University of Central Florida
Department of Electrical Engineering & Computer Science
COP 3402: System Software
Summer 2012

Homework #4 (PL/0 Compiler)

Due Tuesday, July 31, 2012 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. This is especially important for this assignment in order to accomplish the compiler directives task.

Objective:
In this assignment, you must extend the functionality of Assignment 3 to include the additional grammatical constructs highlighted in yellow in the grammar below. The constructs highlighted in pink (procedures) are extra credit (+20 points). **In addition, your compiler driver must support the following compiler directives**:

-l : print the list of lexemes/tokens (scanner output) to the screen
-a : print the generated assembly code (parser/codegen output) to the screen
-v : print virtual machine execution trace (virtual machine output) to the screen

Example commands:

./compile –l –a –v	Print all types of output to the console
./compile –v	Print only the VM execution trace to the console
./compile	Print nothing to the console except for "in" and "out"

Note: You may want to also print all forms of output to a single output file in order to avoid losing points on the other requirements if your implementation of compiler directives does not work.

Example of a program written in PL/0:

```
int x, w;
begin
        x:= 4;
        in w;
        if w > x then
                w:= w + 1
        else
                w:= x;
        out w;
```

end.

Component Descriptions:

The **compiler driver** is a program that manages the parts of the compiler. It must handle the input, output, and execution of the Scanner (HW2), the Parser (HW3) / Intermediate Code Generator (HW3) and the Virtual Machine (HW1).

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.

Submission Instructions:

1.- Submit via WebCourses:

1. Source code of the PL/0 compiler.
2. A text file with instructions on how to use your program entitled readme.txt.
3. A text file composed of the input file to your Scanner and the output of your Parser to demonstrate a correctly formed PL/0 program. The Parser output should indicate the program is syntactically correct. Following the statement that the program is syntactically correct, the text file should contain the generated code from your intermediate code generator and the stack output from your Virtual Machine running your code.
4. A text file (or screenshots) composed of the input file to your Scanner and the output of your Parser to demonstrate all possible errors. This may require many runs and the Parser output should indicate which error is being identified.
5. All files should be compressed into a single .zip format.
6. **Late assignments will not be accepted.**

Appendix A:

Traces of Execution:

Example 1, if the input is:

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

The output should look like:

1.- A print out of the token (internal representation) file:

29 2 x 17 2 y 18 21 2 x 20 2 y 4 3 56 18 22 19

And it's symbolic representation:

varsym identsym x commasym identsym y semicolon beginsym identsym x becomessym identsym y plussym numbersym 56 semicolonsym endsym periodsym

2.- Print out the message " No errors, program is syntactically correct"

3.- Print out the generated code

4.- Run the program on the VM virtual machine (HW1)

Example 2, if the input is:

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

The output should look like:

1.- A print out of the token (internal representation) file:

29 2 1 17 2 2 18 21 2 1 20 2 2 4 3 56 18 22

And its symbolic representation:

varsym identsym x commasym identsym y semicolon beginsym identsym x becomessym identsym y plussym numbersym 56 semicolonsym endsym

2.- Print the message “Error number xxx, period expected”

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

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

```
int f, n;
procedure fact;
        int 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:

```
int x,y,z,v,w;
procedure a;
  int x,y,u,v;
  procedure b;
    int y,z,v;
    procedure c;
      int 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:

Items highlighted in yellow are new requirements for this assignment.
Items highlighted in pink are extra credit for this assignment (20 points).

EBNF of PL/0:

program ::= block "**.**" **.**
block ::= const-declaration var-declaration procedure-declaration statement**.**
constdeclaration ::= [**"const"** ident "=" number {"**,**" ident "=" number} "**;**"]**.**
var-declaration ::= [**"int** "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"** expression
| **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**, **int**, **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 != "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
   } 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.