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

June 14^{th}, 2016

Lexical Analysis

Exercise 1 – 3pt

Consider the automaton $\mathcal{A} =< \mathcal{Q}, \Sigma, \delta, \mathcal{F}, q_0 >$ accepting the regular language $\mathcal{L}$. The language $\overline{\mathcal{L}}$ on alphabet Σ includes all the strings in Σ^* that do not belong to $\mathcal{L}$. Is $\overline{\mathcal{L}}$ a regular language? If not, why? if yes, provide a definition for the elements of automaton $\overline{\mathcal{A}}$ accepting $\overline{\mathcal{L}}$

Solution:

In order to derive a language $\overline{\mathcal{L}}$ that includes all and only the strings that do not belong to a language, it is enough to revise the definition of the final state set to include all and only the states in $\mathcal{Q}$ that are not in $\mathcal{F}$. So the elements of the new automaton $\overline{\mathcal{A}}$ will be:

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

Exercise 2 – 6pt

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:

1. The language $\mathcal{L}$ on the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ including only even numbers with no leading zeros.
   (e.g. 2, 4, 146, 1000 $\in \mathcal{L}$, while ϵ, 15, 016 $\notin \mathcal{L}$)

2. The language $\mathcal{L}$ on the alphabet $\Sigma = \{0, 1\}$ representing numbers that are multiple of 2 but not those that are also multiple of 8 (obviously in binary format - no leading 0 should be included)
   (e.g. 10, 100, 1010, 10100 $\in \mathcal{L}$, while 1000, 11000, 110000, 1, 101, 0, 010, $\epsilon \notin \mathcal{L}$)

3. The language $\mathcal{L}$ including strings over the alphabet $\Sigma = \{a, b\}$ containing at least one 'a' and one 'b'
   (e.g. ab, aab, ba, $babba \in \mathcal{L}$, while ϵ, a, b, aaa, $bb \notin \mathcal{L}$)

Solution:

1. $([1-9][0-9]^* \mid \epsilon)[0, 2, 4, 6, 8]$
2. $(1(0|1)^*|\epsilon)1(0|00)$
3. $\neg(a^*|b^*)$

Syntax Analysis

Exercise 3 – 10pts

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

$$S \longrightarrow Az \quad A \longrightarrow aA \mid azB \quad B \longrightarrow bA \mid bB \mid \epsilon \tag{1}$$

1. Discuss the applicability of parsing LL(1). In case the parser is not applicable, discuss why is not applicable. 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 revised grammar $\mathcal{G}'$.

2. Discuss the applicability of parser LR(0) and SLR(1) for the original grammar $\mathcal{G}$.

Solution:

- LL Parsing strategies cannot be applied since the grammar presents productions with the same starting sub-string for the same non-terminal. The issue could be solved applying the left-factoring strategy to the following produtions:
  - $A \longrightarrow aA \mid azB$
  - $B \longrightarrow bA \mid bB$

  Applying the left-factoring rules the previous productions will be substituted by the following ones:
  - $A \longrightarrow aX \quad X \rightarrow A \mid zB$
  - $B \longrightarrow bY \quad Y \rightarrow A \mid B$

  After the modification there are no evident issues to the applicability of LL(1) parsing therefore we proceed deriving the FIRST, FOLLOW, NULLABLE sets that is shown in Table 1.

	FIRST	**FOLLOW**	**NULLABLE**
S	a	$	
A	a	z	
B	b	z	yes
X	a, z	z	
Y	a, b	z	yes

Table 1: FIRST, FOLLOW, NULLABLE sets

The resulting LL(1) parsing table is shown in Table 2, and since it does not contain conflicts we can conclude that LL(1) parsing is applicable.

	a	b	z	$
S	$S \to Az$			
A	$A \to aX$			
B		$B \to bY$	$B \to \epsilon$	
X	$X \to A$		$X \to zB$	
Y	$Y \to A$	$Y \to B$	$Y \to B$	

Table 2: LL(1) parsing table

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

  1. $S \to Az$ **2.** $A \to aA$ **3.** $A \to azB$ **4.** $B \to bA$ **5.** $B \to bB$ **6.** $B \to \epsilon$

  From the table we observe that it includes conflicts on states 5 and 7 that impedes the applicability of such a kind of parser.

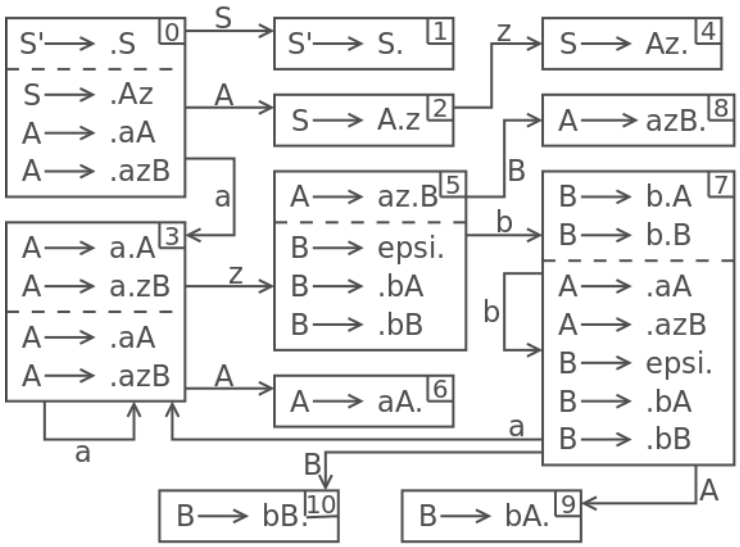

Figure 1: LR(0) automaton

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

	a	b	z	$	S	A	B
0	s3				G1	G2	
1	acc.	acc.	acc.	acc.			
2			s4				
3	s3		s5			G6	
4	r1	r1	r1	r1			
5	r6	**r6/s7**	r6	r6			G8
6	r2	r2	r2	r2			
7	**r6/s3**	**r6/s7**	r6	r6		G9	G10
8	r3	r3	r3	r3			
9	r4	r4	r4	r4			
10	r5	r5	r5	r5			

Table 3: LR(0) parsing table

	FIRST	**FOLLOW**	**NULLABLE**
S	a	$	
A	a	z	
B	b	z	yes

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

	a	b	z	$	S	A	B
0	s3				G1	G2	
1				acc.			
2			s4				
3	s3		s5			G6	
4				r1			
5		s7	r6				G8
6			r2				
7	s3	s7	r6			G9	G10
8			r3				
9			r4				
10			r5				

Table 5: SLR(1) parsing table

Semantic Analysis

Exercise 4 – 14pts

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

$$S \rightarrow \textbf{for } (\textbf{id} = \textbf{num}_1 \textbf{ to num}_2) \textbf{ do } S_1 \textbf{ rof} \tag{2}$$

The command permits to define a cycle that will be executed a fixed number of time, given by the difference of the two numbers ($\textbf{num}_2 - \textbf{num}_1$). In particular

in case the second number is smaller or equal to the first one the cycle will not be executed at all. Exiting from the cycle the variable used to index the cycle will have a value equal to the starting value plus the number of times the cycle has been executed. In fact after each cycle is executed the value of **id** should be increased by one.

- Provide an L-attributed SDD for the command that permits to translate it in a three-address code program behaving 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.

```
for (i = 5 to 10) do
  v = v + i
rof
```

Solution:

The command can be translated taking inspiration from the translation for arithmetic and boolean expressions. The idea is to instantiate the variable with the initial value, then to check the condition and in case the cycle is entered to increase the indexing variable before exiting. As a result the following is a possible SDD satisfying the request:

$$
\begin{array}{lcl}
addr & = & top.get(id.lexeme), \\
t_1 & = & new\ Temp(), \\
start & = & new\ Label(), \\
end & = & new\ Label(), \\
S_1.next & = & end, \\
S.code & = & gen(addr\ '='\ num_1.val)||label(start)|| \\
 & & gen(\textbf{if}\ addr \geq num_2.val\ \textbf{goto}\ S.next)||S_1.code||label(end)|| \\
 & & gen(t_1\ '='\ addr\ '+'\ 1)||gen(addr\ '='\ t_1)||gen(\textbf{goto}\ start)
\end{array}
$$

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:

```
      i = 5
start if i>=10 goto snext
      t2 = v + i
      v = t2
end   t1 = i + 1
      i = t1
      goto start
snext ...
```

[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.

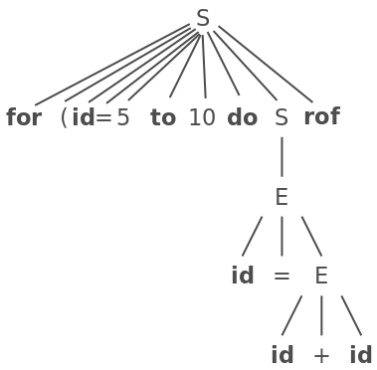

Figure 2: Parse tree for the code snippet