PL/0 User's Manual

Authors:

Nathaniel Bates

Michael Benton

Andres Posadas

Table of Contents

A. Introduction to the PL/0 Programming Language

What is PL/O?

PL/0 is a programming language that is simple to understand for someone who has never programmed before. Its concise, straightforward grammar will set a good precedent on the fundamentals of most programming languages. With this manual, you will learn the basic programmatic structures that are commonly found and how to write basic programs.

Sample PL/0 Program

```
const a = 4;
var x, y, z;
  begin
        y := 3;
        x := y + 56
        z : x + a
  end.
```

There is a common structure to every PL/O program. Here is the structure of this program:

- Constants are values that never change and are declared before variables.
- In this program the constant is the identifier ‘a’.
- Variables are declared next and are not assigned values until the main section of the program. The main section starts after the first begin after declared procedures which will be discussed later. A variable’s value can change when an assignment is carried out upon it which will be discussed later.
- Variables declared here would be the identifiers ‘x’, ‘y’, and ‘z’.
- The begin statement means that the program starts performing operations on the next line.
- The end statement means that the previous statement was the end of the operations the program performs. If it is the final end statement, then a period must follow allowing the compiler to know that it is at the very end of the program.

After this program runs y would be equal to 3, x = 59, and z would be equal to 63.

Blocks

Constants

Constants must be declared before any variable or procedure declarations. The format for declaring constants is as follows:

```
const a = 1, b = 2, …, f = 6;
```

A comma must separate each constant declaration and the final declaration must be followed by a semicolon. Constants that are declared before procedures reside in the main function of the program.

Variables
Variables must be declared after constants and before procedures are declared. The format for declaring variables is as follows:
var a, b, c, …, f;
A comma must separate each constant declaration and the final declaration must be followed by a semicolon. Variables that are declared before procedures reside in the main function of the program.

Procedures
We can have procedures, which are basically sub-programs or functions of a program. These will allow us to call a block of code whenever we need to. A sample of a procedure declaration is found below.

```
var x;
procedure A;
  begin
        x := 5;
  end;
  begin
        x:=3;
         call A;
  end.
```

This program will call the procedure declared as "A" after setting x equal to 3. Since the procedure then assigns the value 5 to x, the program will end with x being equal to 5. Note that once the procedure is declared, a begin is listed, the code for the procedure comes right after, then an "end;" follows. Do not confuse this with "end;" with this "end.". When using multiple procedures, make sure to have the "begin" following immediately after it and to make sure to only recursively call programs that are statically enclosed with it. You can experiment with this since the compiler will detect incorrect references to procedures. The call statement will be discussed later in the manual.

Statements

Assignments
Assignment, or giving a value to a variable, is very easy in PL/0. All you have to do is type the name of the variable you wish to assign a value to, followed by ":=", then by whatever you want to put into this variable. You can have any number of mathematical operations after the ":=", as long as your line ends with a semicolon and you close all parenthesis. PL/0 follows the standard order of operations utilized in mathematics.

Example:
var x;
x := (7 + 4) – 10 * 5;
The value of x would be 5. Remember that a semicolon must follow the assignment statement.

Begins and Ends

A block of code that that is written PL/0 must be contained within a "begin" and end statement. Procedures that must have a begin and end statement as well as the main function. The main function is the implied procedure that will start and end the program. Main is not declared in PL/O, it simply starts after the final procedure declaration. The end statement in main will have a period following it, indicating the end of the program and it is required. The end statement after a procedure declaration must have a semicolon following it. Ahead you will learn about conditional if/else statements and while loops. Begin and end statements must be used for blocks of code that are in those programmatic structures. You will see examples of their use next!

If and Else

If-then statements are imperative to create good programs in any language. We can set a specific condition, and must meet that condition to execute a statement as shown below:

```
const y = 4;
var x;
  begin
        x := 0;
        if x = 0 then
          x := y;
  end.
```

In this PL/O program x was initially set equal to 0. The condition was that if x = 0, then execute the statement x := y;. x is now equal to 4. Had we initially set x = 1, then when the program finishes, x would still be equal to 1.
In this PL/O program we will introduce the else statement. If a certain condition is not met, then we can have the program execute a different statement.

```
const y = 4;
var x;
  begin
        x := 1;
        if x = 0 then
          x := y;
        else
          x := 7;
end.
```

Since x does not equal 0, the statement in the else segment will be executed resulting in x being set equal to 7. Another thing that needs to be addressed are multiple statements in if and else segments. Say we have a PL/O program as follows:

```
const y = 4, a = 2;
var x, b;
  begin
        x := 1;
        if x = 0 then
          begin
             x := y;
             b = a
           end;
        else
          x := 7;
  end.
```

When multiple statements are included in an if or else statement, we use a begin and end block to encapsulate the lines of code to be executed based on the condition. This same rule applies to while loops which we will see in the next section. Notice that the last statement before the end statement in the if segment does not have a semicolon following it. This is a standard in the PL/O language.

While and Do

We can have a program execute many lines repeatedly based on a condition.

```
var x;
  begin
        x := 100;
        while x > 0  do
          x :=x - 1;
  end.
```

This program will execute x := x - 1 until x becomes 0. This is very useful because you can repeatedly execute mathematical, I/O, and many other expressions or lines of code. The same conditions apply with multiple statements to be executed. Meaning such as with an if or else statement, we need to use the begin and end statements to execute may lines of code with in a while loop following the same syntactic structure.

Read and Write

There may come a time where you want the user to input a value into your program. This is where the read statement will come in handy. When a read statement is executed, the program will halt and wait for the user to enter a number. That input value will be assigned to a variable that can be used later in the program. Here is an example of a program with a read statement:

```
const a = 10;
var x, y;
```

```
begin
        read x;
        if x >= a then
          y := 5;
        else
          y := 1;
        write y
end.
```

Notice that a new statement was introduced at the end of the program. The write statement just displays the value of the variable to the screen.

Calls

The call statement will just execute a procedure. The syntax for the call statement is shown in the above program under the section titled "Procedures". After the procedure finishes executing, the next line of code after will begin executing.

Putting it All Together

A standard PL/O program must be syntactically correct in order for it to execute from start to finish. The compiler, which will be discussed in the next section, will display an error if one is found in your program. After you create your program, it must be saved as a text file with the .txt extension. Here is a sample of a PL/O program that you may copy and paste into the directory of the compiler. Make sure it has the .txt extension and after reviewing the next section, run the code. Alter the file to find errors and experiment with before writing your very own PL/O program. You are on your way to becoming an official programmer!

B. How to compile and run the PL/0 compiler

Setting up the Environment

Open up the terminal and change the directory to the folder that contains the compiler and PL/O program. The compiler itself consists of 4 files: P-Machine.c, LexicalAnalyzer.c, compiler.c, and header.h. Don't forget that your text file with you program must be in the same folder as well.

Creating an Executable

The Compiler

The compiler is basically the combination of the LexicalAnalyzer.c and compiler.c files. It will parse through your program and generate a string of tokens that can later be read. The string of tokens will allow the parser portion of the compiler to determine where an error may be located. If an error is encountered, it will tell you the type of error it is and it must be fixed in order to run

your program. As your program parses along the string of tokens, it will generate code that the virtual machine will be able to execute.

Code Generation
The main objective of the compiler is to take a high-level language, such as PL/O, and transform it into a set of instructions that a computer, or in this case a virtual machine, would be able to execute. A set of step by step instruction is read individually by the virtual machine, and the virtual machine uses a stack architecture for storing and retrieving values in memory. More information on the stack architecture can easily be found online.

The Virtual Machine
The virtual machine will execute each assembly instruction it encounters and update its memory as needed. When the virtual machine executes you have the option of displaying the assembly code as well as what the machine's memory stack looks like after each instruction is executed.

Running Your Program
Running from a Linux terminal, compile the three included c files as so:

gcc compiler.c LexicalAnalyzer.c P-Machine.c

This will create an executable file called a.out.

In the terminal simply run the executable as so:

./a.out <filename>.txt

This will run your PL/O program. If there is not read or write statements in your program, then you will just see a message that there were not any errors if your program was syntactically correct. If there were any errors, then there will be output saying so. However, there is a lot that goes on behind the scenes. To see what that is you must use directives as follows:

-l will display the list of tokens along with the identifier symbols.
-a will display all the assembly code that is generated.
-v will display the contents of the memory stack

When you run your program, make sure the directives are not separated as the compiler will read it in as a single string.

Example:

./a.out <filename>.txt -l-a-v

This will print everything associated with your PL/O program included in the text file.

C. Appendices

PL/O Grammar

program ::= block "**.**" .
block ::= const-declaration var-declaration statement**.**
constdeclaration ::= ["**const"** ident "**=**" number {"**,**" ident "**=**" number} "**;**"]**.**
var-declaration ::= ["**var**" ident {"**,**" ident} "**;**"]**.**
statement ::= [ident "**:=**" expression
| "**begin**" statement { "**;**" statement } "**end**"
| "**if**" condition "**then**" 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}**.**
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.

Lexical Conventions

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.

Reserved Words: const, var, procedure, call, begin, end, if, then, else, while, do, read,write.

Special Symbols: ‘+’, ‘-‘, ‘*’, ‘/’, ‘**(**‘, ‘**)**’, ‘=’, ’**,**’ , ‘**.’,** ‘ <’, ‘>’, ‘**;**’ , ’:’ .
Identifiers: identsym = letter (letter | digit)*
Numbers: numbersym = $(\text{digit})^{+}$
Invisible Characters: tab, white spaces, newline
Comments denoted by: /* . . . */
*Refer to **Appendix B** for a declaration of the token symbols that may be useful.*