c) Description of the complement language $\overline{L}$:

$$\overline{L} = \underbrace{\Sigma^*\ b\ a\ \Sigma^*}_{\text{order change}} \cup \Big\{ \quad \underbrace{a^r\ b^s}_{\text{bad count}} \quad | \quad \overbrace{\underbrace{0 \le s < r}_{\text{too few } b\text{'s}} \vee \underbrace{0 \le 2r < s}_{\text{too many } b\text{'s}}}^{\text{characteristic predicate}} \Big\}$$

The first component is regular, for the strings with a letter b that occurs before a letter a. The other two components (one per each part of the characteristic predicate) are for the strings with a wrong counting of letters b.

(d) Language $\overline{L}$ is context-free: the first component is regular, the other two are well-known to be free, and $\overline{L}$ is the finite union thereof. Here is a grammar $\overline{G}$ (axiom S):

$$\overline{G} \begin{cases} S \to X\ b\ a\ X \mid U \mid V \\ X \to a\ X \mid b\ X \mid \varepsilon \\ U \to a\ U\ b \mid a\ U \mid a \\ V \to a\ V\ b\ b \mid V\ b \mid b \end{cases}$$

The language union for $\overline{L}$ above is non-ambiguous, as the three sub-languages are mutually disjoint. Yet grammar $\overline{G}$ is ambiguous, due to the rule $X\ b\ a\ X$ (the leftmost letter pair $b\,a$ can be generated by the left instance of X as well), to the rules $U \to a\ U\ b \mid a\ U$ (commute alternatives) and to the rules $V \to$

$a\ V\ b\ b\ \mid\ V\ b$ (idem). Anyway, it is easy to disambiguate $\overline{G}$:

$$\overline{G}_{\text{non-amb.}} \begin{cases} S \rightarrow X'\ b\ a\ X \mid U \mid V \\ X' \rightarrow a\ X' \mid \varepsilon \\ X \rightarrow a\ X \mid b\ X \mid \varepsilon \\ U \rightarrow a\ U\ b \mid U' \\ U' \rightarrow a\ U' \mid a \\ V \rightarrow a\ V\ b\ b \mid V' \\ V' \rightarrow V'\ b \mid b \end{cases}$$

For nonterminals U and V, the disambiguation is similar to that of grammar G.

2. A *matQuadTree* is a data structure that recursively encodes a $2^n \times 2^n$ matrix of numbers. The matrix is represented as a tree of depth $n \geq 1$, the internal nodes of which have a maximum of four child nodes. Each square region of the matrix can be represented as follows:

   - A leaf node (keyword *val*) that includes one number, if all the elements of the region have the same value. This includes also the case of a region consisting of one element, i.e., a 1×1 submatrix.
   - A node (keyword *fourSplit*) with four child nodes, each of these being the root of a subtree that represents one of the four squares obtained by splitting the region vertically and horizontally into four parts of the same size. These four parts are listed counterclockwise and are denoted by the keywords *tr* (top right), *tl* (top left), *bl* (bottom left) and *br* (bottom right).
   - A node with two subtrees when the square region is split, either horizontally or vertically (keywords *verticalSplit* or *horizontalSplit*), into two rectangular regions. Each of these regions is composed of two square regions with the same contents.

   We report below a 8×8 matrix named *alpha* and its encoding as a *matQuadTree*:

<table>
<tr><td colspan="3" rowspan="3">5</td><td>4</td><td colspan="2">4</td></tr>
<tr><td>5</td><td>7</td><td>5</td><td>7</td></tr>
<tr><td>1</td><td>3</td><td>1</td><td>3</td></tr>
<tr><td>6</td><td>7</td><td rowspan="2">1</td><td rowspan="2">3</td><td colspan="2" rowspan="2">2</td></tr>
<tr><td>3</td><td>5</td></tr>
<tr><td>6</td><td>7</td><td rowspan="2">1</td><td rowspan="2">7</td><td colspan="2" rowspan="2">9</td></tr>
<tr><td>3</td><td>5</td></tr>
</table>

```
matQuadTree alpha is
   fourSplit
      tr: horizontalSplit
             top: val 4;
             bottom: fourSplit
                        tr: val 7;  tl: val 5;  bl: val 1;  br: val 3;
                     end fourSplit;
          end horizontalSplit;
      tl: val 5;
      bl: verticalSplit
             right: val 1;
             left: fourSplit
                      tr: val 7;  tl: val 6;  bl: val 3;  br: val 5;
                   end fourSplit;
          end verticalSplit;
      br: fourSplit
             tr: val 2;  tl: val 3;  bl: val 7;  br: val 9;
          end fourSplit;
   end fourSplit;
end matQuadTree alpha
```

Please infer from this sample script any information that is not explicitly given above, in particular any remaining symbols for keywords, delimiters and their placement, etc.

(a) Write a grammar, possibly of type *EBNF* and not ambiguous, that generates the script language described above, where a phrase consists of one *matQuadTree*.

(b) Modify the above grammar by imposing the constraint that the tree nodes of type *verticalSplit* and *horizontalSplit* may not have a child node of type *fourSplit*.

In the grammar, schematize the identifiers and numbers by terminals id *and* num*, respectively, without expanding them with rules.*

Solution

(a) Here is a viable grammar, of type *EBNF* and not ambiguous (axiom MQ_DEF):

⟨MQ_DEF⟩ → matQuadTree id is ⟨SPLIT⟩ ; end matQuadTree id

⟨SPLIT⟩ → ⟨F_SPL⟩ | ⟨V_SPL⟩ | ⟨H_SPL⟩ | ⟨VAL⟩

⟨F_SPL⟩ → fourSplit
tr : ⟨SPLIT⟩ ;
tl : ⟨SPLIT⟩ ;
bl : ⟨SPLIT⟩ ;
br : ⟨SPLIT⟩ ;
end fourSplit

⟨V_SPL⟩ → verticalSplit
left : ⟨SPLIT⟩ ;
right : ⟨SPLIT⟩ ;
end verticalSplit

⟨H_SPL⟩ → horizontalSplit
top : ⟨SPLIT⟩ ;
bottom : ⟨SPLIT⟩ ;
end horizontalSplit

⟨VAL⟩ → val num

(b) Just remove alternative *fourSplit* from the relevant rules (in red):

⟨MQ_DEF⟩ → matQuadTree id is ⟨SPLIT⟩ ; end matQuadTree id

⟨SPLIT⟩ → ⟨F_SPL⟩ | ⟨V_SPL⟩ | ⟨H_SPL⟩ | ⟨VAL⟩

⟨HALF⟩ → ⟨V_SPL⟩ | ⟨H_SPL⟩ | ⟨VAL⟩

⟨F_SPL⟩ → fourSplit
tr : ⟨SPLIT⟩ ;
tl : ⟨SPLIT⟩ ;
bl : ⟨SPLIT⟩ ;
br : ⟨SPLIT⟩ ;
end fourSplit

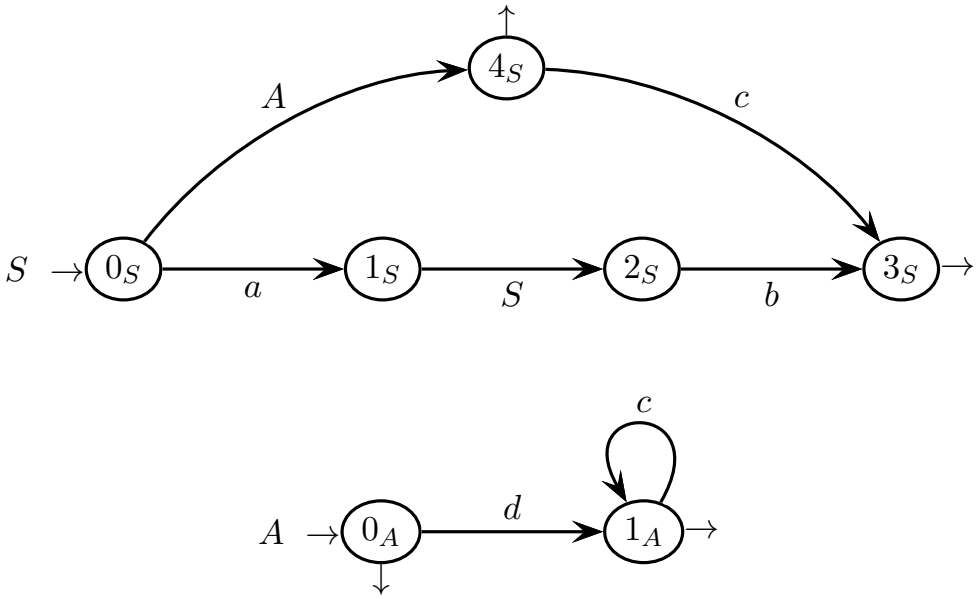
⟨V_SPL⟩ → verticalSplit
left : ⟨HALF⟩ ;
right : ⟨HALF⟩ ;
end verticalSplit

⟨H_SPL⟩ → horizontalSplit
top : ⟨HALF⟩ ;
bottom : ⟨HALF⟩ ;
end horizontalSplit

⟨VAL⟩ → val num

3 Syntax Analysis and Parsing Methodologies 20%

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



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

(a) Draw the complete pilot of grammar G and determine, based on the pilot, whether grammar G is of type $ELR(1)$. Suitably explain your answer.

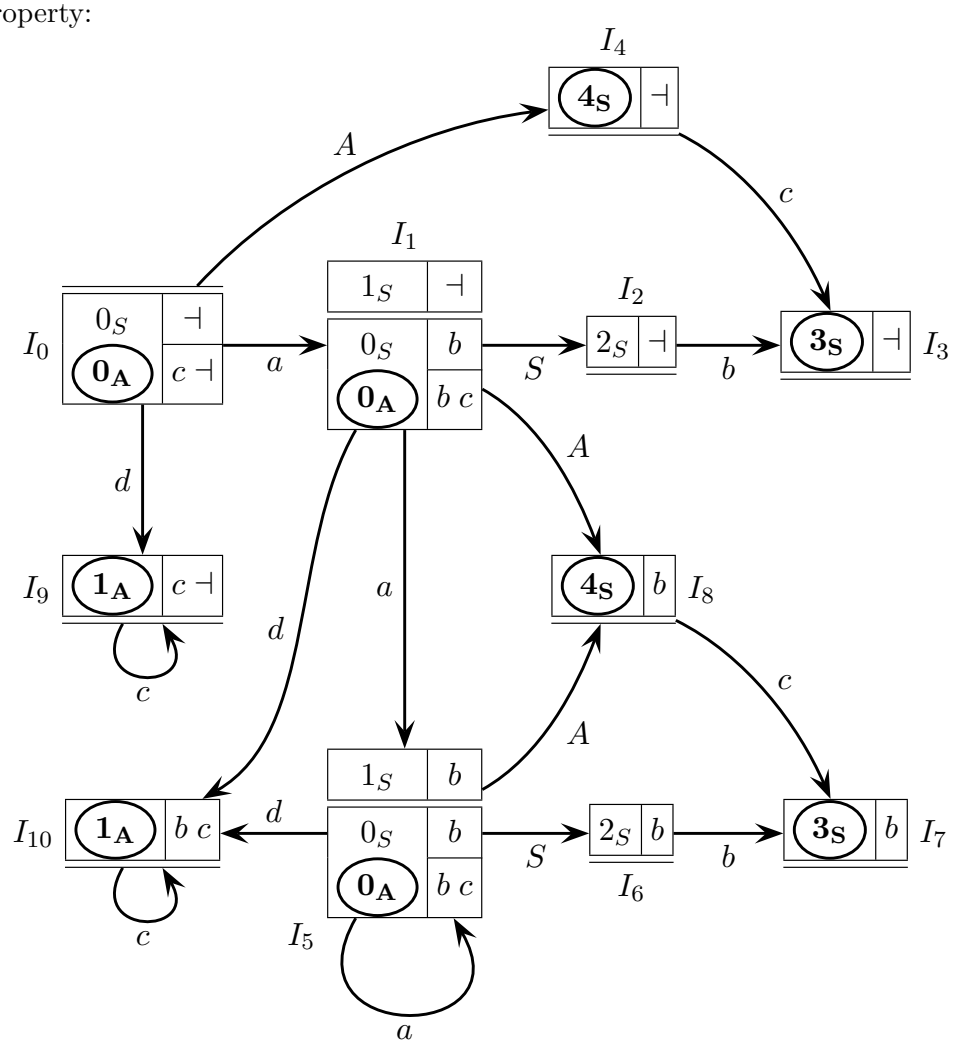
(b) Find all the guide sets of the machine net of grammar G (on the call arcs and exit arrows) and, based on such sets, show that grammar G is not of type $ELL(1)$. Suitably explain your answer.

(c) Determine whether grammar G is of type $ELL(k)$ for some $k \geq 2$. Suitably explain your answer.

(d) (optional) Examine language $L(G)$ and describe it in a mathematically precise way as a set of strings. Is it regular or not ? Suitably explain your answer.

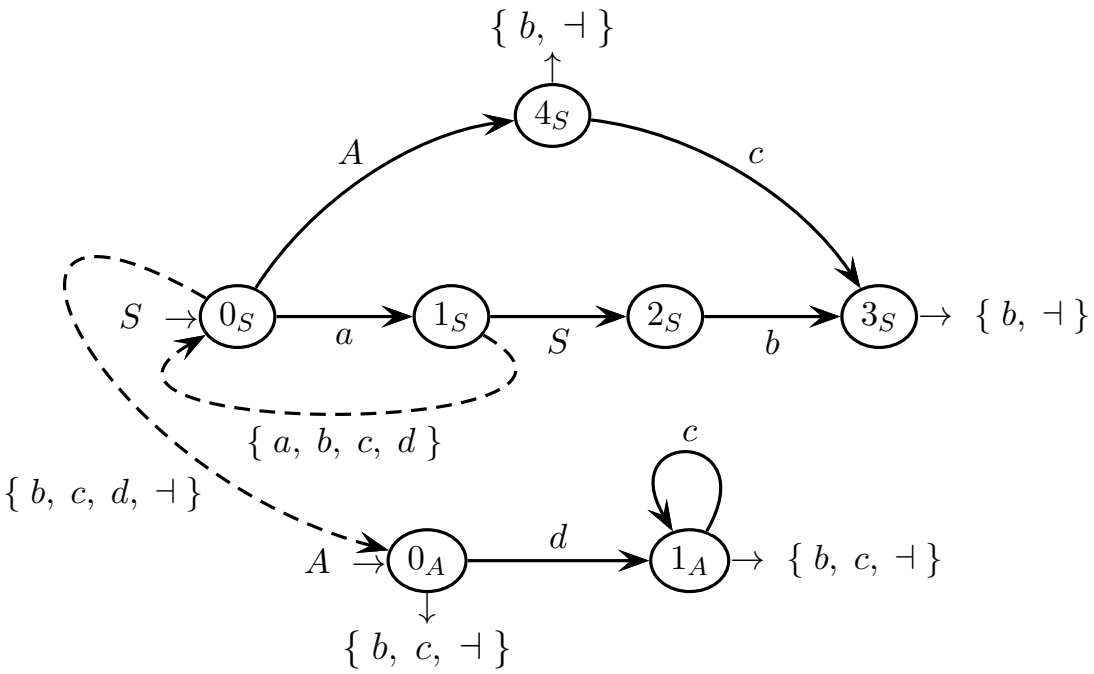
Solution

(a) Here is the pilot of grammar G, with 11 m-states, which disproves the $ELR\,(1)$ property:



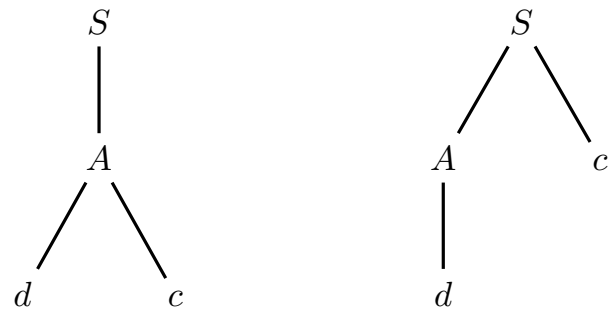
There are only two (identical) shift-reduce conflicts in the two m-states I_9 and I_{10} of the final state 1_A, on terminal c. Thus grammar G is not $ELR\,(1)$.

(b) Notice that both nonterminals A and S are nullable. Here are all the guide sets on the call arcs and exit arrows of the machine net of grammar G:



There is only one guide conflict conflict on state 1_A, on terminal c. Thus grammar G is not $ELL\,(1)$. This was expected, of course, since grammar G is not $ELR\,(1)$.

(c) The guide set conflict on state 1_A is unsolvable for every $k \geq 2$, as grammar G is ambiguous. For instance, string $d\,c \in L\,(G)$ has two syntax trees:



(d) Since it holds $L\,(A) = [\,d\ c^*\,]$ (the square bracket mean optionality), and the alternative $S \to a\ S\ b$ is self-embedding and ends with A optionally followed by c, i.e., $S \to A\ [\,c\,]$, it holds $L = \{\ a^n\ [\,d\ c^*\,]\ [\,c\,]\ b^n \mid \quad n \geq 0\,\}$. Clearly, language L is not regular because of the required equal number of letters a and b.

More formally, clearly it holds $L\ \cap\ L\,(\,a^*\,b^*\,) = \{\ a^n\ b^n \mid \quad n \geq 0\ \}$. Now, if language L were regular, also the intersection result should be regular, as regular languages are closed under intersection. Yet, since it is well known that language $a^n\ b^n$ is not regular, language L is not regular either.

4 Language Translation and Semantic Analysis 20%

1. Consider the two-letter terminal alphabet $\Sigma = \{\ a,\ b\ \}$, the set of digrams over Σ, i.e., $\Sigma^2 = \{\ a\,a,\ a\,b,\ b\,a,\ b\,b\ \}$, and a source language $L_s = \left(\Sigma^2\right)^*$, the strings of which consist of the concatenation of any number of such digrams, plus ε. We associate each digram of Σ^2 with a positive integer: $a\,a \leftrightarrow 1$, $a\,b \leftrightarrow 2$, $b\,a \leftrightarrow 3$ and $b\,b \leftrightarrow 4$.

   Answer the following questions:

   (a) Consider a translation $\tau_1\,(x)$ that maps each source string x of language L_s to the sequence of the integers that are associated with the concatenated digrams in the string x, in the same order as they appear in x. Thus, for instance:

   $$\tau_1\,(\underbrace{a\,b}_{2}\ \underbrace{b\,a}_{3}\ \underbrace{b\,b}_{4}) = 2\,3\,4$$

   Design a transducer, of the least powerful type, that computes translation τ_1.

   (b) Say if translation τ_1 is one-valued, and suitably justify your answer.

   (c) The source grammar G_s below (axiom S) generates language L_s as a Dyck language with parenthesis pairs of four types, which correspond to the digrams in the previous question:

   $$G_s \begin{cases} S \rightarrow \varepsilon & // \text{ pair type} \\ S \rightarrow a\ S\ a\ S & a\,a \\ S \rightarrow a\ S\ b\ S & a\,b \\ S \rightarrow b\ S\ a\ S & b\,a \\ S \rightarrow b\ S\ b\ S & b\,b \end{cases}$$

   Assume that the four types of parenthesis pairs are associated with integers as before: $a\,a \leftrightarrow 1$, $a\,b \leftrightarrow 2$, $b\,a \leftrightarrow 3$ and $b\,b \leftrightarrow 4$. Consider a translation $\tau_2\,(x)$ that maps each source string x of language L_s to a sequence of integers associated with the parenthesis pairs in the string x, as follows:

   - the integer associated with a parenthesis pair is written after writing the translation of the substring (if any) enclosed in the pair
   - the translations of concatenated parenthesis pairs are concatenated

   Thus, for instance ($b\,b$ is enclosed in $a\,a$ and $a\,b$ is concatenated to $a\,a$):

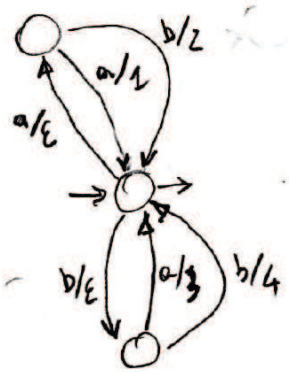
   $$\tau_2\,(\underbrace{a\ \underbrace{b\,b}_{4}\ a}_{1}\ \underbrace{a\,b}_{2}) = 4\,1\,2$$

   Write a translation scheme (or grammar) G_{τ_2} that defines the translation τ_2 described above. Do not change the source grammar G_s. Test scheme G_{τ_2} by drawing the syntax translation tree of the sample translation above.

   (d) (optional) Determine whether translation τ_2 can be computed deterministically, and reasonably motivate your answer.

Solution

(a) A simple finite-state transducer with three states suffices:



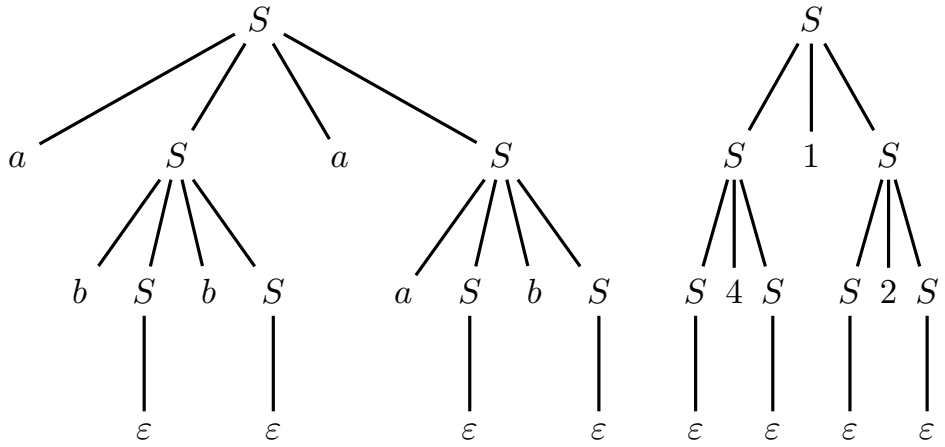
Of course, it could be written in the form of a unilinear translation grammar or scheme. The transducer is clearly deterministic.

(b) Translation τ_1 is one-valued, i.e., a function from strings to strings, because the decomposition of the source string into digrams is unique, and each digram is associated with a different integer. This is confirmed by the fact that the above finite transducer is deterministic.

(c) Here is the destination (or target) grammar G_t:

$$G_t \begin{cases} S \rightarrow \varepsilon \\ S \rightarrow S\,1\,S \\ S \rightarrow S\,2\,S \\ S \rightarrow S\,3\,S \\ S \rightarrow S\,4\,S \end{cases}$$

and the trees for the source and target strings:



(d) Translation τ_2 cannot be computed deterministically because the source grammar is ambiguous, and actually translation τ_2 is not even a function. In fact, consider the three possible Dyck decompositions below, for the valid string $a\,b\,b\,a\,a\,b$, and the related translation:

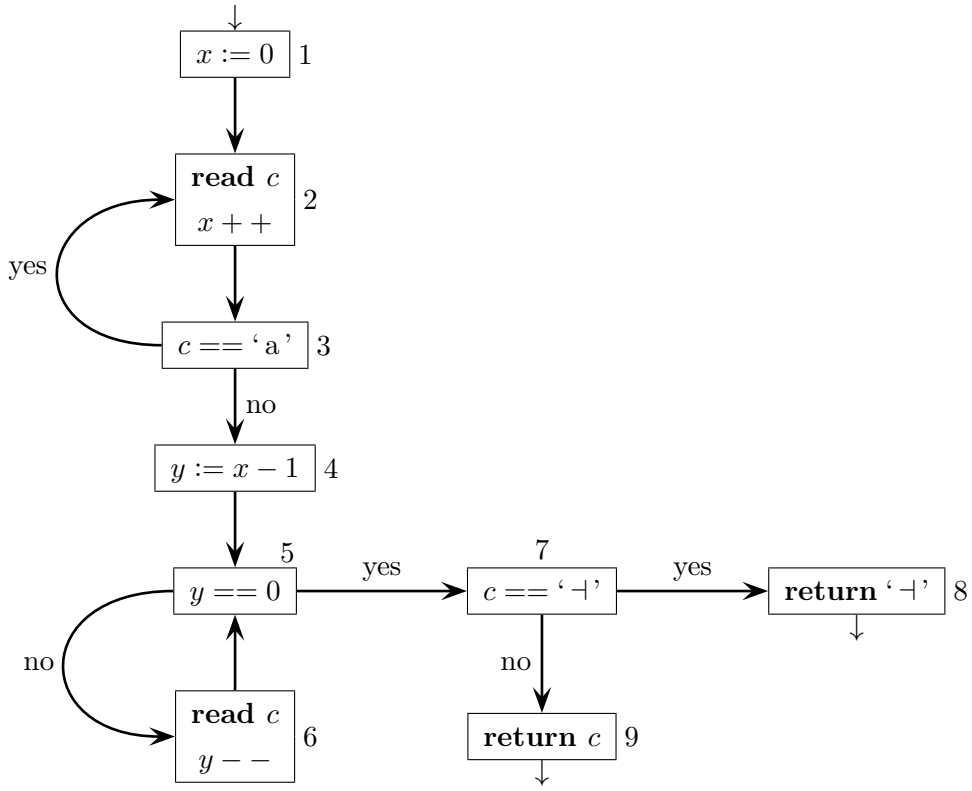
$\underbrace{a\,\underbrace{b\,\underbrace{b}}\,a\,\underbrace{a\,\underbrace{b}}}$ 4 1 2

$\underbrace{a\,\underbrace{b}}\,\underbrace{b\,\underbrace{a}}\,\underbrace{a\,\underbrace{b}}$ 2 3 2

$\underbrace{a\,\underbrace{b}}\,b\,\underbrace{a\,\underbrace{a}\,b}$ 2 1 4

The source string has three possible syntax trees, and consequently three possible translations. Therefore translation τ_2 is multi-valued, that is, as said, it is not a function from a string to one string.

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



The program has three variables: x and y (both integer), and c (character). By means of variable c, the program reads a one-way input tape, which contains a (possibly empty) string $w \in \Sigma^*$, with $\Sigma = \{\ a,\ b\ \}$. String w is terminated by $\dashv$, i.e., the input tape contains $w\ \dashv$. The first read operation executed sets the variable c to the initial character of w if $w \neq \varepsilon$, or to the terminator $\dashv$ if $w = \varepsilon$.

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

(a) Write the regular expression E that represents the execution traces of the *CFG*, disregarding the program semantics, over the node name alphabet $\{\ 1, \ldots, 9\ \}$.

(b) Informally find the live variables at each node and write them by each node (the live variables of a node are those live at the node input). Can the found liveness be exploited to optimize the memory allocation for the program ?

(c) Find the variables used and defined at each node. Then write the flow equations of the live variables, but only for the nodes 2, 4 and 5 (it is not required to solve such equations).

(d) (optional) Verify that the solution found at point (b) satisfies the flow equations of nodes 2, 4 and 5. Remember that each node has two equations: *in* and *out*.

Solution

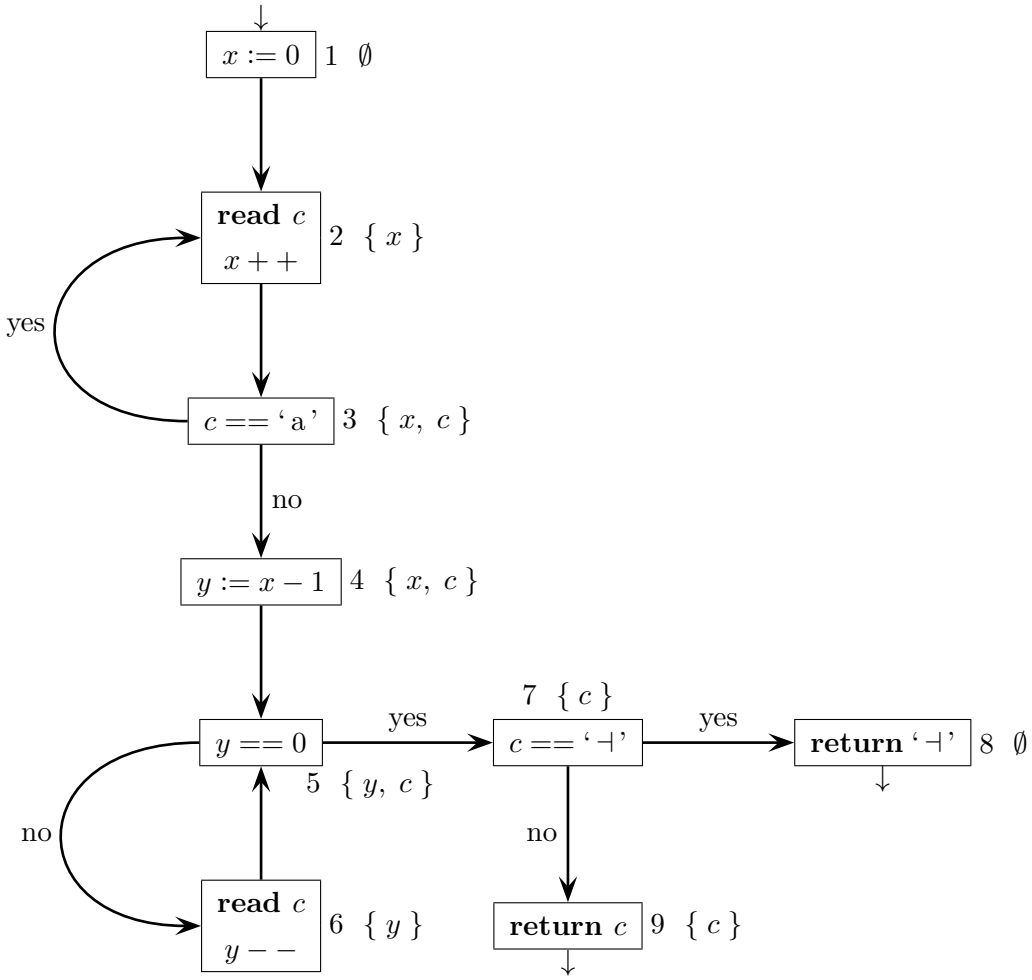
(a) Regular expression E, disregarding semantics:

$$E = 1\ (\ 2\ 3\ )^{+}\ 4\ 5\ (\ 6\ 5\ )^{*}\ 7\ (\ 8\ \mid\ 9\ )$$

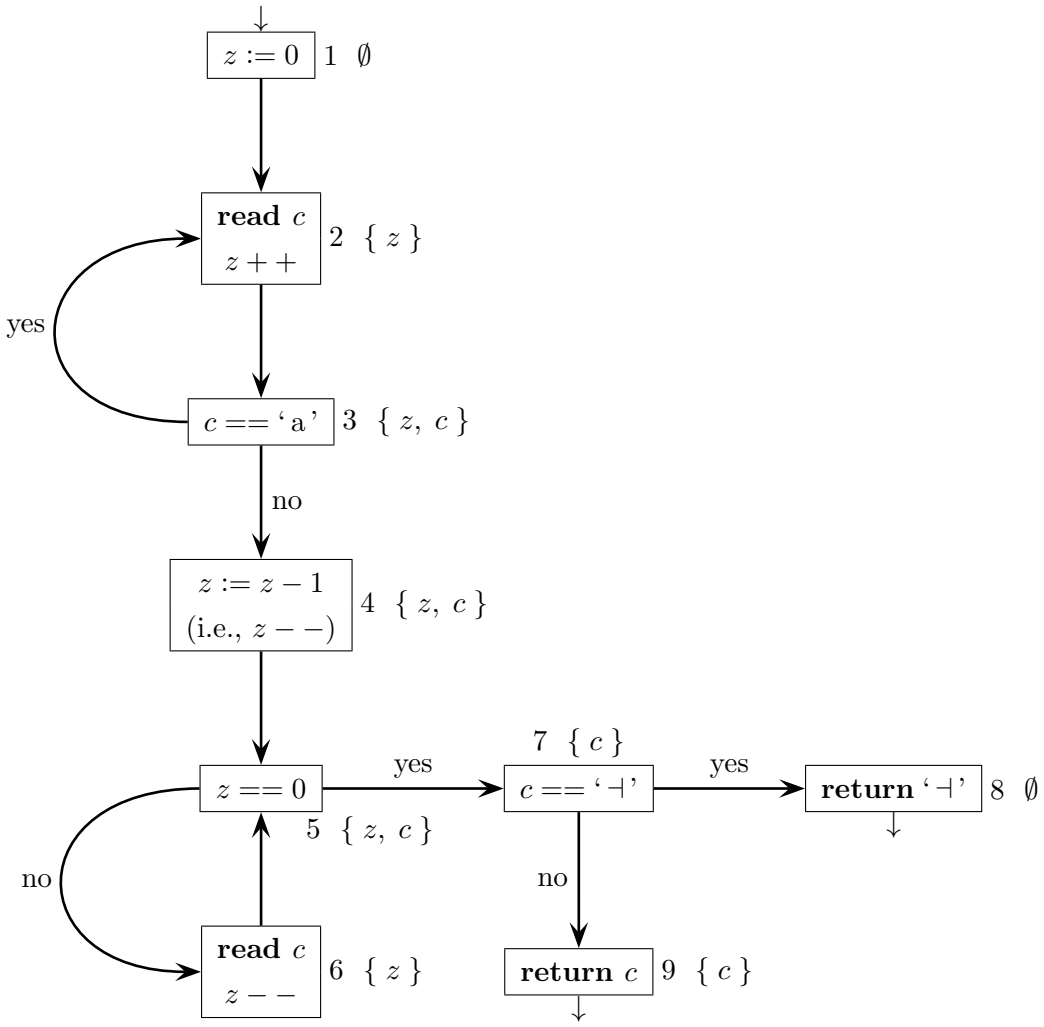
or, equivalently:

$$E = 1\ (\ 2\ 3\ )^{+}\ 4\ (\ 5\ 6\ )^{*}\ 5\ 7\ (\ 8\ \mid\ 9\ )$$

(b) Live variables:



The two variables x and y are never simultaneously live. Since both are of integer type, they can be directly unified into one variable, say z (this is to say they can share the same memory cell or processor register). Thus the program can be optimized as follows:



Writing $z--$ at node 4 would just be a short notation for the assignment $z := z - 1$, the operation being the same. The optimized program uses only two variables, i.e., z and c, instead of three.

(c) Definitions and usages at the nodes:

node	*defined*
1	$\{ x \}$
2	$\{ c, x \}$
3	$\emptyset$
4	$\{ y \}$
5	$\emptyset$
6	$\{ c, y \}$
7	$\emptyset$
8	$\emptyset$
9	$\emptyset$

node	*used*
1	$\emptyset$
2	$\{ x \}$
3	$\{ c \}$
4	$\{ x \}$
5	$\{ y \}$
6	$\{ y \}$
7	$\{ c \}$
8	$\emptyset$
9	$\{ c \}$

system of data-flow equations

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

(d) At each node, the live variables directly provide the corresponding *in* set. The verification just consists of substituting the informal solution into the *out* equations, thus obtaining the *out* sets, and then of checking that the *in* equations become identities. Here is the verification (node 2 is analyzed stepwise, the others more speedily):

- For node 2: since from the solution it is $in(3) = \{\ x,\ c\ \}$, from the *out* equation we have $out(2) = \{\ x,\ c\ \}$; and since from the solution it is $in(2) = \{\ x\ \}$, the *in* equation becomes $\{\ x\ \} = \{\ x\ \} \cup \big(\ \{\ x,\ c\ \} - \{\ x,\ c\ \}\ \big)$, i.e., $\{\ x\ \} = \{\ x\ \} \cup \emptyset$, which is an identity.
- For node 4: the *out* equation yields $out(4) = in(5) = \{\ y,\ c\ \}$, from which the *in* equation becomes $\{\ x,\ c\ \} = \{\ x\ \} \cup \big(\ \{\ y,\ c\ \} - \{\ y\ \}\ \big)$, i.e., $\{\ x,\ c\ \} = \{\ x\ \} \cup \{\ c\ \}$, an identity.
- For node 5: the *out* equation yields $out(5) = in(6) \cup in(7) = \{\ y\ \} \cup \{\ c\ \} = \{\ y,\ c\ \}$, from which the *in* equation becomes $\{\ y,\ c\ \} = \{\ y\ \} \cup \{\ y,\ c\ \}$, an identity.

Therefore, all the listed equations pass the verification test and thus confirm the correctness of the informal solution (or of the part considered).