PL/0 Compiler: User's Guide

Guillermo Alicea

07.23.2016

Table of contents

Introduction

The PL/0 language was created by Niklaus Wirth in 1975, as a simpler version of Pascal, with a purpose to serve as an educational programming language, most notably serving as an example of how to construct a compiler. This user's manual will help you understand the general syntax and grammar of the language, as well as supply you with ample examples to aid in your understanding of the language. Additionally, this manual will show you how to compile and run the PL/0 compiler that I have written as part of a project for Systems Software (COP 3402) at the University of Central Florida. An explanation of how the compiler works, as well as how it handles errors that may be in input files containing PL/0 code, is actually included.

How to Compile/Run

The intended environment for this PL/0 compiler is a UNIX-like environment, specifically the Eustis server at UCF. Therefore, the compiling method discussed in this manual will use the GNU compiler collection compiler (gcc) for C, as the PL/0 compiler is written entirely in C.

1) To compile the PL/0 compiler, using a terminal window, ensure that you are in the directory containing the compiler's source code.

2) In the terminal, enter "gcc CompileDriver.c" to compile the program with a default executable file name of a.out. If you wish to give the executable your own name, you would do this by entering "gcc CompileDriver.c –o <name>".

3) Before attempting to run the PL/0 compiler, ensure that a file containing your PL/0 source code is in the same directory as the executable, and that you are in that directory as well. The PL/0 source code **MUST** be named “input.txt”, or else the compiler will report and error and it will stop.

4) This PL/0 compiler supports three directives and how you run your executable will depend on which, if any, you want to use. The directives are:

-l : output the lexeme list

-m : output the machine code

-s : output the symbol table

5) Running the program can be done like so:

- No directive:

  “./a.out” or “./<name>”

- -l :

  “./a.out –l” or “./<name> -l”

- -m :

  “./a.out –m” or “./<name> -m”

- -s :

  “./a.out –s” or “./<name> -s”

- All directives :

  “./a.out –l –m -s” or “./<name> –l –m -s”

*In the event that a directive other than the two discussed is passed, the compiler will simply ignore them and compile the source code normally.

6) Once it is running, and under the condition that the compiler was successful in the lexical analysis and parsing processes, you may be prompted to input a number value depending on your PL/0 program. Once you are given the option to input into your terminal window, simply enter the value you wish to use (anything other than a numerical value is not supported by PL/0). After compilation, your output will be contained in the output files, discussed later, and on your screen, depending on whether or not you decided to use directives.

How to use the PL/0 language

The 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}.

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.

digit ;;= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

letter ::= "a" | "b" | … | "y" | "z" | "A" | "B" | ... | "Y" | "Z".

From the EBNF of PL/0, you can see that programs written in this language are modeled as such:

Consts ..	Any number of consts
Vars ..	Any number of vars
Procedures ..	Any number of procedures
Statements ..	Any number of statements

Examples of the different declarations and statements:

Constant declaration

Const x = 5;	If only one
Const x = 5, y = 4, z = 3;	If many

Variable declaration

Var x;	If only one
Var x, y, z;	If many

Procedure declaration

"…" denotes the fact that a block is held within a procedure and so it can contain everything that the main program can, however it must end with a ";"

```
Procedure foo;
        …
;
Procedure foo2;
        …
;
```

Statements:

Assignment

```
X := 4;
```

where x is a variable that has already been declared

Call

```
call foo;
```

Where foo is a procedure that has already been declared

Begin

```
begin
        x := 4;
        call foo;
end
```

It is worth noting that the final statement does not need to have a ";" after it – this is entirely optional and will not affect the compiler

If

```
If x = 4 then call foo
```

"odd" can be used as a condition as well:

```
If odd x then call foo
```

If – else

```
If x = 4 then call foo
else call foo1
```

While

```
While x = 4 do
        x := x + 1
```

Read

```
read x
```

Write

```
write x
```

Relations:

Equal to:	=
Not equal to:	<>
Less than:	<
Less than or equal to:	<=
Greater than:	>
Greater than or equal to:	>
Odd:	odd

Comments

```
/* this is a comment */
```

Included here is a fully formed program that makes use of all these conditions (excerpt from the second edition of Wirth's book Compilerbau)

```
const
  m =  7,
  n = 85;

var
  x, y, z, q, r;
/* multiply */
procedure multiply;
var a, b;

begin
  a := x;
  b := y;
```

```
  z := 0;
  while b > 0 do begin
    if odd b then z := z + a;
    a := 2 * a;
    b := b / 2
  end
end;
/* divide */
procedure divide;
var w;
begin
  r := x;
  q := 0;
  w := y;
  while w <= r do w := 2 * w;
  while w > y do begin
    q := 2 * q;
    w := w / 2;
    if w <= r then begin
      r := r - w;
      q := q + 1
    end
  end
end;
/* greatest common denominator */
procedure gcd;
var f, g;
begin
  f := x;
  g := y;
  while f <> g do begin
    if f < g then g := g - f
    else f := f - g
  end;
  write f
end;

begin
  x := m;
  y := n;
```

```
  call multiply;
  x := 25;
  y :=  3;
  call divide;
  x := 84;
  y := 36;
  call gcd
end.
```

Scope:

This compiler handles scope correctly – precedence in scope will be given to the most local declaration. For example,

```
const x = 5;
var a, b;
procedure foo1;
        var a;
        begin
                a := 4;
                write a
        end;
begin
        a := 6;
        call foo1;
        write a;
end.
```

Output:

4

6

Recursion and nested procedures:

Recursion is also allowed in PL/0, as are nested procedures.

Examples of both (provided to us in the assignment pdf):

Recursion -

```
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.
```

Nested Procedures -

```
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.
```

How the PL/0 compiler works

The compiler's procedure is very simple to understand, especially given the fact that it's major functions are separated into four different files – CompileDriver.c, LexicalAnalyzer.h, Parser.h, and VirtualMachine.h.

Compile Driver:

The compiler driver is the first step in the compilation process. It is responsible for reading in any directives and then, in order, directing the different functions of the compiler, beginning with lexical analysis, then parsing, then calling the virtual machine.

Lexical Analyzer:

The lexical analyzer simply reads in the input.txt file and strips it of any comments, creating cleaninput.txt in the process. Cleaninput.txt is then read and use to create a lexeme list and lexeme table which will be its primary output. Parserinput.txt is also created, which serves as the input for the parser. The lexical analyzer also has a bit of error handling in place for certain restrictions in the code naming and length of the variable names, as well as handling any invalid symbols. Variable names must begin with a letter and be no more than 11 characters long. Numbers must be no more than 5 digits long. Valid symbols are only those in the EBNF of PL/0, all others will generate an error. If any error is generated in the lexical analysis process, compilation will stop and parsing will not begin. Lastly, the lexeme list will be printed to the screen if the proper directive is given during execution.

Parser:

The parser is implemented recursively, structured in a way that coincides very closely with the EBNF of PL/0. Error handling and code generation is interleaved in the parsing process. Any error encountered in this step will terminate the compilation after reporting the first error encountered. If no errors are found, a success message will be printed to the screen to inform you that parsing was successful. The output of this step is mcode.txt, which is used in the virtual machine, and, optionally, output to the screen in the form of the machine code created in the code generation process and symbol table, should the proper directives be used during execution.

Virtual machine:

The virtual machine is the last step in the compilation process, and it uses the output from the parser (mcode.txt) to create the stack trace and any input (read) or output (write) of the source code. This is done by following, very closely, the PM/0 instruction set architecture. This step will output stacktrace.txt.

Compiler error handling

The full list of errors handled in the compilation process can be found in errorlist.txt, however I will also include it in this section, along with an explanation of each.

- **Error 1: No period at end of file**

  The character encountered after the final end should be a '.' to signify the end of the program

- **Error 2: Missing identifier after const**

  Const should be followed by some sort of identifier in the form "const = x". If x were missing, for example, this error would be thrown.

- **Error 3: Use = instead of :=**

  Constants are defined using "=" instead of ":="

- **Error 4: Const's should be assigned using =**

  Same as error 3, except something other than ":=" was found

- **Error 5: = should be followed by number**

  In a Constant assignment, you should be using a number and nothing else

- **Error 6: Declaration must end with ;**

  Declarations in PL/0 must end with a ;

- **Error 7: Missing identifier after var**

  Var should be followed by an identifier which will be declared as a var

- **Error 8: Missing identifier after procedure**

  Procedure should be followed by an identifier which will be declared as a procedure

- **Error 9: Procedure declaration must end with ;**

  Declarations in PL/0 must end with a ; (this is specifically for the case that this happens in a procedure declaration)

- **Error 10: No ; at end of block**

  Procedure blocks must end with a ;

- **Error 11: := missing in statement**

  Assignment is missing a ":="

- **Error 12: Missing identifier after call**

  Something other than an identifier is after call

- **Error 13: Begin must be closed with end**

  Every begin must be closed with an end

- **Error 14: If condition must be followed by then**

  If must be followed by a then after it's condition

- **Error 15: While condition must be followed by do**

  While must be followed by a do after its condition

- **Error 16: Missing identifier after read**

  Read must be followed by an identifier

- **Error 17: Missing identifier after write**

  Write must be followed by an identifer

- **Error 18: Relational operator missing in conditional statement**

  Conditions must contain one of the seven relations

- **Error 19: Left (has not been closed**

  A (must be closed by a corresponding)

- **Error 20: Identifier, (, or number expected**

  In an assignment, we found something other than an identifier, number, or (

- **Error 21: Attempting to use undeclared identifier**

  Identifiers must be declared before they can be used

- **Error 22: := assignment to constant or procedure is not allowed**

  Assignment can only be done on variables

- **Error 23: Call to const or var is meaningless**

  Calls can only be made to procedures

- **Error 24: Read must be followed by a var**

  The identifier after read must be a variable

- **Error 25: Write must be followed by a var**

  The identifier after write must be a variable

- **Error 26: Instruction count has exceeded maximum instructions permitted**

  The maximum instruction count allowed is 500

- **Error 27: Program ends with ; and not end.**

  Case where the last character in the source code is a ; (we are stuck in statement loop forever otherwise)

- **Error 28: Use := instead of =**

  Assignments must be done using ":=" and not "="

- **Error 29: Name is too long**

  Maximum identifier name is 11 characters

- **Error 30: Name does not start with letter**

  Identifiers must start with a letter

- **Error 31: Number is too long**

  Numbers cannot be more than 5 digits long

- **Error 32: : is an invalid symbol**

  Specific case were ":" is encounter with no "=" after it, which makes it an invalid symbol

- **Error 33: %c is an invalid symbol**

  Any symbol encountered at this point is invalid

- **Error 34: Consts, Vars, and Procs must be declared before statements**

  Declarations must be done before statements

- **Error 35: input.txt not found, Compilation process has stopped**

  PL/0 source code must be in a file named "input.txt"

- **Error 36: Identifiers cannot be re declared in the same scope**

  An example of this would be declaring a const named x and then a variable named x in the same scope, which is illegal