

University of Central Florida

Department of Computer Science

COP 3402: System Software

Fall 2023

Homework #4 (PL/0 Compiler)

Due on November 22nd, 2023 by 11:59 p.m.

NEW REQUIREMENT:

All assignments must compile and run on the Eustis server. Please see course website for details concerning use of Eustis.

Objective:

In this assignment, you must extend the functionality of Assignment 3 (HW3) to include the additional grammatical constructs highlighted in yellow in the grammar on Appendix B.

Example of a program written in PL/0:

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

Component Descriptions:

The compiler must read a program written in PL/0 and generate code for 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. If you do, your grade will be zero.

Submission Instructions and rubric:

1.- Submit via WebCourses:

1. Source code of the PL/0 compiler. (hw4compiler.c)
2. A text file with instructions on how to use your program entitled readme.txt.
3. A text file composed of the input file (a PL/0 program) to your Compiler to demonstrate a correctly formed PL/0 program. The Compiler output should Display on screen: The input program and a message indicating that the program is syntactically correct. Otherwise, show an error message.

4. Following the statement that the program is syntactically correct, then the generated code must be shown (VM assembly language). In addition, an output file (elf.text) must be created with the executable code for the VM (HW1), but instead of using instructions mnemonics, generate 3 numeric character to match the input file for your VM. For example, instead of the JMP 0 45 shown on the screen, in the file that instruction should be stored as: 7 0 45. Test your elf.text file in your VM.

Do not print the symbol table in this assignment.

5. Test cases for error: only associated to procedures and call
6. All files should be compressed into a single .zip format.
7. **Late assignments will not be accepted (for this project there is not a two day extension after the due date).**
8. Name your program: hw4compiler.c

Rubric:

Deduction	Description
------------------	--------------------

Deduction	Description
------------------	--------------------

-100	Does not compile on Eustis.
-100	Does not accept input filename from command line.
-100	If the compiler follows a different grammar.
-100	Submitting HW3 again without implementing procedures and call.
-15	Crashes handling statements (-15 for each statement that is not handled).
-80	Compiles and does nothing.
-70	Produces some instructions before segfaulting or looping infinitely
-10	Not Supporting error handling for procedures.(including error messages).
-10	No README.txt containing author names
-30	Not implementing correctly procedures in “block”
-30	Not implementing correctly call statements
-20	Does not generate executable file (elf.text) for the VM.

Appendix A:

Traces of Execution:

Example 1, if the input is (program no errors):

```
var x, y;  
begin  
  x := y + 56  
end.
```

The output should look like:

- 1.- Display the input (program in PL/0)
- 2.- Display the message “No errors, program is syntactically correct”
- 3.- Display the generated code (Assembly code for the VM)
- 4.- Create file with executable for your VM virtual machine (HW1). Call the file **elf.txt**

Example 2, if the input is (program with errors):

```
var x, y;  
begin  
  x := y + 56  
end           ← (notice period expected after the “end” reserved word)
```

The output should look like:

- 1.- Display the message “Error number xxx, period expected”

```
var x, y;  
begin  
  x := y + 56  
end  
  ***** Error number xxx, period expected
```

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
                | "while" condition "do" statement
                | "read" ident
                | "write" expression
                | empty ] .
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:

Suggested error messages for the PL/0 compiler:

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 **end** 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.
26. Identifier too long.
27. Invalid symbol.

Note:

1. Identifiers: Maximum 11 characters.
2. Numbers: Maximum 5 digits.
3. Invalid symbols are not accepted (or example % does not belong to PL/0 grammar).
4. Comments and invisible characters must be ignored and not tokenized.

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 should be used as a guidance for modifying your tiny compiler. Just focus in the lines highlighted in yellow for procedures and call. Those lines will give you an idea about where your compiler (HW3) should be modified.

A PL/0 compiler written in pascal will be posted to help you out.

Some pseudo code help you out in the implementation of procedures will be posted as well.

```
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 = "var" 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 != "semicolomsym" then ERROR;
    GET(TOKEN);
    BLOCK;
    if TOKEN != "semicolomsym" 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 = "semicolomsym" 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;

```

```

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;
    end
end;

```

```
        if TOKEN != ")" then ERROR;  
        GET(TOKEN)  
    end  
    else ERROR  
end;  

```

Appendix E:

Symbol Table

Recommended data structure for the symbol.

typedef struct

```
{
    int kind;           // const = 1, var = 2, proc = 3
    char name[10];      // name up to 11 chars
    int val;            // number (ASCII value)
    int level;          // L level
    int addr;           // M address
    int mark;           // to indicate that code has been generated already for a block.
```

```
} 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.