Assignment 3

All sections: Submit by 12/16 (Monday) 11:59 pm

Assignment 3 consists of 1) Symbol table handling and
2) Generating an assembly code for the simplified version of our Grammar from assignment2 (arithmetic expressions and assignment statement for 80% of grade plus documentation(10%).

Or implement a Bottom-Up approach for the Syntax Analyzer.

Or add implementation to your Top-Down Syntax Analyzer.

100% of grade for all the production rules for declarative, while and if and assignment.
Extra credit for:

Integration of all the 3 stages.

Separate working iterations using different approaches.

Our Grammar:

- The **grammar** 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 5000 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 Rat18F 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

PUSHI **{Integer Value}** Pushes the {Integer Value} onto the Top of the Stack (TOS)
PUSHM **{ML - Memory Location**} Pushes the value stored at {ML} onto TOS
POPM **{ML**} Pops the value from the top of the stack and stores it at {ML}
STDOUT Pops the value from TOS and outputs it to the standard output
STDIN Get the value from the standard input and place in onto the TOS
ADD Pop the first two items from stack and push the sum onto the TOS
SUB Pop the first two items from stack and push the difference onto the TOS (Second item - First item)
MUL Pop the first two items from stack and push the product onto the TOS
DIV Pop the first two items from stack and push the result onto the TOS (Second item / First item and ignore the remainder)
GRT Pops two items from the stack and pushes 1 onto TOS if second item is larger otherwise push 0
LES Pops two items from the stack and pushes 1 onto TOS if the second item is smaller than first item otherwise push 0
EQU Pops two items from the stack and pushes 1 onto TOS if they are equal otherwise push 0
NEQ Pops two items from the stack and pushes 1 onto TOS if they are not equal otherwise push 0
GEQ Pops two items from the stack and pushes 1 onto TOS if second item is larger or equal, otherwise push 0
LEQ Pops two items from the stack and pushes 1 onto TOS if second item is less or equal, otherwise push 0
JUMPZ **{IL - Instruction Location}** Pop the stack and if the value is 0 then jump to {IL}
JUMP **{IL**} Unconditionally jump to {IL}
LABEL Empty Instruction; Provides the instruction location to jump to.

A Sample Source Code

```
[* this is comment for this sample code
   for assignment 3 *]

$$
int     i, max, sum;   [* declarations *]

sum = 0;
i = 1;
```

```
input ( max);
while (i <  max)  {
     sum = sum + i;
     i  = i + 1;
 }
output (sum + max);
$$
```

One Possible Assembly Code Listing

```
1    PUSHI     0
2    POPM     5002
3    PUSHI     1
4    POPM     5000
5    STDIN
6    POPM      5001
7    LABEL
8    PUSHM    5000
9    PUSHM    5001
10    LES
11    JUMPZ         21
12   PUSHM    5002
13   PUSHM    5000
14   ADD
15   POPM      5002
17   PUSHM    5000
17   PUSHI      1
18    ADD
19    POPM     5000
20    JUMP        7
21    PUSHM   5002
22    PUSHM   5001
23    ADD
24    STDOUT
```

Symbol Table

Identifier	MemoryLocation	Type
i	5000	integer
max	5001	integer
sum	5002	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.**