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*)	$reg \leftarrow reg_1 - reg_2$

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

It cannot be known if it is LR(1) since it is necessary to generate the table LR(1) and check if there is any problem. We start from the grammar, creating a new symbol S 'and a new production S' :: = S. The 15 States, along with the transitions generated would be:

State 0	Action	Go To
S'::= ·S		[0,S]=1
S::= ·CS		[0,C]=2
S::= ·ES		[0,E]=3
S::= λ	[0,$]=R3	
C::= ·¿(E')=>ZW	[0,¿]=D4	
E::= ·E'->V		[0,E']=5
E'::= ·OU		[0,O]=6
O::= ·Id	[0,Id]=D7	
O::= ·Num	[0,Num]=D8	
State 1	Action	Go To
S'::= S·	[1,$]=ACP	
State 2	Action	Go To
S::= C·S		[2,S]=9
S::= ·CS		[2,S]=2
S::= ·ES		[2,S]=3
S::= λ	[2,$]=R3	
C::= ·¿(E')=>ZW	[2,¿]=D4	
E::= ·E'->V		[2,E']=5
E'::= ·OU		[2,O]=6
O::= ·Id	[2,Id]=D7	
O::= ·Num	[2,Num]=D8	
State 3	Action	Go To
S::= E·S		[3,E]=6
S::= ·CS		[3,C]=6
S::= ·ES		[3,E]=6
S::= λ	[3,$]=R3	
C::= ·¿(E')=>ZW	[3,¿]=D4	
E::= ·E'->V		[3,E']=6
E'::= ·OU		[3,O]=6
O::= ·Id	[3,Id]=D7	
O::= ·Num	[3,Num]=D8	
State 4	Action	Go To
C::= ¿·(E')=>ZW	[4,(]=D11	
State 5	Action	Go To
E::= E'·->V	[5,->]=D12	
State 6	Action	Go To
E'::= O·U		[6,U]=13
U::= ·O'E'		[6,O]=14
U::= λ	[6,->]=R9	
	[6,)]=R9	
O'::= ·+	[6,+]=D15	
O'::= ·-	[6,-]=D?	
O'::= ·*	[6,*]=D?	
O'::= ·/	[6,/]=D?	

O'::= +·	[14,Id]=R10 [14,Num]=R10

O::=Id
O.Value:= Id.Value; O.Code:= "";

O::=Num
O.Value:= Id.Value; O.Code:= "";

Z::=E
Z.Value:=E.Value; Z.Code:=E.Code;

Z::=C
Z.Value:=C.Value; Z.Code:=C.Code;

S::=λ
S.Code:=""

S_0::=CS_1
S_0.Code:=C.Code S_1.Code

S_0::=CS_1
S_0.Code:=E.Code S_1.Code

Use the Stack to know which variables to assign the Value of the expression, Stack = -1 Empty.

T::=λ
T.Code:=(push,-1,,)

T::=;V
T.Code:=V.Code

V::= Id T
V.Value:=newtemp; V.Code:=(push,,,Id)

O'::=+
O'.Code="" O'.Operation="+"

The Operation attribute is included to later know that Operation has to be performed.

O'::=-
O'.Code="" O'.Operation="-"

O'::=*
O'.Code="" O'.Operation="*"

O'::=/
O'.Code="" O'.Operation="/"

U::= λ
U.Code:=""

U::=O' E'
U.Value:=E'.Value; U.Operation=O'.Operation U.Code:=E'.Code

E'::=O U
E'.Value:=newtemp; E'.Code:= O.Code if U.Code="" then (move, O.Value, , E'.Value) else U.Code Select case U.Operation case "+" (move, O.Value,,A) (move, U.Value,,B) (+, O.Value,,A) case "-" (move, O.Value,,A) (move, U.Value,,B) (-, O.Value,,A) case "*" (move, O.Value,,A) (move, U.Value,,B) (*, O.Value,,A) case "/" (move, O.Value,,A) (move, U.Value,,B) (/, O.Value,,A) end select (move, A,, E'.Value)

E::=E'->V
E.Start:=newlabel; E.Stack_Empty:=newlabel; E.Code:=(label,,,Start) (pop,,,A) (if,A,,Stack_Empty) (move,A,E'.Value,) (goto,,,Start) (label,,,Stack_Empty)

C::=¿(E')=>ZW
C.False:=newlabel; C.Exit:=newlabel; C.Code:=E'.Code (<,E'.Value,,C.False) Z.Code (goto,,,C.Exit) (label,,,C.False) W.Code (label,,,C.Exit)

W::=?
W.Code:=""

W::=/=>Z?
W.Code:= Z.Code

State 7	Action	Go To
O::= Id·	[7,+]=R6 [7,-]=R6 [7,*]=R6 [7,/]=R6 [7,->]=R6 [7,)]=R6	
State 8	Action	Go To
O::= Num·	[8,+]=R7 [8,-]=R7 [8,*]=R7 [8,/]=R7 [8,->]=R7 [8,)]=R7	
State 9	Action	Go To
S::= CS·	[9,$]=R1	
State 10	Action	Go To
S::= ES·	[10,$]=R2	
State 11	Action	Go To
C::=¿(·E')=>ZW E'::= ·OU O::= ·Id O::= ·Num	 [11,Id]=D7 [11,Num]=D7	[11,E']=? [11,O]=6
State 12	Action	Go To
E::= E'->·V V::= ·Id T	 [12,Id]=D?	[12,V]=?
State 13	Action	Go To
E'::= OU·	[14,->]=R5 [14,)]=R5	
State 14	Action	Go To
U::= O'·E' E'::= ·OU O::= ·Id O::= ·Num	 [14,Id]=D7 [14,Num]=D8	[14,E']=? [14,E']=6
State 15	Action	Go To
O'::= +·	[14,Id]=R10 [14,Num]=R10	