Unit 7 Assignment – A Programming Assignment

Due: Oct 21, 12pm in Dropbox for Unit 7 Assignments
Submit: Your source code in a pdf
Counts for 50 points under “Programs” in the course grade.

In this assignment we are to do an **extended** version Unit 3 Assignment 2 WITHOUT the benefit of the ANTLR generated parser and lexer. That is, you must meet the requirements by creating code from scratch. Recall the sample:

```
LET A = 4
LET B = 5
LET C = A * B + 7
PRINT A
PRINT " * "
PRINT B
PRINT " + 7 = "
PRINTLN C
```

Your program should parse and execute this input as it did for Unit 3.

The output was:

```
4*5+7=27
```

You may use any programming language you like except esoteric and funcational. I recommend that you structure your program in a manner similar to the structure in the grammar, especially for expression evaluation. That is, create functions that mimic the operations performed in the grammar rules.

The “**extended**” bit is the addition of integer division to expression evaluation which you may have noticed was not in expr.g. Recall that multiplcation and division has the same precedence.

A sample:

```
LET H = 8
PRINTLN H / 2
PRINTLN 64/16
```

Output:

```
4
4
```

Do not use any code generated by ANTLR or any other compiler compiler.

BTW, this problem is about 60 years old. Their are thousands of solutions online. But we learn by doing, not by copying. Do yourself a favor, do this on your own.

CECS 524 Unit 8 Assignment 2 Due Nov 4 in Dropbox Unit 8 Assignment 2

Continuation of Unit 7 Assignment

We continue to create our new language, the Simple Interrepted Language (SIL). In this assignment we add:

1. Line numbers – each programming instruction shall be on numbered line. Line numbers are integers. Each line must have a number and the numbers must be in ascending order. There is no limit on the length (except those imposed by your implementation language) of a line number.
2. INTEGER instruction – the INTEGER instruction is used to explicitly define interger variables ***before*** they are used. You might want to create a symbol class and implement a symbol table classes. The rules for variable names are as in the ANTLER grammar. Any number of integer variables can be created in an INTEGER instruction.
   a) INTEGER A
   b) INTEGER B, C, D, X , Y
3. INPUT instruction – the INPUT instruction is used to receive integer input from the keyboard. Only predefined integer variables can be used with the INPUT instruction. All values are inputted by pressing the Enter key. The enter key is pressed only once per INPUT instruction regardless of how many variables are read.
   a) INPUT A //RESULTS IN EXECUTION STOPING UNTIL AN ENTER IS PRESSED. IF YOU VALUE IS FOUND, AN ERROR OCCURS REPORTED AS "Line n missing input value" where n is the line in the SIL program where the error occurred.
   b) INPUT X, Y, Z //inputs three integer values into X, Y and Z in that order. The Enter key is pressed only once for the entire INPUT instruction. If there are insufficient values to match the variables, an error occurs reported as "Line n missing input value".
4. GOTO LN instruction – the GOTO LN instruction transfers control from the current line of the SIL program to the line indicated by LN. This transfer is unconditional.
5. IF … THEN instruction – the IF THEN instruction compares two expressions to determine the relationship of the two. There are three relational operations: =, <, >. There are no compund conditions (AND or OR). The the relationship evaluates to true, then the statement following the THEN (it must be on the same line) is executed. If the relationship is not true, control passes to the statement on the following line.
   a) IF A < B THEN PRINTLN "A < B"
   b) IF A = B THEN PRINTLN "A = B"
   c) IF A> B THEN PRINTLN "A > B"
   d) IF A*3 > X-4*Y THEN GOTO 100
   e) IF A ! B THEN GOTO 10 //! is !=
6. END instruction – tells the interpreter to stop. In Java, **public void End(){ System.exit(0);}**

Implementing the above will require the entire SIL program to be in "memory". Since the language has the ability to go to any line of instruction to any other line, we can no longer simply read and forget the incoming instructions. THERE IS TO BE NO RESETTING THE INPUT FILE AND SKIPPING OVER INPUT TO GET TO THE CORRECT LINE. DO NOT BOTHER TO TURN YOUR CODE IF THAT IS WHAT YOU DO, YOU WILL GET ZERO.

Building a memory structure to hold the program is a second semester programming task. All it requires is a class that holds an array if Line class which itself is only an integer followed by a string. Instead of parsing an external file, your program will parse its internal memory. Here is the structure I used in my program:

```
public class Memory { //this is Java
   class Line_Memory{ //inner class holds one line of the program
      short   lineNumber;
      String  line;

   }
   private Line_Memory[] program;                  //the entire SIL program is in this
array
   private int          curr_line;                 //the current line that is executing
   private boolean      mEOL, mEOF;        //immulate EOL and EOF in the memory
   …. more data and reading and parsing code
} // class Memory
```

Well formatted with lots of space Memory is about 70 lines.

Syntax of the Instructions:

INTEGER ID [,ID]*
INPUT ID [,ID]*
GOTO line_number
IF expression '=' | '>' |' < '| '!' expression THEN statement

Note: Your program already solves for expression! Reuse it!

There is an obvious problem with IF. We can only execute one statement if the relation is true. What if we need to execute more than one statement? There is an easy fix for this – FORTRAN did it for years. Negate the condition and goto the first line past the "block" of statements to be executed. For example, we need to perform a calculation and print the result out with a message if the relation is true. In Java it might look like this:

```
if(a==b) {
   c = a * b;   //assuming c needed later
   System.out.println ("a * b = " +c); //forgive the dumb quotes – this LibreOffice
}
x = c + x;
…
```

In SIL we would do this:

```
50 IF A ! B THEN GOTO 80
60 LET C = A * B
65 PRINT "a * b = "
70 PRINTLN C
80 X = C + X
…
```

Sample on the following pages.

```
10 PRINTLN "Hello, world!"
20 END
```

```
10 PRINT "Enter a number:"
20 INTEGER number
30 INPUT number
40 PRINT "The number is "
50 PRINTLN number
60 END
```

```
10 INTEGER   A, B, C
20 LET  A = 1
30 LET B = 19
40 LET C = A+B*4
50 PRINT "C="
60 PRINTLN C
70 END
```

```
10 println "Testing if then"
20 integer a, b
25 print "Enter a:"
30 input a
35 print "Enter b:"
40 input b
50 if a < b then println "a < b"
60 if a = b then println "a = b"
70 if a > b then println "a > b"
80 end
```

Note: My program upcases input on scanning. All SIL code is in upper case in “memory”

```
10 println "If block test"
20 integer a, b
30 print "Enter a b:"
40 input a,b
50 if a < b then goto 100
60 println "in if block"
70 println "doing a second statement"
80 println "and a third"
100 println "out of the block"
200 end
```

CECS 524 Unit 9 Assignment 1 Due Nov 14, 11pm in Dropbox Unit 9 Assignment 1

Continuation of Unit 8 Assignment 2

We will add to SIL a very simple subroutine capability. To do this we add four new instructions: GOSUB, RET, POP and PUSH.

The syntax of GOSUB is

GOSUB line_number

After GOSUB is executed, control flow transfers to the statement at line_number. If the line_number does not exist, issue this error:

Line number not found - ###

where ### is the non-existent line_number. Control remains in the "subroutine" until a RET instruction is encountered. RET causes execution control to return to the statement after the GOSUB. GOSUB call my be nested, theoretically infinitely, but for practical purposes we will limit the nested calls to 256 deep.

The syntax of RET is

RET

RET returns control to the statement following the last GOSUB. Clearly since there can be nested calls, a data structure dedicated to tracking the calls is required. Remember back in your second programming class you learned abouts stacks. This is perfect chance to use them.

Speaking of stacks, POP and PUSH while require a stack, too. We shall limit this stack to integers. The syntax for PUSH is

PUSH int

where int is an integer ***expression.*** The value resulting from the expression is then placed at the top of the integer stack.

POP is

POP ID

where ID is a declared integer variable.

Note: Declaring variables in a "subroutine" is not currently a good idea. After the first call to the subroutine, variables declared there will be errors. So SIL mimics COBOL and assembly language in declaring variables at the "top" of the program. We'll fix this later.

Samples: This is nested GOSUB

```
10 println "line 10"
20 gosub 200
30 println "line 30"
40 end
200 println "in the sub"
210 gosub 300
220 println "back from 300"
230 ret
300 println "start of 300 sub"
310 ret
320 end
```

```
Finding a series sum:
10 PRINTLN "This program finds the sum of 1 to n where n is entered by the user"
20 INTEGER N, SUM, I
30 PRINT "Enter n:"
40 INPUT n
50 GOSUB 100
60 PRINT "The sum of 1 to n is "
65 PRINTLN SUM
70 PRINT "Enter 0 to quit, 1 to do another sum:"
80 INPUT SUM
90 IF SUM = 1 THEN GOTO 30
95 END
100 PRINTLN "Finding the sum of 1 to ", n
105 LET SUM = 0
110 LET I = 1
120 IF I>N THEN GOTO 160
130 LET SUM = SUM + I
140 LET I = I + 1
150 GOTO 120
160 RET
170 END
```

```
Notes:
1. Reused SUM as the event variable in line 80
2. The classic FORTRAN event controlled loop structure is used in lines 120-150
because SIL does not have a count-controlled loop.
3. RET should be a stand alone statement.

Example output from above:
This program finds the sum of 1 to n where n is entered by the user
Enter n:3
Finding the sum of 1 to 3
The sum of 1 to n is 6
Enter 0 to quit, 1 to do another sum:1
Enter n:4
Finding the sum of 1 to 4
The sum of 1 to n is 10
Enter 0 to quit, 1 to do another sum:1
Enter n:5
Finding the sum of 1 to 5
The sum of 1 to n is 15
Enter 0 to quit, 1 to do another sum:0
```

POP and PUSH example:

```
5  integer x, y
10 println "pop and push test"
15 integer b
18 let b = 23
20 push  50*3 + b
30 integer a
40 pop a
50 println "a=" , a
60 push a
70 pop b
80 println "b=",b
90  push 5
100 push 7
110 gosub 200
11 push 2
12 push 4
13 gosub 200
120 end
200 println "in sub"
210 pop y
220 pop x
230 println "x+y=", x+y
240 ret
250 end
```

Notes:

1. PUSH's argument can be any integer expression
2. POP's destination must be a variable
3. A "subroutine" should not declare variables (for now)
4. I updated PRINT and PRINTLN to do comma separate items. Ex. PRINTLN "a=", a
   You don't have to but you should to make your SIL code clearer

Output from above:

```
pop and push test
a=173
b=173
in sub
x+y=12
in sub
x+y=6
```