UNIT 5: TOP-DOWN PARSING TECHNIQUES

There is a very simple programming language oriented to the arithmetic calculation of a calculator. In this language, programs consist of a sequence of expressions (there may be any expression). Valid expressions are sequences of operators and numbers ending with the = sign. An example of a valid program is: (only operations of +, - and * are valid)

+ * + **6 8 6 9 =**
- + * - 45 23 2 5 2 =

There is no operator precedence. The expression is read from left to right. The result of the compiling process is shown on the screen and consists of the transformation of operations into equivalent sums and subtracts (the same operation, but without multiplications) and the result, following the example, the output on screen would show:

6 + 8 + 6 + 8 + 6 + 8 + 6 + 8 + 6 + 8 + 6 + 8 + 9 = 93
45 – 23 + 45 – 23 + 5 - 2 = 47

Numbers are positive integer.

It is required:

1. Define the grammar G that would generate valid statements of this programming language and the lexical analyzer.

2. Can the grammar G of the first exercise be used to perform an LL(1) analysis? If not, modify it so that it can. Generate the LL(1) table. Does this language require semantic verifications?

Note: Note that there are no quartets for multiplication, they have to be implemented with sums.

Instruction	**Meaning**
(move, pos_1,, pos_2)	$pos_2 \leftarrow pos_1$
(push, pos_1, ,)	incorporates the contents of pos_1 into the Stack
(pop, , , pos_1)	$pos_1 \leftarrow$ top of the Stack
(label, , , *label*)	defines a *label*
(goto, , , *label*)	go to a *label*
(return, , , *reg*)	go to the address in *reg*
(if, *reg*, , *label*)	go to *label* if the content of *reg* is -1
(<, *reg*, , *label*)	go to *label* if the content of *reg* is lower or equal to 0
(+, reg_1, reg_2, *reg*)	$reg \leftarrow reg_1 + reg_2$
(-, reg_1, reg_2, *reg*)	$reg \leftarrow reg_1 - reg_2$

Solution:

A grammar that generates the language of the problem is defined as follows:

G = {**S**, **C**, **Z**, **E**, **E'**, **O**, **O'**, **U**, **V**, **T**, **W** }, {¿, ?, (,), =>, /=>, ->, Id, Num, +, -, /, *, ;}, {**S**}
(1) **S**::= **C S**
(2) **S**::= **E S**
(3) **S**::= λ
(4) **E**::= **E'** -> **V**
(5) **E'**::= **O U**
(6) **O**::= Id
(7) **O**::= Num
(8) **U**::= **O' E'**
(9) **U**::= λ
(10) **O'**::= +
(11) **O'**::= -
(12) **O'**::= *
(13) **O'**::= /
(14) **V**::= Id **T**
(15) **T**::= λ
(16) **T**::= ; **V**
(17) **Z**::= **E**
(18) **Z**::= **C**
(19) **C**::= ¿(**E'**) => **Z W**
(20) **W**::= ?
(21) **W**::= /=> **Z** ?

Σ_N	FIRST					FOLLOW					
S	λ	¿	Id	Num		$					
C	¿					¿	?	Id	Num	/=>	$
E	Id	Num				¿	?	Id	Num	/=>	$
E'	Id	Num				->	)				
O	Id	Num				+	-	*	/	->	)
O'	*	/	-	+		Id	Num				
U	*	/	-	+	λ	->	)				
V	Id					¿	?	Id	Num	/=>	$
T	λ	;				¿	?	Id	Num	/=>	$
Z	¿	Id	Num			?	/=>				
W	?	/=>				¿	?	Id	Num	/=>	$

(22)

Table LL(1)		Σ_T												
		¿	**Id**	**Num**	**?**	**/=>**	**->**	**)**	**+**	**-**	*****	**/**	**$**	**;**
Σ_N	**S**	1	2	2									3	
	C	19												
	E		4	4										
	E'		5	5										
	O		6	7										
	O'								10	11	12	13		
	U						9	9	8	8	8	8		
	V		14											
	T	15	15	15	15	15							15	16
	Z	18	17	17										
	W				20	21								

It can be observed that both the terminal symbols "=>" and "(" do not appear in the LL(1) table, this means that in the lexical analysis both can be discarded, the reason is that they always appear after the “)” and “?” symbols, so it could be taken as a lexical component to the symbols “)=>” and “?(“.

It is necessary to do semantic verification in the assignment of variables, since it would be possible to have sentences in which the result was assigned several times to the same variable: a + 5-> b;b