UNITS 7 AND 8: SEMANTIC ANALYSIS and ERROR HANDLING

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 be used to perform an LR(1) analysis? Otherwise, modify it so that it can. Generate an SLR(1) analyzer that recognizes sentences of the language generated by G '(modified G of section 2).

3. Can the grammar G of the second exercise be used to perform an LR(1) analysis? Otherwise, modify it so that it can. Generate an SLR(1) analyzer that recognizes sentences of the language generated by G '(modified G of section 2).

4. Describe the semantic routines of the grammar G 'to generate intermediate code in quartets with the following instructions, where pos_i are memory addresses, registers, or a number, and *reg*, reg_1 and reg_2 can be a register or a number:

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

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*)	$reg \leftarrow reg_1 - reg_2$