Formal Languages and Compilers (Linguaggi Formali e Compilatori)

prof. L. Breveglieri
(prof. S. Crespi Reghizzi, L. Sbattella)

Exam - 7 march 2007 - Part I: Theory

NAME:

SURNAME:

ID: SIGNATURE:

INSTRUCTIONS - PLEASE READ CAREFULLY:

- The exam consists of two parts:
  - I (80%) Theory:
    1. regular expressions and finite state automata
    2. context-free grammars and pushdown automata
    3. syntax analysis and parsers
    4. transduction and semantic analysis
  - II (20%) Practice on Flex and Bison
- To pass the complete exam the candidate is required to pass both parts (I and II), in whatever order in a single call or in different calls, but to conclude in one year.
- To pass part I (theory) the candidate is required to demonstrate a sufficient knowledge of the topics of each one the four sections (1-4).
- The exam is open-book: textbooks and personal notes are permitted.
- Please write clearly in the free space left on the sheets; do not attach or replace sheets
- Time: Part I (theory): 2h.30m - Part II (practice): 45m

1 Regular expressions and finite automata 20%

1. Each of the two following tables A and B consists of two rows, and each row contains two regular expressions: the former one is in column L_1, the latter one in L_2.

   Do the following points:

   (a) In each row, list by increasing length the three shortest strings belonging to the language defined by the set difference $L_1 \setminus L_2$.

   Table A

L_1	L_2	the three shortest strings of $L_1 \setminus L_2$
$a((b \mid bb)a)^+$	$(ab)^* ba$	
$(a(ab)^* b)^* (\varepsilon \mid a(ab)^* \mid ab)$	$(a(ab)^* b)^*$	

   (b) In each row, write a regular expression generating the language $L_1 \setminus L_2$ and use only union, concatenation, star and cross operators.

   Table B

L_1	L_2	a regular expression of $L_1 \setminus L_2$
$a((b \mid bb)a)^+$	$(ab)^* ba$	
$(a(ab)^* b)^* (\varepsilon \mid a(ab)^* \mid ab)$	$(a(ab)^* b)^*$	

   Write the answers to the questions in the rightmost column of Tables A and B.

2. Given a deterministic finite state automaton M that recognizes the language $L(M)$, consider the following constructions (orderly listed):

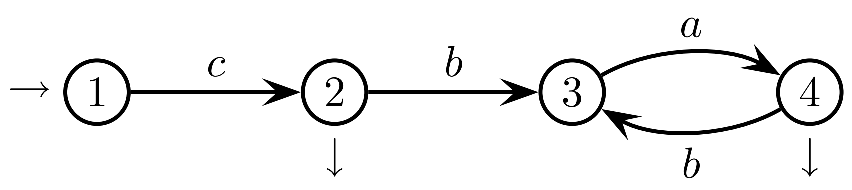
   i. transform automaton M into an automaton N_R that recognizes the mirror language $L(M)^R$

   ii. transform automaton N_R into a deterministic automaton M_R, equivalent to N_R

   iii. transform automaton M_R into an automaton M_{RR} that recognizes the mirror language $L(M_R)^R$

   Do the following points:

   (a) Start from the automaton M shown below, perform the above mentioned constructions (**i**), (**ii**) and (**iii**) (in the specified order), and draw the state-transition graphs of the requested automata N_R, M_R and M_{RR}.



   (b) Compare the two automata M and M_{RR}, and explain what makes them different from each other, and why.

2 Context-free grammars and pushdown automata 20%

1. Consider a subset of the Dyck language over the alphabet { '(', ')', '[', ']', '{', '}' }. Such subset matches both the following constraints:

   - the pairs of square parentheses contain an even number of pairs of round parentheses (at any depth level)
   - the pairs of graph parentheses contain an odd number of pairs of round parentheses (at any depth level)

   Sample correct strings:

   ```
   ( ( ( ) ( ) ) ) ( ( ( ) ) )        ( ( [ ( ) ( ) ] ) ) { ( ( ( ) ) ) }

   [ ( ) ( ) ] [ ( ( ) ) ]            { [ ( { ( ( ) ( ) ) } ) ] ( [ ( { ( ) } ) ] ) }
   ```

   Sample wrong strings:

   ```
   [ ( ) ( ) ( ) ] { ( ( ) ) }     { [ ( ( ( ) ( ) ) ) ] }     [ ( { ( ) ( ) } ) ( ) ]
   ```

   Do the following points:

   (a) Design a grammar in BNF form (not extended) that is not ambiguous and generates the above described language.

   (b) Explain informally (that is, by words), concisely but effectively, how the grammar designed at point (a) works.

---

2. Consider a simplified subset of the C programming language, which exhibits the following features:

   - a program is a purely sequential list (not empty) of assignment statements in C style; there are not any conditional or loop statements
   - the assignment instruction has the following syntax:

     ```
     <variable> = <expression> ;
     ```

   - on the left side of the assignment operator "`=`" there may be an identifier of:
     - a nominal variable, or
     - a pointer variable, preceded by the back-reference operator "`*`"

     and declaring such variables is not required here
   - the expression may contain the following term types:
     - nominal variable
     - back-referenced pointer variable
     - function call
   - the function call has the following syntax:

     ```
     <function-name> ( <parameter-list> )
     ```

     - the function name is an identifier (no need of declaring it)
     - the parameter is either a nominal or a pointer variable, possibly (but not necessarily) back-referenced
     - the function returns an integer or a pointer value; the latter must be back-referenced
     - the parameters are separated by "`,`" (comma)
     - the parameter list may be empty
   - the operators admitted in the expression are "+" (addition) and "/" (division)
   - the expression may contain round parentheses, to change the usual precedence rule of the operators "+" and "/' (division precedes addition)
   - the back-reference operator "`*`" has the highest precedence

   Sample program (of three statements):

   ```
   z = x / (y + f (z, w));

   w = x / (y + *p (l, *m));

   *n = y + x / g (z, *m, *i) + z;
   ```

   Do the following points:

   (a) Design a grammar in EBNF form (extended) that is not ambiguous and models the above described simplified subset of the C language.

   (b) Specify what aspects of the proposed problem can not be modeled by a purely syntactic tool such as an EBNF grammar.

3 Syntax analysis and parsers 20%

1. Consider the following grammar G (in extended form), over the terminal alphabet $\{a, b, c, d\}$ and with axiom S:

$$\begin{aligned} S &\rightarrow A \dashv \\ A &\rightarrow (\, a\, A^*\, A \mid B\,)\; b \\ B &\rightarrow a\, C \mid c \\ C &\rightarrow d\, C \mid d \end{aligned}$$

   Do the following points:

   (a) Represent grammar G as a network of recursive finite state automata.

   (b) Check each of the four non-terminal symbols S, A, B and C separately, and find the smallest integer $k \geq 1$ such that the considered non-terminal is $LL(k)$.

   (c) (optional) If necessary design a new grammar, equivalent to G, such that it is $LL(k)$ for some lower integer k (than that found at point (b)).

---

2. Consider the following grammar G (not extended), over the terminal alphabet $\{a, b\}$ and with axiom S:

$$\begin{aligned} S &\rightarrow A \dashv \\ A &\rightarrow b\, B\, a \\ B &\rightarrow b\, B \mid a\, A \mid a \end{aligned}$$

Do the following points:

(a) Discuss briefly whether language $L(G)$ is regular or not.

(b) Prove formally that grammar G is of type $LR(1)$.

(c) Determine formally whether grammar G is of type $LR(0)$ or not.

(d) (optional) Discuss briefly whether grammar G is of type $LALR(1)$ or not.

4 Transduction and semantic analysis 20%

1. The source phrases are non-empty strings over the alphabet $\{a, c\}$, where a and c correspond to open and closed parentheses, respectively. Such phrases may be well or ill parenthesised. In the latter case (ill) there may be one or more c letters in excess (like for instance in the phrase $ac\overset{\downarrow}{c}ac$, where the little vertical dart points to the exceeding c), while having a letters in excess is not permitted (a string like aac is excluded). Caution: the vertical dart pointing to the exceeding c is not part of the language; here it is included only a a visual facility for the reader.

   One wishes one had a purely syntactic translator (that is, without attributes) that can correct a source phrase and translate it into a well parenthesised string over the alphabet $\{begin,\ end\}$. Correction is carried out by inserting in the appropriate positions more a letters, to match the letters c in excess (and then both are transilterated to *begin* and *end*, of course). Here follow a few examples, which demonstrate how correction works (b and e are short forms for *begin* and *end*, respectively):

source string	translated and possibly corrected string (and explanation)	
$a\,ac\,ac\,c$	$b\,be\,be\,e$	the source string is already correct and therefore it is simply transliterated to the destination alphabet
$\overset{\downarrow}{c}\,ac$	$\overset{\downarrow}{be}\,be$	the source string is wrong and the missing a letter is inserted by the translator as one b
$a\,ac\,c\,\overset{\downarrow}{c}\,\overset{\downarrow}{c}\,ac$	$b\,be\,e\,\overset{\downarrow}{be}\,\overset{\downarrow}{be}\,be$	the source string is wrong and the two missing a letters are inserted by the translator as two b's

   Do the following points:

   (a) Design the state-transition graph of a pushdown IO-automaton (an automaton with one input and one output tape), preferably deterministic, that computes the above described transduction; freely suppose that the source string has a terminator '$\dashv$'. Explain informally and briefly how the automaton works.

   (b) (optional) Write a transduction grammar or syntax scheme that computes the above described transduction.

2. Consider a program consisting of a list of *statements*. Each statement is modeled by means of two sets: the former one contains the names of the variables that the instruction *defines*, the latter one those that the instruction *uses*. For instance, the assignment statement "$c = c + a$" is modeled by the two sets "def c" and "use a, c", while the I/O statement "$read(a, c)$" is modeled by "def a, c" and "use $\emptyset$".

   The syntactic support is the following (axiom S):

$$
\begin{aligned}
S &\rightarrow L \\
L &\rightarrow \langle instr \rangle \ L \\
L &\rightarrow \langle instr \rangle \\
\langle instr \rangle &\rightarrow \text{`def'} \ D \ \text{`use'} \ U \ \text{`;'} \\
D &\rightarrow \text{`}\emptyset\text{'} \\
D &\rightarrow V \\
U &\rightarrow \text{`}\emptyset\text{'} \\
U &\rightarrow V \\
V &\rightarrow \text{`id'} \ \text{`,'} \ V \\
V &\rightarrow \text{`id'}
\end{aligned}
$$

   Symbol "id" is a variable identifier (name) and here it is modeled syntactically as a pure terminal. The name of the identifier is an available lexical attribute, called n (as it is available, one need not take care of computing it)

   Do the following points:

   (a) Define the appropriate attributes and write the semantic functions of an attribute grammar, to compute the set of the *live variables* as an attribute associated with the initial instruction of the program, that is with axiom S (to the purpose, fill the forms in the next pages).

   (b) Decorate the *syntax tree* shown a few pages ahead and write by each node the values of the associated attributes, according to the semantic functions of the attribute grammar designed at point (a).

   (c) Draw the *dependence graphs* of the attributes of the rules (or of the most important rules, at least) and write, at least partially, the pseudocode of the semantic evaluator suited for the identified dependence types.

---

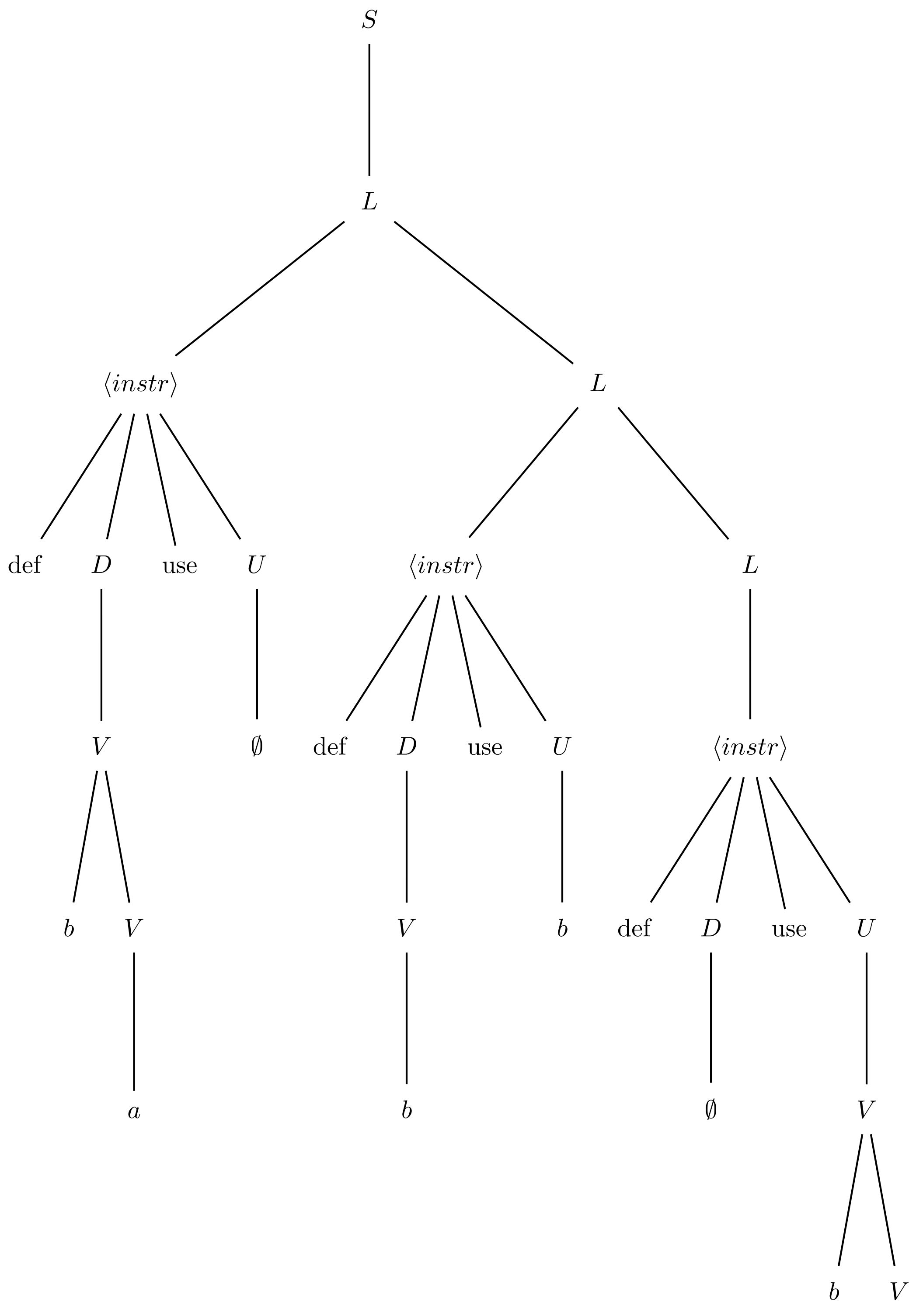
attributes to be used for the grammar

type	name	(non)terminals	domain	meaning
already given in the text of the exercise				
sx	n	id	string	name of the identifier
to be extended or added				

syntax	semantic functions
$S_0 \rightarrow L_1$	
$L_0 \rightarrow \langle instr \rangle_1 \ L_2$	
$L_0 \rightarrow \langle instr \rangle_1$	
$\langle instr \rangle_0 \rightarrow$ ‘def’ D_1 ‘use’ U_2 ‘;’	
$D_0 \rightarrow$ ‘$\emptyset$’	

syntax	semantic functions
$D_0 \quad \rightarrow \quad V_1$	
$U_0 \quad \rightarrow \quad$ ‘$\emptyset$’	
$U_0 \quad \rightarrow \quad V_1$	
$V_0 \quad \rightarrow \quad$ ‘id’ ‘,’ V_1	
$V_0 \quad \rightarrow \quad$ ‘id’	

syntax tree (not complete) to be decorated


S
L
$\langle instr \rangle$
L
def
D
use
U
$\langle instr \rangle$
L
V
$\emptyset$
def
D
use
U
$\langle instr \rangle$
b
V
V
b
def
D
use
U
a
b
$\emptyset$
V
b
V