CEN 324 – Project Part II
Submission Deadline: May 17, 2019, 11:50 pm

Extend the parser (in **JAVA**) you wrote in Project Part-I to generate assembly code for the target machine as explained in **LECTURE NOTES 8** for the following language grammar. You may use the three-address intermediate code (**LECTURE NOTES 9**) to help generate the assembly code. **The rest of the instructions are almost the same (as for Project P-I) except the ones in red.** The extended parser should accept a file (a program written in the language) from the command line: **`java -jar compiler.jar <filename>`**.

```
program --> stmts
stmts --> stmt stmts | stmt
stmt -->  if-stmt
        | repeat-stmt eos
        | assign-stmt eos
        | read-stmt eos
        | write-stmt eos

if-stmt -->  if expr then stmts end
           | if expr then stmts else stmts end
repeat-stmt --> repeat stmts until expr
assign-stmt --> id assignop expr
read-stmt --> read id
write-stmt --> write expr

expr --> simple-expr compop simple-expr | simple-expr
simple-expr --> term { addop term }
term --> factor { mulop factor }
factor --> id | num

id --> letter+
num --> digit+

letter --> [a-zA-Z]
digit --> [0-9]
eos --> ';'
compop --> '<' | '='
addop --> '+' | '-'
mulop --> '*' | '/'
assignop --> ':='
```

All the reserved words are underlined and all the non-terminals are shown in **bold**. The input/output statements begin with the reserved words **read** and **write**. The read statement can read only one

variable (the id) at a time, and a write statement can write only one expression at a time.

Make sure you generate the right assembly code for the target machine (TM) i.e., it should run, and produce the input and output, in the same way as the code in the program being compiled. Read the instructions and examples for the TM from LECTURE NOTES 8. Make sure you understand them fully before working on this project. In addition to the other outputs (for Project P-I) the compiler should generate an assembly file, for example <filename>.tm. This assembly file will be run in the TM simulator to test your compiler.

Semantic rules:

Variables are declared implicitly by use. All variables have integer data type. No nested scopes, only global scope. Therefore the symbol table does not need to maintain any scope information. There are only two simple types, integer and boolean. The only boolean values are those that result from the comparison of two integer values. They can only appear in the test expression of the if-stmt or repeat-stmt. A boolean value cannot be output using a write-stmt.

A typical **symbol table** for the following program:

```
read x;
y = x + 10;
write x;
write y;
x = x + y;
write x;
```

```
Variable name   Location          Line numbers
-------------------------------------------------
x                    0             0  1  2  4  4  5
y                    1             1  3  4
```

During code generation the variables will need to be allocated memory locations, we store this information in the symbol table. For now the locations can be viewed as simple integer indices that are incremented each time a new variable is encountered. We also store a cross reference listing of line numbers where variables are accessed. Note multiple references to the same variable (in this case x in line 4) in the same line generate multiple entries for that line in the symbol table.

One of the programs (**sample.txt**) written using the above grammar is as follows. The first part computes and print the FIBONACCI numbers upto the input number (n) and the other part computes the FACTORIAL of the input number (x) if it is less than 32.

```
read n;
next := 0;
first := 0;
second := 1;
c := 0;
repeat
      if c < 1 then
```

```
                next := c;
            end
            if c = 1 then
                next := c;
            else
                next := first + second;
                first := second;
                second := next;
            end
            c := c + 1;
            write next;
        until c = n;

        read x;
        if x < 32 then
            fact := 1;
            repeat
                fact := fact * x;
                x := x - 1;
            until x = 0;
            write fact;
        end
```

The extended parser should output the **Source Code (with annotations – tokens etc)**, the **Syntax Tree** and the **Symbol Table** of the program. In addition to other errors (syntax, language grammar, etc) the extended parser should check for the types and if type checking fails, it should also print an error message with the line number. As an example, the annotated source code, the syntax tree and the symbol table of the above program is listed below:

Source code: sample.txt

```
   1: read n;
        1: reserved word: read
        1: ID, name= n
        1: ;
   2: next := 0;
        2: ID, name= next
        2: :=
        2: NUM, val= 0
        2: ;
   3: first := 0;
        3: ID, name= first
        3: :=
        3: NUM, val= 0
        3: ;
   4: second := 1;
        4: ID, name= second
        4: :=
        4: NUM, val= 1
        4: ;
```

```
 5: c := 0;
      5: ID, name= c
      5: :=
      5: NUM, val= 0
      5: ;
 6: repeat
      6: reserved word: repeat
 7:   if c < 1 then
      7: reserved word: if
      7: ID, name= c
      7: <
      7: NUM, val= 1
      7: reserved word: then
 8:           next := c
      8: ID, name= next
      8: :=
      8: ID, name= c
 9:   end;
      9: reserved word: end
      9: ;
10:   if c = 1 then
      10: reserved word: if
      10: ID, name= c
      10: =
      10: NUM, val= 1
      10: reserved word: then
11:           next := c
      11: ID, name= next
      11: :=
      11: ID, name= c
12:   else
      12: reserved word: else
13:           next := first + second;
      13: ID, name= next
      13: :=
      13: ID, name= first
      13: +
      13: ID, name= second
      13: ;
14:           first := second;
      14: ID, name= first
      14: :=
      14: ID, name= second
      14: ;
15:           second := next
      15: ID, name= second
      15: :=
      15: ID, name= next
16:   end;
```

```
      16: reserved word: end
      16: ;
17:   c := c + 1;
      17: ID, name= c
      17: :=
      17: ID, name= c
      17: +
      17: NUM, val= 1
      17: ;
18:   write next
      18: reserved word: write
      18: ID, name= next
19: until c = n;
      19: reserved word: until
      19: ID, name= c
      19: =
      19: ID, name= n
      19: ;
20:
21: read x;
      21: reserved word: read
      21: ID, name= x
      21: ;
22: if x < 32 then
      22: reserved word: if
      22: ID, name= x
      22: <
      22: NUM, val= 32
      22: reserved word: then
23:   fact := 1;
      23: ID, name= fact
      23: :=
      23: NUM, val= 1
      23: ;
24:   repeat
      24: reserved word: repeat
25:           fact := fact * x;
      25: ID, name= fact
      25: :=
      25: ID, name= fact
      25: *
      25: ID, name= x
      25: ;
26:           x := x - 1
      26: ID, name= x
      26: :=
      26: ID, name= x
      26: -
      26: NUM, val= 1
```

```
  27:   until x = 0;
        27: reserved word: until
        27: ID, name= x
        27: =
        27: NUM, val= 0
        27: ;
  28:   write fact
        28: reserved word: write
        28: ID, name= fact
  29: end
        29: reserved word: end
        30: EOF
```

Syntax tree: sample.txt

```
  Read: n
  Assign to: next
    Const: 0
  Assign to: first
    Const: 0
  Assign to: second
    Const: 1
  Assign to: c
    Const: 0
  Repeat
    If
      Op: <
        Id: c
        Const: 1
      Assign to: next
        Id: c
    If
      Op: =
        Id: c
        Const: 1
      Assign to: next
        Id: c
      Assign to: next
        Op: +
          Id: first
          Id: second
      Assign to: first
        Id: second
      Assign to: second
        Id: next
    Assign to: c
      Op: +
        Id: c
        Const: 1
    Write
```

```
      Id: next
    Op: =
      Id: c
      Id: n
  Read: x
  If
    Op: <
      Id: x
      Const: 32
    Assign to: fact
      Const: 1
    Repeat
      Assign to: fact
        Op: *
          Id: fact
          Id: x
      Assign to: x
        Op: -
          Id: x
          Const: 1
      Op: =
        Id: x
        Const: 0
    Write
      Id: fact
```

Symbol table: samples.txt

Variable Name	Location	Line Numbers
n	0	1 19
next	1	2 8 11 13 15 18
first	2	3 13 14
second	3	4 13 14 15
c	4	5 7 8 10 11 17 17 19
x	5	21 22 25 26 26 27
fact	6	23 25 25 28

```
Checking Types...

Type Checking Finished
```

If line 22 in the source code is changed to:

```
      if x + 32 then
```

then type checking should print:

```
Checking Types...
Type error at line 22: if test is not Boolean

Type Checking Finished
```

RULES:

1. Obey honor code principles.
2. Read your homework carefully and follow the directives about the I/O format (data file names, file formats, etc.) and submission format strictly. Violating any of these directives will be penalized.
3. Obey coding convention.
4. Your **online submission** should include the following file and NOTHING MORE (no data files, object files, etc):
   P_2<Firstname>_<Lastname>_<student number>_compiler.zip.
5. Do not use non-English characters in any part of your homework (in body, file name, etc.).

IMPORTANT

I want everyone to divide their code into different phases (at least: lexical, semantic and code gen) of the compiler. Make a different sub-directory (sub-package) for each phase.

Submit a Makefile (*make all* command should build a self executable jar file e.g., compiler.jar) and a manifest file. It is recommended to code each production rule (if-stmt, repeat-stmt, read-stmt, write-stmt, assign-stmt, expr, simple-expr, term, etc) in a separate file/class/object.

At the minimum, your source code should contain at least **four** files (e.g: main.java, lexical.java, semantic.java and codegen.java) and **three** sub-directories (e.g: compiler/lexical, compiler/semantic and compiler/codegen where compiler is the root directory/package).

Submissions with less than three java files or without a Makefile will **loose 50% of the marks**. This is not to penalize you but to make you learn a very important practical skill that will help you latter in life as a software engineer.

Contact me through e-mail or in office if you need further clarifications.

Notes: Develop on either Linux or cygwin (www.cygwin.com) on Windows to use the GNU make utility.

GOOD LUCK.