

# COP 3402: System Software

## Summer 2014

### Homework #4 (PL/0 Compiler)

**Due on 07/26/2015 (Sunday) at 11:59PM.**

**All assignments must compile and run on the Eustis server. You have to submit this assignment before the above deadline or you will receive 0 points.**

You must extend the functionality of Homework #3 to include the additional grammatical constructs highlighted in yellow in the grammar below (Appendix B). Also, you must create a User's Guide for your PL/0 compiler.

#### **Example of a program written in PL/0:**

```
var x, w;  
begin  
    x:= 4;  
    read w;  
    if w > x then  
        w:= w + 1  
    else  
        w:= x;  
    write w;  
end.
```

#### **Component Descriptions:**

The compiler must read a program written in PL/0, generate PM/0 machine code and run it in the Virtual Machine (VM) you implemented in HW1. Your compiler must neither parse nor generate code for programming constructs that are not in the grammar described below (Appendix B).

### **Submission Instructions:**

#### **1.- Submit via Webcourses:**

1. Source code of the PL/0 compiler (including source code for scanner, parser/code generator, virtual machine and compile driver).
2. A text file (named “readme.txt”) with instructions on how to compile your project on Eustis, and how to use your compiler (instructions for all components must be included). The name of both students must be clearly indicated on this file.
3. A text file containing a correctly formed PL/0 program (one which has procedures), and the corresponding output files generated by your PL/0 compiler: cleaninput.txt, lexemelist.txt, lexemetable.txt, mcode.txt and stacktrace.txt.
4. A document (in Microsoft Word or PDF format, the latter is preferred) with incorrectly formed PL/0 programs and screenshots of the output of your PL/0 compiler showing the corresponding error message to each incorrect PL/0 program. This document must demonstrate at least ten different errors.
5. A User’s Guide for the PL/0 compiler (in Microsoft Word or PDF format, the latter is preferred).
6. All files should be compressed into a single file, in .zip format.

The User’s Guide should assume that the user has no experience whatsoever with the compiler and the PL/0 language and very little experience with the platform on which the compiler is run. At the very least, the User’s Guide should cover:

1. How to compile and run the PL/0 compiler.
2. How to use the PL/0 compiler once it is running.
3. How to use the PL/0 language.
  - a. E.g. how to use an “if” statement, how to assign a value to a variable, etc.
  - b. Examples of PL/0 code are very helpful.

## Appendix A:

Example 3: Use this example (recursive program) to test your compiler:

```
var f, n;
procedure fact;
  var ans1;
  begin
    ans1:= n;
    n:= n-1;
    if n = 0 then f := 1;
    if n > 0 then call fact;
    f:=f*ans1;
  end;
begin
  n:=3;
  call fact;
  write f;
end.
```

Example 4: Use this example (nested procedures program) to test your compiler:

```
var x,y,z,v,w;
procedure a;
  var x,y,u,v;
  procedure b;
    var y,z,v;
    procedure c;
      var y,z;
      begin
        z:=1;
        x:=y+z+w;
      end;
    begin
      y:=x+u+w;
      call c;
    end;
  begin
    z:=2;
    u:=z+w;
    call b;
  end;
begin
  x:=1; y:=2; z:=3; v:=4; w:=5;
  x:=v+w;
  write z;
  call a;
end.
```

## Appendix B:

### EBNF of PL/0:

```
program ::= block "." .
block ::= const-declaration var-declaration procedure-declaration statement.
constdeclaration ::= [ "const" ident "=" number { "," ident "=" number } ";" ].
var-declaration ::= [ "var" ident { "," ident } ";" ].
procedure-declaration ::= { "procedure" ident ";" block ";" }
statement ::= [ ident ":" expression
    | "call" ident
    | "begin" statement { ";" statement } "end"
    | "if" condition "then" statement ["else" statement]
    | "while" condition "do" statement
    | "read" ident
    | "write" ident
    | e ] .
condition ::= "odd" expression
    | expression rel-op expression.
rel-op ::= "=" | "<" | "<=" | ">" | ">=" .
expression ::= [ "+" | "-" ] term { ("+" | "-") term } .
term ::= factor { ("*" | "/" ) factor } .
factor ::= ident | number | "(" expression ")" .
number ::= digit { digit } .
ident ::= letter { letter | digit } .
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
letter ::= "a" | "b" | ... | "y" | "z" | "A" | "B" | ... | "Y" | "Z" .
```

**Based on Wirth's definition for EBNF we have the following rule:**

**[ ] means an optional item.**

**{ } means repeat 0 or more times.**

**Terminal symbols are enclosed in quote marks.**

**A period is used to indicate the end of the definition of a syntactic class.**

## Appendix C:

### Error messages for the tiny PL/0 Parser:

1. Use = instead of :=.
2. = must be followed by a number.
3. Identifier must be followed by =.
4. **const**, **var**, **procedure** must be followed by identifier.
5. Semicolon or comma missing.
6. Incorrect symbol after procedure declaration.
7. Statement expected.
8. Incorrect symbol after statement part in block.
9. Period expected.
10. Semicolon between statements missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
13. Assignment operator expected.
14. **call** must be followed by an identifier.
15. Call of a constant or variable is meaningless.
16. **then** expected.
17. Semicolon or } expected.
18. **do** expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot begin with this symbol.
24. An expression cannot begin with this symbol.
25. This number is too large.

**Note: Not all of these error messages may be used, and you may choose to create some error messages of your own to more accurately represent certain situations.**

## Appendix D:

### Recursive Descent Parser for a PL/0 like programming language in pseudo code:

As follows you will find the pseudo code for a PL/0 like parser. This pseudo code will help you out to develop your parser and intermediate code generator for tiny PL/0:

```
procedure PROGRAM;
begin
  GET(TOKEN);
  BLOCK;
  if TOKEN != "periodsym" then ERROR
end;

procedure BLOCK;
begin
  if TOKEN = "constsym" then begin
    repeat
      GET(TOKEN);
      if TOKEN != "identsym" then ERROR;
      GET(TOKEN);
      if TOKEN != "eqsym" then ERROR;
      GET(TOKEN);
      if TOKEN != NUMBER then ERROR;
      GET(TOKEN)
    until TOKEN != "commasym";
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN)
  end;
  if TOKEN = "varsym" then begin
    repeat
      GET(TOKEN);
      if TOKEN != "identsym" then ERROR;
      GET(TOKEN)
    until TOKEN != "commasym";
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN)
  end;
  while TOKEN = "procsym" do begin
    GET(TOKEN);
    if TOKEN != "identsym" then ERROR;
    GET(TOKEN);
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN);
    BLOCK;
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN)
```

```
    end;  
    STATEMENT  
end;
```

```
procedure STATEMENT;  
begin  
    if TOKEN = "identsym" then begin  
        GET(TOKEN);  
        if TOKEN != "becomessym" then ERROR;  
        GET(TOKEN);  
        EXPRESSION  
    end  
    else if TOKEN = "callsym" then begin  
        GET(TOKEN);  
        if TOKEN != "identsym" then ERROR;  
        GET(TOKEN)  
    end  
    else if TOKEN = "beginsym" then begin  
        GET TOKEN;  
        STATEMENT;  
        while TOKEN = "semicolon" do begin  
            GET(TOKEN);  
            STATEMENT  
        end;  
        if TOKEN != "endsym" then ERROR;  
  
        GET(TOKEN)  
    end  
    else if TOKEN = "ifsym" then begin  
        GET(TOKEN);  
        CONDITION;  
        if TOKEN != "thensym" then ERROR;  
        GET(TOKEN);  
        STATEMENT  
    end  
    else if TOKEN = "whilesym" then begin  
        GET(TOKEN);  
        CONDITION;  
        if TOKEN != "dosym" then ERROR;  
        GET(TOKEN);  
        STATEMENT  
    end  
end;  
end;
```

```
procedure CONDITION;  
begin  
    if TOKEN = "oddsym" then begin  
        GET(TOKEN);
```

```

        EXPRESSION
    else begin
        EXPRESSION;
        if TOKEN != RELATION then ERROR;
        GET(TOKEN);
        EXPRESSION
    end
end;

```

```

procedure EXPRESSION;
begin
    if TOKEN = "plussym" or "minussym" then GET(TOKEN);
    TERM;
    while TOKEN = "plussym" or "slashsym" do begin
        GET(TOKEN);
        TERM
    end
end;

```

```

procedure TERM;
begin
    FACTOR;
    while TOKEN = "multsym" or "slashsym" do begin
        GET(TOKEN);
        FACTOR
    end
end;

```

```

procedure FACTOR;
begin
    if TOKEN = "identsym" then
        GET(TOKEN)
    else if TOKEN = NUMBER then
        GET(TOKEN)
    else if TOKEN = "(" then begin
        GET(TOKEN);
        EXPRESSION;
        if TOKEN != ")" then ERROR;
        GET(TOKEN)
    end
    else ERROR
end;

```



## Appendix E:

### Symbol Table

Recommended data structure for the symbol.

```
#define MAX_SYMBOL_TABLE_SIZE 100

typedef struct symbol{
    int kind;          // const = 1, var = 2, proc = 3
    char name[12];     // name up to 11 chars
    int val;           // number (ASCII value)
    int level;         // L level
    int addr;          // M address
} symbol;

symbol_table[MAX_SYMBOL_TABLE_SIZE];
```

For constants, you must store kind, name and value.

For variables, you must store kind, name, L and M.

For procedures, you must store kind, name, L and M.