PL/0 User's Manual

By: Michael Pfeiffer, Yared Espinosa, and Megan Herrera
Date: 4/16/2017

Table of Contents

1.0 How to Program in PL/0

Every PL/0 program must end with a period. The main block of the program supports constant, variable, and procedure declarations, as well as statements. Whitespace is ignored typically unless you need to differentiate a keyword from an identifier. Comments are also ignored and is defined using /* */ notation. See Figure 1.0.

Figure 1.0

```
var x, y;

/*here is a comment.
Everything in between  the
delimiters are ignored
*/
begin
  x := y + 56
end.
```

1.1 Declarations

1.1.1 Constants

Constants are defined as numbers, any sequence of digits (0-9), and may only be defined once in a program. You cannot assign values once they have been defined. These declarations have the following structure:

1.0 Start with the keyword “const” followed by an identifier.
    -An identifier must begin with a letter (lowercase or uppercase “a” – “Z”) and be followed by 0 or more letters or digits.
2.0 “=”
3.0 number
4.0 “;”

You may declare more than one constant per line. A “,” separates the previous number and the following identifier. See Figure 2.0.

1.1.2 Variables

Variables are defined as identifiers. Unlike a constant, you may assign an identifier to this variable more than once in the program using a statement which is covered later in this document. Here is the basic structure:

1.0 Start with the keyword “var” followed by an identifier

2.0 “;”

Again, you may declare more than one variable per line, but they must be separated by a “,”. See Figure 2.0.

1.1.3 Procedures

Procedures are like functions within the main program. The structure looks like this:

1.0 Start with the keyword “procedure” followed by an identifier
2.0 “;”
3.0 Followed by a block
-block supports constant, variable, and procedure declarations, as well as statements.
4.0”;”

It is possible to have nested procedures. Each procedure will use variables that are in the scope of itself or in blocks above it.

For example, if we have a statement that assigns an identifier to a variable in the first procedure declaration, and another statement that assigns for the same variable in the second procedure declaration, we will use the second assignment for the second procedure since it was the most recent statement.

The scope is different for the first procedure. It will use the first statement since that is within its’ scope.

See Figure 2.0 for an example

Figure 2.0

```
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;
                end;
        begin
                z:=2;
                u:=z+w;
        end;
begin
        x:=1; y:=2; z:=3; v:=4; w:=5;
        x:=v+w;
end.
```

1.2 Factor

Factor is defined in three possible ways.

Definition 1: identifier
Definition 2: number
Definition 3:

1.0 Starts with a "("
2.0 followed by expression
3.0 followed by ")"

This will be useful when trying to build an expression.

1.3 Term

Term must always begin with a factor. It is possible to multiply or divide by another factor, which may be repeated 0 or more times. This is useful when trying to create an equation in Pl/0. For example,

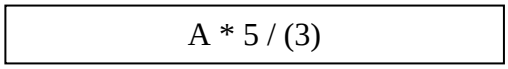


Figure 3.0

"A", "5", and "(3)" are all factors which create a term when adding in the multiply and divide symbols. This can be inserted into an expression which will be inserted into a statement.

1.4 Expression

Expression takes all of the values together and creates a complete equation similar to one in mathematics. The structure is as follows:

1.0 "+" or "-". This is an optional item, but if you want to include positive or negative numbers in your equation then you can do so, but only to the first term in an expression.

2.0 term

3.0 "+" or "-"

4.0 term

Structure number 3.0 and 4.0 may be repeated zero or more times. Below is an example of an expression. This includes the optional unary operator.

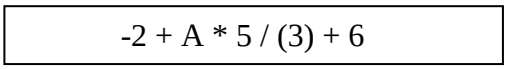


Figure 4.0

1.5 Condition

A condition is useful when you need to check if an answer is true or not. PL/0 includes relational operators you can use to make a comparison between two expressions. Condition does have two definitions.

Definition 1:

Starts off with the keyword “odd” This is used to determine if a number is even or odd, followed by an expression. It will return true if the expression is odd and false if it is not.

Definition 2:

1.0 expression
2.0 relational operator
3.0 expression

Here is a list of valid relational operators

1. “=” are the expressions equal to each other
2. “!=” are the expressions not equal to each other
3. “<” is the first expression in condition less than the second
4. “<=”is the first expression in condition less or equal to the second
5. “>=”is the first expression in condition greater than or equal to the second
6. “>”is the first expression in condition greater than the second

1.6 Statements

Here is where the magic happens. Statements can make it possible to program any kind of problem.

1.6.0

1.0 The first definition must start with an identifier
2.0 “:=” which sets the identifier equal to some expression

3.0 expression

1.6.1 call

1.0 “call” which is a keyword that executes a procedure.
Behind the scenes, the compiler will set up a whole new block on the stack. The stack is necessary for nested procedures to function properly in the PL/0 language.

2.0 identifier

1.6.2 begin

1.0 “begin” keyword which marks the beginning of a block of code

2.0 statement

3.0 “;”

4.0 statement

5.0 “end” keyword which marks the end of that block of code

1.6.3 if-then-else

This is useful if you only want to execute something when a value is in a specific condition. If the condition returns true then the statement gets executed, otherwise it jumps to the next line of code or skips to the else and executes the statement after else. Structure numbers 5.0 and 6.0 are optional.

1.0 “if”

2.0 condition

3.0 “then”

4.0 statement

5.0 “else”

6.0 statement

1.6.4 while-do

This is similar to an if-statement except that the statement will get executed as many times as the condition stays true. Typically you will change the values in the condition after each statement execution; otherwise you will get an infinite loop.

1.0 “while”
2.0 condition

3.0 “do”

4.0 statement

1.6.5 read

1.0 “read”

2.0 identifier

The input that is given by the user will be assigned to the identifier specified.

1.6.6 write

1.0 “write”

2.0 identifier

For this statement, the identifier specified will be written to the user on the screen. On the following page Figure 5.0 shows every possible statement definition.

*Note: It is possible to have a statement defined as nothing.

Figure 5.0

```
/* factorial program */
/*keywords are highlighted in red*/

var fact, i, z;
begin

  read fact;                                    /*reads user's input*/
  i := fact;

  while i >= 2 do
  begin
        fact := fact * (i - 1);
        i := i - 1;
  end;

  write fact;                                   /*writes to the screen*/

  if odd (fact - 1) then                        /*if-then-else statement*/
        z := 3;
else
  write z;

end.
```

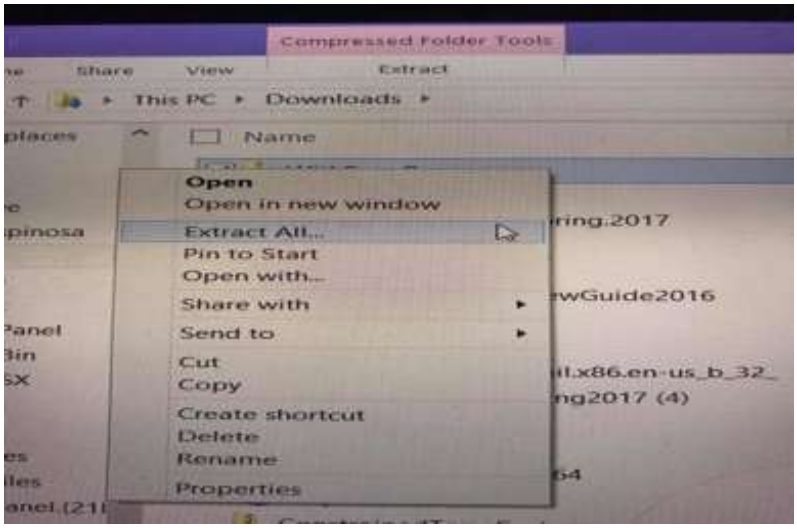
2.0 How to compile and run the PL/0 compiler

The PL/0 Compiler has been created to be as user/programmer friendly as possible, so no-need to worry if your experience programming is limited. The Compiler program you'll be using is a PL/0 compiler and a virtual machine. You, as the programmer, will be able to compile and run a successful program with the next set of helpful instructions.

1.0.Pre-sets:

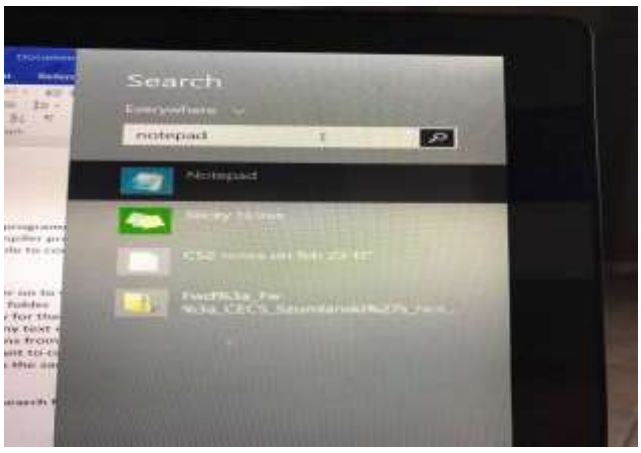
1.1.Download file PL/0 Compiler on to your computer

1.2 Extract all files from the zip folder



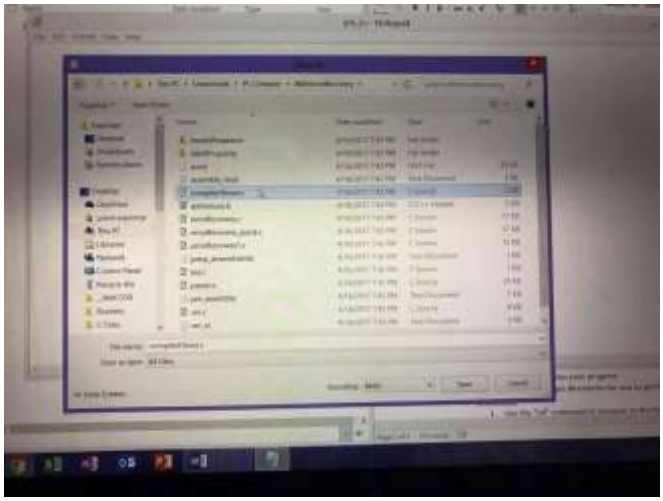
2.0 Getting your program ready

2.1 Open Notepad or any text editor of your choice



2.2 Using the instructions from "How to use the PL/0 language" to create the program you want to compile

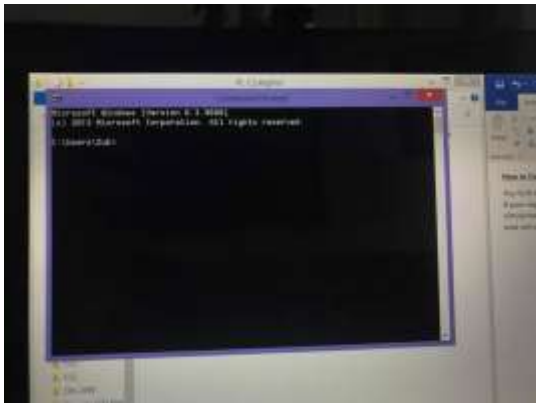
2.3 Save your text file in the same folder as where the “compilerDriver.c” program is saved



3.0 Command Prompt

3.1 On your computer, search for your command prompt or the terminal of your choice

3.2 Open such terminal



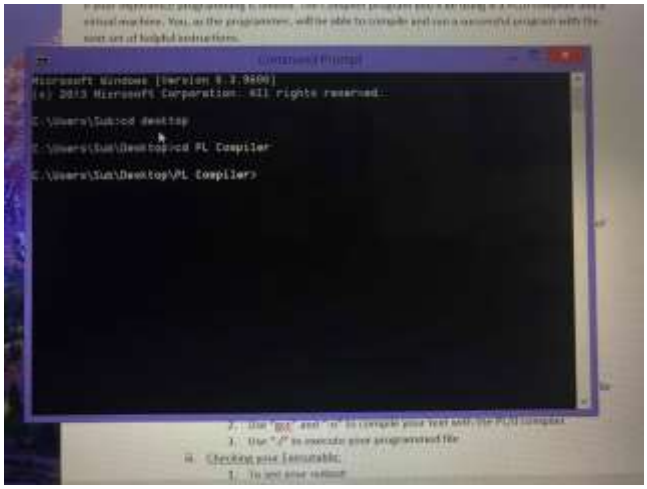
3.3 Compiling:

3.3.1 Basic Need-to-know Terminal Commands

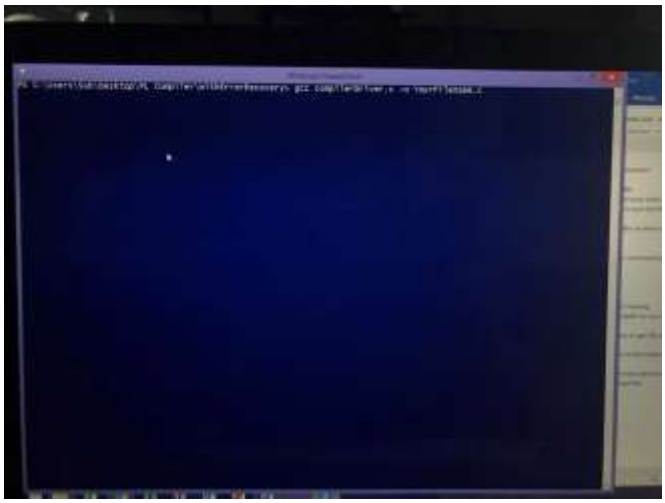
1. gcc : will get the PL/0 Compiler running
2. -o : gets the file you want compile to run with the PL/0 Compiler
3. ./ : Executes your program
4. cd : changes directories for you to get file you need

3.3.2 In Terminal:

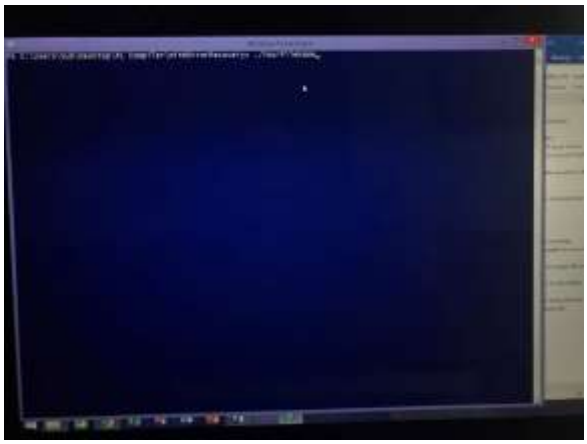
1. Use the “cd” command to navigate to the folder you saved your text file at



2. Use “gcc” and “-o” to compile your text with the PL/0 compiler

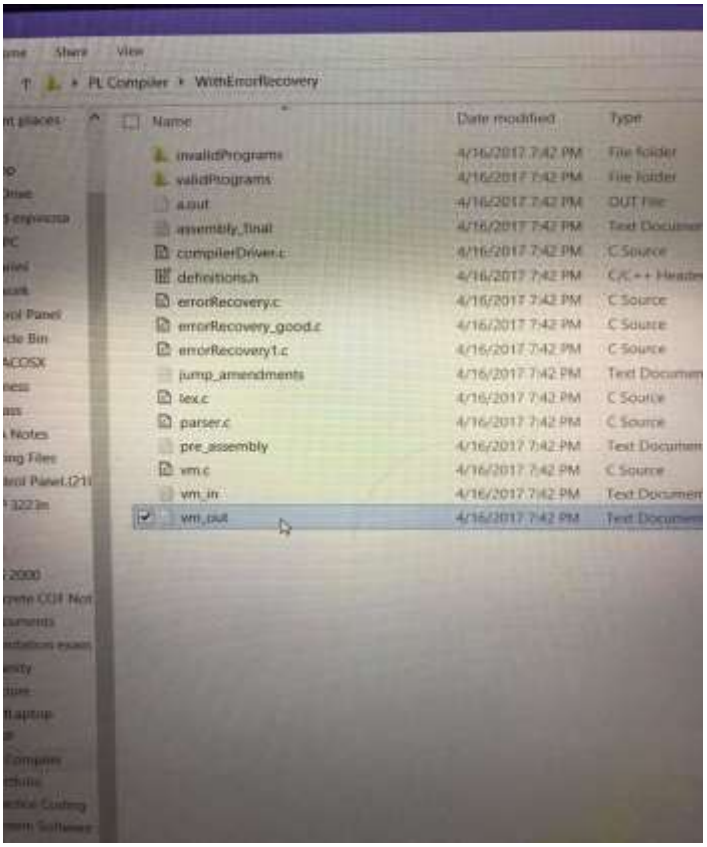


3. Use "./" to execute your programmed file



3.3.3 Checking your Executable:

1. Open the file "vm_out" to see the output of program



2. Open the file "vm-in" to see the input used

3.0 How to use the PL/0 once the compiler is running

Once the compiler starts running with the input file that contains PL/0 code you are able to look at a few things. If you would like to look at the assembly code you can do two things:

1: When you run the executable you can follow the input filename (PL/0 code text file) with *–a* and press enter on the keyboard. This will write the generated assembly code to the screen.

```
./a.out test1.txt - a
```

2: You can also use the *cat* command followed by *jump_amendments.txt* (This is the line numbers where we want to jump to on a JPC instruction), pre-*assembly.txt* (this contains the assembly code before the jump on condition is dealt with), or *assembly_final.txt* (This has the final correct assembly code. Please refer to the Tables section at the end of the document for more information.

```
cat jump_amendments.txt
```

```
cat preassembly.txt
```

```
cat assembly_final.txt
```

If you would like to see what the lexical graphical analyzer interprets you can use the –l flag after you run the executable. This will print out the lex table, followed by the lex list, and the symbolic version of lex list. Please refer to the table section at the end of the document for a comprehensive lex list.

```
./a.out test1.txt - l
```

If you would like to see how the virtual machine interprets the assembly instructions given to it you can use the flag –v after you run the executable. It will be in the format of Line number, opcode, R, L and M. Then it will show you step by step the executed assembly with the program counter, base pointer, stack pointer, and the stack itself. For further information refer to the Table section at the end of the document.

4.0 Tables

Instruction Set Architecture Pseudo Code

01 – **LIT R, 0, M** R[i] ← **M;**

02 – **RTN 0, 0, 0**
```
sp ← bp - 1;
bp ← stack[sp + 3];
pc ← stack[sp + 4];
```

03 – **LOD R, L, M** R[i] ← stack[base(**L, bp**) + **M**];

04 – **STO R, L, M** stack[base(**L, bp**) + **M]** ← R[i];

05 - **CAL 0, L, M**
```
stack[sp + 1] ← 0;              /* space to return value
stack[sp + 2] ← base(L, bp);    /* static link (SL)
stack[sp + 3] ← bp;             /* dynamic link (DL)
stack[sp + 4] ← pc;             /* return address (RA)
bp ← sp + 1;
pc ← M;
```

06 – **INC 0, 0, M** sp ← sp + **M**;

07 – **JMP 0, 0, M** pc ← **M**;

08 – **JPC R, 0, M**
```
if R[i] == 0 then
{
        pc ← M;
}
```

09 – **SIO R, 0, 1** print(R[i]);

10 - **SIO R, 0, 2** read(R[i]);

11 – **SIO R, 0, 3** **Set Halt flag to one;**

12 - **NEG** (R[i] ← -R[j])
13 - **ADD** (R[i] ← R[j] + R[k])
14 - **SUB** (R[i] ← R[j] - R[k])
15 - **MUL** (R[i] ← R[j] * R[k])
16 - **DIV** (R[i] ← R[j] / R[k])
17 - **ODD** (R[i] ← R[i] mod 2) or ord(odd(R[i]))
18 - **MOD** (R[i] ← R[j] mod R[k])
19 - **EQL** (R[i] ← R[j] = = R[k])
20 - **NEQ** (R[i] ← R[j] != R[k])
21 - **LSS** (R[i] ← R[j] < R[k])
22 - **LEQ** (R[i] ← R[j] <= R[k])
23 - **GTR** (R[i] ← R[j] > R[k])

Lexical Conventions for PL/0:

A numerical value is assigned to each token (internal representation) as follows:

nulsym = 1,
identsym = 2,
numbersym = 3,
plussym = 4,
minussym = 5,
multsym = 6,
slashsym = 7,
oddsym = 8,
eqlsym = 9,
neqsym = 10,
lessym = 11,
leqsym = 12,
gtrsym = 13,
geqsym = 14,
lparentsym = 15,
rparentsym = 16,
commasym = 17,
semicolonsym = 18,
periodsym = 19,
becomessym = 20,
beginsym = 21,
endsym = 22,
ifsym = 23,
thensym = 24,
whilesym = 25,
dosym = 26,
callsym = 27,
constsym = 28,
varsym = 29,
procsym = 30,
writesym = 31,
readsym = 32,
elsesym = 33.