UNITS 7 AND 8: SEMANTIC ANALYSIS and ERROR HANDLING

We want to construct a compiler for a language of definition and application of sequential machines. In the language you can define as many machines and process as many strings as you want. The language format is as follows:

- To declare a sequential machine, the MS instruction is used:

```
MS name_of_the_sequential_machine
{
inputs { symbol1, symbol2,…,symboln}
outputs { symbol1, symbol2,…,symbolk}
states { state1, state2, …}
transitions {
(state1, symbol_input1, state2, symbol_output1)
(state1, symbol_input1, state2, symbol_output1)
…
(state1, symbol_input1, state2, symbol_output1)
}
}
```

- For the sequential machine to process a string, it is used:

```
process ( name_of_automaton, string, initial_state)
```

The name of the sequential machine is a string of alphabetic characters. The statement of the set of states, the set of input and output symbols, and the set of transitions can be in any order, but they must always be included in every statement. The set of states, transitions, input and output symbols must have at least one value. An example of a sequential machine definition in this language would be:

```
MS Afirst
{
outputs {1,0}
states {a, c}
inputs {L,M,N}
transitions { (a,L,a,1)
(a,M,a,1)
(a,N,c,0)
(c,L,a,0)
(c,M,a,0)
(c,N,c,1)
}
}
process (Afirst, LLM, a)
process (Afirst, LLM, c)
```

To use the process function, the sequential machine used must be previously declared. The function displays, for the previous example, the following information:

```
MS: Afirst     Input: LLM  Output: 111
MS: Afirst     Input: LLM  Output: 011
```

It is required:

1. Define the grammar G that would generate valid sentences of this programming language.

2. Generate the first 15 states (including the state initial) of an LR (1) parser that recognizes statements of the language generated by G' (modified G of section 2). Show, for the elements of those states ("items"), what transitions of the LR (1) table would be generated with the created states.

3. Describe the semantic routines (in pseudocode) of the productions involved in the G or G' grammar that allow semantic control to verify that the MS instruction has the four sections (inputs, outputs, states and transitions) declared. If additional data structures are required, explain their usefulness.

SOLUTION:

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

G = {**A**, **B**, **D**, **E**, **R**, **S**, **V**, **W**, **Z** }, {( ) , { } automatonFD string states final initial name recognize t transitions}, {**S**}

(1) **A**::= λ
(2) **A**::= **S**
(3) **B**::= states { **V** }
(4) **B**::= final { **U** }
(5) **B**::= initial { t }
(6) **B**::= transitions { **W** }
(7) **D**::= automatonFD name { **B B B B** }
(8) **E**::= **D**
(9) **E**::= **R**
(10) **R**::= recognize (name, string)
(11) **S**::= **E A**
(12) **U**::= λ
(13) **U**::= **V**
(14) **V**::= t **Z**
(15) **W**::= λ
(16) **W**::= (t , t , t) **W**
(17) **Z**::= λ
(18) **Z**::= , **V**

State 0	Action	Go To
S’::= ·S		[0,S]=1
S::= ·EA		[0,E]=2
E::= ·D		[0,D]=3
E::= ·R		[0,R]=4
D::= · automatonFD name { B B B B }	[0,automatonFD]=D5	
R::= · recognize (name , string)	[0,recognize]=D6	

State 1	Action	Go To
S’::= S·	[1,$]=ACP	

State 2	Action	Go To
S::= E·A		[2,A]=7
A::= ·S		[2,S]=8
A::= λ	[2,$]=R1	
S::= ·EA		[2,E]=2
E::= ·D		[2,D]=3
E::= ·R		[2,R]=4
D::= · automatonFD name { B B B B }	[2,automatonFD]=D5	
R::= · recognize (name , string)	[2,recognize]=D6	

State 3	Action	Go To
E::= D·	[3,automatonFD]=R8 [3,recognize]=R8 [3,$]=R8	

State 4	Action	Go To
E::= R·	[4,automatonFD]=R9 [4,recognize]=R9 [4,$]=R9	

State 5	Action	Go To
D::= automatonFD · name { B B B B }	[5,name]=D11	

State 6	Action	Go To
R::= recognize · (name , string)	[6,(]=D11	

State 7	Action	Go To
S::= E A ·	[7,$]=R11	

State 8	Action	Go To
A::= S ·	[8,$]=R2	

State 9	Action	Go To
A::= λ	[9,$]=R1	

State 10	Action	Go To
D::= automatonFD name · { B B B B }	[10,{]=D12	

State 11	Action	Go To
R::= recognize (· name , string)	[5,name]=D13	

State 12	Action	Go To
D::= automatonFD name { · B B B B }		[12,B]=14
B::= · states { V }	[12,states]=D?	
B::= · final { U }	[12,final]=D?	
B::= · initial { t }	[12,initial]=D?	
B::= · transitions { W }	[12,transitions]=D?	

State 13	Action	Go To
R::= recognize (name · , string)	[13,","]=D?	

State 14	Action	Go To
D::= automatonFD name { B · B B B }		[14,B]=?
B::= · states { V }	[14,states]=D?	
B::= · final { U }	[14,final]=D?	
B::= · initial { t }	[14,initial]=D?	
B::= · transitions { W }	[14,transitions]=D?	

To generate the code for automatonFD, with the grammar described in section 2, it is necessary to do the semantic control that verifies that all sections are present. From the definition of the grammar of section 2 it is observed that the production in which this semantic control must be realized is:

D::= automatonFD name { **B B B B** }

When the reduction takes place, then the semantic actions associated with the production will be executed.

In addition to checking if the four sections (which is what is requested at this point), controls can also be performed at this time that the initial state, the final ones and those that appear in the transitions are declared in the set of states. By adding an attribute, section, to the non-terminal symbols, it will be possible to determine which production has been applied:

$$B.\text{section} = \begin{cases} 2 & \text{if initial reduced} \\ 3 & \text{if states reduced} \\ 5 & \text{if final reduced} \\ 7 & \text{if transitions reduced} \end{cases}$$

To avoid a very large set of nested "if" statements, we will produce the product of the different "B.section", if the result is 210, then each section will have been reduced regardless of the order. If a section is missing (because another is repeated) then the result will be different from 210. If correct, then a data structure containing the automaton definition must be created and then a function will be invoked to check for other semantic errors that can occur.

D::= automatonFD name { **B_1 B_2 B_3 B_4** }

```
{
k=newtemp();
k=B1.section * B2.section * B3.section * B4.section
IF k < > 210 then
        ERROR(Missing sections to declare)
ELSE
        new_automaton=new_structure()
        new_automaton=create_automaton(B1.list, B2.list, B3.list, B4.list)
        IF (verify_semantics(new_automaton))=F then
                ERROR(Incorrect declaration of the automaton)
        ELSE
                D.code = instructions_declaration_of_automaton()
        ENDIF
ENDIF
}
```

Las funciones:

newtemp → creates a temporal variable
ERROR → generates a call to the error manager
new_structure → declaration and memory required to contain an automaton
create_automaton → generates a data structure to store the definition of an automaton
verify_semantics → function that returns F if there is an error in the definition of the automaton
instructions_declaration_of_automaton → function that returns an string with the declaration of the automaton in the object code.

The data structure of the automaton can be as follows:

```
automaton{
        list_of_states          // definition of a list that contains the symbols of the states
        initial                 // variable with the symbol of the initial state
        list_de_final           // definition of a list that contains the symbols of the final states
        list_de_transitions     // definition of a list that contains the transitions
        }
```