Assignment   3

Due by 12/12   (Friday)   11:59 pm

Assignment 3 consists   of   1) Symbol table handling and 2) Generating an assembly code for the simplified version of Rat25F.

Simplified Rat25F:
- The **simplified Rat25F** is essentially the same except that the program has NO <Function Definitions>
- No "real" type is allowed

Some Semantics:
- Consider that "true" has an integer value of 1 and "false" has an integer value of 0.
- No arithmetic operations are allowed for booleans.
- The types must match for arithmetic operations (no conversions)

Part 1) Symbol table Handling (2%):   Every identifier declared in the program should be placed in a symbol table and accessed by the symbol table handling procedures.
a) Each entry in the symbol table should hold the lexeme, and a "memory address" where an identifier is placed within the symbol table. For example, define a global integer variable called "Memory_address" and set initially 10000 and increment it by one when a new identifier is declared and placed into the table.
b) You need to write a procedure that will check to see if a particular identifier is already in the table, a procedure that will insert into the table and a procedure that will printout all identifiers in the table. If an identifier is used without declaring it, then the parser should provide an error message. Also, if an identifier is already in the table and wants to declare it for the second time, then the parser should provide an error message. Also, you should check the type match.

 Part 2) Generating the assembly code (8%):
Modify your parser according to the simplified Rat25F and add code to your parser that will produce the assembly code instructions. The instructions should be kept in an array and at the end, the content of the array is printed out to produce the listing of assembly code. Your array should hold at least 1000 assembly instructions. The instruction starts from 1.
The listing should include an array index for each entry so that it serves as label
to jump to. *The compiler should also produce a listing of all the identifiers.*

Our target machine is a virtual machine based on a stack with the following instructions

I1) **PUSHI** {**Integer Value**} Pushes the {Integer Value} onto the Top of the Stack (TOS)

I2) **PUSHM** {**ML - Memory Location**} Pushes the value stored at {ML} onto TOS

I3) **POPM** {**ML**} Pops the value from the top of the stack and stores it at {ML}

I4) **STDOUT** Pops the value from TOS and outputs it to the standard output

I5) **STDIN** Get the value from the standard input and place in onto the TOS

I6) **ADD** Pop the first two items from stack and push the sum onto the TOS

I7) **SUB** Pop the first two items from stack and push the difference onto the TOS
(Second item - First item)

I8) **MUL** Pop the first two items from stack and push the product onto the TOS

I9) **DIV** Pop the first two items from stack and push the result onto the TOS
(Second item / First item and ignore the remainder)

I10) **GRT** Pops two items from the stack and pushes 1 onto TOS if second item is
larger otherwise push 0

I11) **LES** Pops two items from the stack and pushes 1 onto TOS if the second item is
smaller than first item otherwise push 0

I12) **EQU** Pops two items from the stack and pushes 1 onto TOS if they are equal
otherwise push 0

I13) **NEQ** Pops two items from the stack and pushes 1 onto TOS if they are not equal
otherwise push 0

I14) **GEQ** Pops two items from the stack and pushes 1 onto TOS if second item is
larger or equal otherwise push 0

I15) **LEQ** Pops two items from the stack and pushes 1 onto TOS if second item is
Less or equal otherwise push 0

I16) **JUMPZ** {**IL - Instruction Location**} Pop the stack and if the value is 0 then jump to
{IL}

I17) **JUMP** {**IL**} Unconditionally jump to {IL}

I18) **LABEL** Empty Instruction; Provides the instruction location to jump to.

A Sample Source Code

"* this is comment for this sample code for assignment 3 "
```
#
integer    i, max, sum;    "declarations"

sum = 0;
i = 1;
get (max);
while (i <  max)  {
     sum = sum + i;
     i  = i + 1;
 }
put (sum + max);
#
```

One Possible Assembly Code Listing

```
1   PUSHI   0
2   POPM    10002
3   PUSHI   1
4   POPM    10000
5   STDIN
6   POPM    10001
7   LABEL
8   PUSHM   10000
9   PUSHM   10001
10  LES
11  JUMPZ   21
12  PUSHM   10002
13  PUSHM   10000
14  ADD
15  POPM    10002
16  PUSHM   10000
17  PUSHI   1
18  ADD
19  POPM    10000
20  JUMP    7
21  PUSHM   10002
22  PUSHM   10001
23  ADD
24  STDOUT
```

Symbol Table

Identifier	MemoryLocation	Type
i	10000	integer
max	10001	integer
sum	10002	integer

NOTE:
- **DO NOT CREATE YOUR OWN ASSEMBLY INSTRUCTIONS.
  USE ONLY PROVIDED INSTRUCTIONS.**

- **Turn in your document according to the instructions given in the project outline.**