# Formal Languages and Compilers
# (Linguaggi Formali e Compilatori)

## prof. Luca Breveglieri
## (prof. S. Crespi Reghizzi, prof. A. Morzenti)

## Written exam - 28 june 2008 - Part I: Theory

NAME:

SURNAME:

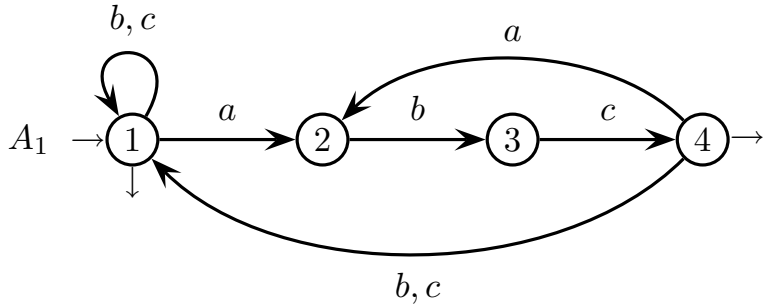
ID: SIGNATURE:

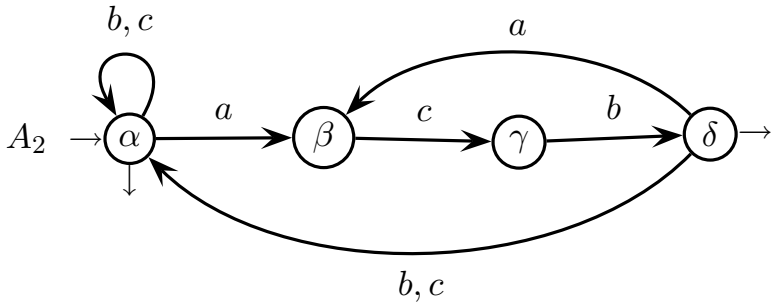
INSTRUCTIONS - READ CAREFULLY:

- The exam consists of two parts:
  - I (80%) Theory:
    1. regular expressions and finite automata
    2. free grammars and pushdown automata
    3. syntax analysis and parsing
    4. translation and semantic analysis
  - II (20%) Practice on Flex and Bison
- To pass the exam, the candidate must succeed in both parts (I and II), in one call or more calls separately, but within one year.
- To pass part I (theory) one must answer the mandatory (not optional) questions.
- The exam is open book (texts and personal notes are admitted).
- Please write in the free space left and if necessary continue on the back side of the sheet; do not attach new sheets nor replace the existing ones.
- Time: Part I (theory): 2h.30m - Part II (practice): 45m

# 1 Regular Expressions and Finite Automata 20%

1. Consider the two following finite state automata $A_1$ and $A_2$ recognising languages $L_1$ and $L_2$, respectively, both over alphabet $\{ a, b, c \}$.




The strings of the two languages $L_1$ and $L_2$ have the following respective description:

- in $L_1$ every letter $a$ is followed first by a letter $b$ and then by a letter $c$
- in $L_2$ every letter $a$ is followed first by a letter $c$ and then by a letter $b$

Answer the following questions:

(a) Check whether automata $A_1$ and $A_2$ are minimal, and if it is not already so, minimise them.

(b) Obtain a regular expression $R_3$ generating the intersection language $L_3 = L_1 \cap L_2$, in either one of the following two ways (at choice): reasoning informally on the descriptions of the two languages; or proceeding algorithmically starting from the two automata (for instance find the automaton of $L_3$ and obtain from it the regular expression).

(c) (optional) Obtain the two regular expressions $R_1$ and $R_2$ generating the two languages $L_1$ and $L_2$, respectively, in either one of the following two ways (at choice): reasoning informally on the descriptions of the two languages; or proceeding algorithmically starting from the two automata.

---

2. Give the following regular expression $R$, over alphabet $\{a, b, c\}$:

$$R = \big( \, ( \, a \, b \, )^+ \;\mid\; ( \, a \;\mid\; c \, )^* \, \big) \; ( \, c \, b \;\mid\; c \, a \, )^+$$

Answer the following questions:

(a) From regular expression $R$ obtain a finite deterministic recogniser $A$, by means of the Berry and Sethi method.

(b) (optional) Check whether the deterministic automaton $A$ obtained at point (a) is minimal and, if it is not already so, minimise it.

---

## 2 Free Grammars and Pushdown Automata 20%

1. Give a language $L$ over alphabet $\{a, b, c\}$, defined as follows:

$$L = \{\ a^n\, b\, c^m\, a\ \} \cup \{\ a^n\, b\, c^m\, a\, a\ \} \cup \{\ a^n\, b\, b\, c^m\, a\, a\ \}$$

   Language $L$ is expressed as the union of three components, with $0 \leq m \leq n$.

   Answer the following questions:

   (a) Write a grammar $G$, not ambiguous and not in extended form (BNF), generating language $L$.

   (b) (optional) Write a grammar $G'$, not ambiguous and not in extended form (BNF), generating the star language $L^*$ (that is the Kleene closure of $L$).

---

2. Consider the restriction of a high level programming language (like C, Pascal, etc) including the following syntactic features:

   - there are nominal variables, of integer type, and integer constants
   - variable and constant are denoted simply by terminal symbols "var" and "const", respectively, without further expanding them
   - there are expressions with variables, constants, and the addition "+" and subtraction "−" operations
   - the program statement is of two types, as follows:
     - assignment statement, with the following syntax:

       ```
       variable-name = expression
       ```

     - conditional construct of type `if-then` or `if-then-else`, like:

       ```
       if ( expression ) then
          statement-block
       ```

       ```
       if ( expression ) then
          statement-block
       else
          statement-block
       ```

   - a statement block is a list, not empty and enclosed in graph brackets "{" and"}", of statements separated by character ";" (semicolon); the separator must not be placed after the last statement of the list; for example:

     ```
     { statement; statement; statement }
     ```

   - if the block consists of only one statement (of any type), the enclosing graph brackets are optional (but may still be specified if one wants so)
   - in the case of a dangling else way, as in these two examples:

     ```
     if ( expression ) then
        if ( expression ) then
           statement-block
     else
        statement-block
     ```

     ```
     if ( expression ) then {
        if ( expression ) then
           if ( expression ) then
              statement-block
        else
           statement-block
     }
     ```

     the dangling `else` way should be syntactically linked to the outer `if` construct, compatibly with the nesting of statement blocks; the justification of the two examples above demonstrates the idea - CAUTION: SUCH A SPECIFICATION IS DIFFERENT FROM THE STANDARD C LANGUAGE
   - and finally, the program consists of only one statement block (there is not any variable declaration section)

   It is required to write the grammar, not ambiguous and in extended form (EBNF), generating the restricted programming language described above.

---

## 3 Syntax Analysis and Parsing 20%

1. The following grammar $G$ (in extended form EBNF) generates lists, the elements of which are separated by character $d$ and are well formed strings over alphabet $\{\, a,\, c \,\}$ (axiom $S$).

$$G \begin{cases} S & \rightarrow & P\ (\ d\ P\ )^* \\ P & \rightarrow & a\ P\ c\ \mid\ P\ a\ c\ \mid\ \varepsilon \end{cases}$$

   Answer the following questions:

   (a) Represent grammar $G$ in the form of a recursive network of finite state automata and draw the corresponding state-transition graphs.

   (b) Find the lookahead sets, of type $LL(1)$, for every arc and final dart of the network, and point out all the existing conflicts (if any).

   (c) (optional) If it is necessary, modify grammar $G$ and obtain an equivalent one satisfying the $LL(1)$ condition.

---

2. One wishes one designed a deterministic parser for the language $L$ defined by the following grammar $G$, over alphabet $\{a, b\}$ (axiom $S$):

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

Answer the following questions:

(a) Draw the driver (pilot) graph, of type $LR(1)$, of grammar $G$; just do what helps for answering the next question (to exhibit a conflict).

(b) Point out an existing conflict in the driver (pilot) graph, of type $LR(1)$, of grammar $G$ and explain of what kind it is

(c) (optional) Describe, informally or by means of a state-transition graph, a deterministic pushdown automaton $A$ recognising language $L$.

---

# 4 Translation and Semantic Analysis 20%

1. The input string to a transduction is a sequence of data tokens of integer (int) or character (char) type denoted by symbols $i$ and $c$, which have size of 2 or 1 bytes respectively. The transduction transforms the source string into a destination string consisting of a sequence of records. The record has a fixed size of 4 bytes. Every record must contain as many data tokens as possible up to size 4 and, if it is impossible to fill it exactly, must be completed by one padding symbol pad (see the example below). The start and end borders of every record are delimited by symbols beg and end, respectively. Here is an example (one may put a terminator $\dashv$ in the source string):

   Source string:

   $$c\ i\ i\ i\ c\ c\ i\ c\ \dashv$$

   Destination string:

   $$\mathsf{beg}\ c\ i\ \mathsf{pad}\ \mathsf{end}\ \mathsf{beg}\ i\ i\ \mathsf{end}\ \mathsf{beg}\ c\ c\ i\ \mathsf{end}\ \mathsf{beg}\ c\ \mathsf{pad}\ \mathsf{end}$$

   If the source string is empty (with terminator), the destination string is beg pad end.

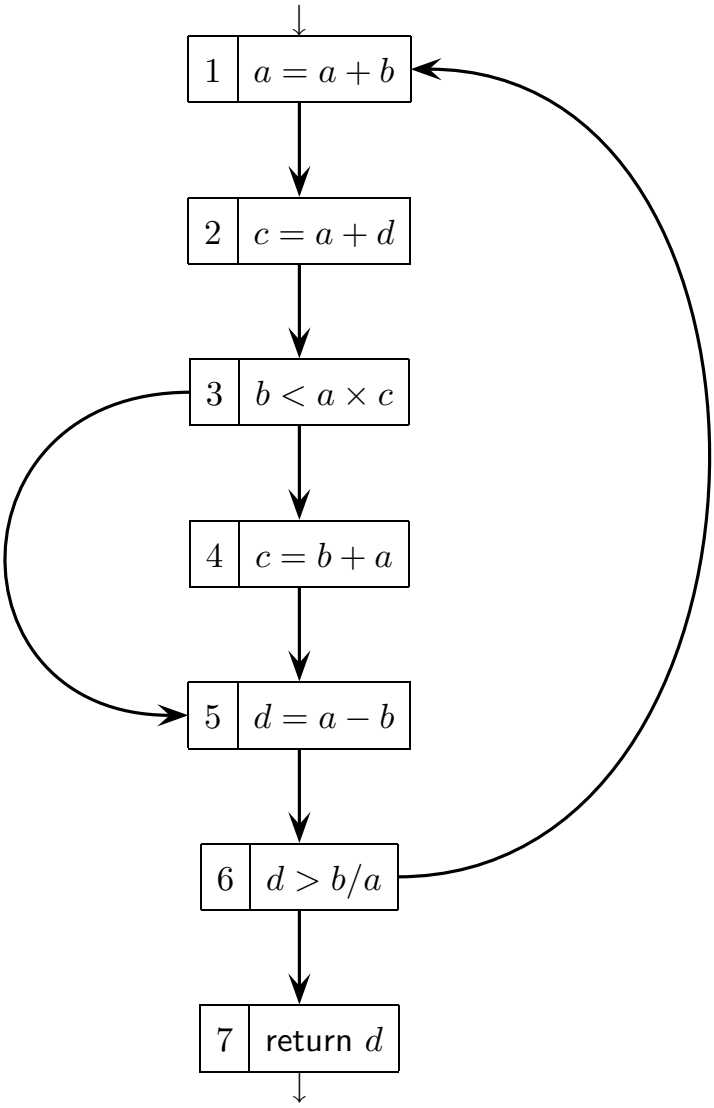
   Answer the following questions:

   (a) Choose the model most suited to the proposed transduction, of the three ones: IO-automaton, syntax transduction scheme (or grammar) or transduction regular expression; and design the transducer according to the chosen model.

   (b) (optional) If it is not already so, explain or demonstrate how to determinise the previously obtained transducer (one may change the model if a different one seems to be more convenient).

---

2. Consider the following control flow graph of a routine (with seven statements):




Answer the following questions:

(a) Write the flow equations for computing the liveness intervals of the variables (write in the table and grid on the next page - space is not significant).

(b) Use such equations and compute the sets of live variables in every point of the program (write in the grid on the next page - space is not significant).

(c) Still use the sets of live variables and point out the program statements that can be removed as they correspond to unuseful definitions.

(d) (optional) Show whether and how to use the sets of live variables to minimise the memory occupation of the variables.

---

| # | **def** | **use** |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

| | | | |
|---|---|---|---|
| $in(1)$ | $=$ | $use(1) \cup (out(1) - def(1)) =$ | – input liveness definition |
| | | | |
| | | | |
| $out(1)$ | $=$ | | – node 1 has ... |
| $in(2)$ | $=$ | | – input liveness definition |
| | | | |
| | | | |
| $out(2)$ | $=$ | | – nodo 2 has ... |
| $in(3)$ | $=$ | | – input liveness definition |
| | | | |
| | | | |
| $out(3)$ | $=$ | | – nodo 3 has ... |
| $in(4)$ | $=$ | | – input liveness definition |
| | | | |
| | | | |
| $out(4)$ | $=$ | | – nodo 4 has ... |
| $in(5)$ | $=$ | | – input liveness definition |
| | | | |
| | | | |
| $out(5)$ | $=$ | | – nodo 5 has ... |
| $in(6)$ | $=$ | | – input liveness definition |
| | | | |
| | | | |
| $out(6)$ | $=$ | | – nodo 6 has ... |
| $in(7)$ | $=$ | | – input liveness definition |
| | | | |
| | | | |
| $out(7)$ | $=$ | | |

| | **pass** 0 | | **pass** 1 | | **pass** 2 | | **pass** 3 | |
|---|---|---|---|---|---|---|---|---|
| # | *in* | *out* | *in* | *out* | *in* | *out* | *in* | *out* |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |

| | **pass** 3 | | **pass** 4 | | **pass** 5 | | **pass** 6 | |
|---|---|---|---|---|---|---|---|---|
| # | *in* | *out* | *in* | *out* | *in* | *out* | *in* | *out* |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |