Formal Languages and Compilers (A.Y. 2015/2016) Solutions

July 5th, 2016

First name: **Last name:**

Matriculation n.: **e-mail:**

Lexical Analysis

Exercise 1 – 5pt

Consider the following deterministic finite automata on the alphabet Σ (with :

- $\mathcal{A}^1 =< \mathcal{Q}^1, \Sigma, \delta^1, \mathcal{F}^1, q_0^1 >$ accepting the regular language $\mathcal{L}^1$
- $\mathcal{A}^2 =< \mathcal{Q}^2, \Sigma, \delta^2, \mathcal{F}^2, q_0^2 >$ accepting the regular language $\mathcal{L}^2$

The language $\mathcal{L}^\wedge$ on alphabet Σ includes all the strings in Σ^* that belong to $\mathcal{L}^1 \cap \mathcal{L}^2$. Is $\mathcal{L}^\wedge$ a regular language? If not, why? if yes, provide a definition for the elements of automaton $\mathcal{A}^\wedge$ accepting $\mathcal{L}^\wedge$.

In defining an answer to the exercise maybe can be useful to consider that the language $\mathcal{L}^1 \cup \mathcal{L}^2$ is a regular language for which an accepting deterministic finite automaton can be derived transforming in a DFA the following NDFA:

- $\mathcal{Q}^\vee = \mathcal{Q}^1 \cup \mathcal{Q}^2 \cup \{q_0^\vee\}$
- $\Sigma^\vee = \Sigma \cup \epsilon$
- $\delta^\vee(q,a) = \begin{cases} \{\delta^1(q,a)\} & : q \in Q^1 \\ \{\delta^2(q,a)\} & : q \in Q^2 \\ \{q_0^1, q_0^2\} & : q = q_0^\vee \wedge a = \epsilon \end{cases}$
- $\mathcal{F}^\vee = \begin{cases} \mathcal{F}^1 \cup \mathcal{F}^2 & : \epsilon \notin \mathcal{L}^1 \cup \mathcal{L}^2 \\ \mathcal{F}^1 \cup \mathcal{F}^2 \cup \{q_0^\vee\} & : \epsilon \in \mathcal{L}^1 \cup \mathcal{L}^2 \end{cases}$

Moreover, given that the complementary language of a regular language is still a regular language for which an accepting automaton can be easily defined starting from the automaton for the original language, it could be an idea to define an intersection in term of union and complement of regular languages.

Solution:
In defining the solution it is useful to rewrite the intersection in terms of complementary sets and union. In particular it is well known that the following property holds:

$$\mathcal{L}^1 \cap \mathcal{L}^2 = \overline{(\overline{\mathcal{L}^1} \cup \overline{\mathcal{L}^2})} \tag{1}$$

Given an automaton $\mathcal{A} =< \mathcal{Q}, \Sigma, \delta, \mathcal{F}, q_0 >$ recognizing language $\mathcal{L}$ it results that the automaton accepting language $\overline{\mathcal{L}}$ can be built in the following way:

- $\overline{\mathcal{Q}} = \mathcal{Q}$
- $\overline{\Sigma} = \Sigma$
- $\overline{\delta} = \delta$
- $\overline{\mathcal{F}} = \mathcal{Q} - \mathcal{F}$
- $\overline{q}_0 = q_0$

As a result the automaton $\mathcal{A}^{\vee}$ recognizing language $\overline{\mathcal{L}^1} \cup \overline{\mathcal{L}^2}$ can be defined in the following way:

- $\mathcal{Q}^{\vee} = \mathcal{Q}^1 \cup \mathcal{Q}^2 \cup \{q_0^{\vee}\}$
- $\Sigma^{\vee} = \Sigma \cup \epsilon$
- $\delta^{\vee}(q, a) = \begin{cases} \{\delta^1(q, a)\} & : q \in Q^1 \\ \{\delta^2(q, a)\} & : q \in Q^2 \\ \{q_0^1, q_0^2\} & : q = q_0^{\vee} \wedge a = \epsilon \end{cases}$
- $\mathcal{F}^{\vee} = \begin{cases} (\mathcal{Q}^1 - \mathcal{F}^1) \cup (\mathcal{Q}^2 - \mathcal{F}^2) & : \epsilon \notin \overline{\mathcal{L}^1} \cup \overline{\mathcal{L}^2} \\ (\mathcal{Q}^1 - \mathcal{F}^1) \cup (\mathcal{Q}^2 - \mathcal{F}^2) \cup \{q_0^{\vee}\} & : \epsilon \in \overline{\mathcal{L}^1} \cup \overline{\mathcal{L}^2} \end{cases}$

Therefore automaton $\mathcal{A}^{\wedge}$ can be derived from automaton $\mathcal{A}^{\vee}$ deriving again an automaton for a complementary language.

Exercise 2 – 4pt

In the definition of a regular expression real languages generally includes the not operator (in addition to the traditional symbols defining regular expressions). The operator permits to identify the set of strings that do not match the regular expression pattern, given an alphabet Σ. So, given a regular expression r on Σ, the strings matching $\neg r$ on Σ are those not matching r. If useful the operator can be included in the definition of the regular expressions for the following languages:

- The language $\mathcal{L}$ including strings over the alphabet $\Sigma = \{a, b\}$ not containing the subsequence "ab"
  (e.g. $ab,\ aab,\ babba \notin \mathcal{L}$, while $\epsilon,\ a,\ b,\ aaa,\ bb \in \mathcal{L}$)

- The language $\mathcal{L}$ on the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ representing all the possible IPv4 addresses
  (e.g. 192.132.255.5, 25.25.255.1, 1.1.1.1 $\in \mathcal{L}$, 256.13.25.5, 255.255.255.01 $\notin \mathcal{L}$)

Solution:

1. $\neg((a|b)^* a(a|b)^* b(a|b)^*)$
2. $(25[0-5] \mid ((2[0-4] \mid 1[0-9] \mid [1-9])? \mid [0-9]).)^3$
   $(25[0-5] \mid ((2[0-4] \mid 1[0-9] \mid [1-9])? \mid [0-9]))$

Syntax Analysis

Exercise 3 – 10pts

Let's $\mathcal{G}$ the grammar defined by the following productions:

$$S \longrightarrow YXY \mid YX \quad X \longrightarrow XxYy \mid \epsilon \quad Y \longrightarrow yYx \mid yx \tag{2}$$

1. Discuss the applicability of parsing LL(1). In case the parser is not applicable, report all the issues. In such a case revise the grammar to remove possible issues hindering the applicability of LL(1) parsing and check again the applicability of LL(1) parsing, on the revised grammar $\mathcal{G}'$.
2. Discuss the applicability of parser LR(0) and SLR for the original grammar $\mathcal{G}$.

Solution:

1. LL(1) parsing is not directly applicable since:

   a. production $X \longrightarrow XxYy$ presents a left recursion
   b. productions $S \longrightarrow YXY \mid YX$ present left factoring issues
   c. productions $Y \longrightarrow yYx \mid yx$ present left factoring issues

   To solve the issues the listed productions should be substituted by the following ones:

   a. $X \longrightarrow X'$ $X' \longrightarrow xYyX' \mid \epsilon$ (removal of left recursion)
   b. $S \longrightarrow YXS'$ $S' \longrightarrow Y \mid \epsilon$ (left factoring)
   c. $Y \longrightarrow yY'$ $Y' \longrightarrow Yx \mid x$ (left factoring)

   After having removed the immediate causes hindering the applicability of LL(1) we should check the applicability of LL(1) parsing for the new grammar deriving the FIRST and FOLLOW sets (see Table 1) from which we can derive the parsing table (see Table 2). From the table it can be inferred that LL(1) parsing is now applicable since the table does not contain conflicts.

	FIRST	**FOLLOW**	**NULLABLE**
S	y	\$	no
S'	y	\$	yes
X	x	y, \$	yes
X'	x	y, \$	yes
Y	y	x, y, \$	no
Y'	x, y	x, y, \$	no

Table 1: FIRST, FOLLOW and NULLABLE sets

	$	x	y
S			$S \longrightarrow YXS'$
S'	$S' \longrightarrow \epsilon$		$S' \longrightarrow Y$
X	$X \longrightarrow X'$	$X \longrightarrow X'$	$X \longrightarrow X'$
X'	$X' \longrightarrow \epsilon$	$X' \longrightarrow xYyX'$	$X' \longrightarrow \epsilon$
Y			$Y \longrightarrow yY'$
Y'		$Y' \longrightarrow x$	$Y' \longrightarrow Yx$

Table 2: LL(1) parsing table

2. To decide if LR(0) and SLR(1) parsing strategies are applicable we derive the LR(0) automaton that is reported in Figure 1. Successively we can derive the LR(0) parsing table reported in Table 3. To fill the table productions are numbered according to the following order:

   1. $S \rightarrow YXY$ 2. $S \rightarrow YX$ 3. $X \rightarrow XxYy$ 4. $X \rightarrow \epsilon$ 5. $Y \rightarrow yYx$
   6. $Y \rightarrow yx$

   From the table we observe that there are conflicts on state 6 that impedes the applicability of such a kind of parser.

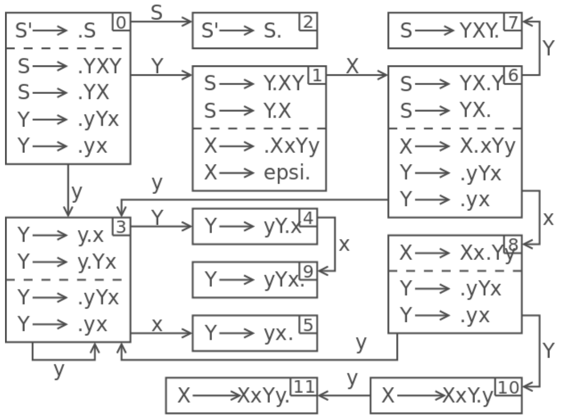

Figure 1: LR(0) automaton

In order to derive the table for the SLR(1) strategy it is necessary to compute the FOLLOW sets for the various non terminals. The resulting set is shown in Table 4. In Table 5 the SLR(1) parsing table is then reported. Since no conflicts are included in the table the parsing strategy can be applied.

	x	y	$	S	X	Y
0		S3		G2		G1
1	R4	R4	R4		G6	
2	acc.	acc.	acc.			
3	S5	S3				G4
4	S9					
5	R6	R6	R6			
6	**S8/R2**	**S3/R2**	R2			G7
7	R1	R1	R1			
8		S3				G10
9	R5	R5	R5			
10		S11				
11	R3	R3	R3			

Table 3: LR(0) parsing table

	FIRST	**FOLLOW**	**NULLABLE**
S	y	$	
X	x	x, y, $	yes
Y	y	x, y, $	

Table 4: FIRST, FOLLOW, NULLABLE sets for the original grammar

	x	y	$	S	X	Y
0		S3		G2		G1
1	R4	R4	R4		G6	
2	acc.					
3	S5	S3				G4
4	S9					
5	R6	R6	R6			
6	S8	S3	R2			G7
7			R1			
8		S3				G10
9	R5	R5	R5			
10		S11				
11	R3	R3	R3			

Table 5: SLR(1) parsing table

Semantic Analysis

Exercise 4 – 14pts

Consider the following excerpt from a grammar for a complex programming language:

$$S \rightarrow \textbf{alternate } S_1 \textbf{ and } S_2 \textbf{ till } (B) \qquad (3)$$

The command permits to define a cycle in which at each successive iteration a different branch is executed. So entering the cycle the first time statement S_1 will be executed, and then the condition checked. If the condition is false the cycle will be executed a second time. Nevertheless this time time statement S_2 will be executed, and then the condition checked again. If the condition is false in the next iteration statement S_1 will be executed. In conclusion the two statements in the command will be executed alternatively till the condition will assume the value true.

- Provide an L-attributed SDD for the command that permits to translate it in a three-address code program that behaves as expected[1]
- Show the parse tree and derive the three address code program for the code snippet below. In doing this refer to the translation schemes for expressions and commands which have been introduced during the course. It is not necessary to check the type of the expressions, and it can be assumed that variables have been declared somewhere before reaching the statement.

```
. . .
alternate
    i = v + 1
  and
    v = v - 2
till ( i < 0 )
. . .
```

Suggestion: in order to store the information about the turn it can be useful to introduce a variable (call it "`t`") that is instantiated to 0 before entering the command. Then when statement S_1 is executed the variable is increased by one, while is decreased by one when statement S_2 is executed. Therefore at each iteration the statement to execute next, in case the condition is false, will be decided according to the value of the variable. In particular if the value is one a jump will redirect the flow to the block of S_2 while if is zero the jump will redirect to statement S_2. So the whole command could be closed generating a three address code intructions that behaves as an if condition on the value of the variable `t`, that clearly can only assume the values `0` or `1`.

[1]For your convenience I recall that the function *top.get(id.lexeme)* permits to retrieve the address of the specified id, while the function *gen(. . .)* is used to generate three-address code in the right format for the different istructions, and finally *new Temp()* and *new Label()* permit to generate a new temporary address and a new label, respectively.

Solution:
The command can be translated taking inspiration from the translation for arithmetic and boolean expressions. In particular a the beginning we define a nen temporary variable that can assume only values 0 and 1 and then represent the turn for the next iteration. The variable is followed by the blocks for the command fragments S_1 and S_2 and then by the condition B, finally we include code for the jump according to the turn rule. As a result the following is a possible SDD satisfying the request:

$$\begin{array}{lcl}
turn & = & new\,Temp(), \\
S1Begin & = & new\,Label(), \\
S1End & = & new\,Label(), \\
S2Begin & = & new\,Label(), \\
S2End & = & new\,Label(), \\
Cond & = & new\,Label(), \\
Jump & = & new\,Label(), \\
B.true & = & S.next, \\
B.false & = & Jump, \\
S_1.next & = & S1End, \\
S_2.next & = & Cond, \\
S.code & = & gen(turn\,'='\;'0')|| \\
 & & label(S1Begin)||S_1.code||label(S1End)||gen(turn\,'='\;'1')|| \\
 & & gen(\mathbf{goto}\;Cond)|| \\
 & & label(S2Begin)||S_2.code||label(S2End)||gen(turn\,'='\;'0')|| \\
 & & label(Cond)||B.Code|| \\
 & & label(Jump)||gen(\mathbf{if}\;turn\;=\;0\;\;\mathbf{goto}\;S1Begin)|| \\
 & & gen(\mathbf{goto}\;S2Begin)
\end{array}$$

The three-address code corresponding to the snippet, and derived according to the parse tree shown in Figure 2, will then look like the following:

```
        turn = 0
S1Begin t1 = v + 1
        i = t1
S1End   turn = 1
        goto Cond
S2Begin t2 = v - 2
        v = t2
S2End   turn = 0
Cond    if i < 0 goto SNext
        goto Jump
Jump    if turn = 0 goto S1Begin
        goto S2Begin
SNext   ...
```

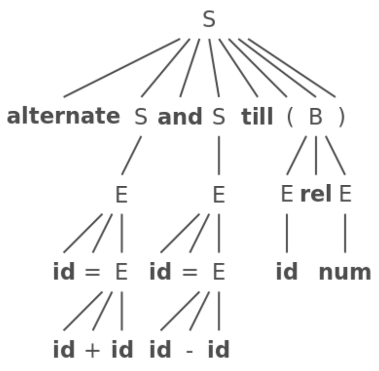

Figure 2: Parse tree for the code snippet