FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Fri 30 September 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%

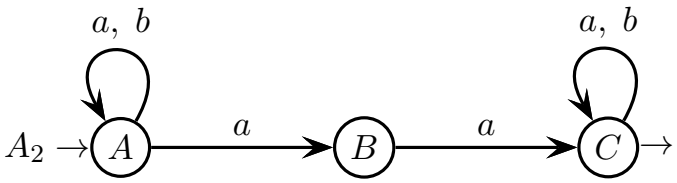
1. Consider a two-letter alphabet $\Sigma = \{ a, b \}$ and answer the following questions:

   (a) Consider the following regular expression R_1 over the alphabet Σ:

   $$R_1 = (a\ b)^* \ (a\ a\ b)^*$$

   - By using the Berry-Sethi (BS) method, derive a deterministic finite-state automaton A_1 that accepts the language $L(R_1)$, and minimize it if necessary.
   - Is the language $L(R_1)$ local ? Give an adequate justification to your answer.

   (b) Consider the following finite-state automaton A_2 over the alphabet Σ.



   - By using the node elimination (BMC) method, derive a regular expression R_2 equivalent to the automaton A_2, and please make explicit every step of the derivation. Furthermore, briefly describe the characteristic feature of the strings that belong to language $L(A_2)$.
   - From the finite-state automaton A_2, derive an equivalent right-unilinear grammar G_2 that has the set of nonterminals $V_N = \{ A, B, C \}$ (axiom A) associated with the states of A_2.
   - From the grammar G_2, derive an equivalent regular expression R_2' by solving an equation system in the variables L_A, L_B and L_C that denote the languages associated with the nonterminals of G_2. Use substitution and the Arden identity, and please make explicit every step of the derivation.

   (c) (optional) By using a systematic method, derive a finite-state automaton A_3 that accepts the intersection language $L(R_1) \cap L(A_2)$.

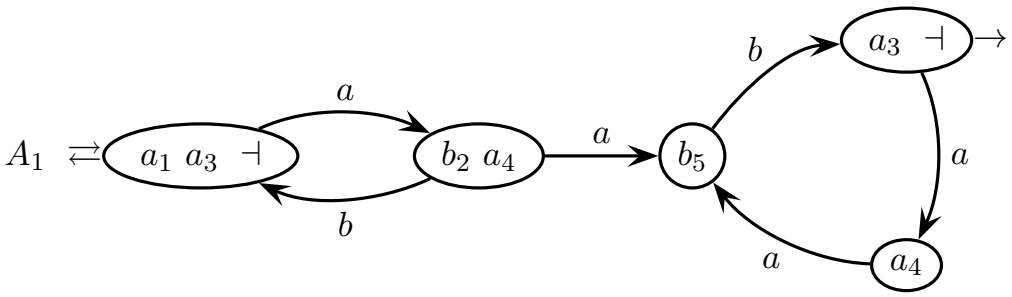
Solution

(a) Constructing a deterministic automaton A_1 is easy and the resulting state-transition graph has five states. First here is the marked regular expression $R_{1,\#}$, and its two sets of initials and followers:

$$R_{1,\#} = (\ a_1\ b_2\ )^*\ (\ a_3\ a_4\ b_5\ )^*\ \dashv$$

initials	a_1	a_3	$\dashv$
terminals	followers		
a_1	b_2		
b_2	a_1	a_3	$\dashv$
a_3	a_4		
a_4	b_5		
b_5	a_3	$\dashv$	

Then here is the *BS* automaton A_1 that recognizes language $L\,(R_1)$:



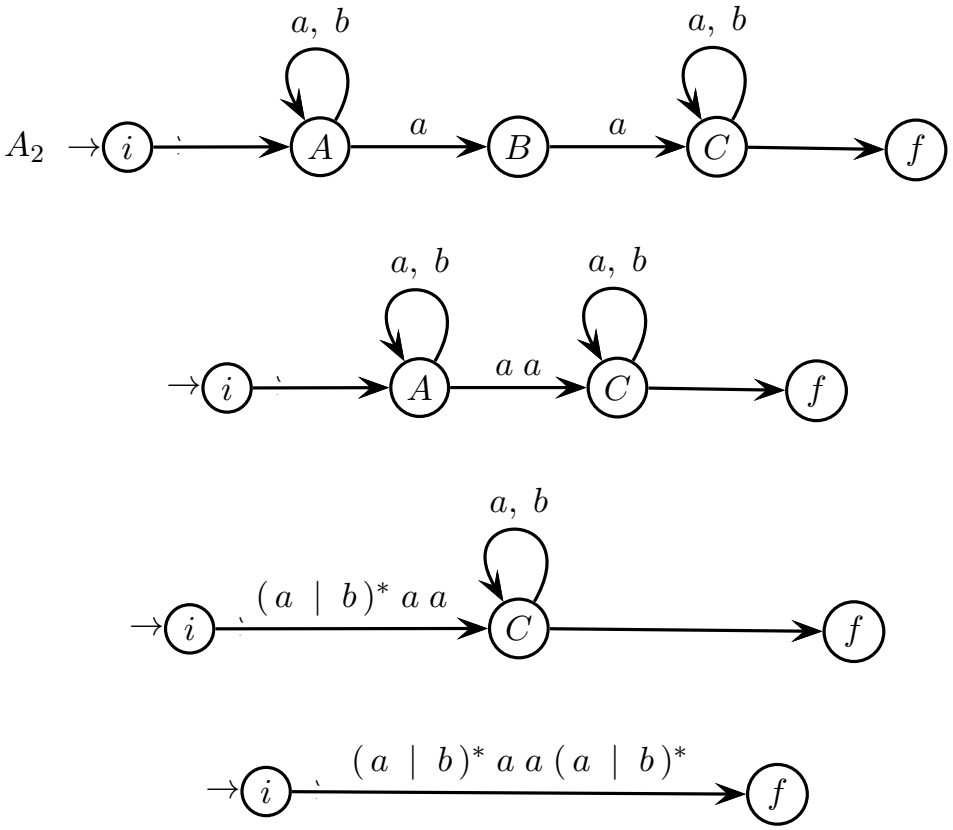
By construction, automaton A_1 is deterministic and in the clean form. It happens to be minimal, too: the three non-final states have different arc labels and thus are distinguishable, and the two final states go to distinguishable (non-final) states and thus are distinguishable as well.

Concerning the locality of language $L\,(R_1)$, see the automaton A_1 above, which recognizes language $L\,(R_1)$. The local sets that can be obtained from A_1 are $Ini(L\,(A_1)) = \{\ a\ \}$, $Fin(L\,(A_1)) = \{\ b\ \}$ and $Dig(L\,(A_1)) = \{\ a\,b,\ b\,a,\ a\,a\ \}$. Thus the local language generated by these local sets includes the strings that have arbitrarily long substrings of consecutive letters a. Yet such strings do not belong to $L\,(R_1)$ as clearly the regular expression R_1 generates only substrings of one or two letters a. Therefore language $L\,(R_1)$ is not local.

(b) An obvious intuitive formulation of the regular expression R_2 is the following:

$$R_2 = (\ a\ \mid\ b\ )^*\ a\ a\ (\ a\ \mid\ b\ )^* = \Sigma^*\ a\ a\ \Sigma^*$$

Here is the fully detailed construction for node elimination (*BMC* method):



The node elimination construction yields the same result as the intuitive one before. One may notice that such a formulation of the regular expression R_2 is ambiguous just like the original automaton A_2 was, too. By inspecting R_2 (or A_2) it is quite evident that the characteristic feature of the strings of language $L(R_2)$, which exactly defines them, is that of having at least one digram $a\,a$.
Here is the right-unilinear grammar G_2 obtained from automaton A_2 (axiom A):

$$G_2 \begin{cases} A \;\to\; a\,A \mid b\,A \mid a\,B \\ B \;\to\; a\,C \\ C \;\to\; a\,C \mid b\,C \mid \varepsilon \end{cases}$$

And here is the system of language equations associated with grammar G_2.

$$\begin{cases} L_A \;=\; \{\, a \,\}\; L_A \cup \{\, a \,\}\; L_A \cup \{\, a \,\}\; L_B \\ L_B \;=\; \{\, a \,\}\; L_C \\ L_C \;=\; \{\, a \,\}\; L_C \cup \{\, a \,\}\; L_C \cup \{\, \varepsilon \,\} \end{cases}$$

that is, by factorizing the language variables on the right:

$$\begin{cases} L_A \;=\; \{\, a,\, b \,\}\; L_A \cup \{\, a \,\}\; L_B \\ L_B \;=\; \{\, a \,\}\; L_C \\ L_C \;=\; \{\, a,\, b \,\}\; L_C \cup \{\, \varepsilon \,\} \end{cases}$$

To solve these equations step-by-step, from the Arden identity it follows:

$$L_C = \{\ a,\ b\ \}^*\ \{\ \varepsilon\ \} = (\ a\ \mid\ b\ )^*$$

Then by substitution it follows:

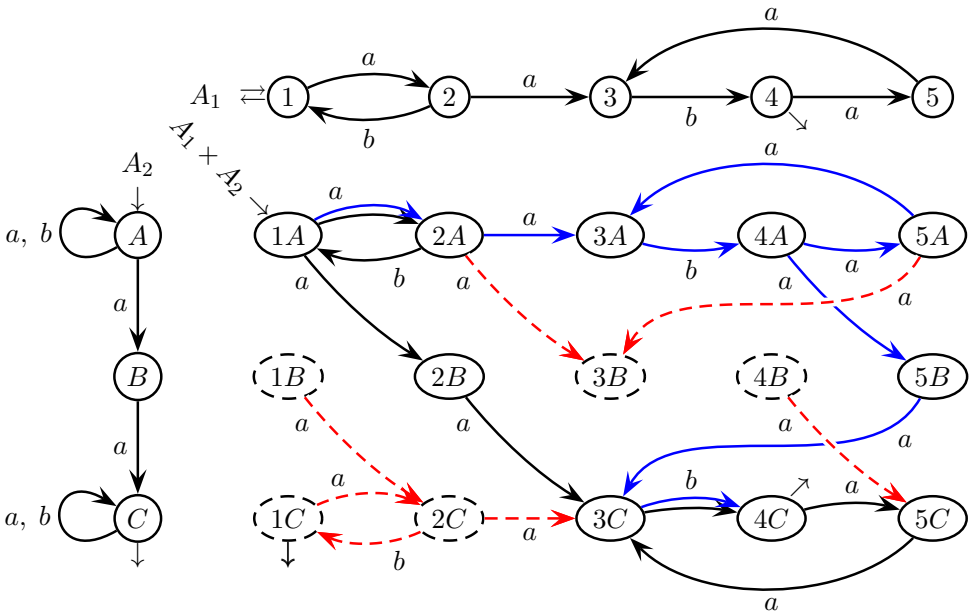
$$L_B = \{\ a\ \}\{\ a,\ b\ \}^*\ \{\ \varepsilon\ \} = a\ (\ a\ \mid\ b\ )^*$$

And finally, again by the Arden identity and then by substitution, it follows:

$$L_A = \{\ a,\ b\ \}^*\ \{\ a\ \}\ L_B = (\ a\ \mid\ b\ )^*\ a\ a\ (\ a\ \mid\ b\ )^*$$

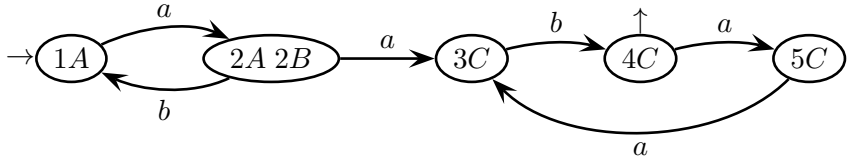
The obtained regular expression for language L_A is identical, hence equivalent, to the regular expression R_2 before. In general one would just obtain an equivalent regular expression, but not necessarily an identical one.

(c) Use the Cartesian product automaton construction as follows, where the states and connections that are accessible and post-accessible are drawn in solid trait, black or blue (read below), while all the others are drawn in dashed trait, red:



The product automaton $A_1 \times A_2$ is ambiguous and *a fortiori* non-deterministic; in fact automaton A_2 is ambiguous as well. The product accepts the strings $(\ a\ b\ )^*\ (\ a\ a\ b\ )^+$, which start by zero, one or more pairs $a\,b$ and end by at least one triplet $a\,a\,b$. Intuitively this characterizes the intersection language.

To complete: states $3A$, $4A$, $5A$ and $5B$ are redundant as their blue paths to $4C$ are covered by the black paths, thus such states can be removed; determinizing the product automaton by the subset construction (or by Berry-Sethi) needs to group states $2A$ and $2B$. The equivalent deterministic machine is so obtained:



This machine is already minimal (the reader can verify by himself). It is again at all evident that the strings of the intersection language are $(\ a\ b\ )^*\ (\ a\ a\ b\ )^+$.

2 Free Grammars and Pushdown Automata 20%

1. Consider a two-letter terminal alphabet $\Sigma = \{ a, b \}$, the free grammar G (non-terminal alphabet $V = \{ S, B \}$ and axiom S) over Σ, and the two regular expressions R_1 and R_2 over Σ, as defined below:

$$G \begin{cases} S & \rightarrow & a\ S\ b \mid B \\ B & \rightarrow & B\ b \mid \varepsilon \end{cases}$$

$$R_1 = (\, a\ a \mid b\ b\,)^* \qquad\qquad R_2 = \left((\, a \mid b\,)^2 \right)^* = \left(\Sigma^2 \right)^*$$

Answer the following questions:

(a) Describe the language $L\,(G)$ as a string set, by giving its characteristic predicate, as follows:

$$L\,(G) = \{\ \text{string description} \mid \quad \text{characteristic predicate}\ \}$$

(b) Write a grammar G_1, *BNF* and unambiguous, that generates the intersection language L_1 below:

$$L_1 = L \cap L\,(R_1)$$

(c) Write a grammar G_2, *BNF* and unambiguous, that generates the intersection language L_2 below:

$$L_2 = L \cap L\,(R_2)$$

(d) (optional) Say if the complement language $\overline{L_1}$ is free or not, and justify your answer. In the case it is free, hint how to write a grammar $\overline{G_1}$ for it.

Solution

(a) In the grammar G the nonterminal S generates an auto-inclusive (nested) structure of type $a^n\ B\ b^n$ $(n \geq 0)$ with *any equal* numbers of letters a and b, and the nonterminal B inserts in the centre *any* number (possibly zero) of additional letters b. Thus all the a's precede all the b's and the number of a's is less than or equal to that of b's. The result is a terminal string of type $a^h\ b^k$ with $0 \leq h \leq k$. Here is a possible characterization of language $L(G)$:

$$L(G) = \{\ \underbrace{a^h\ b^k}_{\text{description}}\ \mid\ \underbrace{0 \leq h \leq k}_{\text{predicate}}\ \}$$

Of course language $L(G)$ may have other characterizations, more or less similar. Here are variations[1] with a more complex description and a simpler predicate:

$$L(G) = \{\ \underbrace{a^m\ b^{m+n}}_{\text{description}}\ \mid\ \underbrace{m,\, n \geq 0}_{\text{predicate}}\ \} = \{\ \underbrace{a^m\ b^m\ b^*}_{\text{description}}\ \mid\ \underbrace{m \geq 0}_{\text{predicate}}\ \}$$

(b) Intersecting language $L(G)$ with the regular language $L(R_1)$ filters out all the strings that have an odd number (1, 3, etc) of letters a, and similarly b's. Anyway language L_1 is not regular as its strings, which are still infinitely many, may not have fewer b's than a's. Thus grammar G_1 has to have an auto-inclusive rule or derivation. Here is a possible grammar G_1 (axiom S), *BNF* and non-ambiguous:

$$G_1\ \begin{cases} S \ \rightarrow\ a\ a\ S\ b\ b\ \mid\ B & \text{generates } a^{2n}\ B\ b^{2n}\ (n \geq 0) \\ B \ \rightarrow\ B\ b\ b\ \mid\ \varepsilon & \text{generates } (\,b\ b\,)^* \end{cases}$$

In the grammar G_1 the nonterminal S generates an auto-inclusive (nested) structure of type $a^{2n}\ B\ b^{2n}$ $(n \geq 0)$ with *equal even* numbers (0, 2, etc) of letters a and b, and the nonterminal B inserts in the centre an *even* number (possibly zero) of additional letters b. Thus grammar G_1 exactly generates language L_1. It is unambiguous as it consists of the classical unambiguous rule for nested structures and of a left-linear rule, which clearly is not ambiguous either.

(c) Intersecting language $L(G)$ with the regular language $L(R_2)$ filters out all the strings that have an odd total length. Anyway language L_2 and grammar G_2 are not regular either, basically for the same reasons explained before for L_1 and G_1. Here is a possible grammar G_2 (axiom S), *BNF* and non-ambiguous:

$$G_2\ \begin{cases} S \ \rightarrow\ a\ S\ b\ \mid\ B & \text{generates } a^n\ B\ b^n\ (n \geq 0) \\ B \ \rightarrow\ B\ b\ b\ \mid\ \varepsilon & \text{generates } (\,b\ b\,)^* \end{cases}$$

In the grammar G_2 the nonterminal S generates an auto-inclusive (nested) structure of type $a^n\ B\ b^n$ $(n \geq 0)$ with *any equal* numbers of letters a and b, which so in total has an even number of terminals, i.e., $|\,a^n\ B\ b^n\,|_{a,\,b} = 2n$, and the nonterminal B inserts in the centre an *even* number (possibly zero) of additional letters b. Thus grammar G_2 exactly generates language L_2. It is unambiguous essentially for the same reasons as for the grammar G_1 above.

[1] The latter variation ought to be more rigorously written $\{\ s \mid\ \ s \in L(a^m\ b^m\ b^*)\ \wedge\ m \geq 0\ \}$, as a *RE* is not one string; however the shortcut form $\{\ a^m\ b^m\ b^* \mid\ \ m \geq 0\ \}$ shown above is at all acceptable.

(d) Despite the family of free languages is not closed under complement, it turns out that the complement language $\overline{L_1}$ is free and this property is provable as follows. Since language L_1 is defined as the intersection of two languages, the De Morgan laws apply and the complement language $\overline{L_1}$ can be expressed as follows:

$$\overline{L_1} = \overline{L \cap L(R_1)} = \overline{L} \cup \overline{L(R_1)}$$

Language $\overline{L}$ is free as it can be expressed as the union of these two languages:

$$\overline{L} = \Sigma^* \, b \, a \, \Sigma^* \cup \left\{ a^h \, b^k \mid \quad 0 \leq k < h \right\}$$

The former language contains the strings with a letter b that precedes a letter a, and the latter one contains the strings with orderly more letters a than b. The former is obviously regular as it is formulated through a regular expression. This regular expression is stood to be ambiguous to keep it easy and short. The latter is free since its characteristic predicate $0 \leq k < h$ is essentially the same as that of language $L(G)$, provided the roles of letters a and b are switched and the letters a exceed the b's. Here is a viable grammar for it (axiom S):

$$\begin{cases} S \;\rightarrow\; a \, S \, b \mid A & \text{generates } a^n \, A \, b^n \; (n \geq 0) \\ A \;\rightarrow\; a \, A \mid a & \text{generates } a^+ \end{cases}$$

directly derived from grammar G, or more simply by using only one nonterminal:

$$S \rightarrow a \, S \, b \mid a \, S \mid a$$

which is also ambiguous because the two recursive rules can be freely commuted. Thus language $\overline{L}$ is free as it is the union of a regular language and a free one. Then language $\overline{L(R_1)}$ is regular as it is the complement of a regular language. Finally the complement language $\overline{L_1}$ is free as it is the union of a free language and a regular one. This proves what has been stated before.
Concerning how to write a grammar for language $\overline{L_1}$. A union grammar $\overline{G}$ can be easily written for the previous language $\overline{L}$, namely (axiom S):

$$\overline{G} \begin{cases} S \;\rightarrow\; X \mid Z & \\ X \;\rightarrow\; Y \, b \, a \, Y & \text{generates } \Sigma^* \, b \, a \, \Sigma^* \\ Y \;\rightarrow\; a \, Y \mid b \, Y \mid \varepsilon & \text{generates } \Sigma^* \\ Z \;\rightarrow\; a \, Z \, b \mid a \, Z \mid a & \text{generates } \left\{ a^h \, b^k \mid \quad 0 \leq k < h \right\} \end{cases}$$

Since language $\overline{L(R_1)}$ is regular, it has a right-linear grammar $\overline{G_1}$, e.g., obtained from the complement regular expression $\overline{R_1}$ below:

$$\overline{R_1} = \Sigma \left(\Sigma^2 \right)^* \mid \left(\Sigma^2 \right)^* (a \, b \mid b \, a) \left(\Sigma^2 \right)^*$$

Expression $\overline{R_1}$ is the union of the strings of odd length and those of even length that certainly contain at least one adjacent pair $a\,b$ or $b\,a$, and it is also formulated ambiguously. To obtain grammar $\overline{G_1}$ it suffices to design a *FSA* equivalent to $\overline{R_1}$, e.g., by the Berry-Sethi method, and then to rewrite the *FSA* as a right-linear grammar. Alternatively one might obtain a grammar (not necessarily a right-linear one) for $\overline{L(R_1)}$ by the structural analysis of $\overline{R_1}$ (see the textbook). Thus a union grammar $\overline{G} \cup \overline{G_1}$ can be written for the complement language $\overline{L_1}$. Notice that grammar $\overline{G}$ is ambiguous and that so surely is the union grammar (it might even have other ambiguity forms); anyway unambiguity is not requested. One might investigate whether an unambiguous grammar can be found, too.

2. An mp3 player can process compilations that describe sound tracks and playlists. There are two musical genres: *classical* and *jazz*. Here are the language specifications:

   - A sound track consists of the following attributes, mandatory or optional:
     - the *genre*, denoted by the keywords **classical** or **jazz** (mandatory)
     - the *composer*, a string (mandatory)
     - the *performer*, a string (optional)
     - the *title*, a string (mandatory)
     - the *duration*, two integers for minutes and seconds (both mandatory)
     - the *size* in KBytes, an integer number (mandatory)
   - A playlist consists of a non-empty list of sound tracks and/or nested playlists, according to the following rules:
     - sound tracks and playlists can occur in any order
     - there are three types of playlist: *miscellaneous*, *classical* and *jazz*
     - a playlist of type miscellaneous can include sound tracks of any genre and playlists of any type
     - a playlist of type classical or jazz can include only sound tracks of the same genre and playlists of the same type

   A compilation consists of a non-empty list of sound tracks and playlists, in any order. Here is a sample compilation:

```
classical track
  composer Brahms  performer Levy  title Ballade  min 3 sec 41  size 4782
end track
miscellaneous playlist
  jazz track
    composer Davis  title SoWhat  min 2 sec 44  size 6423
  end track
  classical playlist
    classical track
      composer JSBach  performer Kremer  title Aria
      min 3 sec 34   size 7239
    end track
    classical playlist
      classical track
        composer WAMozart  performer Bollini  title prelude
        min 5 sec 21  size 8451
      end track
      classical track
        composer Vivaldi  title concert in F  min 12 sec 45  size 42683
      end track
    end playlist
  end playlist
end playlist
```

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

Solution

Here is a reasonable grammar G (axiom COMP), of type *EBNF*, for the sketched language of music compilations for mobile palyers and similar devices:

$$
G \left\{
\begin{array}{rcl}
\langle \mathsf{COMP} \rangle & \to & \big(\langle \mathsf{STRACK} \rangle \mid \langle \mathsf{PLIST} \rangle \big)^{+} \\
\hline
\langle \mathsf{STRACK} \rangle & \to & \big(\mathsf{classical} \mid \mathsf{jazz} \big) \langle \mathsf{STBODY} \rangle \\
\langle \mathsf{PLIST} \rangle & \to & \langle \mathsf{MPL} \rangle \mid \langle \mathsf{CPL} \rangle \mid \langle \mathsf{JPL} \rangle \\
\hline
\langle \mathsf{STBODY} \rangle & \to & \mathsf{track} \\
 & & \quad \mathsf{composer}\ \mathbf{s} \\
 & & \quad \big[\mathsf{performer}\ \mathbf{s} \big] \\
 & & \quad \mathsf{title}\ \mathbf{s} \\
 & & \quad \mathsf{min}\ \mathbf{n}\ \mathsf{sec}\ \mathbf{n} \\
 & & \quad \mathsf{size}\ \mathbf{n} \\
 & & \mathsf{end\ track} \\
\langle \mathsf{MPL} \rangle & \to & \mathsf{miscellaneous\ playlist} \\
 & & \quad \langle \mathsf{COMP} \rangle \\
 & & \mathsf{end\ playlist} \\
\langle \mathsf{CPL} \rangle & \to & \mathsf{classical\ playlist} \\
 & & \quad \big(\mathsf{classical}\ \langle \mathsf{STBODY} \rangle \mid \langle \mathsf{CPL} \rangle \big)^{+} \\
 & & \mathsf{end\ playlist} \\
\langle \mathsf{JPL} \rangle & \to & \mathsf{jazz\ playlist} \\
 & & \quad \big(\mathsf{jazz}\ \langle \mathsf{STBODY} \rangle \mid \langle \mathsf{JPL} \rangle \big)^{+} \\
 & & \mathsf{end\ playlist}
\end{array}
\right.
$$

A terminal $\mathbf{s}$ or $\mathbf{n}$ represents a string or a number, respectively, and is left unexpanded. The square brackets indicate optionality, as usual.

Grammar G reuses the axiom COMP in the rules, yet this could be avoided if necessary by normalizing the grammar as known. By construction, grammar G is reasonably correct and unambiguous, as it consists of structures known to be unambiguous, like lists, disjoint alternatives, etc, and of auto-inclusive (self-embedded or nested) structures that are not ambiguous either, like the rules of nonterminals XPL with $\mathsf{X} = \mathsf{M}, \mathsf{C}, \mathsf{J}$. Of course there may be other formulations, without or with minor syntactic differences in the language, and one of them is shown in the following.

Here is a slightly different equivalent formulation G' of grammar G (axiom COMP). It uses the same syntactic classes as G, which are all reported below, and only changes a few rules a little; anyway it generates the same language as G, i.e., $L(G') = L(G)$:

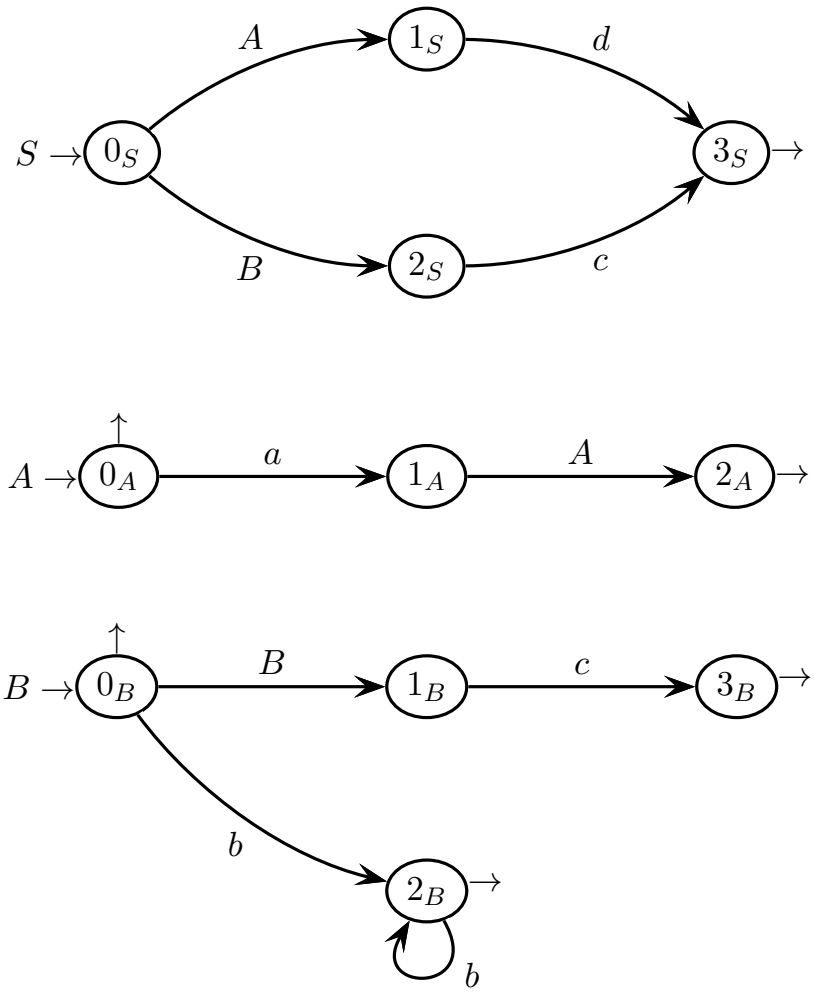
$$
G' \left\{
\begin{array}{rcl}
\langle \mathsf{COMP} \rangle & \rightarrow & \big(\langle \mathsf{STRACK} \rangle \mid \langle \mathsf{PLIST} \rangle \big)^{+} \\
\hline
\langle \mathsf{STRACK} \rangle & \rightarrow & \big(\mathsf{classical} \mid \mathsf{jazz} \big) \langle \mathsf{STBODY} \rangle \\
\langle \mathsf{PLIST} \rangle & \rightarrow & \mathsf{miscellaneous} \ \langle \mathsf{MPL} \rangle \mid \\
 & & \mathsf{classical} \ \langle \mathsf{CPL} \rangle \mid \\
 & & \mathsf{jazz} \ \langle \mathsf{JPL} \rangle \\
\hline
\langle \mathsf{STBODY} \rangle & \rightarrow & \mathsf{track} \\
 & & \quad \mathsf{composer} \ \mathbf{s} \\
 & & \quad \big[\mathsf{performer} \ \mathbf{s} \big] \\
 & & \quad \mathsf{title} \ \mathbf{s} \\
 & & \quad \mathsf{min} \ \mathbf{n} \ \mathsf{sec} \ \mathbf{n} \\
 & & \quad \mathsf{size} \ \mathbf{n} \\
 & & \mathsf{end\ track} \\
\langle \mathsf{MPL} \rangle & \rightarrow & \mathsf{playlist} \ \langle \mathsf{COMP} \rangle \ \mathsf{end\ playlist} \\
\langle \mathsf{CPL} \rangle & \rightarrow & \mathsf{playlist} \\
 & & \quad \Big(\mathsf{classical} \ \big(\langle \mathsf{STBODY} \rangle \mid \langle \mathsf{CPL} \rangle \big) \Big)^{+} \\
 & & \mathsf{end\ playlist} \\
\langle \mathsf{JPL} \rangle & \rightarrow & \mathsf{playlist} \\
 & & \quad \Big(\mathsf{jazz} \ \big(\langle \mathsf{STBODY} \rangle \mid \langle \mathsf{JPL} \rangle \big) \Big)^{+} \\
 & & \mathsf{end\ playlist}
\end{array}
\right.
$$

The difference between grammars G and G' is just in the way of factoring the genre attribute for playlists. Grammar G specifies the list type at a lower level in each playlist rule, whereas grammar G' specifies it at a higher level before expanding each playlist. Only the playlist rules are changed, all the others are left unchanged.

To conclude: though there may be even more numerous and diverse variants, by generalizing the two above solutions a common feature of any grammar for the sketched language seems to be that of having a syntactic class (nonterminal) for each playlist type (here the classes are XPL with $\mathsf{X} = \mathsf{M}, \mathsf{C}, \mathsf{J}$). Such an hypothesis is also supported by a reasoning. A playlist is a recursive and auto-inclusive structure, and for this reason its contents cannot be generated by a regular expression alone. Thus, since there are three different playlist types, as many syntactic classes seem to be a necessity. The same does not apply to the sound tracks, which are finite structures.

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 $V = \{ S, A, B \}$ (axiom S).



Answer the following questions:

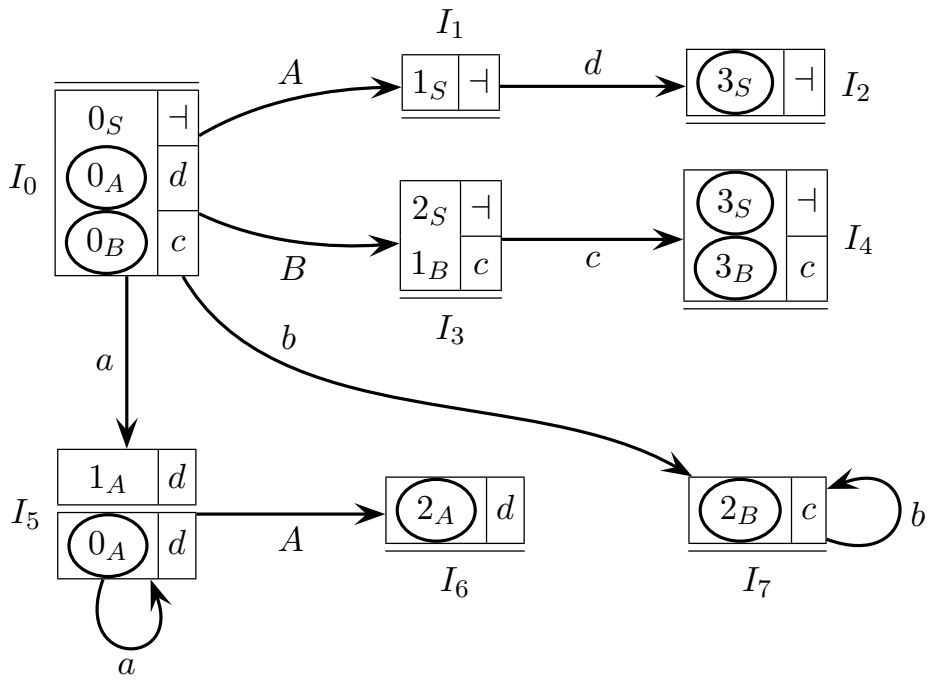
(a) Draw the complete pilot of grammar G, say if grammar G is of type $ELR(1)$ and shortly justify your answer. If the grammar is not $ELR(1)$ then highlight all the conflicts in the pilot.

(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. If you wish, you can use the figure above to add the call arcs and annotate the guide sets.

(c) (optional) Say if the language $L(G)$ is of type $ELL(1)$, and in the case it is provide a grammar G' that has the $ELL(1)$ property.

Solution

(a) Here is the pilot of grammar G, which has 8 m-states (m-state I_0 is initial):

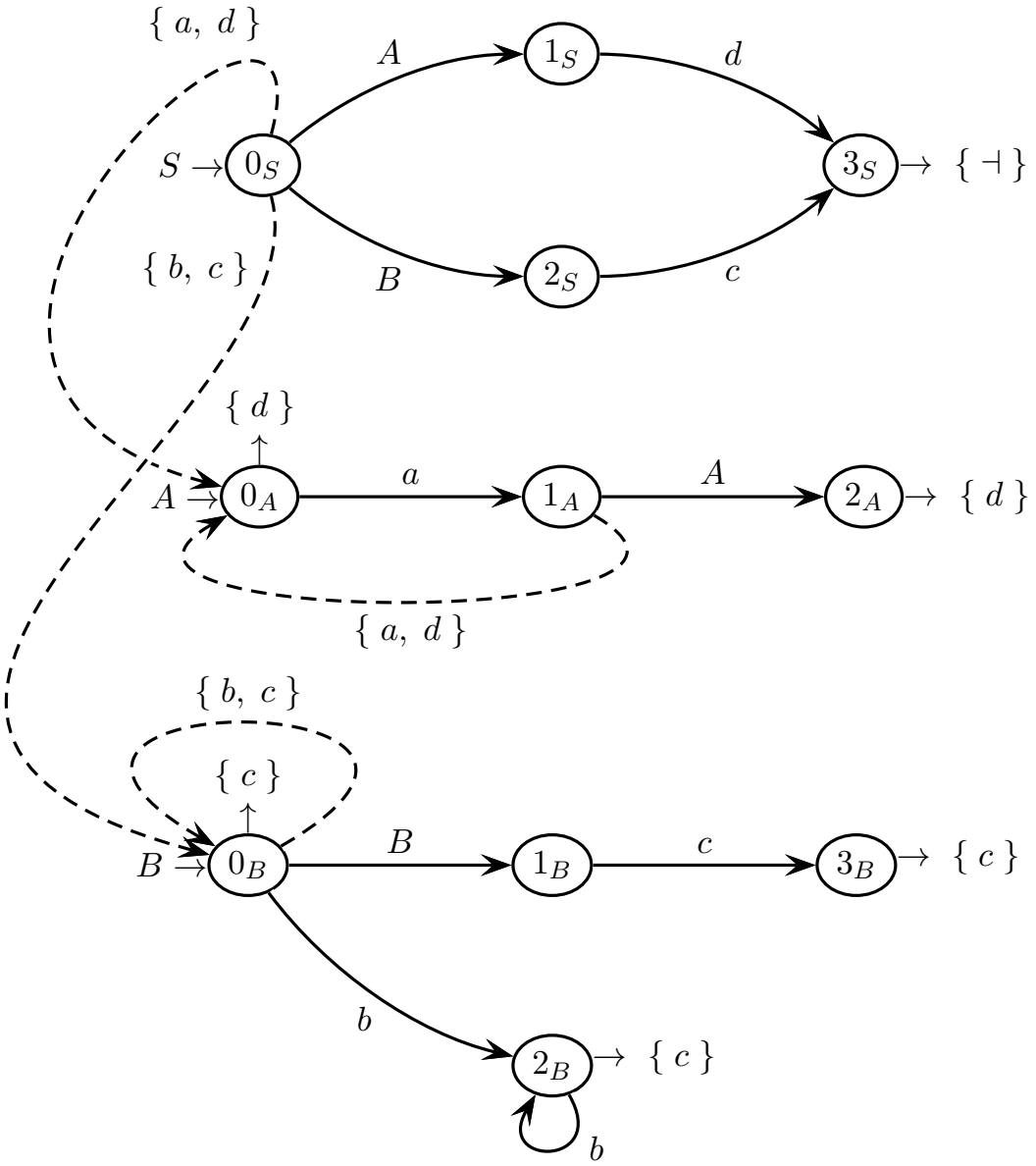


Here is the detailed analysis of the *ELR* potential conflicts in the pilot graph:

- the look-ahead terminals of the final states in the m-states I_0, I_2, I_4, I_5, I_6 and I_7 are different from the terminals on the shift arcs outgoing from the respective m-states, thus there are not any shift-reduce conflicts
- the look-ahead terminals in each pair of final states in the m-states I_0 and I_4 are pairwise different, thus there are not any reduce-reduce conflicts
- there are two multiple transitions (with multiplicity equal to 2), namely $I_0 \xrightarrow{B} I_2$ and $I_2 \xrightarrow{c} I_4$, yet none of them is convergent as their destination states are pairwise different, thus there are not any convergence conflicts

In conclusion there are not any conflicts of type shift-reduce, reduce-reduce or convergence, therefore grammar G is of type $ELR(1)$. Notice that the pilot does not have the single transition property (*STP*) as the bases of the m-states I_3 and I_4 (destinations of the multiple transitions) contain two states each.

(b) Here are all the guide sets of grammar G, those on the call arcs and exit arrows, while those on the terminal shift arcs are obvious and here are omitted:



There are overlapping guide sets in the state 0_B: on the call and terminal shift arcs, the guide sets of which share terminal b, as well as on the call arc and exit arrow, the guide sets of which share terminal c. Indeed some overlapping is expected since the machine M_B of nonterminal B is left-recursive. The immediate left-recursiveness of M_B soon catches the eye because of the call arc self-loop on state 0_B. Such a loop is fatal to the $ELL(1)$ property as its guide set adsorbs all the guide sets of the branching shift arcs, here the outgoing b-arc, and this behaviour alone breaks the $ELL(1)$ property. The presence of terminal c in the same set has a different cause, namely that nonterminal B is nullable. Thus grammar G is not $ELL(1)$ and actually the pilot does not have the STP either. State 0_B is the only point in the net where the $ELL(1)$ property fails.

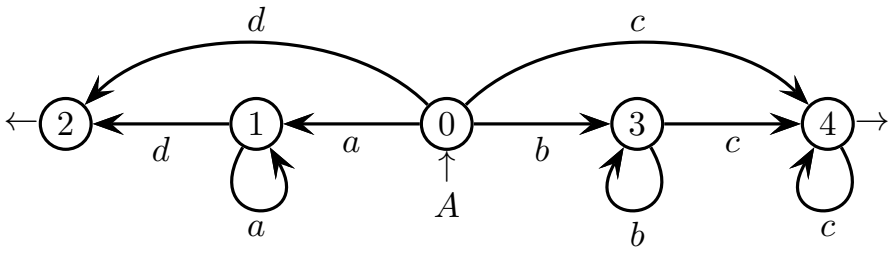
(c) Language $L(G)$ turns out to be regular. In fact it is not difficult to obtain a regular expression or a deterministic *FSA* for this language, and consequently a right-linear grammar. Since the rules of the nonterminals A and B are right and left linear, respectively, the Arden identity yields these regular expressions:

$$L(A) = a^* \qquad\qquad L(B) = b^*\ c^*$$

and consequently by substitution the following regular expression:

$$L(S) = a^*\ d\ \mid\ b^*\ c^+ = L(G)$$

so that language $L(G)$ is actually regular. A deterministic finite-state automaton A equivalent to grammar G can be easily obtained intuitively, as follows:



It is also easy to verify that the (deterministic) automaton A is minimal, though minimality is unnecessary in what follows. Here is the right-unilinear grammar G' associated with A (axiom 0), obviously equivalent to grammar G:

$$G' \begin{cases} 0 \ \rightarrow\ a\,1 \mid d\,2 \mid b\,3 \mid c\,4 \\ 1 \ \rightarrow\ a\,1 \mid d\,2 \\ 2 \ \rightarrow\ \varepsilon \\ 3 \ \rightarrow\ b\,3 \mid c\,4 \\ 4 \ \rightarrow\ c\,4 \mid \varepsilon \end{cases}$$

As automaton A is deterministic the right-unilinear grammar G' is notoriously $LL(1)$ and *a fortiori* it is also $ELL(1)$. Therefore language $L(G)$ is $ELL(1)$.

A different and less radical approach is to replace the left-linear and left-recursive *EBNF* rule $B \rightarrow B\ c\ \mid\ b^*$ of nonterminal B (obtained from machine M_B) with one or more equivalent rules (*BNF* or *EBNF*) that are not left-recursive. Left recursion can always be turned into right recursion by the known grammar transformation (see the textbook), yet here intuition suffices. For instance the two following *BNF* rules, which use two nonterminals instead of only one:

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

generate the same language $b^*\ c^*$ as the original *EBNF* rule does, yet they are right-recursive. Furthermore, clearly they are $LL(2)$ as the look-ahead sets of the alternatives $C \rightarrow c\,C$ and $C \rightarrow \varepsilon$ are $\{\ c\ c\ \}$ and $\{\ c\ \dashv\ \}$, respectively, and *a fortiori* they are even $ELL(2)$. Therefore language $L(G)$ is *ELL*, but with the minor detail that the look-ahead depth obtained is 2 instead of only 1.

4 Language Translation and Semantic Analysis 20%

1. Let string w_k denote the binary encoding of the natural number k (in the following natural numbers are represented in decimal notation, unless otherwise specified), and let string w_k^R denote the mirror image of w_k. For instance:

   $$w_{12} = 1\,1\,0\,0 \qquad \text{and} \qquad w_{12}^R = 0\,0\,1\,1$$

   Consider the following translation function τ, defined on the binary strings of type w_k^R, such that it holds $k > 0$ (i.e., the encoded natural number is strictly positive) and such that the binary encoding does not have any leading zeros:

   $$\tau\left(w_k^R\right) = w_{k+1}$$

   Here are a few translation samples:

   - $\tau\,(0\,0\,1\,1) = 1\,1\,0\,1$ as $w_{12} = 1\,1\,0\,0$ (the encoding of 12 is $1\,1\,0\,0$) and $w_{13} = 1\,1\,0\,1$
   - $\tau\,(1\,1) = 1\,0\,0$ as $w_3 = 1\,1$ (the encoding of 3 is $1\,1$) and $w_4 = 1\,0\,0$
   - $\tau\,(0\,0\,1) = 1\,0\,1$ as $w_4 = 1\,0\,0$ (the encoding of 4 is $1\,0\,0$) and $w_5 = 1\,0\,1$
   - $\tau\,(1\,1\,0\,1) = 1\,1\,0\,0$ as $w_{11} = 1\,0\,1\,1$ (the encoding of 11 is $1\,0\,1\,1$) and $w_{12} = 1\,1\,0\,0$

   On the other hand:

   - $\tau\,(0) =$ *undefined* because the argument is zero
   - $\tau\,(1\,1\,0) =$ *undefined* because of the leading zero in the source string

   A simple *BNF* grammar that generates the source language is as below (axiom L):

   $$\begin{cases} L & \rightarrow & 0\;L \\ L & \rightarrow & 1\;L \\ L & \rightarrow & 1 \end{cases}$$

   Answer the following questions:

   (a) Write a *BNF* translation grammar G_τ, or scheme, that defines the above described translation τ. If necessary you can change the source grammar.

   (b) With reference to grammar G_τ, draw the syntax trees for the following translation samples: $\tau\,(1) = 1\,0$, $\tau\,(1\,1) = 1\,0\,0$ and $\tau\,(1\,1\,0\,1) = 1\,1\,0\,0$

   (c) Argue that grammar G_τ is such that the translation samples $\tau\,(0)$ and $\tau\,(1\,1\,0)$ are both undefined.

   (d) (optional) Is translation τ deterministic ? Briefly justify your answer.

Solution

(a) A possibility is to model translation τ on the well known binary addition algorithm, based on carry propagation (carry ripple). Since the carry bit is either 0 or 1, two nonterminals can represent it, say U and C, respectively. Here is a working translation grammar G_τ with three nonterminals (axiom L):

G_τ

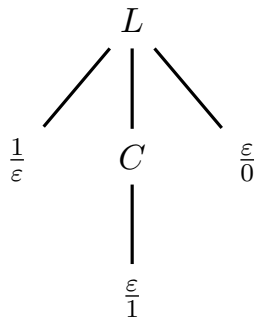
translation rule			*comment*
L	$\to$	$\frac{0}{\varepsilon}\ U\ \frac{\varepsilon}{1}$	lsb 0: switch to 1 without carry out
L	$\to$	$\frac{1}{\varepsilon}\ C\ \frac{\varepsilon}{0}$	lsb 1: switch to 0 with carry out
U	$\to$	$\frac{0}{\varepsilon}\ U\ \frac{\varepsilon}{0}$	0 without carry in: keep as 0
U	$\to$	$\frac{1}{\varepsilon}\ U\ \frac{\varepsilon}{1}$	1 without carry in: keep as 1
U	$\to$	$\frac{1}{1}$	msb 1 without carry in: keep as 1 and finish
C	$\to$	$\frac{0}{\varepsilon}\ U\ \frac{\varepsilon}{1}$	0 with carry in: switch to 1 without carry out
C	$\to$	$\frac{1}{\varepsilon}\ C\ \frac{\varepsilon}{0}$	1 with carry in: switch to 0 with carry out
C	$\to$	$\frac{\varepsilon}{1}$	overflow: prepend 1 and finish

As customary in computer arithmetic the nicknames "lsb" and "msb" mean least and most significant bit, respectively. Grammar G_τ models the well known binary addition algorithm (carry ripple). Since here what is wanted is just to increment the source string, a bit 1 is injected into the position of the lsb, it is added to the lsb and the carry is possibly propagated and added in the direction of the msb. Overflow may occur, in which case the translation will be longer than the source. If the result wants one more bit, this is prepended to its msb.

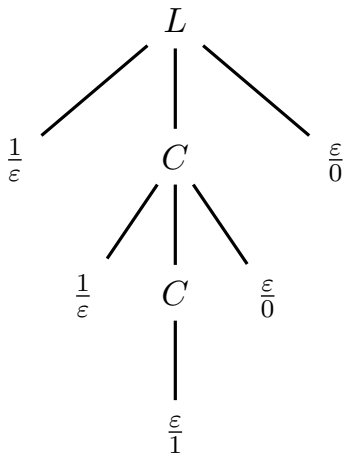
The source grammar is modified by adding two nonterminals: U (unchanged), to mean that there is not any carry propagation (carry 0) and thus that the rest of the string will be unchanged; and C (changed), to mean that there is carry propagation (carry 1) and a bit 1 has to be added to the next position, and thus that the rest of the string will be more or less changed. Of course once the carry propagation ends, it will not be resumed any longer.

The source and destination terminals are placed on the left and right ends of the right parts of the rules, because the source string is generated in mirrored form to have the lsb at the left end, whereas the destination string is generated as usual with the lsb on the right end. Consequently grammar G_τ has a seemingly palindromic structure, where the arithmetic orderings of input and output are mirrored. However the source and destination grammars are right and left linear, respectively, and thus the source and destination languages are regular.

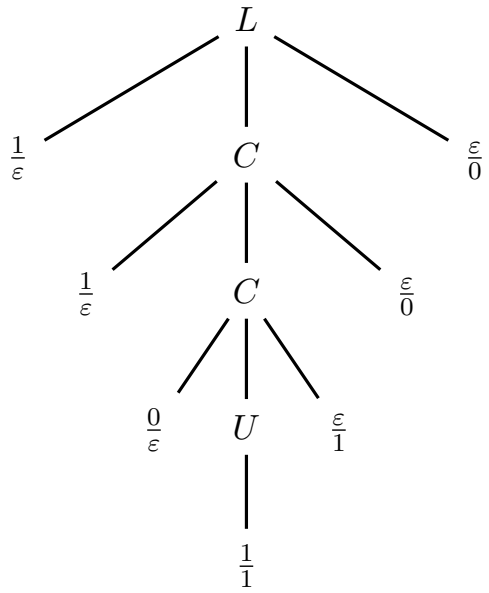
(b) Here are the three requested translation trees (source and destination unified):



$\tau\,(1) = 1\,0$



$\tau\,(1\,1) = 1\,0\,0$



$\tau\,(1\,1\,0\,1) = 1\,1\,0\,0$

Remember that the binary number that figures as the argument of translation τ has the lsb on the left end (or at the tree top), that is, opposite to the usual position, whereas the translation result has the lsb on the right end (or at the tree bottom), as usual. The tree stem encodes the carry ripple chain, if any.

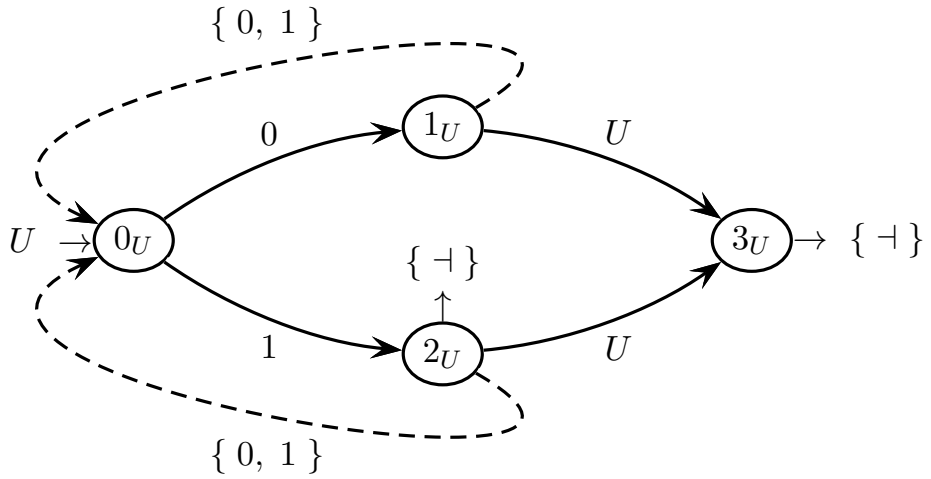
(c) Translation τ is undefined for the source strings 0 (zero) and 1 1 0 (it has a leading zero) as these are invalid strings, not generated by the source grammar. The non-membership check is trivial and it is left to the reader.

(d) Translation τ is deterministic because the source grammar is $LL(2)$. Here are the guide sets of each *BNF* source rules (of course right-linear):

source rule	*guide set*	*look-ahead depth* $k =$
$L \rightarrow 0\,U$	$\{\ 0\ \}$	1
$L \rightarrow 1\,C$	$\{\ 1\ \}$	1
$U \rightarrow 0\,U$	$\{\ 0\ \}$	1
$U \rightarrow 1\,U$	$\{\ 1\,0,\ 1\,1\ \}$	2
$U \rightarrow 1$	$\{\ 1 \dashv\ \}$	2
$C \rightarrow 0\,U$	$\{\ 0\ \}$	1
$C \rightarrow 1\,C$	$\{\ 1\ \}$	1
$C \rightarrow \varepsilon$	$\{\ \dashv\ \}$	1

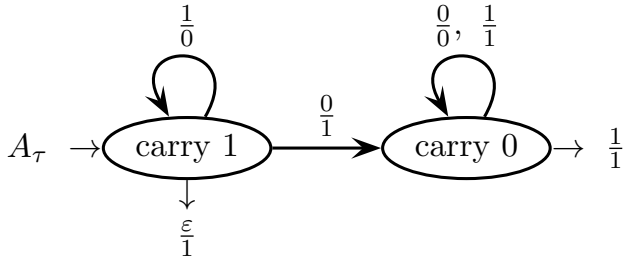
Notice that all the guide sets of the alternative rules are disjoint. Actually only the second and third alternative rules of nonterminal U need a look-ahead depth $k = 2$ for them to be disjoint, whereas for all the other rules a look-ahead depth $k = 1$ is enough. However the source grammar is $LL(2)$ as a whole and thus translation τ is deterministic. It is easy to design a recursive descent translator for τ, by starting from the analyzer and finalizing it with write actions.

To continue: since the (source grammar of the) translation grammar G_τ is $LL(2)$, *a fortiori* it is $ELL(2)$ and so $ELR(2)$. Is it even $ELL(1)$ and so $ELR(1)$? The answer is yes. First unify the three source rules of nonterminal U into a deterministic machine completed with call arcs and guide sets, as follows:

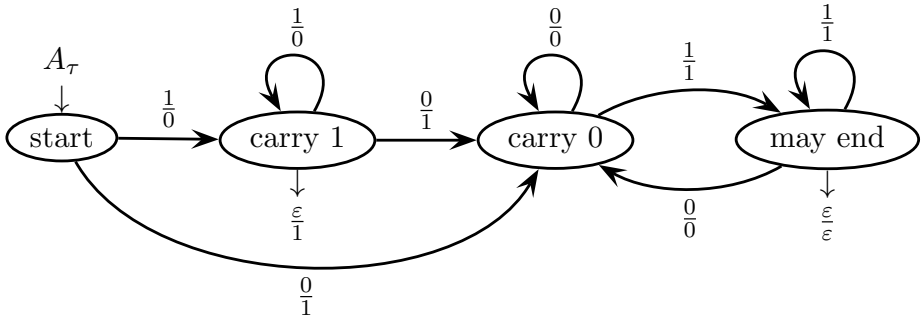


Then see that now the guide sets with look-ahead depth $k = 1$ on state 2_U are disjoint because the shared terminal 1 of the 2-nd and 3-rd alternative rules of U is factored. Thus the source grammar is $ELL(1)$ and *a fortiori* it is $ELR(1)$. Second notice that the destination grammar (write it !) is left-linear and thus that it is in the postfix from: the output symbols, if any, occur only at the end of the destination rules or equivalently on the exit arrows of the corresponding machines. Therefore the translation grammar G_τ is $ELR(1)$, too. It is easy to design a bottom-up translator for τ, by starting from the bottom-up analyzer and finalizing it with write actions in the reduction m-states of the pilot.

To complete: one might question whether translation τ has a generic pushdown deterministic translator, not necessarily one derived from a grammar, or even whether it has a finite-state translator. The answer is yes again. In fact translation τ is purely rational, despite at a first glance grammar G_τ purports it may be strictly free. A generic translator for τ has to scan the input from lsb to msb and to output the result again from lsb to msb, possibly with one more bit 1 prepended to the msb. Such a machine needs a one-bit memory for the carry. Thus a finite-state sequential translator A_τ suffices. Here is a (tentative) draft:

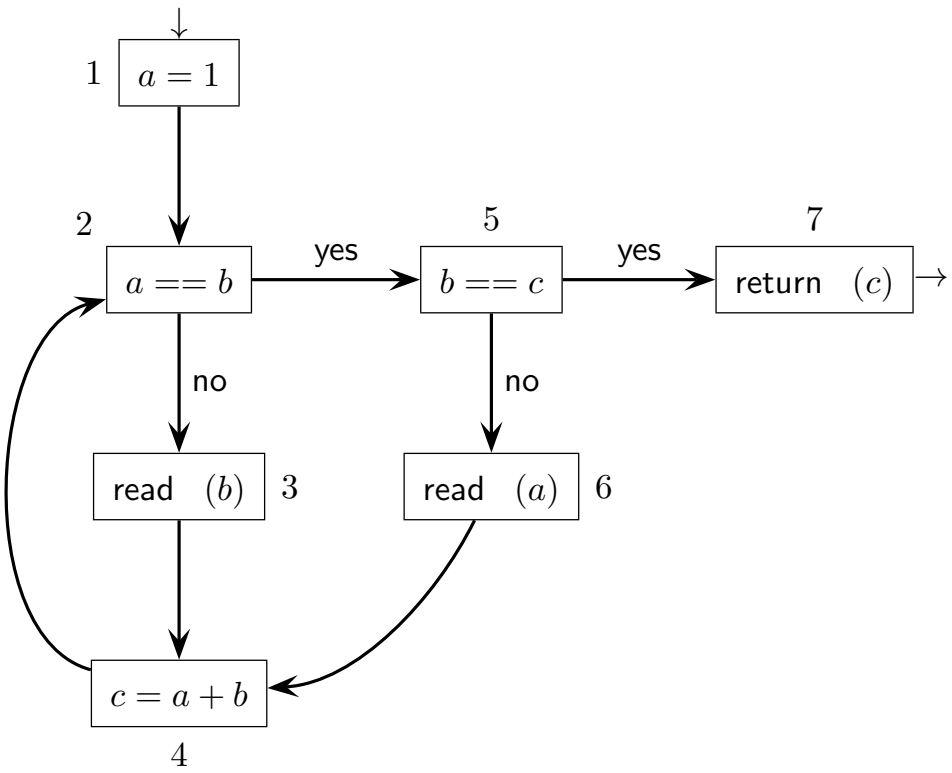


This draft of A_τ has only two states, i.e., a one-bit memory (see the state names). State "carry 1" complements and outputs the source string, whereas state "carry 0" outputs it unchanged. The draft is slightly erroneous as it accepts the source empty string ε and translates ε to 1, which is nonsense. Furthermore it is limitedly nondeterministic in the state "carry 0", where it has two 1-transitions (one final). It can be corrected and determinized by using more states, as follows:



Notice that states "start" and "may end" help reject the source empty string and avoid nondeterminism, respectively. Now translator A_τ is deterministic as the underlying recognizer A is such. Thus translation τ is deterministic as well, and a pushdown translator is even too powerful for τ as a finite-state one suffices. Of course translator A_τ can be rewritten as a right-linear translation grammar, which will be of type $LL(1)$ as the automaton it comes from is deterministic.

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



Variables b and c are input parameters, and variable c is also output parameter, while variable a is local.

Answer the following questions:

(a) Find the *live variables* of the given *CFG*, by directly applying the definition of live variable. Write the live variables on the *CFG* prepared on the next pages.

(b) Find the *reaching definitions* of the given *CFG*, by means of the systematic method of the flow equations for reaching definitions: first write the equations, then iteratively solve them (use the tables prepared on the next pages). Write the reaching definitions on the *CFG* prepared on the next pages.

---

please here write the **LIVE VARIABLES** (at the node inputs) obtained directly

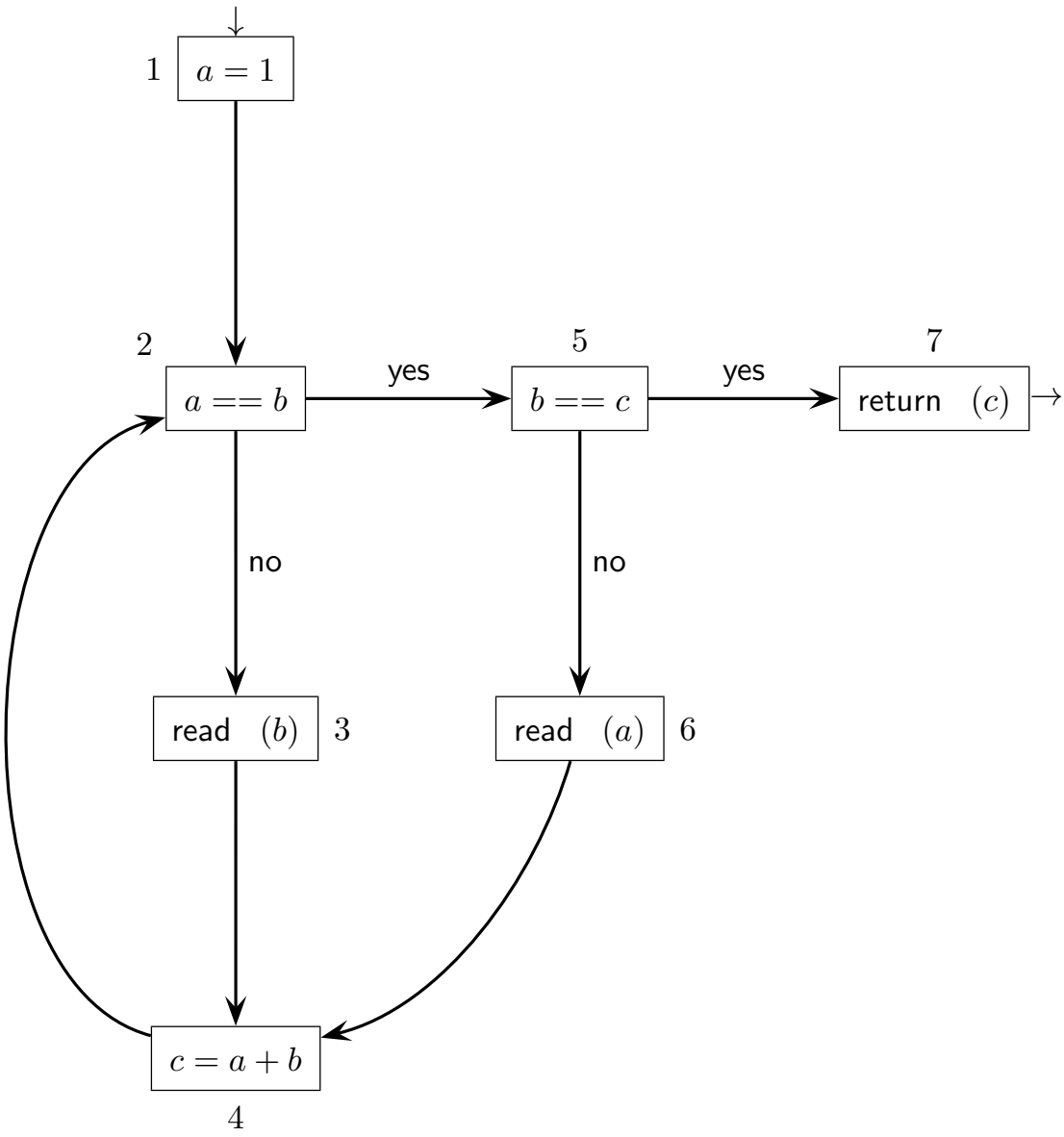


table of definitions and suppressions at the nodes
for **REACHING DEFINITIONS**

node	*defined*
1	
2	
3	
4	
5	
6	
7	

node	*suppressed*
1	
2	
3	
4	
5	
6	
7	

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

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

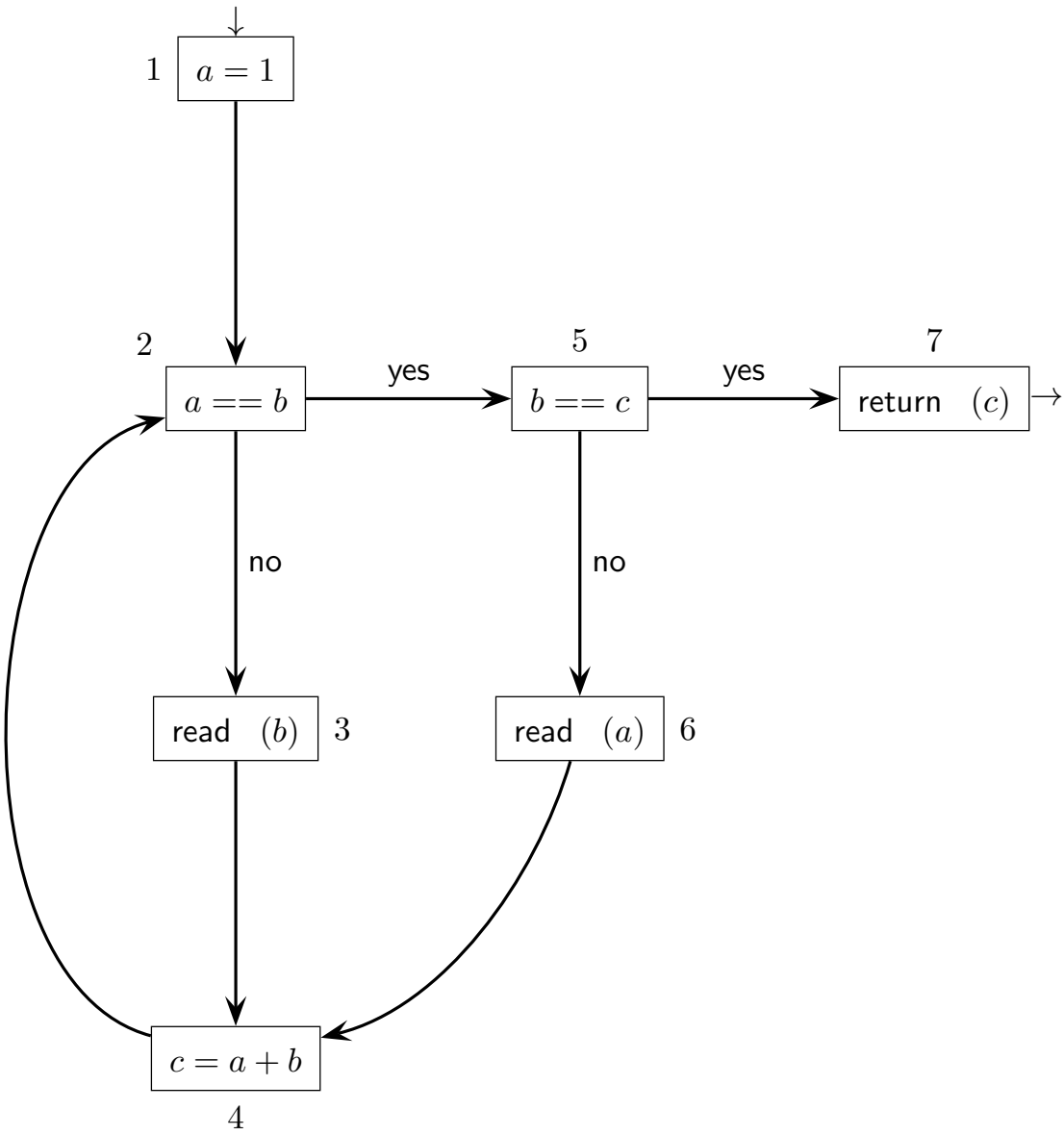
iterative solution table of the system of data-flow equations (**REACHING DEF.S**)

(the number of tables and columns is not significant)

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

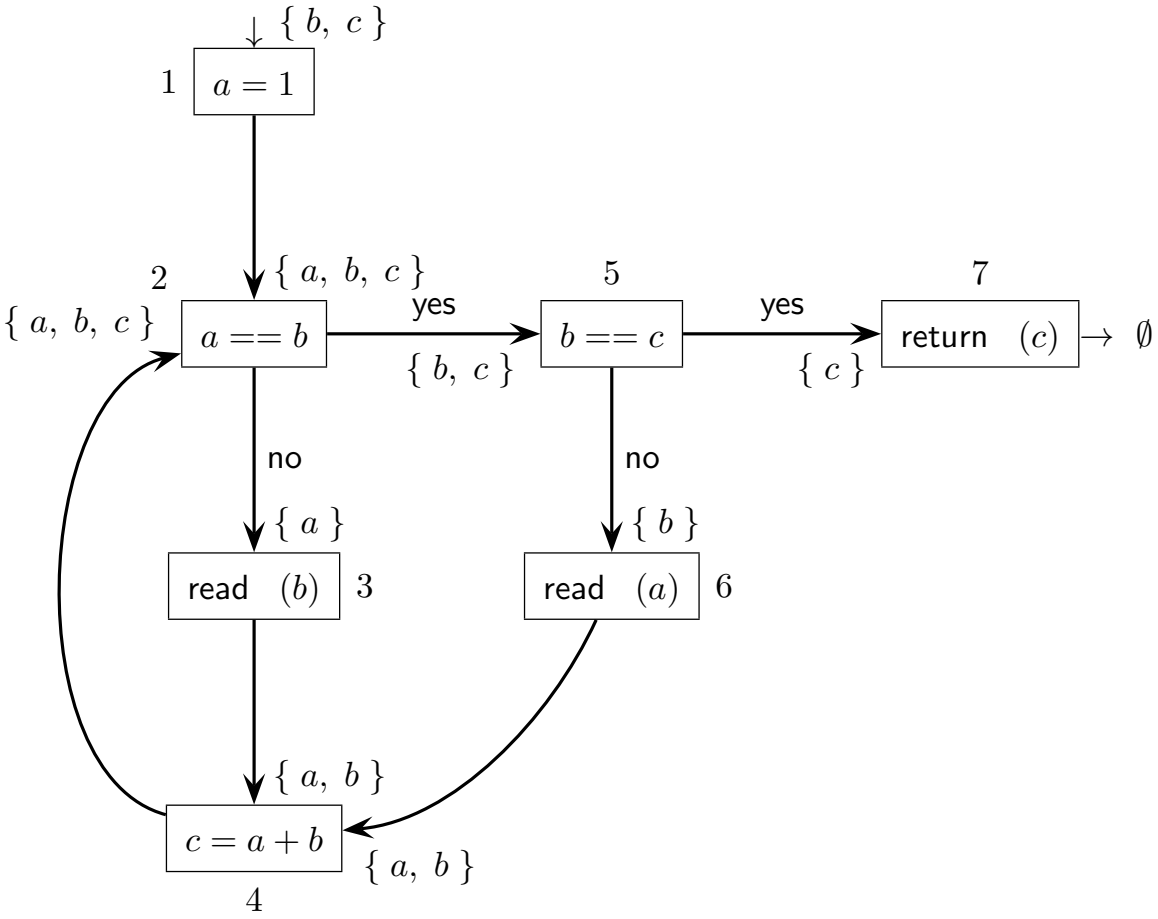
	7		8		9		10		11		12		13	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1														
2														
3														
4														
5														
6														
7														

please here write the **REACHING DEFINITIONS** (at the node outputs)
obtained systematically



Solution

(a) Here are the live variables at the node inputs, obtained directly from the definition of liveness (annotated next to the arrow tips to mean they are input):



Of course all the inputs to the same node bear the same live variables. Variables b and c are live on the input node of the *CFG*, as they are input and then used in the *CFG*. Variable c is defined in the *CFG* and then is output, yet it is not live at the output node of the *CFG*, as whether it is used subsequently is unknown.

To examine a few cases: variable a is not live at the input of node 1 as 1 defines a; b and c are live at 1 since they are input parameters assigned outside of the program and are used at 2 successor of 1; a, b and c are live at 2 since all of them have been previously assigned, a and b are used at 2, and c is used at 3 successor of 2; only a is live at 3 since it has been previously assigned and is used at 4 successor of 3, while b and c are assigned at 3 and 4 successor of 3 without being used; and so on. The reader may wish to systematically verify the solution by himself, by means of the data-flow equation method for live variables.

(b) Here is the systematic computation of the reaching definitions. The definitions in the *CFG* are a_1, b_3, c_4 and a_6, plus the input parameters $b_?$ and $c_?$, which are defined outside of the *CFG*. First here are the definition and suppression tables:

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

node	*defined*
1	a_1
2	
3	b_3
4	c_4
5	
6	a_6
7	

node	*suppressed*
1	a_6
2	
3	$b_?$
4	$c_?$
5	
6	a_1
7	

As said, it is necessary to consider the definitions $b_?$ and $c_?$ for the input parameters of the program. The rest of the definitions and suppressions is clear. Second here are the data-flow equations, divided into input and output:

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

node	*in equations*	*out equations*
1	$in(1) = \{\ b_?,\ c_?\ \}$	$out(1) = \{\ a_1\ \} \cup \big(\ in(1) - \{\ a_6\ \}\ \big)$
2	$in(2) = out(1) \cup out(4)$	$out(2) = in(2)$
3	$in(3) = out(2)$	$out(3) = \{\ b_3\ \} \cup \big(\ in(3) - \{\ b_?\ \}\ \big)$
4	$in(4) = out(3) \cup out(6)$	$out(4) = \{\ c_4\ \} \cup \big(\ in(4) - \{\ c_?\ \}\ \big)$
5	$in(5) = out(2)$	$out(5) = in(5)$
6	$in(6) = out(5)$	$out(6) = \{\ a_6\ \} \cup \big(\ in(6) - \{\ a_1\ \}\ \big)$
7	$in(7) = out(5)$	$out(7) = in(7)$

Here it happens that every definition suppresses another definition of the same variable or input parameter (see the definition and suppression tables computed above). The rest of the system of data-flow equations is clear.

iterative solution table of the system of data-flow equations
(**REACHING DEF.S**)

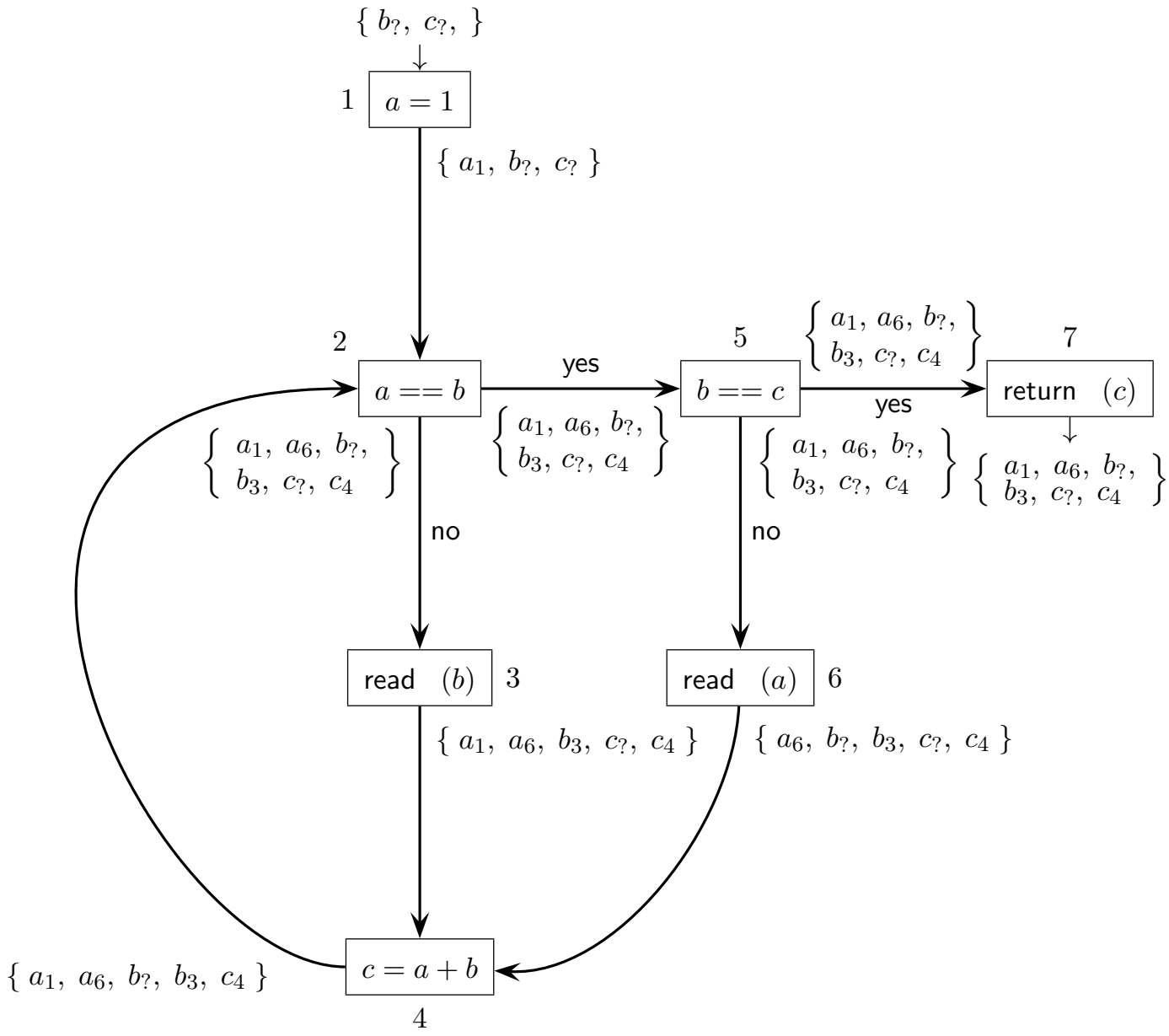
	initialization		1		2		3		4		5	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$b_?$ $c_?$	a_1 $b_?$ $c_?$	$b_?$ $c_?$	a_1 $b_?$ $c_?$	$b_?$ $c_?$	a_1 $b_?$ $c_?$	$b_?$ $c_?$	a_1 $b_?$ $c_?$	$b_?$ $c_?$	a_1 $b_?$ $c_?$	$b_?$ $c_?$	
2	$\emptyset$	$\emptyset$	a_1 $b_?$ $c_?$ c_4	a_1 $b_?$ $c_?$ c_4	a_1 a_6 $b_?$ b_3 $c_?$ c_4	a_1 a_6 $b_?$ b_3 $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	
3	$\emptyset$	b_3	$\emptyset$	b_3	a_1 $b_?$ $c_?$ c_4	a_1 b_3 $c_?$ c_4	a_1 a_6 $b_?$ b_3 $c_?$ c_4	a_1 a_6 b_3 $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	
4	$\emptyset$	c_4	a_6 b_3	a_6 b_3 c_4	a_6 b_3	a_6 b_3 c_4	a_1 a_6 b_3 $c_?$ c_4	a_1 a_6 b_3 c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	
5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	a_1 $b_?$ $c_?$ c_4	a_1 $b_?$ $c_?$ c_4	a_1 a_6 $b_?$ b_3 $c_?$ c_4	a_1 a_6 $b_?$ b_3 $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	
6	$\emptyset$	a_6	$\emptyset$	a_6	$\emptyset$	a_6	a_1 $b_?$ $c_?$ c_4	a_6 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	
7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	a_1 $b_?$ $c_?$ c_4	a_1 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	a_1 a_6 b_3 $b_?$ $c_?$ c_4	

The *in* columns of steps 4 and 5 are identical, thus the solution procedure terminates in four steps. The contents of the *out* column of step 4 represent the definitions reaching the *CFG* nodes, plus the contents of the *in* cell of node 1.

Notice that the solution procedure for reaching definitions propagates forward. For this reason, the solution procedure starts from the input definitions.

The definitions that are the slowest to reach the output of their farthest node are the following: $b_?$ to reach node 4, b_3 to reach node 6, both b_3 and a_6 to reach node 7. All of them take exactly four steps to propagate as far as possible. In fact a quick look shows that they have to traverse a path of four nodes. For instance: $b_?$ has to traverse the four nodes 2, 5, 6 and 4; instead on path 2, 3 and 4, which has length only three, $b_?$ would be suppressed by b_3 at node 3; and there are not any other acyclic paths that connect the input node 1 to node 4. There is not any definition that can traverse a path of five nodes before looping on a node already reached, or before being suppressed, or before exiting the *CFG*. This is why the solution procedure converges in exactly four steps.

Here are the reaching definitions at the node outputs, obtained systematically (annotated next to the arrow exits to mean they are output):



Of course all the outputs to the same node bear the same reaching definitions. Definitions $b_?$ and $c_?$ are reaching on the input node of the *CFG* as they are input parameters to the program. The reader may wish to verify the systematic solution directly on the *CFG* by applying the definition of reaching definition.