University of Central Florida
Department of Computer Science
COP 3402: System Software
Summer 2024

Homework #4 (PL/0 Compiler)

Due on July 21st, 2024 by 11:59 p.m.

NEW REQUIRMENT:
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:

- Submit the following files via WebCourses:
  - Source code of the PL/0 compiler (named hw4compiler.c)
  - A text file named readme.txt containing instructions on how to use your program
  - A text file with a sample input file (a PL/0 program) to demonstrate a correctly formed PL/0 program
    - This file should serve as an input for testing your program
  - Along with the text file for the sample input file, please provide the expected output file (elf.txt) that your program should generate
  - A folder containing test cases for errors only associated with procedures and calls (where it may contain more than two test case inputs)
    - Example: A test case for the error "call must be followed by an identifier" might be:
    -
      ```
      var a, b;
      begin
            call;
      end.
      ```
  - Compress all the above files into a single .zip file before submitting
- Your PL/0 compiler should function as follows:
  - If the input test case contains an error:
    - Display the specific error message in the terminal
    - Example:
      - Error: Call must be followed by an identifier
  - If the input test case is fully correct:
    - Display the input test case PL0 program in the terminal
    - Display the generated instructions in the terminal with the following format for each line:
      - instruction_name L M (e.g., JMP 0 30)
    - Output an **elf.txt** file containing the executable code for the VM from HW1
      - Each line of the instruction should be in the format:
        - opcode L M (e.g., 7 0 30, where 7 is the opcode for the JMP instruction)
- Hint: Before submitting your implementation, please test your generated code in your VM to ensure the generated instructions are correct
- **Do not** print the symbol table because you may opt for the deletion algorithm for symbol table management
- No late submissions will be accepted, as this project is due on Sunday, which already includes a two-day extension

Rubric:

Deduction	Description
-100	Does not compile on Eustis. Not compiling means the compilation process creates errors that prevent the generation of the a.out (executable file), or while running the program, the executable produces an immediate segmentation fault or crashes while running the grading test cases.
-100	Does not accept input filename from the command line. After compiling, the graders should be able to run your program using the command "./a.out input_file.txt" to test any grading test cases. If your program does not support this schema or asks for manual input of the input file name, it will result in this deduction.
-100	If the compiler follows a different grammar. Make sure you are following the provided grammar rather than the pseudo-code. If the grammar does not include certain functionality that is present in the pseudo-code, do not include it; otherwise, it will result in this deduction.
-100	Submitting HW3 again without implementing procedures and call.
-15	Incorrect implementation that generates wrong instructions for each incorrect if statement part in the statement function.
-80	Compiles but does nothing.
-70	Produces some instructions before segfaulting or looping infinitely.
-10	Not supporting error handling for procedures (including error messages).
-10	Not supporting error handling for call (including error messages).
-5	No README.txt containing author names.
-2.5	No sample input file and sample output file.
-2.5	No test cases folder.
-30	Not implementing procedures in the "block" correctly.
-30	Not implementing call statements correctly.
-10	Does not generate the elf.txt executable file for the VM.
-10	Does not display the generated instructions in the terminal.
-5	JMP instruction's M, JPC instruction's M, or CAL instruction's M not fully divisible by 3 (each occurrence).
-5	JMP, JPC, or CAL instruction not leading to the correct index in the code list (each occurrence).
-10	Program does not handle variables with the same name at different levels correctly.
-10	Level information not managed correctly (e.g., global environment level should be 0, and in a procedure, levels should increment accordingly).
-10	Marking or deletion algorithm for symbol table management does not work correctly. If using marking, every symbol should have a mark of 0 upon initial insertion and a mark of 1 once they are no longer usable.

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 := 1fi;
                if n > 0 then call fact fi;
                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 "**fi**"
| "**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.

This pseudo code might have programming construct that are not in HW4 grammar

```
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 != "semicolomsym" 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 != "semicolomsym" 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;
```

```
        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.