FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Tue 3 March 2015 - Part Theory

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

NAME:

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 following regular expression R, over the two-letter alphabet $\{ a, \; b \}$:

$$R = (a^{+} \;\mid\; b \, a \;\mid\; a \, b \, a)^{*} \; b$$

Answer the following questions:

(a) Provide evidence that the regular expression R is ambiguous, by referring to the string $a \, b \, a \, b$.

(b) By using the Berry-Sethi method, construct a deterministic finite state automaton A that recognizes the language $L(R)$.

(c) Minimize the number of states of the automaton A obtained at the previous point (b) and find the equivalent minimal automaton A_{min}.

(d) (optional) Define a strictly right unilinear grammar G that generates the language $L(R)$. Grammar G should be as simple as possible in terms of the number of nonterminal symbols and rules. Also write a derivation of the string $a \, b \, a \, b$ according to grammar G.

Solution

(a) To show that the regular expression R is ambiguous, consider its subscripted version $R_\#$ below:

$$R_\# = \left(a_1^+ \mid b_2\, a_3 \mid a_4\, b_5\, a_6 \right)^* \; b_7$$

from which the two subscripted strings below:

$$a_1\, b_2\, a_3\, b_7 \qquad \text{and} \qquad a_4\, b_5\, a_6\, b_7$$

can be derived. These strings become identical when the numerical subscripts are erased, thus the regular expression R is ambiguous.

A little more insight would show that the ambiguity degree of expression R is unlimited, as for instance the strings $(a\, b\, a)^n\, b \in L(R)$ have an ambiguity degree lower bounded by 2^n (with $n \geq 1$), since every substring $a\, b\, a$ has degree two[1], though no string of language $L(R)$ has an infinite ambiguity degree.

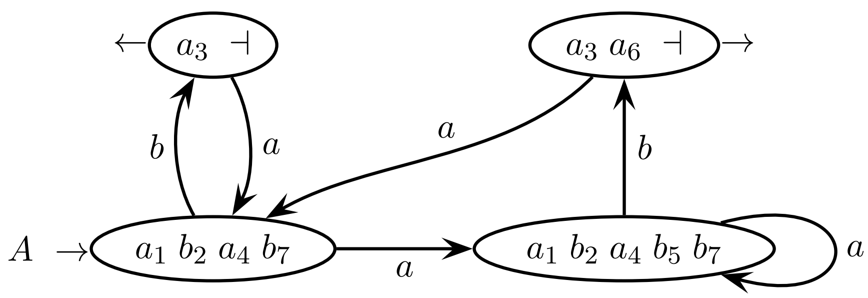
(b) Consider again the subscripted regular expression $R_\#$:

$$R_\# = \left(a_1^+ \mid b_2\, a_3 \mid a_4\, b_5\, a_6 \right)^* \; b_7$$

The initials and followers are:

initials	$a_1\ b_2\ a_4\ b_7$
generator	*follows*
a_1	$a_1\ b_2\ a_4\ b_7$
b_2	a_3
a_3	$a_1\ b_2\ a_4\ b_7$
a_4	b_5
b_5	a_6
a_6	$a_1\ b_2\ a_4\ b_7$
b_7	$\dashv$

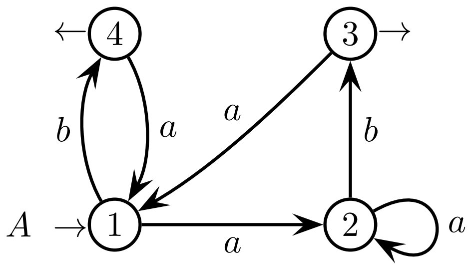
Here is the Berry-Sethi deterministic automaton A:



with four states.

[1] Formula 2^n is only a lower bound to the ambiguity degree, as such strings also have other ambiguous formulations, so that their actual degree is even higher.

(c) First rename the states of automaton A:



Then notice that the final states 3 and 4 are undistinguishable as by letter a both go to state 1, and that the non-final states 1 and 2 are undistinguishable as by letter a both do not change state group and by letter b both go to states 4 and 3, which are already known to be undistinguishable. More systematically, here is the undistinguishability table of automaton A and its resolution:

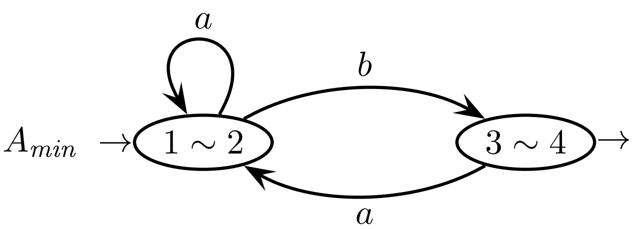
2	$1 \sim 2$ $3 \sim 4$		
3	$\times$	$\times$	
4	$\times$	$\times$	$1 \sim 1$
	1	2	3

initial

2	$\sim$		
3	$\times$	$\times$	
4	$\times$	$\times$	$\sim$
	1	2	3

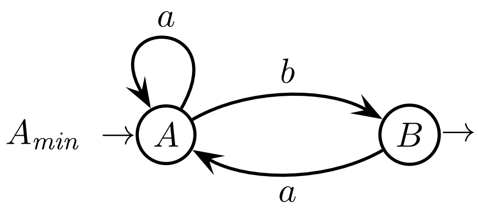
resolved

The state equivalence classes are: [1, 2] and [3, 4]. Therefore we obtain the minimal automaton A_{min} below:



with two states only.

(d) Change again the state names of the minimal automaton A_{min}:



We obtain the following right unilinear grammar G (axiom A):

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

Here is the derivation of string $a\,b\,a\,b$:

$$A \xRightarrow{A \rightarrow aA} a\,A \xRightarrow{A \rightarrow bB} a\,b\,B \xRightarrow{B \rightarrow aA} a\,b\,a\,A \xRightarrow{A \rightarrow bB} b\,a\,b\,A \xRightarrow{B \rightarrow \varepsilon} a\,b\,a\,b$$

which obviously corresponds to a computation in the automaton A_{min}.

2 Free Grammars and Pushdown Automata 20%

1. Consider a language L_1 of nested lists, possibly empty and of arbitrary depth. Every (sub)list is embraced in round brackets "$($" and "$)$", and may contain elements schematized by letter e. The whole string is one list embraced in round brackets.

   Here are four sample nested lists s_i, with $i = 1, 2, 3, 4$:

   $$s_1 = (\,) \qquad s_2 = (\,e\,e\,) \qquad s_3 = \big(\,(\,e\,)\,\big) \qquad s_4 = \big(\,e\,(\,e\,)\,e\,e\,\big)$$

   Answer the following questions:

   (a) Write a non-extended (*BNF*) grammar G_1, not ambiguous, that generates the language L_1 described above, and draw the syntax tree of the sample string s_4.

   (b) Consider a modified list language L_2, similar to L_1 but with hash separators "$\#$", which must be inserted between two consecutive letters e and nowhere else. For instance, the sample string s_4 above becomes as the string s_5 below:

   $$s_5 = \big(\,e\,(\,e\,)\,e\,\#\,e\,\big)$$

   Write a non-extended (*BNF*) grammar G_2, not ambiguous, that generates the language L_2 described above, and draw the syntax tree of the sample string s_5.

   (c) (optional) Consider another modified list language L_3, which is similar to L_1 and contains all the strings of L_1, but such that one closed square bracket "$]$" can be used at the string end to close all the preceding round brackets "$($" that are still pending open. For instance, here is a sample string s_6 of such kind:

   $$s_6 = \Big(\,\big(\,(\,)\,e\,\big)\,e\,\big(\,e\,\Big]$$

   where the 1^{st} and 4^{th} open round brackets are both closed by "$]$". Write a non-extended (*BNF*) grammar G_3, no matter if it is ambiguous, that generates the language L_3 described above, and draw the syntax tree of the sample string s_6.

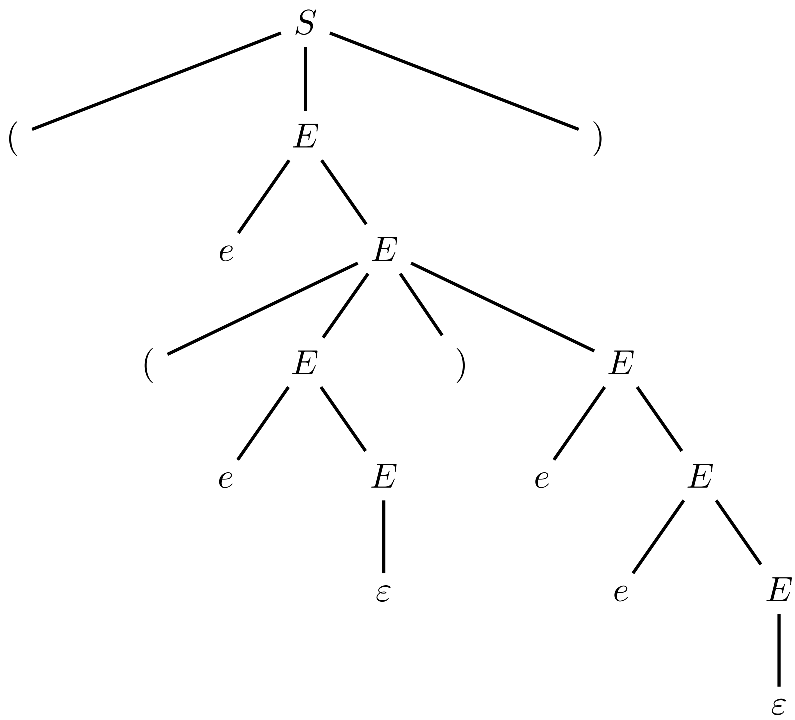
Solution

(a) An indicative observation, valid for the three questions altogether, is that the list language sketched and its variants are quite similar to the Dyck language with round brackets, extended to also allow elements e (question (a)) as well as element separators (question (b)), and to match some brackets altogether with the final square one (question (c)). Thus the Dyck rule form (non-ambiguous) is likely to play a role in the grammars of such a list language and variants thereof.

Grammar G_1 (axiom S):

$$G_1 \left\{ \begin{array}{lcl} S & \to & \overbrace{\text{‘(’}\ E\ \text{‘)’}}^{\substack{\textit{outermost} \\ \textit{brackets}}} \\ E & \to & \underbrace{e\ E}_{\substack{\textit{list of} \\ \textit{elem.s}}} \ \mid\ \underbrace{\text{‘(’}\ E\ \text{‘)’}\ E}_{\textit{Dyck}} \ \mid\ \underbrace{\varepsilon}_{\textit{term.}} \end{array} \right.$$

Grammar G_1 is not ambiguous, as its rules are the usual self-embedding (auto-inclusive) one for the outermost brackets, the right-linear one for lists, and the well known non-ambiguous one for Dyck. Here is the syntax tree of string s_4:

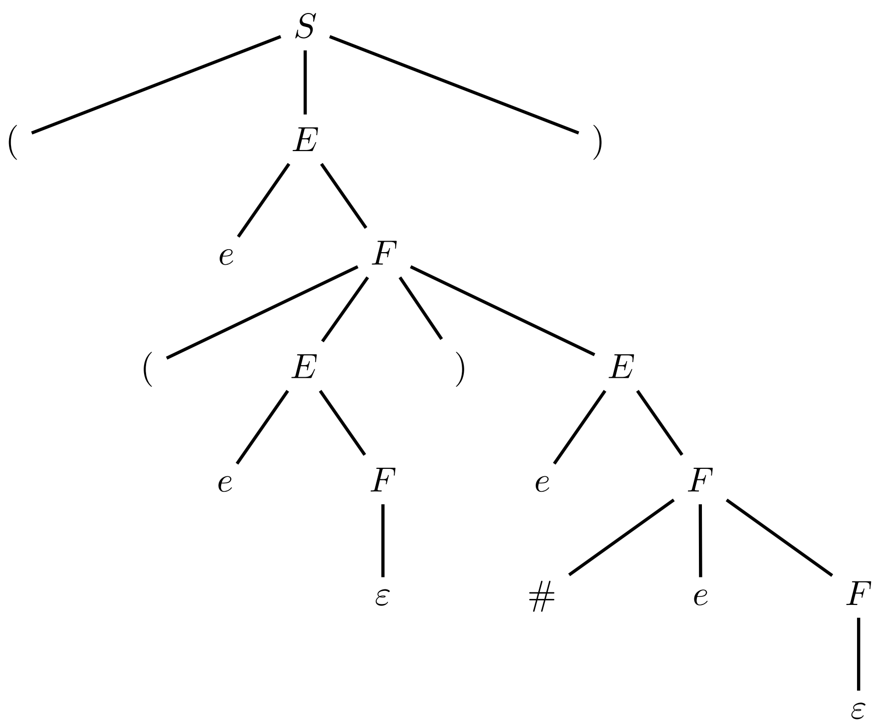


which does not deserve any special comment.

(b) After answering question (a), variant (b) is quite easy to model, since the separators "#" do not really interfere with the nested bracket structure. A slight change in the right-linear rules will achieve the result. Grammar G_2 (axiom S):

$$G_2 \left\{ \begin{array}{lcl} S & \rightarrow & \text{`('}\, E\, \text{`)'} \\ E & \rightarrow & e\, F \mid \text{`('}\, E\, \text{`)'}\, E \mid \varepsilon \\ F & \rightarrow & \underbrace{\text{`\#'}\, e\, F}_{\substack{\textit{put in the} \\ \textit{separators}}} \mid \text{`('}\, E\, \text{`)'}\, E \mid \varepsilon \end{array} \right.$$

Grammar G_2 is not ambiguous, basically for the same reasons as explained at the previous point (a). It just splits the generation of a list into the two cases with separator "#" and without it. Here is the syntax tree of string s_5:



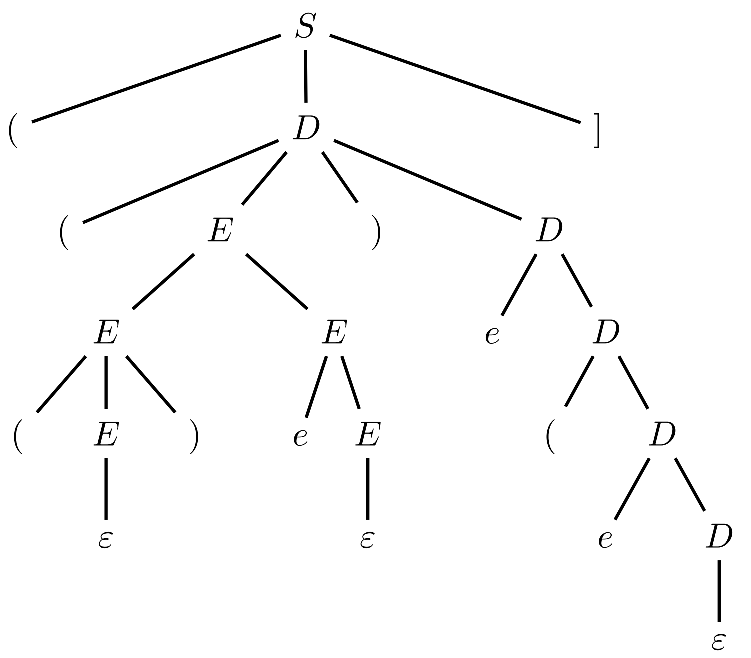
which does not deserve any special comment either.

(c) This variant deserves a little more insight, to understand how to correctly deal with the unbalanced round brackets closed by the single square one at the end. Grammar G_3 (axiom S):

$$G_3 \left\{ \begin{array}{lcl} S & \rightarrow & \text{‘(’}\, E \,\text{‘)’} \mid \text{‘(’}\, D \,\text{‘]’} \\ E & \rightarrow & e\, E \mid \text{‘(’}\, E \,\text{‘)’}\, E \mid \varepsilon \\ D & \rightarrow & e\, D \mid \underbrace{\text{‘(’}\, D}_{\substack{\textit{closed} \\ \textit{at end}}} \mid \underbrace{\text{‘(’}\, E \,\text{‘)’}}_{\substack{\textit{only regular} \\ \textit{inner lists}}} \; \underbrace{D}_{\textit{more}} \mid \varepsilon \end{array} \right.$$

Actually grammar G_3 is not ambiguous, though this property was not requested. The reason is similar to that of cases (a) and (b) above: the rules are all of non-ambiguous type and are not ambiguously combined. The difference with respect to grammar G_1 is that grammar G_3 immediately decides whether either to generate standard lists all regularly closed with their own round bracket, or to generate a few lists altogether closed by one final square bracket (as well as to generate regular lists). In the latter case the nonterminal D may generate a list with or without its own closing round bracket.

Yet notice that an inner list of one that has its own closing round bracket must be forced to be also regularly closed, lest otherwise there might be an inner list closed by the final square bracket, which would figure as being longer than the regular list it is contained in. Here is the syntax tree of string s_6:



In this tree the decision is for a list with final square bracket. From the left, both the first and the fourth open round brackets are closed by the final square bracket, and the tree structure clearly reflects this fact. Instead, the second and third open round brackets are regularly closed, as clearly apparent in the tree.

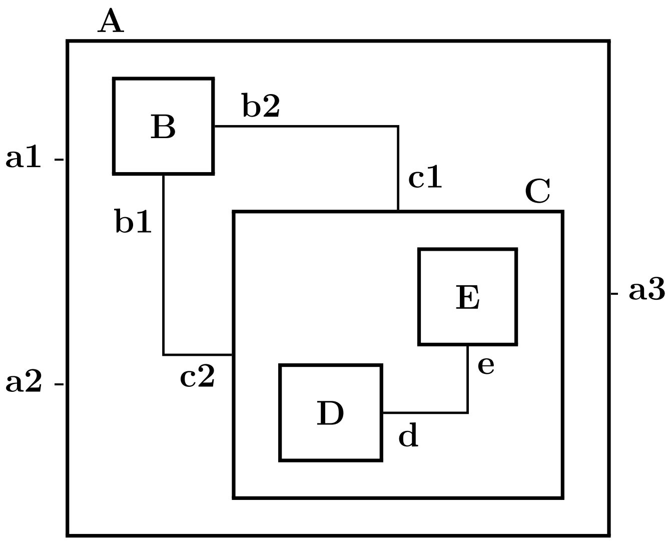
2. A textual language defines the structure of nested modules with connections. A program in such a language consists of the definition of one module, with these features:

   - every module has an *interface* that consists of one or more items
   - a module may include *nested modules*, which in turn may include more modules, and so on at an arbitrary depth
   - a module may specify one or more *connections*: each connection links two interface items of immediately nested modules (one item per module)

   The module and interface names are identifiers schematized by the terminal `id`.

   For instance, the modular structure depicted in the figure below on the right is represented by the program reported on the left. Notice that the program includes only structural information related to the nesting relation between modules, to their interface items and to their connections, but that it ignores any other geometric/typographic information possibly conveyed by a graphic representation like that in the figure.

```
module A
  interface a1, a2, a3;
  component
    module B
      interface b1, b2;
    end module B;
    module C
      interface c1, c2;
      component
        module D
          interface d;
        end module D;
        module E
          interface e;
        end module E;
      connection (D.d, E.e);
    end module C;
  connection (B.b1, C.c2) (B.b2, C.c1);
end module A.
```



   Answer the following questions:

   (a) Write an extended grammar (*EBNF*) G_1, not ambiguous, that reasonably models the language sketched above.

   (b) (optional) Write an extended grammar (*EBNF*) G_2, not ambiguous, for the following variation of the language:

   - a module may or may not have interface items
   - every module that immediately includes two or more modules with some interface items, necessarily includes also a *connection* section

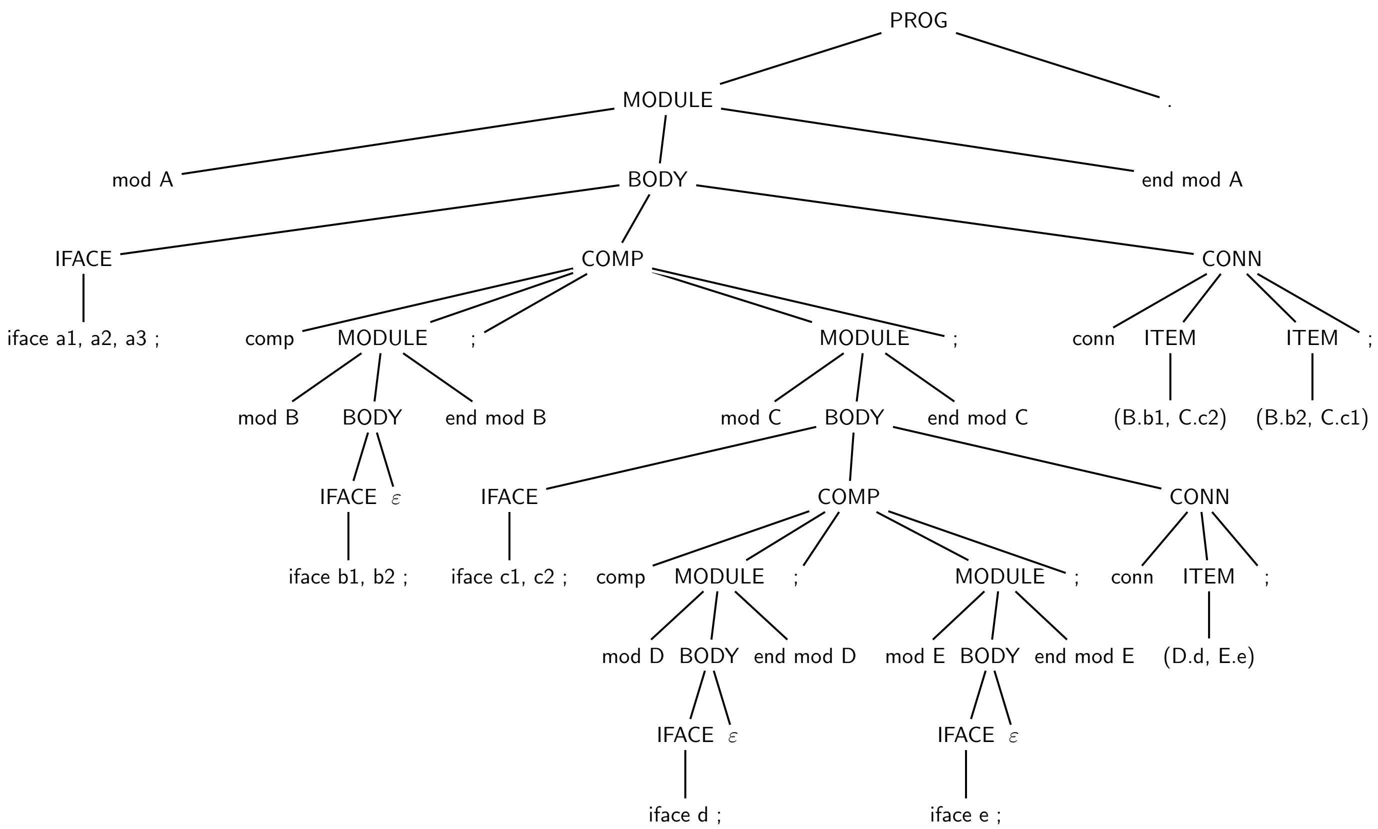
Solution

(a) Here is the requested grammar G_1 (axiom PROG):

$$
G_1 \left\{
\begin{array}{rcl}
\langle \mathsf{PROG} \rangle & \rightarrow & \langle \mathsf{MODULE} \rangle \ `\,.\,\text{'} \\
\hline
\langle \mathsf{MODULE} \rangle & \rightarrow & \mathsf{module\ id}\ \langle \mathsf{BODY} \rangle\ \mathsf{end\ module\ id} \\
\hline
\langle \mathsf{BODY} \rangle & \rightarrow & \langle \mathsf{IFACE} \rangle \left[\langle \mathsf{COMP} \rangle \left[\langle \mathsf{CONN} \rangle \right] \right] \\
\hline
\langle \mathsf{IFACE} \rangle & \rightarrow & \mathsf{interface\ id} \left(`\,,\,\text{'}\ \mathsf{id} \right)^{*} `\,;\,\text{'} \\
\langle \mathsf{COMP} \rangle & \rightarrow & \mathsf{component} \left(\langle \mathsf{MODULE} \rangle\ `\,;\,\text{'} \right)^{+} \\
\langle \mathsf{CONN} \rangle & \rightarrow & \mathsf{connection}\ \langle \mathsf{ITEM} \rangle^{+}\ `\,;\,\text{'} \\
\hline
\langle \mathsf{ITEM} \rangle & \rightarrow & `\,(\,\text{'}\ \mathsf{id}\ `\,.\,\text{'}\ \mathsf{id}\ `\,,\,\text{'}\ \mathsf{id}\ `\,.\,\text{'}\ \mathsf{id}\ `\,)\,\text{'}
\end{array}
\right.
$$

The square brackets indicate optionality. The interface section is mandatory, but the component and connection ones are optional. Obviously, if there are no components, then it makes no sense to have connections. The interface, component and connection sections may not be empty. Grammar G_1 is not ambiguous, as it is made of non-ambiguous rules that are not ambiguously combined.

For completeness, though it was not requested, here is the simplified syntax tree of the sample program, where a few leaf nodes are shortened or grouped:



(b) Here is the requested grammar G_2 (axiom PG), derived from grammar G_1:

$$
G_2 \left\{ \begin{array}{rcl}
\langle \mathsf{PG} \rangle & \to & \langle \mathsf{MX} \rangle \; `\,.\,' \\
\hline
\langle \mathsf{MX} \rangle & \to & \langle \mathsf{M0} \rangle \;\mid\; \langle \mathsf{M1} \rangle \\
\hline
\langle \mathsf{M0} \rangle & \to & \mathsf{module\ id}\; \langle \mathsf{B0} \rangle \; \mathsf{end\ module\ id} \quad \text{// module without iface} \\
\langle \mathsf{M1} \rangle & \to & \mathsf{module\ id}\; \langle \mathsf{B1} \rangle \; \mathsf{end\ module\ id} \quad \text{// module with iface} \\
\hline
\langle \mathsf{B0} \rangle & \to & \big[\, \langle \mathsf{CX} \rangle \,\big] \\
\langle \mathsf{B1} \rangle & \to & \langle \mathsf{IF} \rangle \; \big[\, \langle \mathsf{CX} \rangle \,\big] \\
\hline
\langle \mathsf{CX} \rangle & \to & \langle \mathsf{C1} \rangle \; \big[\, \langle \mathsf{CN} \rangle \,\big] \;\mid\; \langle \mathsf{C2} \rangle \; \langle \mathsf{CN} \rangle \\
\hline
\langle \mathsf{IF} \rangle & \to & \mathsf{interface\ id}\; \big(\, `\,,\,' \; \mathsf{id} \,\big)^{*} \; `\,;\,' \\
\langle \mathsf{C1} \rangle & \to & \mathsf{component}\; \langle \mathsf{L1} \rangle \quad \text{// none or one module with iface} \\
\langle \mathsf{C2} \rangle & \to & \mathsf{component}\; \langle \mathsf{L2} \rangle \quad \text{// two or more modules with iface} \\
\langle \mathsf{CN} \rangle & \to & \mathsf{connection}\; \langle \mathsf{IT} \rangle^{+} \; `\,;\,' \\
\hline
\langle \mathsf{L1} \rangle & \to & \big(\, \langle \mathsf{M0} \rangle \; `\,;\,' \,\big)^{+} \;\mid\; \big(\, \langle \mathsf{M0} \rangle \; `\,;\,' \,\big)^{*} \; \langle \mathsf{M1} \rangle \; `\,;\,' \, \big(\, \langle \mathsf{M0} \rangle \; `\,;\,' \,\big)^{*} \\
\langle \mathsf{L2} \rangle & \to & \big(\, \langle \mathsf{M0} \rangle \; `\,;\,' \,\big)^{*} \; \langle \mathsf{M1} \rangle \; `\,;\,' \, \big(\, \langle \mathsf{M0} \rangle \; `\,;\,' \,\big)^{*} \; \langle \mathsf{M1} \rangle \; `\,;\,' \, \big(\, \langle \mathsf{MX} \rangle \; `\,;\,' \,\big)^{*} \\
\hline
\langle \mathsf{IT} \rangle & \to & `\,(\,' \; \mathsf{id} \; `\,.\,' \; \mathsf{id} \; `\,,\,' \; \mathsf{id} \; `\,.\,' \; \mathsf{id} \; `\,)\,'
\end{array} \right.
$$

Since now the interface section is optional as well, grammar G_2 can generate empty modules (but no empty sections). For the rest, it has the same features as grammar G_1. Some nonterminals are split in two versions and the comments nearby help to understand the motivation. Notice there are copy rules, which as well known may avoid rules with long and repetitive right parts and thus help to compact the grammar. Furthermore, with respect to G_1 a few more nonterminals have been introduced to isolate certain crucial rule parts.

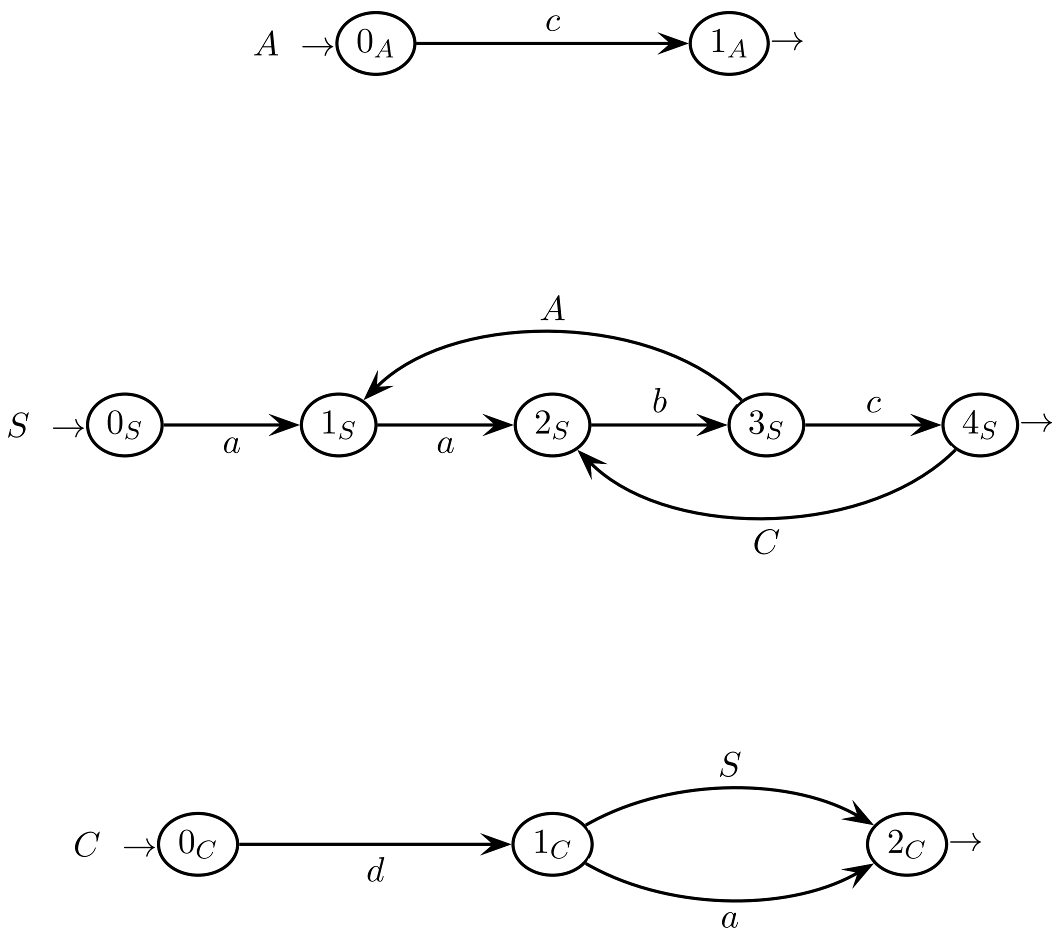
Grammar G_2 is not ambiguous either: the reasons are the same as before, and furthermore care has been taken to make sure that the regular expressions in the rule right members (more complex than in G_1) are not ambiguous.

The sample program fulfills also the variant requirements, thus it can be generated by grammar G_2. The reader may wish to redraw the previous syntax tree and adapt it to G_2 by himself; the changes to do are somewhat limited.

Of course, there may be other as acceptable solutions. For instance, the semicolons may be placed in the rules in a few different positions.

3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the following machine net (axiom S), over the four-letter alphabet $\{ a, \ b, \ c, \ d \}$:



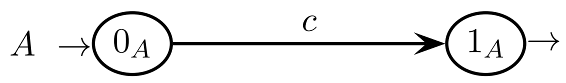
Answer the following questions:

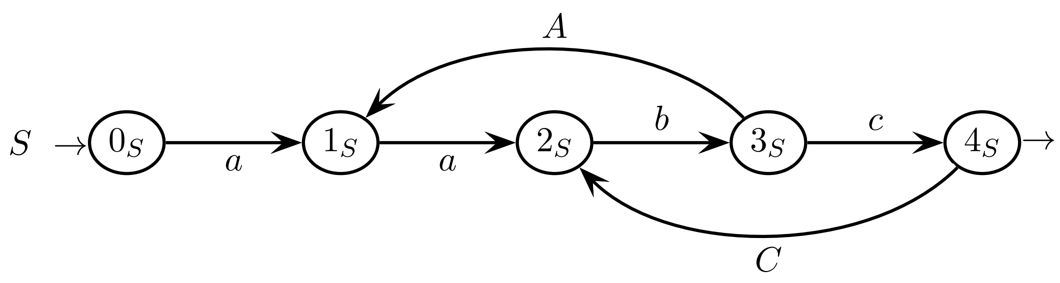
(a) Draw the complete pilot graph of the machine net and say if the net is of type $ELR\,(1)$ or not (explain your answer). Is the $ELL\,(1)$ condition satisfied by the pilot or not (explain your answer) ? Use the space left on the next pages.

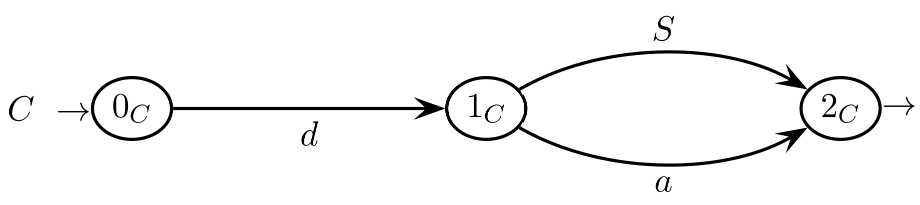
(b) Write the guide sets on each call and exit arrow of the machine net, and using these guide sets (as well as those on the terminal shift arcs) say if the net is of type $ELL\,(1)$ or not (explain your answer). Use the net on the next pages.

(c) (optional) Examine the net and say if it is of type $ELL\,(k)$ for some $k \geq 2$ or not. This may require to find a few guide sets of a length greater than one, and to reason using them. Please answer concisely but rigorously. If you wish, you can use the net on the next pages.

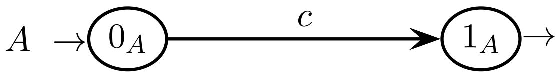
question (a) - please draw here the **PILOT GRAPH** and write your explanation question (b) - please compute here the **GUIDE SETS** and write your explanation

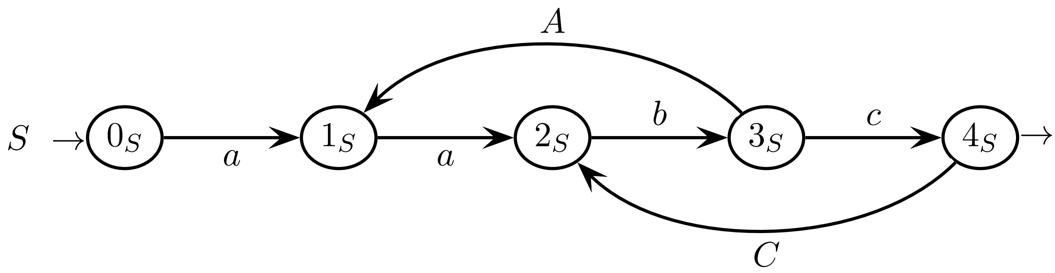


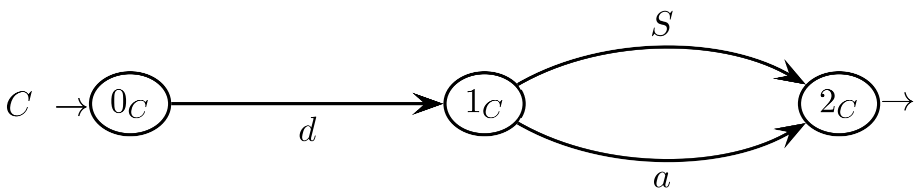




question (c) - please compute here the **GUIDE SETS** and write your explanation

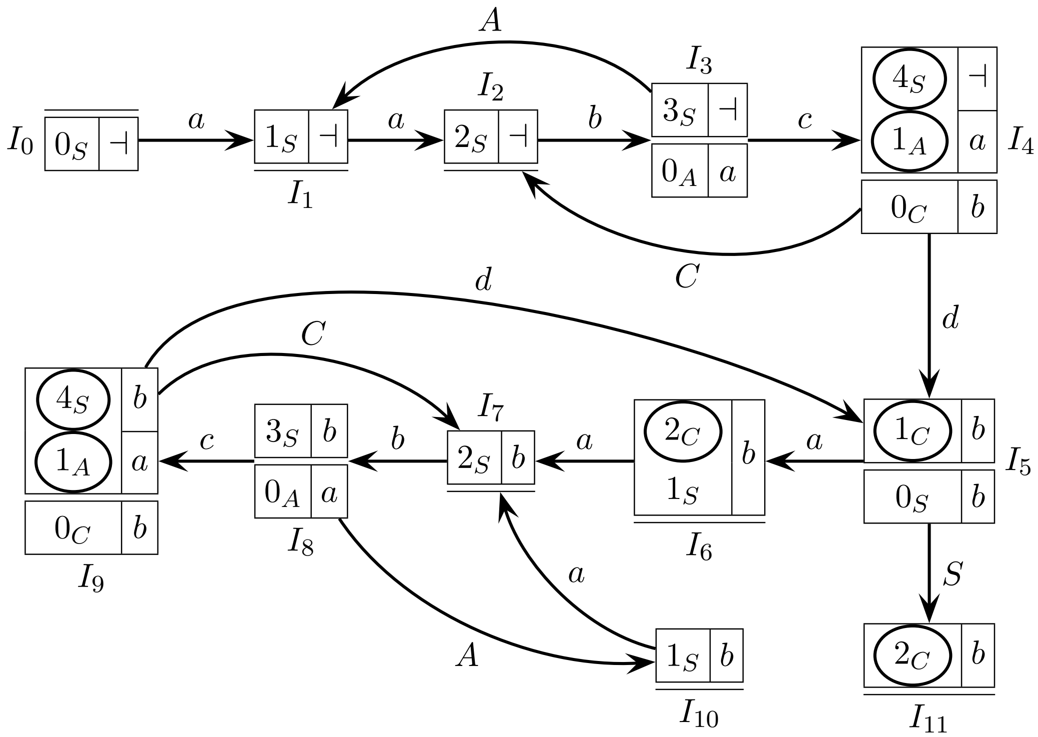






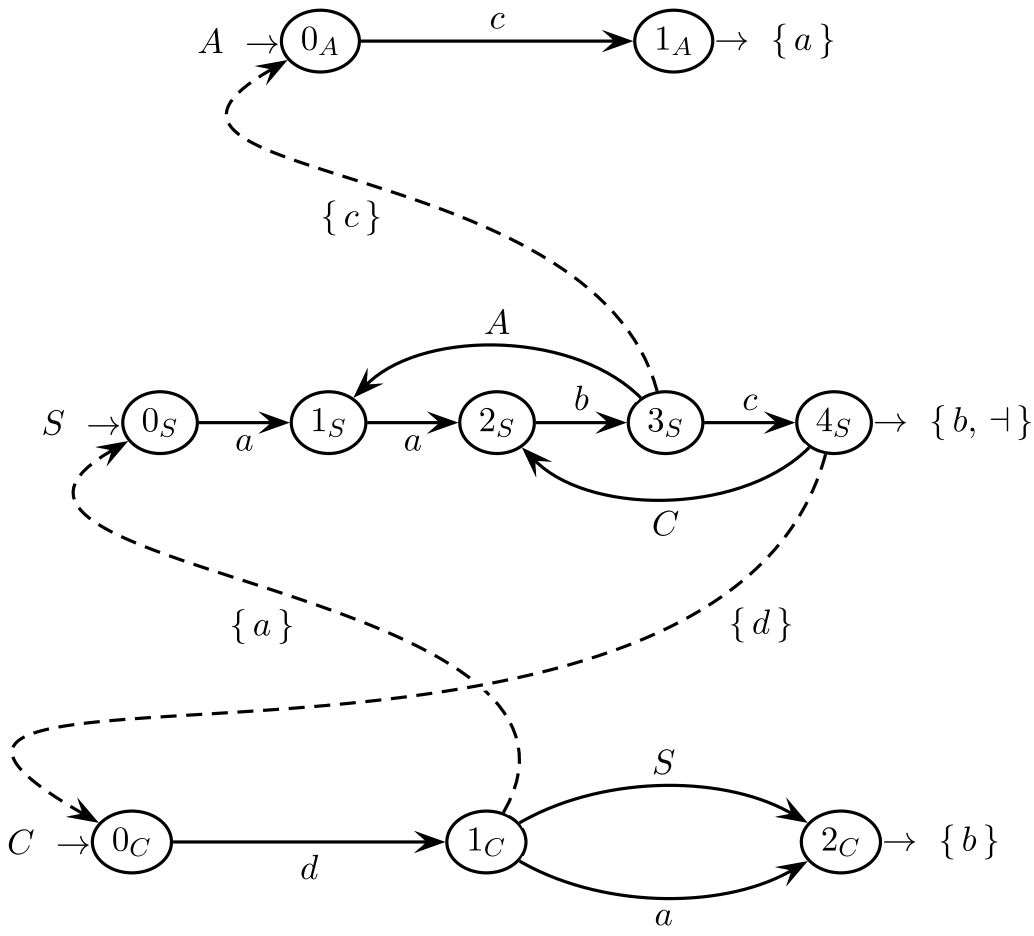
Solution

(a) Here is the requested pilot, with 12 m-states:



The net is of type $ELR(1)$, as it does not have any conflicts of whatever kind. Anyway, the $ELL(1)$ condition is violated by the presence of three multiple (double) transitions: $I_3 \xrightarrow{c} I_4$, $I_5 \xrightarrow{a} I_6$ and $I_8 \xrightarrow{c} I_9$, i.e., the net does not have the Single Transition Property (STP). This can be also argued by noticing that the three destination m-states I_4, I_6 and I_9 have bases with two items.

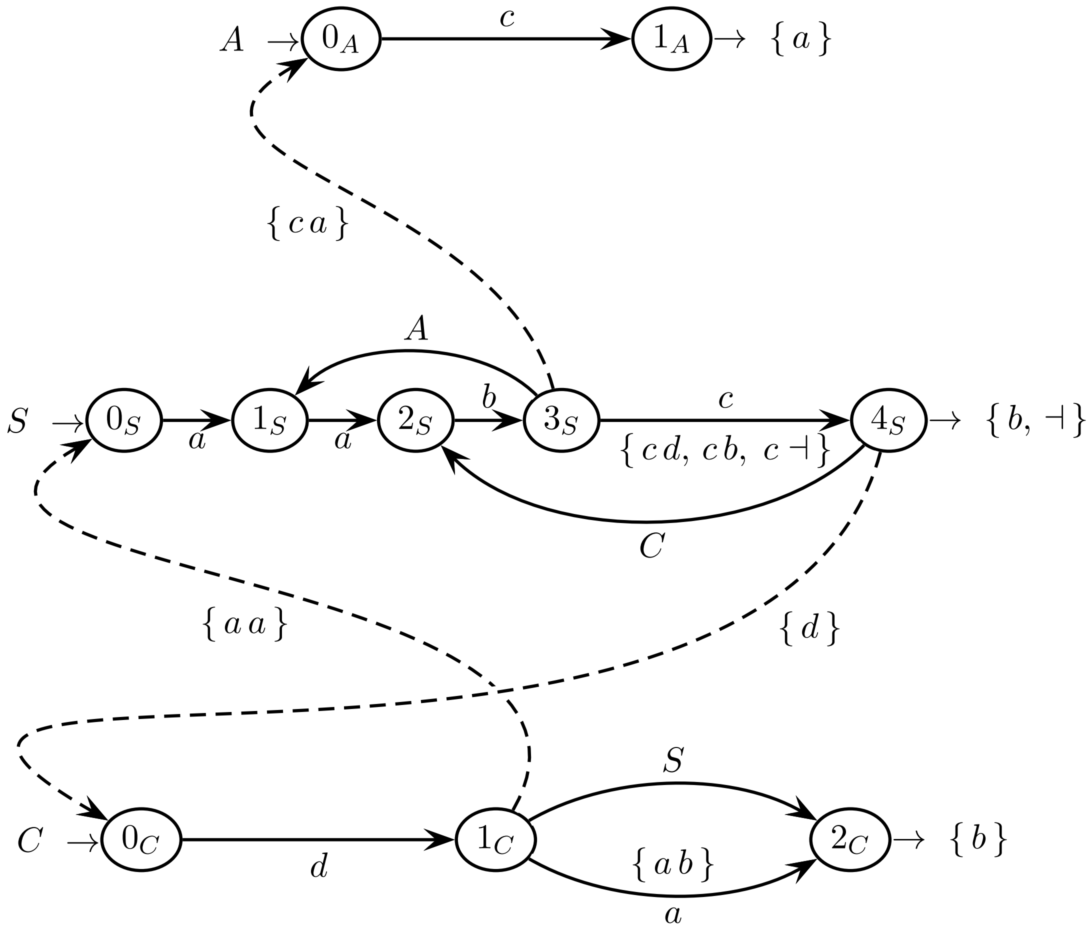
(b) Here are the requested guide sets with $k = 1$ (those of the terminal shift arcs coincide with the terminal):



The net is not of type $ELL\,(1)$, as the guide sets at the states 3_S and 1_C are not disjoint.

Intuitively, the guide set overlap of letter c at the state 3_S is in relation with the pilot multiple transitions $I_3 \xrightarrow{c} I_4$ and $I_8 \xrightarrow{c} I_9$, and that the guide set overlap of letter a at the state 1_C is in relation with the pilot multiple transition $I_5 \xrightarrow{a} I_6$, since such transitions break the $ELL\,(1)$ condition; for the pilot see question (a).

(c) Here are the guide sets with $k = 2$ on the critical states, those where with $k = 1$ there were conflicts:



The net is of type $ELL\,(2)$, as now the guide sets of length 2 at the states 3_S and 1_C are disjoint.

Intuitively, this is in relation with the observation that the pilot multiple transitions are isolated, not concatenated to form multiple paths of length two or more; for the pilot see question (a).

4 Language Translation and Semantic Analysis 20%

1. Consider the following source grammar G_s (axiom S):

$$G_s \begin{cases} S & \rightarrow & A\, X \mid A \\ X & \rightarrow & C\, S \mid C \\ A & \rightarrow & a\, A \mid a \\ C & \rightarrow & c\, C \mid c \end{cases}$$

   which generates strings composed by alternated groups of letters a and c.

   Answer the following questions:

   (a) Write a destination grammar G_d that defines a translation τ of the strings of the language $L\,(G_s)$, such that:

   - every group of letters a of ***length two or more*** is reduced to one letter a
   - and every group of letters c is enclosed within a pair of letters b and e

   For instance:

$$a\,a\,c\,c\,c\,a\,c\,a \overset{\tau}{\mapsto} a\,b\,c\,c\,c\,e\,a\,b\,c\,e\,a$$

   (b) Modify the previous syntactic scheme, and if necessary also change the source grammar, in such a way that the source language is left unchanged and the translation τ is redefined as a new translation τ', such that:

   - every group of letters a of length two or more is reduced to one letter a
   - and every group of letters c of ***length two or more*** is enclosed within a pair of letters b and e

   For instance:

$$a\,a\,c\,c\,c\,a\,c\,a \overset{\tau'}{\mapsto} a\,b\,c\,c\,c\,e\,a\,c\,a$$

   (c) (optional) Define a deterministic sequential (i.e., finite state) transducer T that computes the translation τ' defined at the previous point (b).

Solution

(a) Immediate basic case. Here is the destination grammar G_d for translation τ:

$$G_d \begin{cases} S \rightarrow A\,X \mid A \\ X \rightarrow b\,C\,e\,S \mid b\,C\,e \\ A \rightarrow A \mid a \\ C \rightarrow c\,C \mid c \end{cases}$$

It translates unchanged only the last letter a of each group of one or more letters a, and it encapsulates between b and e the nonterminal C that generates a group of one or more letters c, which are all translated unchanged.

(b) For translation τ', the source grammar G_s has to be modified to separately treat the groups of two or more letters c. An additional nonterminal D is introduced to this purpose. The new source grammar G'_s is the following:

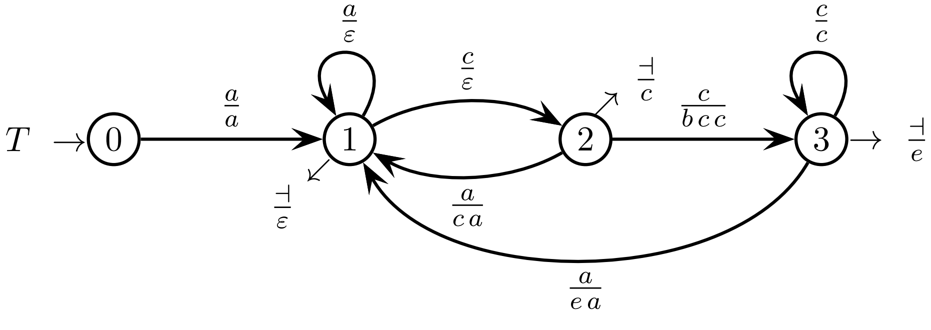
$$G'_s \begin{cases} S \rightarrow AX \mid A \\ X \rightarrow C\,S \mid C \\ A \rightarrow a\,A \mid a \\ C \rightarrow c\,D \mid c \\ D \rightarrow c\,D \mid c \end{cases}$$

and the corresponding destination grammar G'_d is the following:

$$G'_d \begin{cases} S \rightarrow A\,X \mid A \\ X \rightarrow C\,S \mid C \\ A \rightarrow A \mid a \\ C \rightarrow b\,c\,D\,e \mid c \\ D \rightarrow c\,D \mid c \end{cases}$$

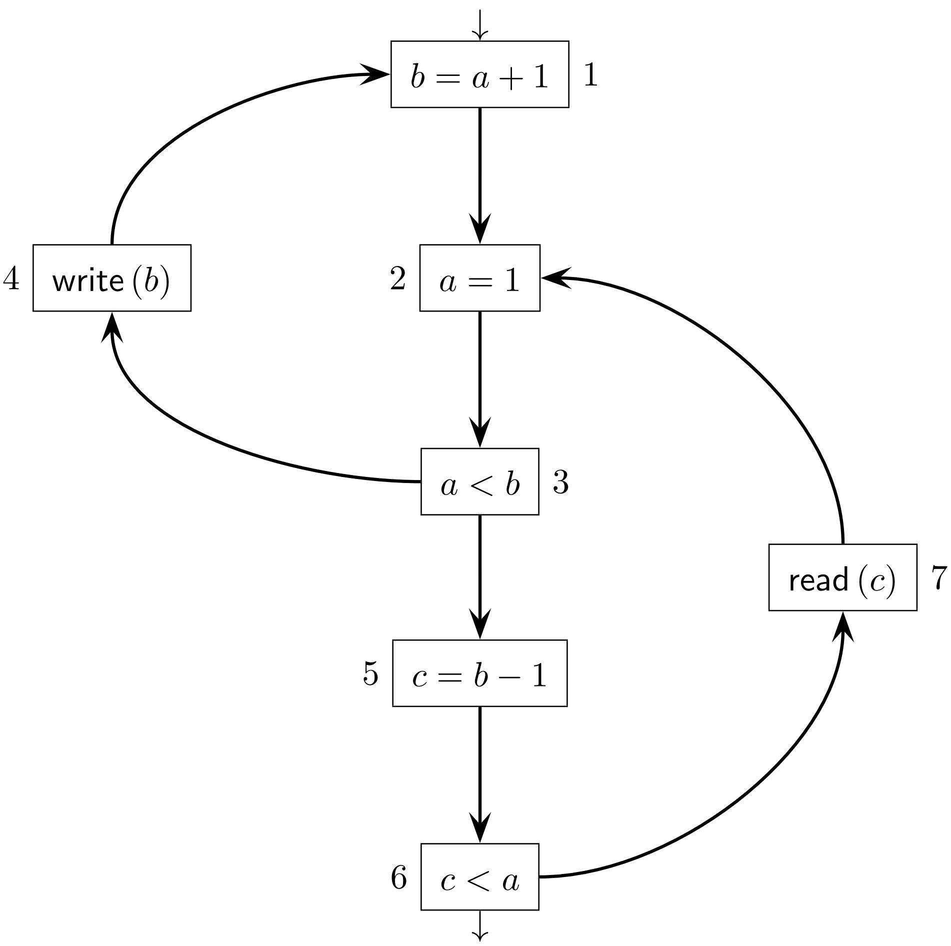
which is structured similarly to the basic syntactic scheme of question (a).

(c) Here is the requested sequential transducer T for translation τ', with three final states 1, 2 and 3:



The transducer T is deterministic, as the underlying recognizer is so. It outputs only the first letter a of a group of letters a, and waits to see if there are one or more letters c before undertaking the output actions appropriate to either case. Notice it needs to produce more output on exiting the final states 2 and 3.

2. Consider the following control flow graph (*CFG*) of a program, with seven nodes (input node 1 and output node 6):



Answer the following questions:

(a) Find the live variables at all the nodes of the *CFG*, by using the flow equation method (for live variables). Use the tables on the next pages. Write the solution by the graph nodes (use the graph after the tables).

(b) (optional) Find the reaching definitions in the *CFG*, informally or again by using the flow equation method (for reaching definitions). Use the tables on the next pages. Write the solution by the graph nodes (use the graph after the tables).

node	*defined*
1	
2	
3	
4	
5	
6	
7	

node	*used*
1	
2	
3	
4	
5	
6	
7	

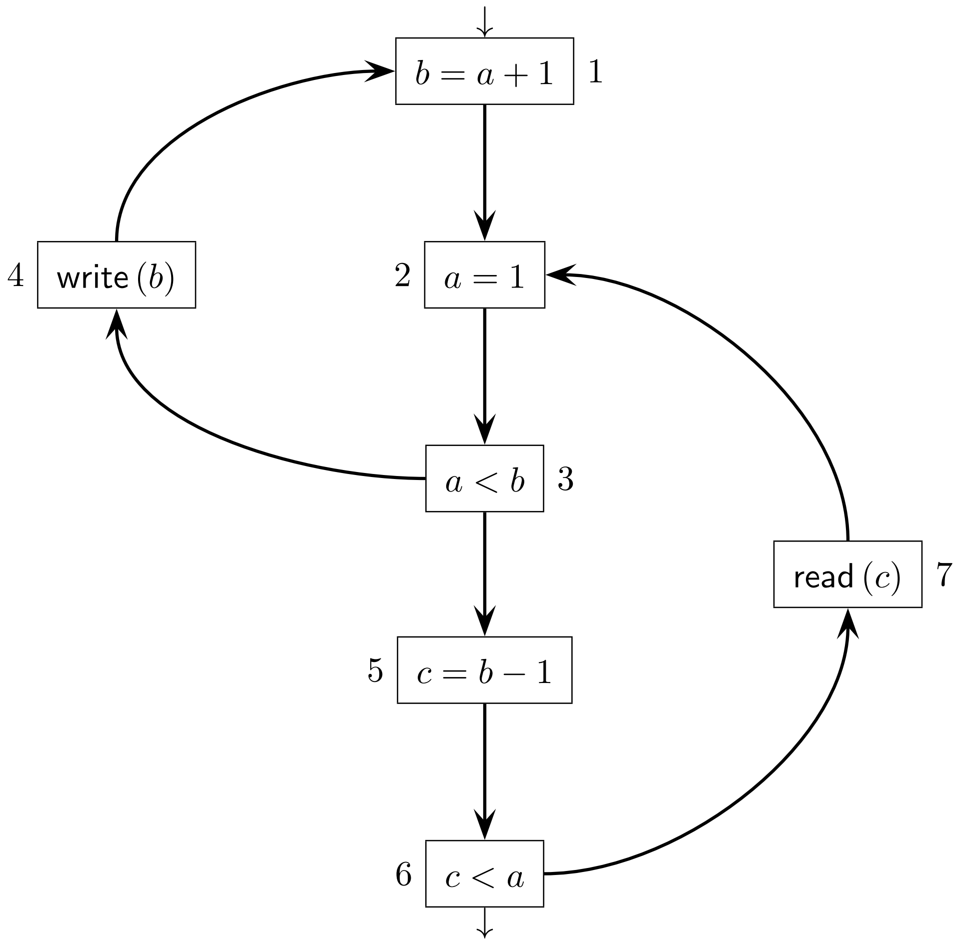
system of data-flow equations for **LIVE VARIABLES**

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

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

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

please here write the **LIVE VARIABLES**



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**

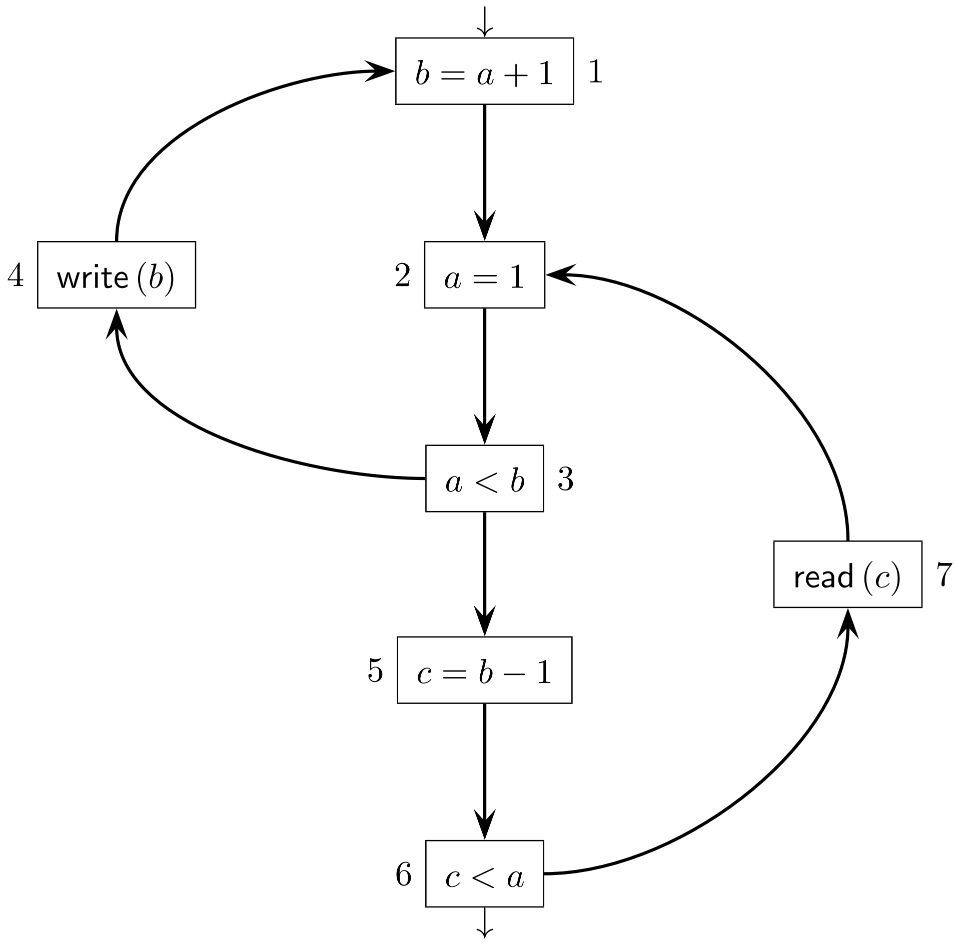
node	*in equations*	*out equations*
1		
2		
3		
4		
5		
6		
7		

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

(the number of 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														

please here write the **REACHING DEFINITIONS**



Solution

(a) Here is the computation of the live variables. First the tables of the defined and used variables, then the data-flow equations obtained from the program *CFG*:

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

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

system of data-flow equations for live variables

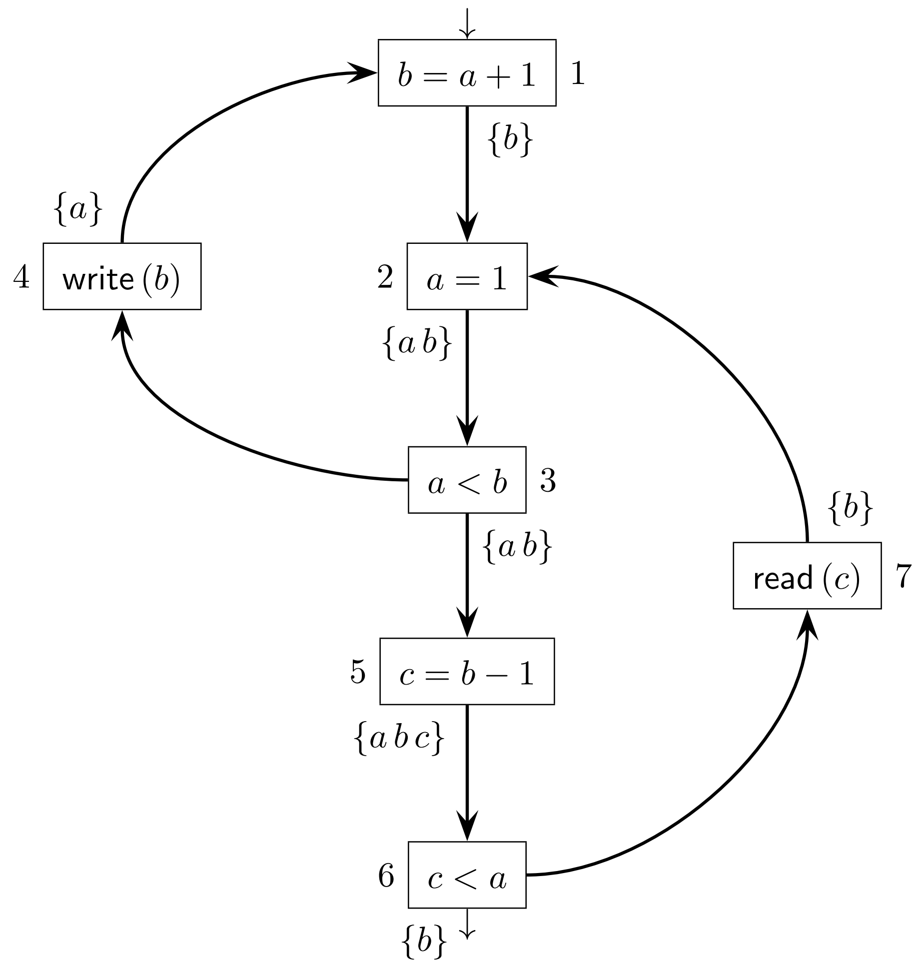
node	*in equations*	*out equations*
1	$in(1) = \{a\} \cup (\, out(1) - \{b\}\,)$	$out(1) = in(2)$
2	$in(2) = out(2) - \{a\}$	$out(2) = in(3)$
3	$in(3) = \{a, b\} \cup out(3)$	$out(3) = in(4) \cup in(5)$
4	$in(4) = \{b\} \cup out(4)$	$out(4) = in(1)$
5	$in(5) = \{b\} \cup (\, out(5) - \{c\}\,)$	$out(5)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

Next the equation system is solved iteratively, by initially setting all the *out* variable to $\emptyset$ and then (re)computing all the variables until convergence is reached.

iterative solution table of the system of data-flow equations

	initialization		1		2		3		4	
#	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*
1	$\emptyset$	a	$\emptyset$	a	b	a	b	a	b	a
2	$\emptyset$	$\emptyset$	$a\ b$	b	$a\ b$	b	$a\ b$	b	$a\ b$	b
3	$\emptyset$	$a\ b$	b	$a\ b$	$a\ b$	$a\ b$	$a\ b$	$a\ b$	$a\ b$	$a\ b$
4	$\emptyset$	b	a	$a\ b$	a	$a\ b$	a	$a\ b$	a	$a\ b$
5	$\emptyset$	b	$a\ c$	$a\ b$	$a\ c$	$a\ b$	$a\ c$	$a\ b$	$a\ b\ c$	$a\ b$
6	$\emptyset$	$a\ c$	$\emptyset$	$a\ c$	$\emptyset$	$a\ c$	b	$a\ b\ c$	b	$a\ b\ c$
7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	b	b	b	b	b	b

The two rightmost *in* columns are identical, thus convergence is reached in four steps. Here is the solution (the rightmost *out* variables) reported on the *CFG*:



Intuitively four steps to converge is right, as variable b is the slowest to propagate to all the node outputs (see row 5 in the table) and it flows back from node 3 to node 5 through four nodes, namely 2, 7, 6 and 5, before reaching node 3.

(b) Here is the computation of the reaching definitions, using the data-flow equation method. First the tables of the defined and suppressed variables, then the data-flow equations obtained from the program *CFG*:

node	*defined*
1	b_1
2	a_2
3	–
4	–
5	c_5
6	–
7	c_7

node	*suppressed*
1	–
2	$a_?$
3	–
4	–
5	c_7
6	–
7	c_5

Notice that we have to introduce a definition $a_?$ to account for the usage of variable a in the node 1, where such a variable is still unassigned when the program starts (when the program execution passes through node 1 again, the variable has been assigned in the node 2). In practice we assume variable a is an input parameter to the program, assigned elsewhere before the program starts.

system of data-flow equations for reaching definitions

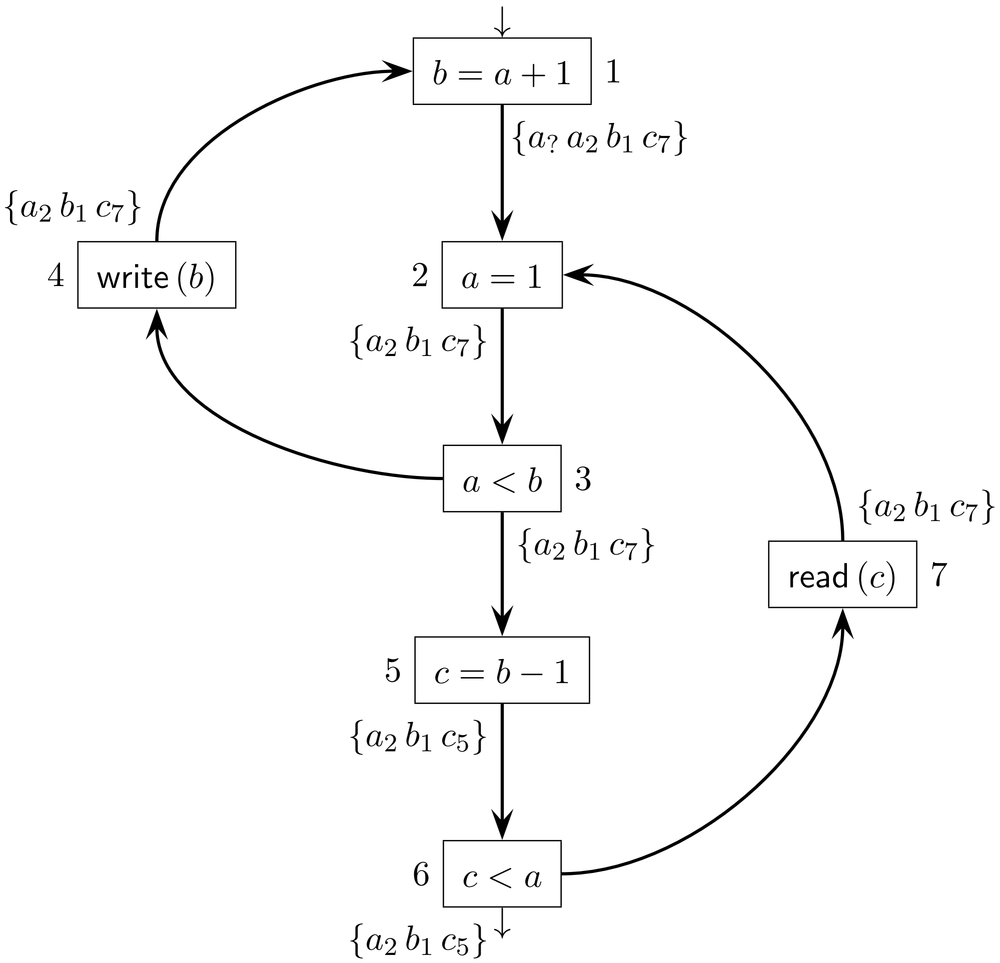
node	*in equations*	*out equations*
1	$in(1) = \{a_?\} \cup out(4)$	$out(1) = \{b_1\} \cup in(1)$
2	$in(2) = out(1) \cup out(7)$	$out(2) = \{a_2\} \cup (in(2) - \{a_?\})$
3	$in(3) = out(2)$	$out(3) = in(3)$
4	$in(4) = out(3)$	$out(4) = in(4)$
5	$in(5) = out(3)$	$out(5) = \{c_5\} \cup (in(5) - \{c_7\})$
6	$in(6) = out(5)$	$out(6) = in(6)$
7	$in(7) = out(6)$	$out(7) = \{c_7\} \cup (in(7) - \{c_5\})$

Next the equation system is solved iteratively, by initially setting all the *in* variable to $\emptyset$ and then (re)computing all the variables until convergence is reached.

iterative solution table of the system of data-flow equations

	initialization		1		2		3		4		5	
#	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*	*in*	*out*
1	$\emptyset$	b_1	$a_?$	$a_?\ b_1$	$a_?$	$a_?\ b_1$	$a_?a_2b_1$	$a_?a_2b_1$	$a_?a_2b_1c_7$	$a_?a_2b_1c_7$	$a_?a_2b_1c_7$	$a_?a_2b_1c_7$
2	$\emptyset$	a_2	$b_1\ c_7$	$a_2b_1c_7$	$a_?b_1c_7$	$a_2b_1c_7$	$a_?b_1c_7$	$a_2b_1c_7$	$a_?a_2b_1c_7$	$a_2b_1c_7$	$a_?a_2b_1c_7$	$a_2b_1c_7$
3	$\emptyset$	$\emptyset$	a_2	a_2	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$
4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	a_2	a_2	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$	$a_2b_1c_7$
5	$\emptyset$	c_5	$\emptyset$	c_5	a_2	$a_2\ c_5$	$a_2b_1c_7$	$a_2b_1c_5$	$a_2b_1c_7$	$a_2b_1c_5$	$a_2b_1c_7$	$a_2b_1c_5$
6	$\emptyset$	$\emptyset$	c_5	c_5	c_5	c_5	$a_2\ c_5$	$a_2\ c_5$	$a_2b_1c_5$	$a_2b_1c_5$	$a_2b_1c_5$	$a_2b_1c_5$
7	$\emptyset$	c_7	$\emptyset$	c_7	c_5	c_7	c_5	c_7	$a_2\ c_5$	$a_2\ c_7$	$a_2b_1c_5$	$a_2b_1c_7$

We can stop here, because one more step would show that the rightmost *in* column is stable. Therefore convergence is reached in five steps. Here is the solution (the rightmost *out* variables) reported on the *CFG*:



Intuitively five steps to converge is right, as definition b_1 is the slowest to completely propagate (see row 7 in the table) and it flows forth from node 1 to node 7 through five nodes, namely 2, 3, 5, 6 and 7, before reaching node 2 again.