Formal Languages and Compilers
(A.Y. 2015/2016)
2h30m

July 26^{th}, 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 $\mathcal{L}^-$ on alphabet Σ includes all the strings in Σ^* that belong to $\mathcal{L}^1$ but not to $\mathcal{L}^2$, i.e. $\mathcal{L}^- = \{s \in \Sigma^* | s \in \mathcal{L}^1 \wedge s \notin \mathcal{L}^2\}$. Is $\mathcal{L}^-$ a regular language? If not, why? if yes, provide a definition for the elements of automaton $\mathcal{A}^-$ accepting $\mathcal{L}^-$.

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.

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 $\Sigma = \{a, b\}$ containing at least one occurrence of the character "a"
  (e.g. $\epsilon,\ b,\ bbb \notin \mathcal{L}$, while $\epsilon,\ a,\ ab,\ aaa,\ ba \in \mathcal{L}$)
- The language $\mathcal{L}$ on the alphabet $\Sigma = \{a, b, c, <, >\}$ representing all the possible couples of strings on $\{a, b, c\}$ ending with the same symbol.
  (e.g. $< a, aba >,\ < abb, cb >,\ < cca, ba > \in \mathcal{L},\ < a, b >,\ < ab, ac >,\ < aa, c > \notin \mathcal{L}$)

Syntax Analysis

Exercise 3 – 10pts

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

$$S \longrightarrow aSa \mid bSb \mid A \quad A \longrightarrow aBc \quad B \longrightarrow bAc \mid bc \mid \epsilon \tag{1}$$

1. Discuss the applicability of parser LR(0) and SLR for the original grammar $\mathcal{G}$.
2. show the steps of the parser SLR in the acceptance or rejection of the string *ababccba*

Semantic Analysis

Exercise 4 – 14pts

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

$$S \rightarrow \textbf{alternate } t \textbf{ times } S_1 \textbf{ and } S_2 \tag{2}$$

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 if t has not be reached S_2 will be executed. In conclusion the two statements in the command will be executed alternatively till the number of execution of the two statement is equal to t.

- 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 t times
    i = v + 1
  and
    v = i - 2
```

[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.