FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 13 July 2016 - Part Theory

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

NAME (capital letters pls.):

MATRICOLA: SIGNATURE:

INSTRUCTIONS - READ CAREFULLY:

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

1 Regular Expressions and Finite Automata 20%

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

   (a) Design a (preferably minimal) finite-state automaton A_1 that accepts all and only the strings of odd length over the alphabet Σ.

   (b) By using the Berry-Sethi method (BS), find a deterministic finite-state automaton A_2 equivalent to the regular expression R below:

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

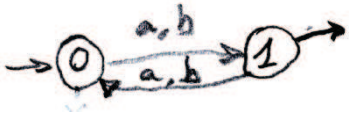
   (c) In a systematic way, obtain the finite-state automaton A cartesian product of the previous automata A_1 and A_2, i.e., $A = A_1 \times A_2$.

   (d) If necessary, minimize the automaton A previously obtained.

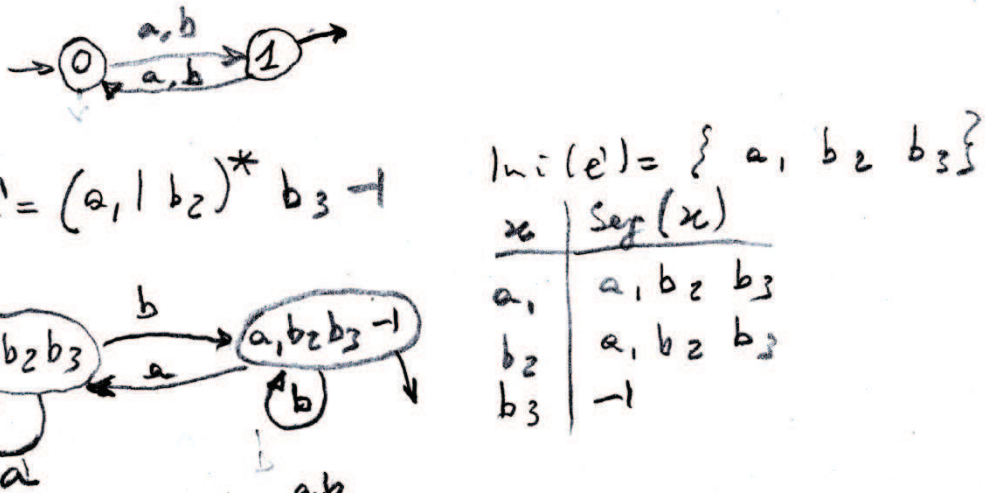
   (e) (optional) From the minimal automaton A and using the BMC method (node elimination method), derive an equivalent regular expression R'. If necessary, rewrite this regular expression R' in order to simplify it and to highlight the characteristic properties of the language generated by R'.

Solution

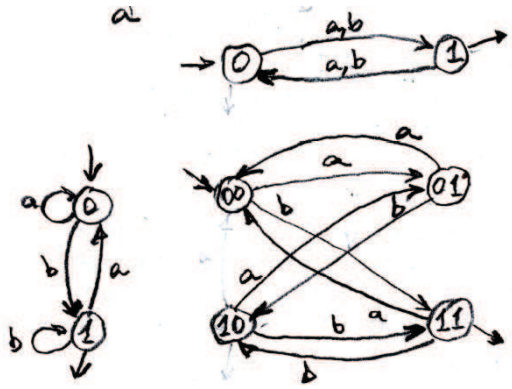
(a) Here is the minimal automaton that accepts only the strings of odd length:



(b) Here is the deterministic automaton obtained by means of the *BS* method:

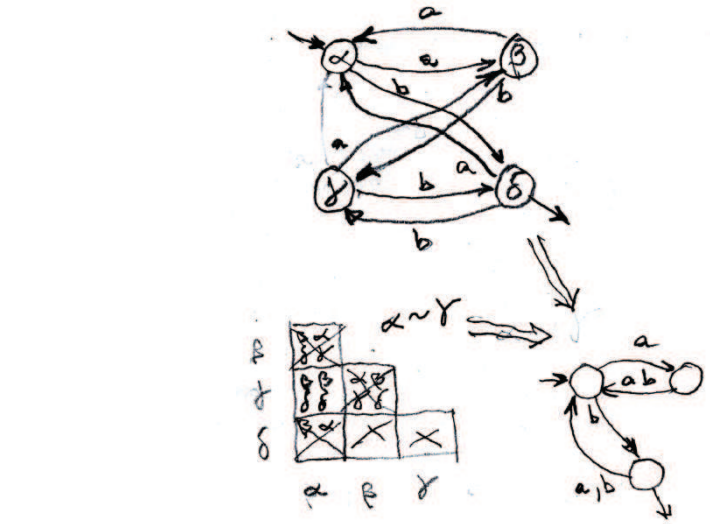


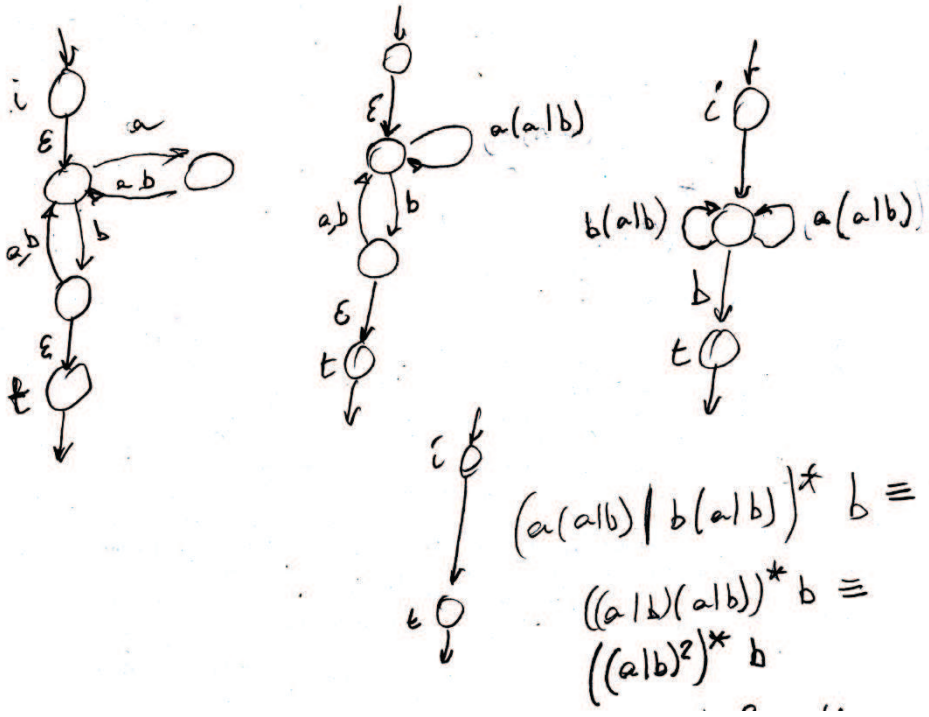
(c) Here is the cartesian product automaton that accepts the intersection of the two languages:



(d) Here is the minimized cartesian product automaton:

(e) The obtained *RE* $\left((a \mid b)^2\right)^* b$ shows that the language of the product automaton consists of all and only the strings of odd length ending with a letter b.


a
b
α
β
γ
δ
α~γ
a
a,b
b
a,b


i
ε
a
a,b
a,b
b
ε
t
b
ε
a(a|b)
a,b
b
ε
t
i
b(a|b)
a(a|b)
b
t
i
ε
(a(a|b) | b(a|b))* b ≡
((a|b)(a|b))* b ≡
((a|b)²)* b

2 Free Grammars and Pushdown Automata 20%

1. Consider the Dyck language with round open and closed parentheses '(' and ')', generated by the well known *BNF* unambiguous grammar G below (axiom S):

   $$G\colon S \to (\ S\ )\ S \ \mid\ \varepsilon$$

   Answer the following questions:

   (a) Write a grammar G_1, *BNF* and unambiguous, that generates all and only the Dyck strings of language $L(G)$ whose substrings of consecutive open parentheses have an even length (0, 2, 4, ...).
   Examples:

   ε $\qquad$ $\overbrace{(\,(\,)\,)}^{2}$ $\qquad$ $(\overbrace{(\,)}^{2}\ \overbrace{(\,(\,)\,)}^{2}\,)$

   Counterexamples:

   $\overbrace{(\quad)}^{1}$ $\qquad$ $(\overbrace{(\,)}^{2}\ \overbrace{(\,(\,(\,)\,)\,)}^{3}\,)$

   To test grammar G_1, draw the syntax trees of the three above examples; also try to draw the syntax trees of the two counterexamples and shortly explain why it is impossible to complete the trees by means of grammar G_1.

   (b) (optional) Write a grammar G_2, *BNF* and unambiguous, that generates all and only the Dyck strings of language $L(G)$ whose substrings of consecutive closed parentheses have an odd length (1, 3, 5, ...).
   Examples:

   $(\overbrace{\quad}^{1})$ $\qquad$ $(\,(\overbrace{\quad}^{1})\ \overbrace{(\,)\,}^{3}\,)$

   Counterexamples:

   ε $\qquad$ $(\,(\overbrace{\,)}^{2}\,)$ $\qquad$ $\overbrace{(\quad)}^{1}\ (\overbrace{(\,)}^{2}\,)$

   To test grammar G_2, draw the syntax trees of the three examples; also try to draw the syntax trees of the two counterexamples and shortly explain why it is impossible to complete the trees by means of grammar G_2.

Solution

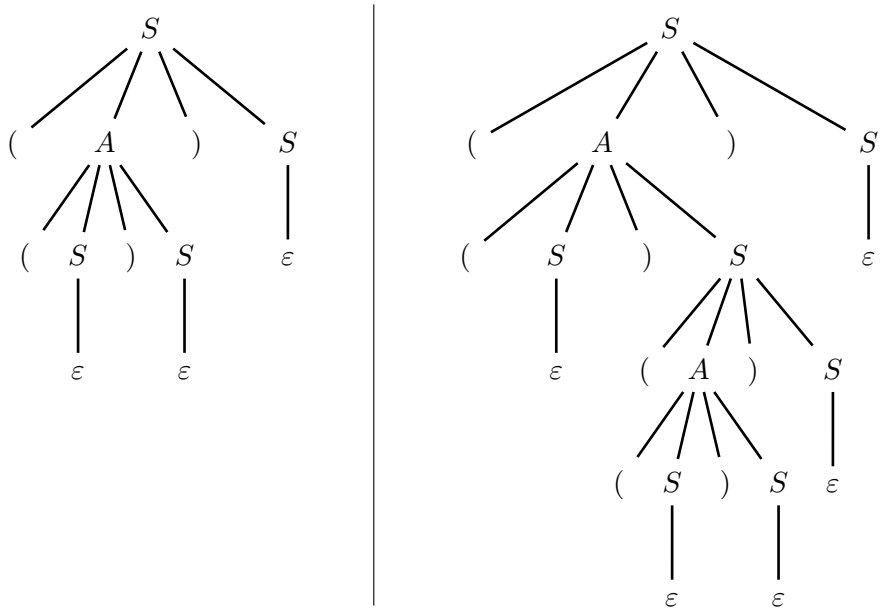
(a) Here is grammar G_1 (axiom S), inspired to grammar G:

$$G_1 \begin{cases} S & \to & (\ A\ )\ S \ \mid\ \varepsilon \\ A & \to & (\ S\ )\ S \end{cases}$$

Notice that grammar G_1 is right-recursive. Intuitively, grammar G_1 forces the consecutive open parentheses to come two-by-two (starting from zero). Since grammar G_1 is obtained by restricting the unambiguous grammar G, it is unambiguous as well.

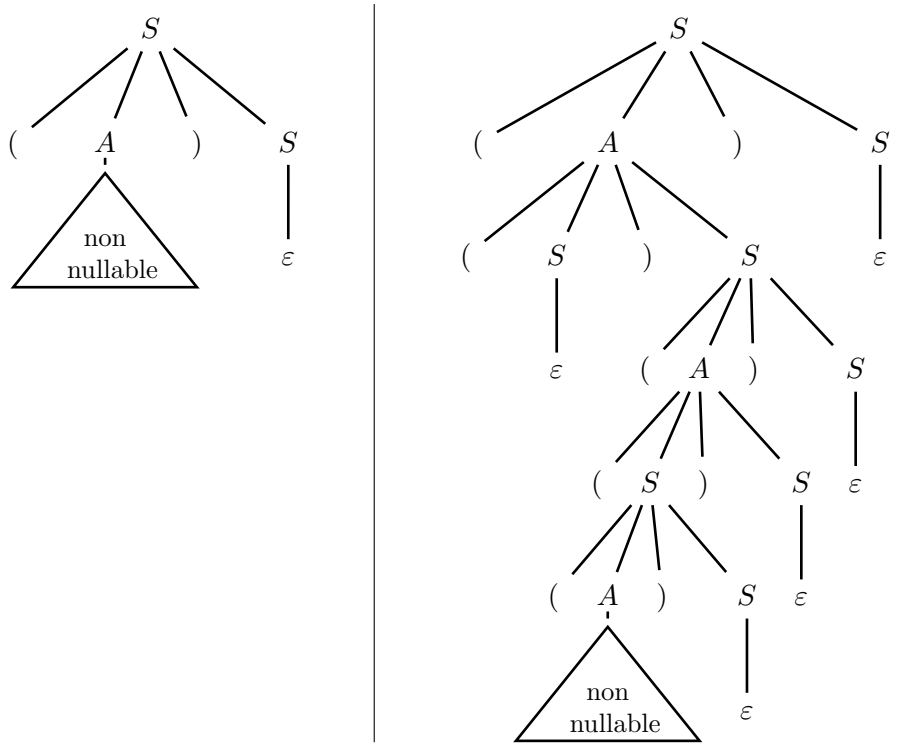
The syntax trees of the examples are easy to draw, and those of the counterexamples cannot be completed as nonterminal A may not generate the empty string and prevents the tree from having an odd number of consecutive open parentheses on the fronteer.

Here are the two valid sample trees (the tree of string ε is trivial and is skipped) for grammar G_1:



One can see that on every path from root to ε the nested nonterminals S and A alternate, but only S is nullable, thus they can generate substrings of open parentheses only of even length.

As for the two invalid sample strings, their most fitting tentative syntax trees (yet incomplete) are as below:



Other larger tentative trees would fail even worse. In both cases, for having the desired fronteer it would be necessary to nullify nonterminal A, which is not nullable. This happens wherever an odd number of consecutive open parentheses has been placed on the fronteer, that is, it has been generated by grammar G_1.

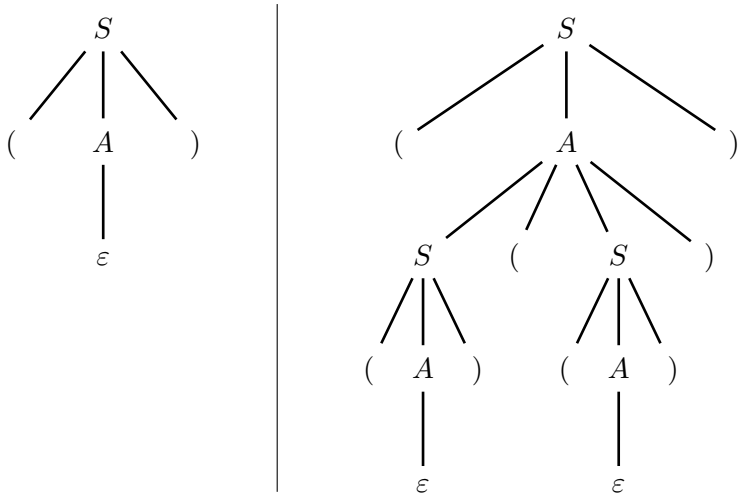
(b) Here is grammar G_2 (axiom S), also inspired to grammar G, but with some more relevant changes with respect to grammar G_1:

$$G_2 \begin{cases} S \;\to\; S\ (\ A\ )\ \mid\ (\ A\ ) \\ A \;\to\; S\ (\ S\ )\ \mid\ (\ S\ )\ \mid\ \varepsilon \end{cases}$$

Notice that grammar G_2 is left-recursive; in fact, it is well known that grammar G can be equivalently written in left-recursive form, i.e., "$S \to S\ (\ S\ )\ \mid\ \varepsilon$". This change allows us to easily control the number of closed parentheses, instead of the open ones. Intuitively, grammar G_2 forces the consecutive closed parentheses to come two-by-two, plus one more closed parenthesis as nonterminal S is unable to generate the empty string ε. Notice also that the non-nullability of nonterminal S makes it necessary to introduce some alternatives, to consider the cases where there are not any concatenated parenthesis pairs, but only nested pairs. Since grammar G_2 is obtained by restricting the unambiguous left-recursive version of grammar G, it is unambiguous as well.

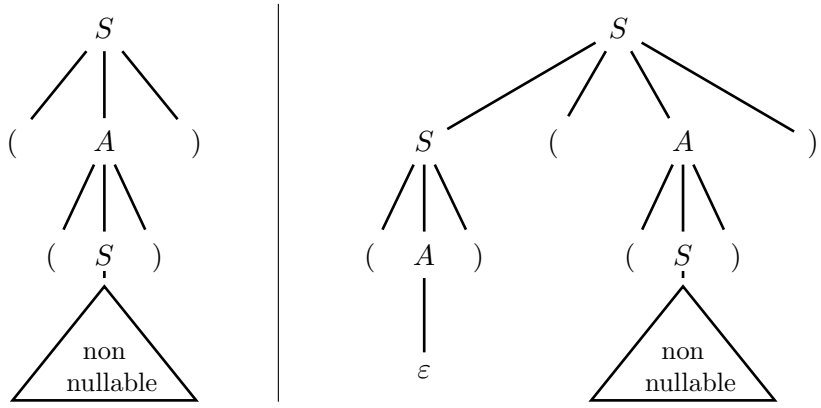
The syntax trees of the examples are easy to draw, and those of the counterexamples cannot be completed as nonterminal S may not generate the empty string and prevents the tree from having an even number of consecutive closed parentheses on the fronteer.

Here are the two valid sample trees for grammar G_2:



One can see that on every path from root to ε the nested nonterminals S and A alternate, but only A is nullable, thus they can generate substrings of closed parentheses only of odd length.

As for the two invalid sample strings (clearly the empty string ε cannot be generated), their most fitting tentative syntax trees (yet incomplete) are as below:



Other larger tentative trees would fail even worse. In both cases, for having the desired fronteer it would be necessary to nullify nonterminal S, which is not nullable. This happens wherever an even number of consecutive closed parentheses has been placed on the fronteer, that is, it has been generated by grammar G_2.

2. Consider a language designed to describe a sport game organized by elimination tournaments, where at each stage of the game only the winners will play in the next stage, until exactly one competitor becomes the final winner.

   - In each stage, a player can be either a new team or the winner team of a previous stage, at any depth level.
   - Every match has two players and these attributes:
     - an associated date, defined by a number for the year, a string for the month and a number for the day
     - a scheduled time, defined by two numbers for the hour and the minute
     - and a location, defined by a string

     For simplicity, assume that in the grammar all the numbers and strings are schematized by the terminals $\boldsymbol{n}$ and $\boldsymbol{s}$, respectively.
   - A document (i.e., a string of the language) consists of a non-empty list of tournament descriptions.

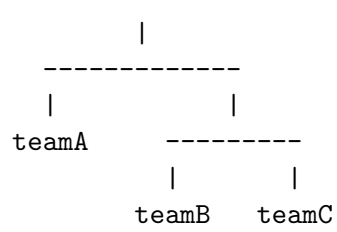
   Answer the following questions:

   (a) Write a grammar G, *EBNF* and unambiguous, that models the language described above.

   (b) Briefly illustrate how your grammar G can be modified in a systematic way to put a limit k on the height of the tree that represents the matches played in the tournament; exemplify the modification for the case $k = 3$.

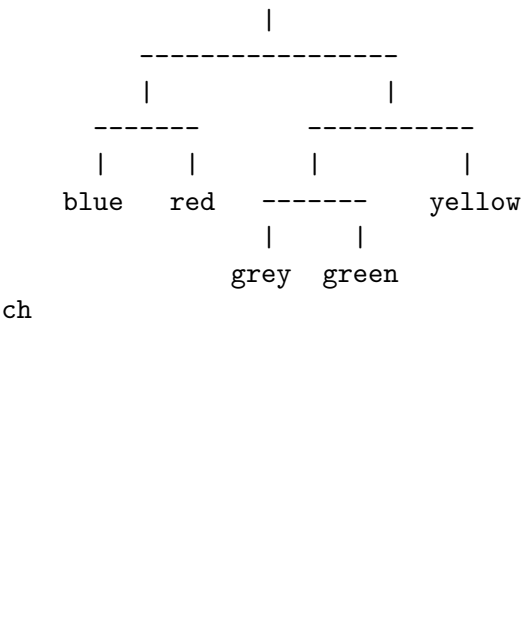
   Two sample language fragments are reported below. On the right side of each sample it is depicted a tree that represents the matches of the tournament.

```
tournament julyFun is
   match
      date 2016 july 29;
      time 21 : 30;
      location Rome;
      player1 is teamA;
      player2 is winner of match
         date 2016 july 23;
         time 17 : 00;
         location Milan;
         player1 is teamB;
         player2 is teamC;
      end match;
   end match;
end tournament julyFun;
```



See the next page for one more example.

```
tournament springOutdoor is
   match
      ...
      player1 is winner of match
         ...
         player1 is blue
         player2 is red;
      end match;
      player2 is winner of match
         ...
         player1 is winner of match
            ...
            player1 is grey;
            player2 is green;
         end match;
         player2 is yellow;
      end match;
   end match;
end tournament springOutdoor;
```



Solution

(a) Here is a reasonable grammar G (axiom DOCUM):

$$
G \left\{ \begin{array}{rcl}
\langle \mathsf{DOCUM} \rangle & \to & \big(\ \langle \mathsf{TOURN} \rangle \ \text{`;'} \ \big)^{+} \\
\hline
\langle \mathsf{TOURN} \rangle & \to & \mathsf{tournament} \ \boldsymbol{s} \ \mathsf{is} \\
 & & \qquad \langle \mathsf{MATCH} \rangle \ \text{`;'} \\
 & & \mathsf{end \ tournament} \ \boldsymbol{s} \\
\hline
\langle \mathsf{MATCH} \rangle & \to & \mathsf{match} \\
 & & \qquad \langle \mathsf{DATE} \rangle \ \text{`;'} \\
 & & \qquad \langle \mathsf{TIME} \rangle \ \text{`;'} \\
 & & \qquad \langle \mathsf{LOCATION} \rangle \ \text{`;'} \\
 & & \qquad \langle \mathsf{PLAYER1} \rangle \ \text{`;'} \\
 & & \qquad \langle \mathsf{PLAYER2} \rangle \ \text{`;'} \\
 & & \mathsf{end \ match} \\
\hline
\langle \mathsf{PLAYER1} \rangle & \to & \mathsf{player1 \ is} \ \big(\ \boldsymbol{s} \ \mid \ \mathsf{winner \ of} \ \langle \mathsf{MATCH} \rangle \ \big) \\
\langle \mathsf{PLAYER2} \rangle & \to & \mathsf{player2 \ is} \ \big(\ \boldsymbol{s} \ \mid \ \mathsf{winner \ of} \ \langle \mathsf{MATCH} \rangle \ \big) \\
\langle \mathsf{DATE} \rangle & \to & \mathsf{date} \ \boldsymbol{n} \ \boldsymbol{s} \ \boldsymbol{n} \\
\langle \mathsf{TIME} \rangle & \to & \mathsf{time} \ \boldsymbol{n} \ \text{`:'} \ \boldsymbol{n} \\
\langle \mathsf{LOCATION} \rangle & \to & \mathsf{location} \ \boldsymbol{s}
\end{array} \right.
$$

Grammar G is of type *EBNF* and is unambiguous, as it is made of well known unambiguous constructs, like regular lists with separators and disjoint alternatives. Recursion involves only the nonterminals PLAYERx (with $x = 1, 2$) and MATCH, and it does not cause any ambiguity either.

(b) A simple solution to bound the depth of the competition tree, consists of introducing distinct nonterminals for the matches that take place at different tree levels. For the case $k = 3$, we will have three such nonterminals MATCH1, MATCH2 and MATCH3, where the last one can only be expanded as a player name, not as the winner of a further match.

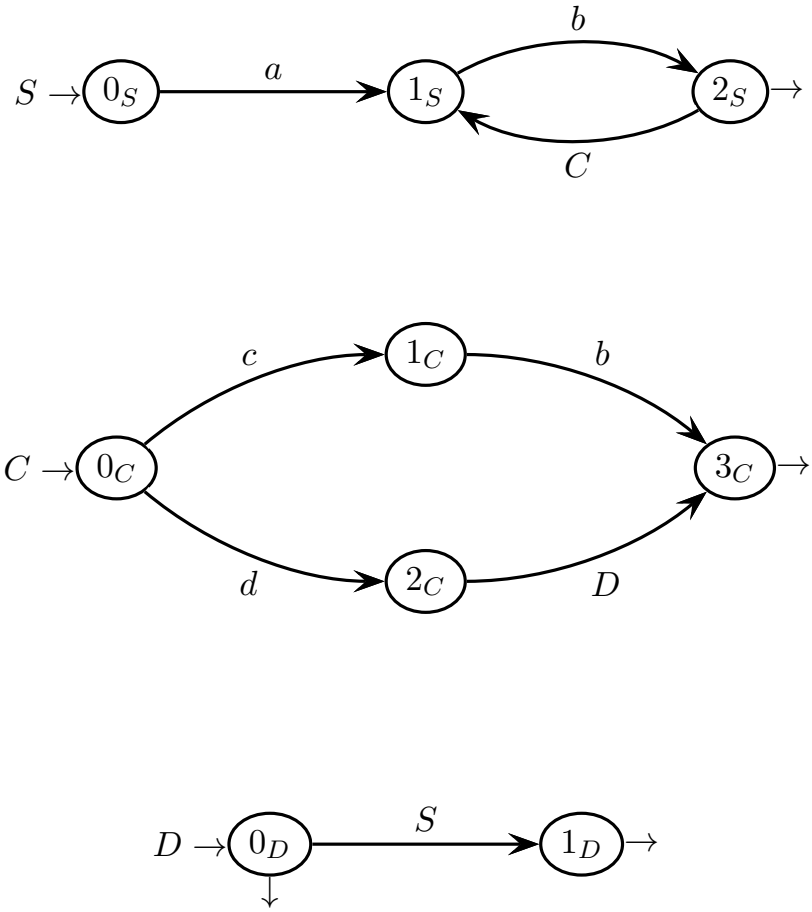
Below there is a sketch of the modified grammar (only the new or updated rules, or part thereof). To limit the grammar size, the expansions of nonterminals PLAYER1 and PLAYER2 are incorporated into the MATCHx rules (with $x = 1, 2, 3$), and the nonterminals PLAYER1 and PLAYER2 are eliminated.

modified grammar G

$$
\left\{
\begin{array}{lcl}
\langle \text{TOURN} \rangle & \rightarrow & \text{tournament } \boldsymbol{s} \text{ is} \\
& & \quad \langle \text{MATCH1} \rangle \text{ `;'} \\
& & \text{end tournament } \boldsymbol{s} \\
\hline
\langle \text{MATCH1} \rangle & \rightarrow & \text{match} \\
& & \quad \ldots \\
& & \quad \text{player1 is } \big(\boldsymbol{s} \mid \text{winner of } \langle \text{MATCH2} \rangle \big) \text{ `;'} \\
& & \quad \ldots \\
& & \text{end match} \\
\hline
\langle \text{MATCH2} \rangle & \rightarrow & \text{match} \\
& & \quad \ldots \\
& & \quad \text{player1 is } \big(\boldsymbol{s} \mid \text{winner of } \langle \text{MATCH3} \rangle \big) \text{ `;'} \\
& & \quad \ldots \\
& & \text{end match} \\
\hline
\langle \text{MATCH3} \rangle & \rightarrow & \text{match } \ldots \text{ player1 is } \boldsymbol{s} \text{ `;' } \ldots \text{ end match}
\end{array}
\right.
$$

3 Syntax Analysis and Parsing Methodologies 20%

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



Answer the following questions:

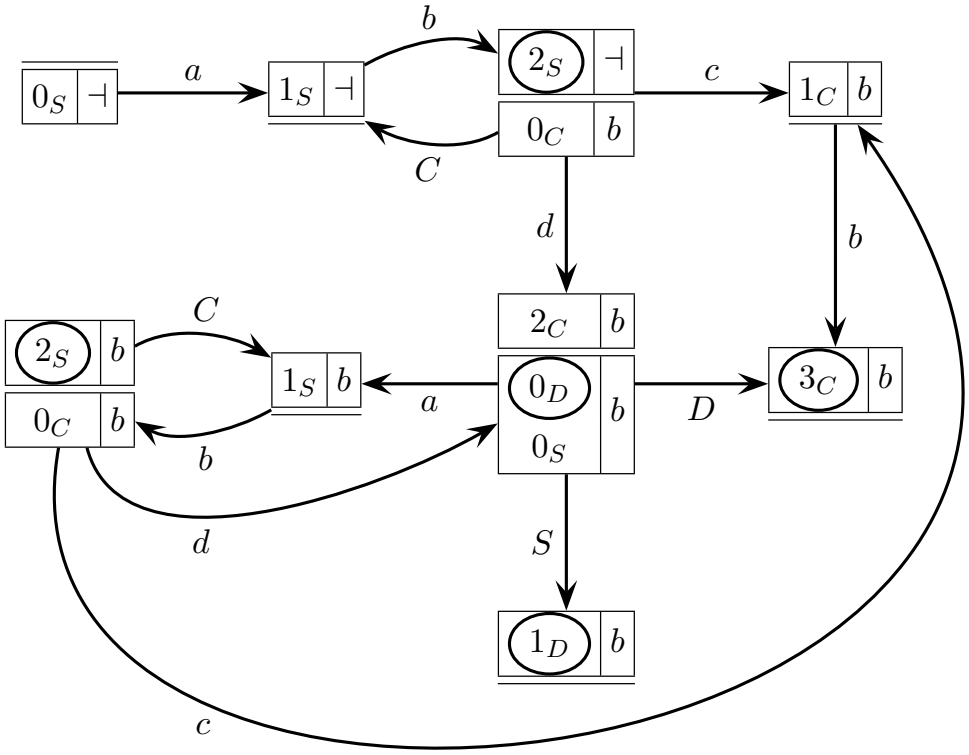
(a) Draw the complete pilot of grammar G, say if grammar G is of type $ELR(1)$ and shortly justify your answer.

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

(c) (optional) Write two (of three) syntactic procedures of the recursive descent syntax analyzer of grammar G.

Solution

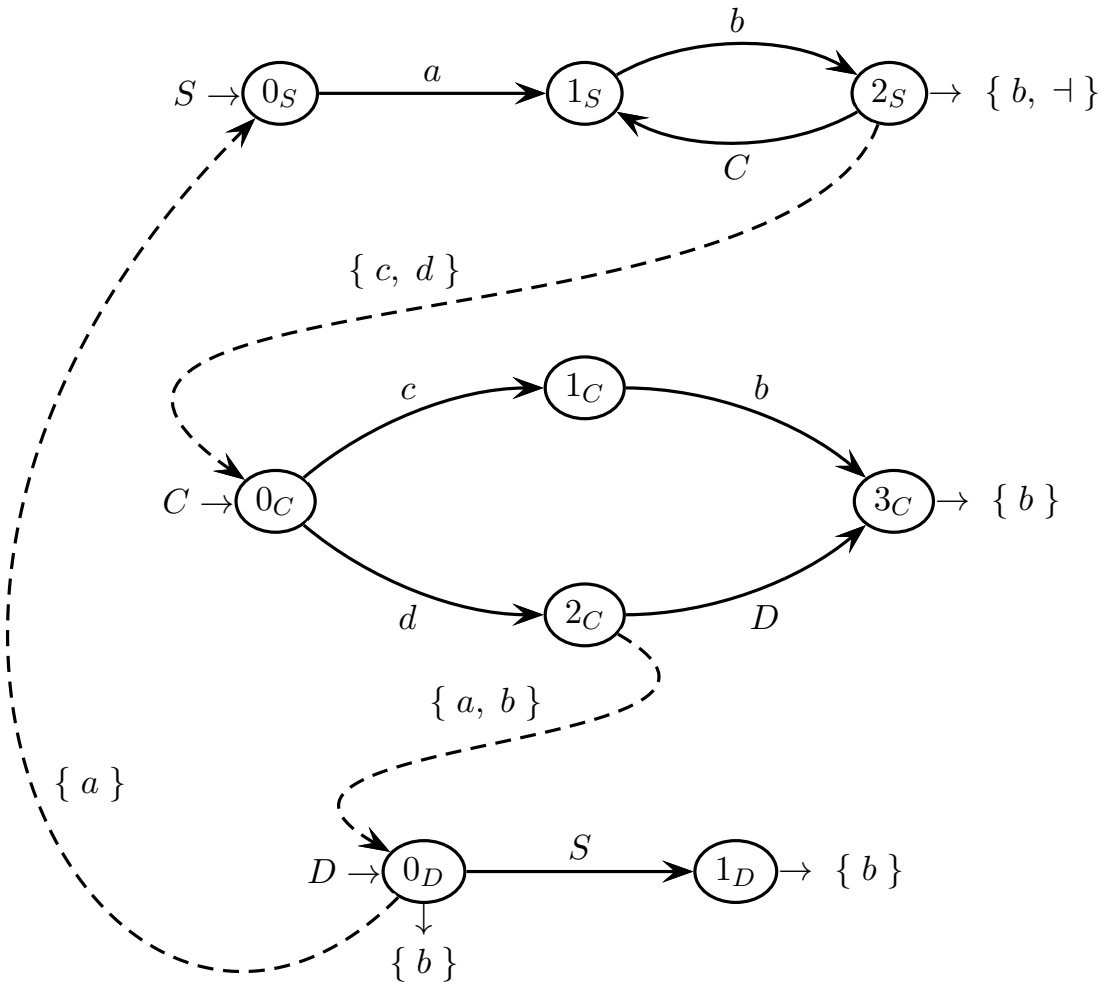
(a) Here is the requested pilot, with 9 m-states (for simplicity here the m-state numbering is omitted and item $\langle\, 0_S,\, \dashv \rangle$ is the initial m-state)). The pilot is conflict-free, thus grammar G is $ELR\,(1)$.



Incidentally, one may notice that the pilot has the STP (all the m-state bases are singletons, which implies the STP) and that the grammar is not left-recursive, so that it is already apparent that the grammar is $ELL\,(1)$ as well.

To complete the picture, one might wonder whether language $L\,(G)$ is regular or not. Notice that the grammar has an auto-inclusive (self-embedding) recursive derivation scheme $S \Rightarrow a\ b\ C\ b \Rightarrow a\ b\ d\ D\ b \Rightarrow a\ b\ d\ S\ b$, which will terminate either with $C \Rightarrow c\ b$ or with $D \Rightarrow \varepsilon$ or with $S \Rightarrow a\ b$. This is a necessary condition for a free language not to be regular. Actually this condition is not sufficient, yet here there are not any other derivation schemes, so there will be non-regular string sets like $(\, a\ b\ d\,)^n\ a\ b\ b^n$ for any $n \geq 0$ (all the other string types being just variations thereof, depending on which rule the derivation terminates by), thus one can reasonably conclude that language $L\,(G)$ is very unlikely to be regular.

(b) Here are all the guide sets, not difficult to compute. The machine net completed in this way constitutes the so-called *PCFG* (Predictive Control Flow Graph) of the *ELL* syntax analyzer.



Since the pilot is already drawn and available, the prospect sets (also called guide sets) on the exit arrows of the final states in the machine net can be obtained by collecting the look-ahead terminals of the reduction items in the pilot m-states. The reader may verify in this way the coherence of pilot and *PCFG*.

The guide set on call arc $2_C \xrightarrow{\{\,a,\,b\,\}} 0_D$ contains two elements, namely: letter a as it holds $a \in \mathit{Ini}\big(L\,(0_S)\big)$ or equivalently the call arc is followed by call arc $0_D \xrightarrow{\{\,a\,\}} 0_S$; and letter b as nonterminal D is nullable, language $L\,(3_C)$ is also nullable and it holds $b \in \pi_{3_C}$, i.e., b is in the prospect set π of state 3_C. **Caution**: the reason for b to be in the guide set of $2_C \dashrightarrow 0_D$, *is not* that the call arc is followed by an exit arrow from state 0_D with a prospect set containing b; remember that the guide sets of exit arrows do not back-propagate onto call arc chains; see the textbook for the guide set equations and their enabling conditions.

The other guide sets on the call arcs are simple. Furthermore, the guide sets on the terminal shift arcs are immediate and therefore they are not shown here.

The guide sets on the three bifurcation states 2_S, 0_C and 0_D of the *PCFG* are disjoint, thus grammar G is of type $ELL\,(1)$ (actually this it was already known by verifying the *ELL* condition on the pilot).

(c) For completeness, here all three syntactic procedures S, C and D of the recursive descent analyzer are listed. Here are the plain non-optimized versions, obtained by testing each guide set case separately.

```
procedure S
if cc ∈ { a }                                          // state 0
|  cc = next                                           // goto 1
|  loop                                          // unconditional loop
|  |  if cc ∈ { b }                                    // state 1
|  |  |  cc = next                                     // goto 2
|  |  |  if cc ∈ { c, d }                             // state 2
|  |  |  |  call C                                 // go back to 1
|  |  |  else if cc ∈ { b, ⊣ }                       // state 2
|  |  |  |  return
|  |  |  else                                        // state 2
|  |  |  |  error
|  |  |  endif
|  |  else                                           // state 1
|  |  |  error
|  |  endif
|  end
else                                                   // state 0
|  error
endif
end procedure
```

Procedure S is the axiomatic one. The main program is the usual one (see the textbook) and it is not shown here. Here are the two other procedures.

```
procedure C
if cc ∈ { c }                    // state 0
|  cc = next                      // goto 1
|  if cc ∈ { b }                 // state 1
|  |  cc = next                   // goto 3
|  |  if cc ∈ { b }              // state 3
|  |  |  return
|  |  else                       // state 3
|  |  |  error
|  |  endif
|  else                          // state 1
|  |  error
|  endif
else if cc ∈ { d }               // state 0
|  cc = next                      // goto 2
|  if cc ∈ { a, b }               // state 2
|  |  call D                     // goto 3
|  |  if cc ∈ { b }              // state 3
|  |  |  return
|  |  else                       // state 3
|  |  |  error
|  |  endif
|  else                          // state 2
|  |  error
|  endif
else                             // state 0
|  error
endif
end procedure
```

```
procedure D
if cc ∈ { a }                    // state 0
|  call S                        // goto 1
|  if cc ∈ { b }                 // state 1
|  |  return
|  else                          // state 1
|  |  error
|  endif
else if cc ∈ { b }               // state 0
|  return
else                             // state 0
|  error
endif
end procedure
```

Of course there are several possible code optimizations, like grouping most error cases. Here is an optimized version of the axiomatic procedure S.

```
procedure S
if cc ∈ { a }                                        // state 0
|  cc = next                                         // goto 1
|  while cc ∈ { b } do                              // conditional loop
|  |  cc = next                                      // goto 2
|  |  if cc ∈ { c, d }                              // state 2
|  |  |  call C                                     // go back to 1
|  |  else if cc ∈ { b, ⊣ }                        // state 2
|  |  |  return
|  |  endif
|  end
endif
error                                                // grouped error cases
end procedure
```

The reader may implement the other optimized procedure versions by himself.

4 Language Translation and Semantic Analysis 20%

1. Consider the following source grammar G_s (axiom S), for a language over the three-letter terminal alphabet $\{\ a,\ b,\ c\ \}$ and the two-letter nonterminal alphabet $\{\ S,\ A\ \}$:

$$G_s \begin{cases} S & \rightarrow & A\ c\ S \\ S & \rightarrow & A \\ A & \rightarrow & a\ a\ A\ b\ b \\ A & \rightarrow & c\ b \end{cases}$$

Grammar G_s generates strings composed of a sequence of groups of letters separated by letters c, like for instance string:

$$a^4\ c\ b^5\ c\ a^2\ c\ b^3$$

where each group consists of an *even* number (say $2k$ with $k \geq 0$) of letters a immediately followed by one letter c and then by $2k+1$ letters b.

(a) Write a syntactic translation scheme G_τ (or a translation grammar) that computes the following translation τ:

$$\tau\,(a^{2k_1}\ c\ b^{2k_1+1}\ c\ \ldots\ c\ a^{2k_n}\ c\ b^{2k_n+1}) = b^{2k_1+1}\ d\ a^{2k_1}\ e\ \ldots\ e\ b^{2k_n+1}\ d\ a^{2k_n}\ f$$

over the five-letter target alphabet $\{\ a,\ b,\ d,\ e,\ f\ \}$, with $k_i \geq 0$ and $1 \leq i \leq n$. Example:

$$\tau\,(a^4\ c\ b^5\ c\ a^2\ c\ b^3) = b^5\ d\ a^4\ e\ b^3\ d\ a^2\ f$$

(b) Draw the source and target syntax trees for the sample string $a^4\,c\,b^5\,c\,a^2\,c\,b^3$ shown above.

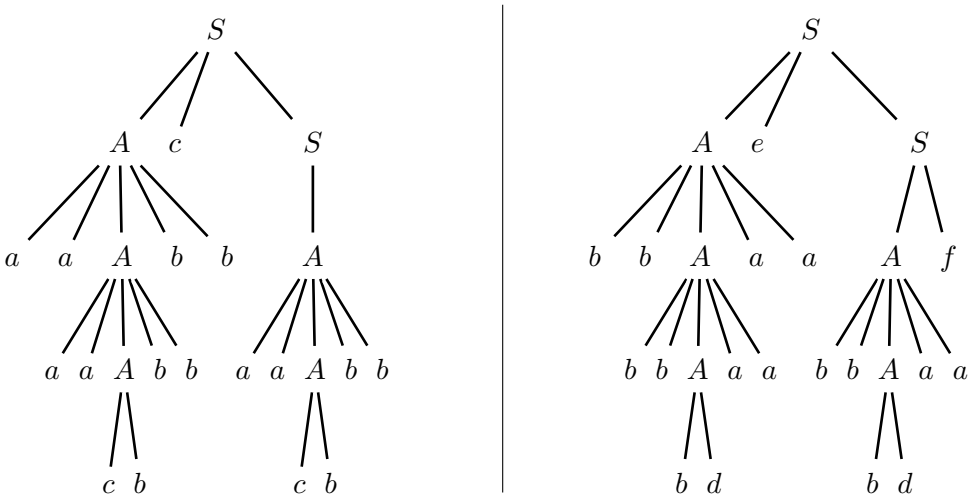
(c) Determine if the above translation τ can be defined by means of a finite-state translator T, and provide a suitable explanation for your answer.

Solution

(a) Here is a working translation scheme G_τ (axiom S):

$$G_\tau \begin{cases} S \to A\ c\ S & S \to A\ e\ S \\ S \to A & S \to A\ f \\ A \to a\ a\ A\ b\ b & A \to b\ b\ A\ a\ a \\ A \to c\ b & A \to b\ d \end{cases}$$

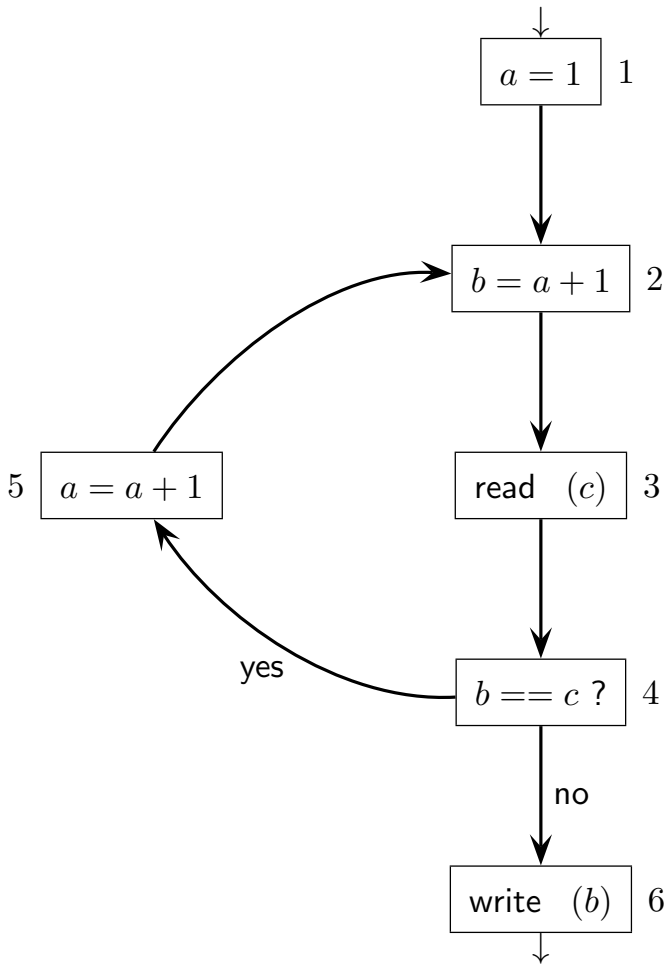
(b) Here are the two requested source and target syntax trees:



(c) Translation τ cannot be defined by a finite-state translator T, because the underlying finite-state recognizer A of T would be unable to count the matching letters a and b of each group in the source string, hence it could not check if a given input string belongs to the source language. In fact, it is evident that each input group of matching letters a and b is of the type $a^n\ c\ b^{n+1}$ (with $n \geq 0$), that is, the group has a well known free non-regular structure.

It would be relatively easy to design a finite-state translator that, given an input string *certainly belonging to the source language*, translates it into the correct matching string of the target language, but such a translator would be unable to properly compute the translation when it were fed with a generic input string.

2. Consider the following control flow graph (*CFG*) of a simple program:



Answer the following questions:

(a) Find the *live variables* of the given *CFG*, by means of the the systematic method of the liveness flow equations: first write the equations, then iteratively solve them. Write the live variables on the *CFG* prepared on the next pages.

(b) Find the *reaching definitions* of the given *CFG*, by intuitively applying the definition of reaching definition. Write the reaching definitions on the *CFG* prepared on the next pages.

(c) (optional) Write the flow equations for the reaching definitions.

---

table of definitions and usages at the nodes for **LIVE VARIABLES**

node	*defined*
1	
2	
3	
4	
5	
6	

node	*used*
1	
2	
3	
4	
5	
6	

system of data-flow equations for **LIVE VARIABLES**

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

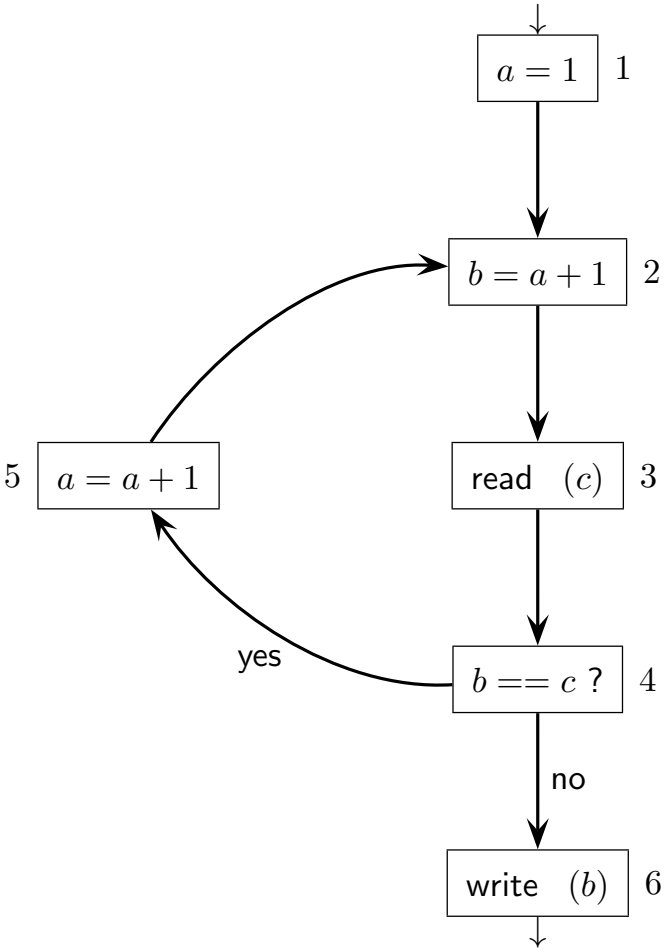
iterative solution table of the system of data-flow equations (**LIVE VAR.S**)

(the number of tables and 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														
6														

	7		8		9		10		11		12		13	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1														
2														
3														
4														
5														
6														

please here write the **LIVE VARIABLES** obtained systematically



please here write the **REACHING DEFINITIONS** obtained intuitively

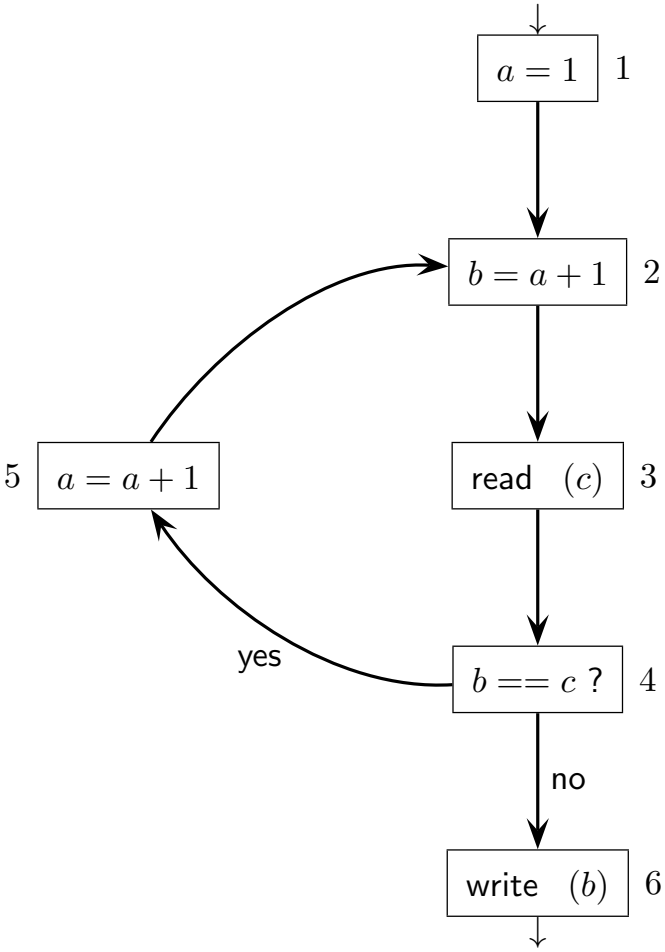


table of definitions and suppressions at the nodes
for **REACHING DEFINITIONS**

node	*defined*
1	
2	
3	
4	
5	
6	

node	*suppressed*
1	
2	
3	
4	
5	
6	

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

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

Solution

(a) Here is the systematic application of the flow equation method for live variables.

table of definitions and usages at the nodes for **LIVE VARIABLES**

node	*defined*
1	a
2	b
3	c
4	$-$
5	a
6	$-$

node	*used*
1	$-$
2	a
3	$-$
4	$b\ c$
5	a
6	b

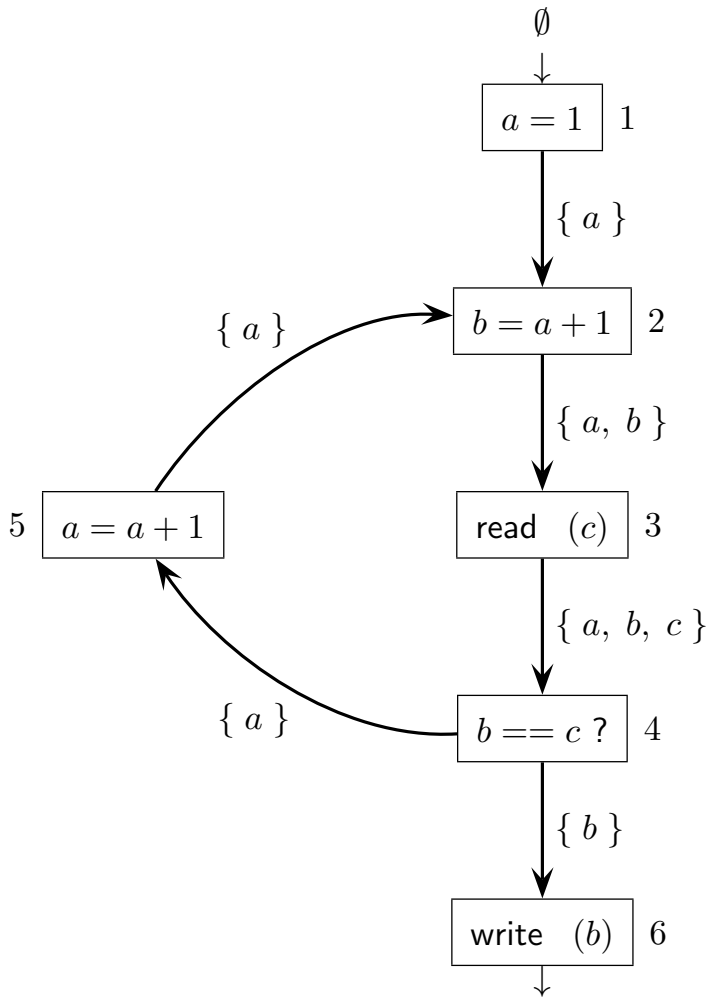
system of data-flow equations for **LIVE VARIABLES**

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

iterative solution table of the system of data-flow equations (**LIVE VAR.S**)

	initialization		1		2		3		4		5	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1	$\emptyset$	$-$	a	$\emptyset$	a	$\emptyset$	a	$\emptyset$				
2	$\emptyset$	a	$-$	a	b	a	$a\ b$	a				
3	$\emptyset$	$-$	$b\ c$	b	$a\ b\ c$	$a\ b$	$a\ b\ c$	$a\ b$				
4	$\emptyset$	$b\ c$	$a\ b$	$a\ b\ c$	$a\ b$	$a\ b\ c$	$a\ b$	$a\ b\ c$				
5	$\emptyset$	a	a	a	a	a	a	a				
6	$\emptyset$	b	$\emptyset$	b	$\emptyset$	b	$\emptyset$	b				

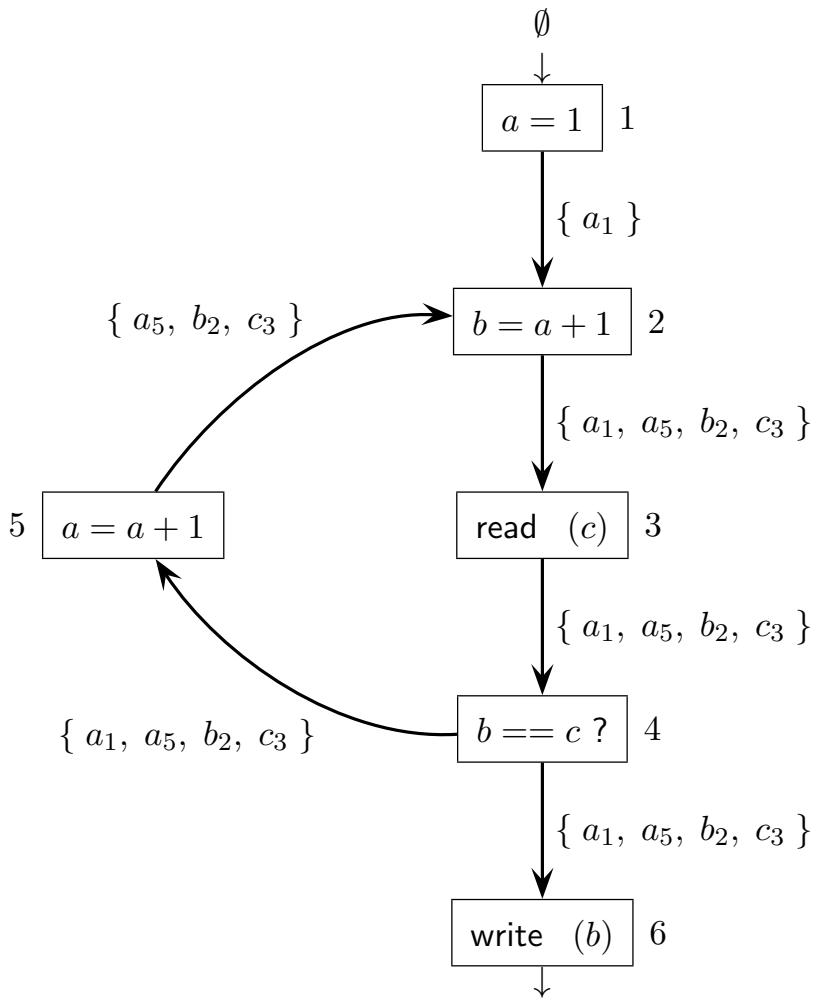
LIVE VARIABLES obtained systematically



The iterative solution procedure converges in three steps. The variables shown here are those that are LIVE at the INPUT of each node (including the initial one, which does not have any).

(b) Here is the intuitive solution of the reaching definitions.

REACHING DEFINITIONS obtained intuitively



The definitions shown here are those that are REACHING at the INPUT of each node (including the initial one, which does not have any).

(c) Here is the systematic application of the flow equation method for reaching definitions.

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

node	*defined*
1	a_1
2	b_2
3	c_3
4	–
5	a_5
6	–

node	*suppressed*
1	a_5
2	–
3	–
4	–
5	a_1
6	–

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

node	*in equations*	*out equations*
1	$in(1) = \emptyset$	$out(1) = \{\ a_1\ \} \cup \big(\ in(1) - \{\ a_5\ \}\ \big)$
2	$in(2) = out(1) \cup out(5)$	$out(2) = \{\ b_2\ \} \cup in(2)$
3	$in(3) = out(2)$	$out(3) = \{\ c_3\ \} \cup in(3)$
4	$in(4) = out(3)$	$out(4) = in(4)$
5	$in(5) = out(4)$	$out(5) = \{\ a_5\ \} \cup \big(\ in(5) - \{\ a_1\ \}\ \big)$
6	$in(6) = out(4)$	$out(6) = in(6)$