PL/0 User's Guide

Authors: Travis Le and Allysia Freeman

Table of Contents

Index of Figures and Tables

1.0: How to Program with PL/0

Programming in PL/0 consists of three main sections in a basic program. They are constant declarations, variable declarations, procedure declarations, and statements.

```
const x = 5;
var y;
begin
  y := x;
end.
```

Figure 1: A Simple PL/0 Program

Here we can see these things in action. The first line, `const x = 5;` is the constant declaration. The second line, `var y;`, is the variable declaration. The `begin` and `end.` designate the main program, and `y := x;` is a statement contained within this main program. Here is the same code with labels:

Type	Code	Line
Constant declaration	`const x = 5;`	0
Variable declaration	`var y;`	1
Beginning of main program	`begin`	2
Statement	`  y := x;`	3
End of main program	`end.`	4

Table 1: A Simple PL/0 Program

It is important to notice a few things about this code:

- Constants and variables are declared once only. You will see in section 1.5, however, that they can be used more than once inside procedures.
- Every line ends with a semicolon, except begin and end. These are statements, and every statement must be terminated with a semicolon, as seen in section 3.
- The final end has a period after it. Periods are used after the final end to designate the end of the program only.

1.1 Data Types

Though many of these words have not been explained yet, they are built from the most core of PL/0: data types. These are numbers (integers), variables, and constants. Procedures are also included in this, but they are explained in their own section, 1.5.

1.1.1 Integer

The most basic of all data types, they are used for everything. PL/0 supports only integers as numbers, and does not support complex numbers or real numbers. This means all numbers used within PL/0 are within the range of {..., -3, -2, -1, 0, 1, 2, 3, ...}, with no fractions or decimal points.

For mathematical operations, integers react differently than real numbers. In a normal division between real numbers, say ¾, the resulting number is 0.75. However, since only integers are

allowed, the value becomes 0. In integer operations, the result truncates all numbers after the decimal place. It is important to remember this for all mathematical operations in PL/0.

1.1.2 Variable (local and global)

A variable is a position on the stack (see section 5.1.1) with a designated name. In more simple terms, it is a space in the program which can hold an integer, whose location is found by calling its name.

We declare a variable by using the syntax `var name;` where `var` designates variable and `name;` designates the name of your variable. These names cannot be duplicated, must consist of only lowercase letters and numbers, and must begin with a letter. A variable name cannot begin with a number. It is also important to remember your semicolon after this declaration.

You may declare variables only once. You cannot write `var x; var y;`. To facilitate this, you may use an unlimited number of commas between names, in the format of `var x,y,…;`.

```
const z = 2;
var w,x,y;
begin
  read w;
  x := 3;
  y := 9;
end.
```

Table 2: Correct Use of Data Types

Here we can see the correct implementation of variables. Ignore all lines except `var w,x,y;` for now.

1.1.3 Constant

A constant is similar to a variable, except that the value it holds cannot be changed, also known as a static variable. It uses the same name and usage conventions as variables, but the declaration is different.

Since a constant needs a value, we can declare a constant using the syntax `const z = 2;`. You can once again use commas between the declarations, in the format `const z = 2, a = 6,…;`. If you look at the sample of code given in the previous section, you can see a correct implementation of a constant.

1.2 Operators

Operators are important for any variables in PL/0. They are how you compare, perform mathematical operations on, or assign values to variables. Expressions will be explained in the following section, so for now, simply become familiar with the concepts introduced in this section.

1.2.1 Assignment

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.

1.2.2 Relational Operators

Relational operators are comparisons between two numbers. This is important for some code you'll see later.

They consist of the following symbols:

Symbol	What it does
Odd	tests if an expression is odd
=	constant definition or check is two expressions are equal
<>	test that two expressions do not equal eachother
<	tests that left expression is less than right expression
<=	tests that left expression is less than or equal to right expression
>	tests that left expression is greater than right expression
>=	tests that left expression is greater than or equal to right expression

Table 3: Relational Operators

Except for odd, they compare the expression on the left of the symbol with the expression to the right of the symbol. Odd simply tests the expression to the right of itself.

1.2.3 Mathematical Operators

These are what you should already know and love: math. All basic operations are supported (adding, subtracting, multiplying, and dividing).

To use use them, you simply enter a number or expression, then the operator (for example, "+"), followed by the number or expression you wish to use for the operation.

The basic order of operations is followed, and parenthesis are included.

1.3 Expressions

Expressions are defined as a series of numbers, variables, and mathematical operators, in appropriate order. Put more simply, they are mathematical expressions.

For example, “3+(4-2)” is an expression.

An example using variables: “a+b-c*(60-5+d)”

It is important to note that PL/0 does not understand the convention that “5(6+1)” is equivalent to “5*(6+1)”, and will error if it sees this used. You must include an operator between every variable and number.

For a more exact definition of an expression, see section 3.

1.4 Statements

A statement is what PL/0 mainly deals with. It is the bulk of the code, and encompasses a large majority of all possible practices.

1.4.1 Assignments

Assignment, though previously explained, is a statement. Here we can see an example with the variable w being assigned the value of 0.

```
var w;
begin
  w := 0;
end.
```

Figure 2: Assignment in PL/0

1.4.2 Read and Write

Reading and writing is important if you want the user's input for the program. If you want the user to enter a value, you can use "read", and if you want the user to see the value of a variable or constant, you can use "write".

```
var x,w;
begin
  x := 99;
  w := 32;
  read x;
  read w;
  write x;
  write w;
end.
```

Figure 3: Read and Write in PL/0

Read and write support only one variable or constant name before the semicolon.

1.4.3 While Do

While do statements are loops with a condition attached. They are declared using the conventions shown below.

For while do, you must present a condition using conditional operators. If this is evaluated as false, the program will end the loop.

```
var w;
begin
  w := 0;
  while w < 5 do w := w + 1;
end.
```

Figure 4: While,Do in PL/0

Anything contained after the do will repeat until the condition is proven false. As with all loops, there is a danger of looping forever if implemented poorly.

```
var x,w;
begin
  w := 0;
  while w < 5 do
  begin
    w := w * 2;
    x := x + 1;
  end;
end.
```

Figure 5: While,Do with Begin,End in PL/0

If you wish to execute more than one statement after the do, you must encompass them in a begin end statement.

1.4.4 If, Then, Else

These operate similarly to while do, except that they do not loop, and only run once.

```
var w;
begin
  if w > 5 then w := w * 2;
end.
```

Figure 6: If,Then in PL/0

There must be a condition after “if”, followed by “then”. The statement after the then is executed if the condition if true, and skipped if not true.

```
var x,w;
begin
  if w > 5 then
  begin
    w := w * 2;
    x := 1;
  end;
  else w := 0;
end.
```

Figure 7: If,Then with Begin,End in PL/0

If you wish to execute more than one statement after the then, you must encompass them in a begin end statement.

If you wish to do some statement if the condition is evaluated as false, you can enter the line “else” followed by the statement you wish to execute. This follows the same conventions as before, where a single line is fine, but multiple lines need a begin end.

1.5 Procedures

1.5.1 Declaration

To declare procedures, you must first designate the name. This is done using `procedure name;` followed by the code that you want your procedure to contain.

This code is encompassed by a begin and end. Notice that the end is followed by a semicolon instead of a period, as seen previously. This is because whatever is contained within a procedure is a series of statements, and not the entire program, like you would expect with main.

It is important to note that procedures can contain local variables and constants. These are declared in the same way that they were in main, except that they must come after you declare the procedure's name. These variables and constants cannot be used in main, and exist only within the procedure. Main cannot see that these variables and constants exist.

```
procedure mult;
const j = 1;
var i;
begin
  read i;
  w := w * i * j;
end;
```

Figure 8: Procedure Declaration in PL/0

1.5.2 Calling

To call a procedure, simply use the statement `call procedurename;` . This can be used anywhere in the code, except for nested procedures, which you will see in section 1.5.4.

Procedures can call themselves (see the next section), and main can call any procedure at the lowest level. "Lowest level" designates that any procedures of, for example, level 2, cannot be called by main.

1.5.3 Recursion

When a procedure calls itself, it practices what is called recursion.

In recursion, you need two main things: a procedure to call and a return condition. This means that a procedure calls itself a limited number of times, and returns each time after a condition is met. If there is no condition to be met, it is likely that the code would run infinitely or run out of memory.

It can be described as setting building blocks upon one another. The bottom block is main, and each new block is a procedure call. You call the procedure once, and a new block is added. The procedure does something, then calls itself again. A new block is added, it does its code, and a

new block is added. This continues until the condition to return is met- which is when you start removing blocks. You then keep removing blocks until you're back to main.

```
procedure add;
begin
  w := w + 1;
  if w < 15 then call add;
end;
```

Figure 9: Recursive Procedure in PL/0

This procedure calls itself if w, a variable from main, is less than 15. In this instance, the procedure adds one to w until it becomes larger than 14. The procedure to run is add, and the return condition is the testing of the variable w, so it is an adequate recursive procedure.

It is also worthy to note that you should do something within the recursive procedure to change something. If you do not change anything, then your procedure will never return, and keep calling itself until it runs out of memory.

1.5.4 Nested Procedures

Nested procedures are extremely similar to normal procedures, except that they operate differently when being called. Here you can see the proper declaration of three nested procedures:

```
procedure highest;
  procedure middle;
  var a;
    procedure lowest;
    const c = 9;
      begin
        w := c;
      end;
  begin
    read a;
    w := 9;
    call lowest;
  end;
begin
  w := 7;
  call middle;
end;
```

Figure 10: Nested Procedure Declaration in PL/0

They work just as normal procedures do, where you declare the name, then the variables or constants you wish to use. However, the difference is that the procedure declaration isn't done when a new procedure is declared, therefore "nesting" it.

This is used in many ways, and its most important property is that the only procedure that can be called by any other part of the program is the one of the lowest level; "highest" in this case. All other procedures, since they are nested into "highest", can be called by highest only.

2.0: Build, Compile, and Execute PL/0 programs

The Compiler is composed of the following files:

1. `compiler.c`
   The main driver of the program, this handles all the other files and executes them as needed.
2. `header.h`
   This holds all the declarations used often throughout all files. It also includes a section for redefining filenames. If you don't know how to use `#define` in C programming, please don't change these names.
3. `scanner.h`
   This scans your input file and generates a list of lexemes (kinds of symbols).
4. `parser.h`
   This takes the result from `scanner.h` and analyzes it for errors. It then generates machine code for the given input file. If it is found to be correct, it sends the generated machine code to `vm.h`
5. `vm.h`
   This takes the machine code from `parser.h` and executes it.
6. `input.pl0`
   This is your PL/0 code. It must be named this exactly, including the file extension .pl0. If you would like to change the input filename, edit *nameCode* in `header.h`, making sure to include the file's extension. Complex file extensions are not supported, and it is strongly suggested to use either a plaintext or simple code editor to write your PL/0 code.

With the following output files, which are all used/generated internally:

1. `cleaninput.txt`
   This is your PL/0 cleaned of all comments.
2. `lexemetable.txt`
   This is a user friendly version of the lexeme list. See section 4 for a list all possible tokens and their meanings.
3. `lexemelist.txt`
   The not so user friendly version of the lexeme table. This is used internally for analysis and machine code generation.
4. `mcode.txt`
   The machine code that was generated. See section 5.1.2 for more information on machine codes.
5. `stacktrace.txt`
   Includes the interpreted machine code, which is used internally, and a stack trace of when the code was executed. See section 5 for an explanation of the stack.

2.1 Building the Compiler

It is assumed you are using a linuix-based terminal for all following instructions.

To build this compiler:

1. Make sure all files listed above are in the same directory, and named exactly as shown.
2. Open your terminal.
3. Run the command "gcc compiler.c –o compile"

You should now have a file called "compiler.o" in your directory, and are ready to execute.

2.2 Executing the Compiler

If you have closed your terminal, please re-open it in order to execute the compiler.

In order to execute the compiler, run the command "./compile"

Alternatively, the following flags are available for your use:

- -l which instructs the compiler to display the lexeme list to the screen. This is the internal interpretation of the input file.
- -a which instructs the compiler to display the generated machine code to the screen. The meaning of these codes and their properties are listed in section five.
- -v which instructs the compiler to display a stack trace for the execution of the input file. The stack is also explained in section five.

To use them, simply enter the flag after the execute command, making sure to put space between all flags. For example, in order to see the lexeme list, enter "./compile -l". To see both the lexeme list and the stack trace, enter "./compile -l -v", and in order to see all of them, enter "./compile -l -a -v". They can be entered in any order.

Provided your PL/0 program is syntactically correct and can execute properly, the compiler will generate the output files described in the previous subsection.

And, if there are any read or write statements in your code, the console will either prompt you or print a value, respectively. Please enter a reasonable number and then press enter for read statements.

3.0: EBNF Grammar of PL/0

program = block "**.**" **.**
block = const-declaration var-declaration procedure-declaration statement **.**
const-declaration = [“**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
| "**while**" condition "**do**" statement
| "**read**" ident
| "**write**" ident] **.**
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**"|"**c**"|**…** |"**x**"|"**y**"|"**z**" **.**

Table 4: Extended Backus–Naur Form (EBNF) for PL/0

The first word of each section is the name of a “syntactic class.”

This is read using the following rules:

1. **|** means “or, but not both”
2. **[ ]** means an optional item
3. **{ }** means repeat zero or more times
4. Special symbols are enclosed in quote marks
5. A period is used to indicate the end of the definition of a syntactic class

4.0: Tokens and Reserved Words

Symbol	Name	Value	Usage
	nulsym	1	n/a
	identsym	2	used for constant, procedure, and variable names
	numbersym	3	used for numbers
+	plussym	4	adds
-	minussym	5	subtracts
*	multsym	6	multiplies
/	slashsym	7	divides
odd	oddsym	8	test if an expression is odd
=	eqlsym	9	constant definition or check is two expressions are equal
<>	neqsym	10	test that two expressions do not equal eachother
<	lessym	11	tests that left expression is less than right expression
<=	leqsym	12	tests that left expression is less than or equal to right expression
>	gtrsym	13	tests that left expression is greater than right expression
>=	geqsym	14	tests that left expression is greater than or equal to right expression
(	lparentsym	15	begin factor
)	rparentsym	16	end factor
,	commasym	17	separates identifiers in declarations
;	semicolonsym	18	ends a statement
.	periodsym	19	ends the program
:=	becomessym	20	assigns a value to a variable
begin	beginsym	21	begins a block of statements

end	endsym	22	ends a block of statements
if	ifsym	23	begins if statement, followed by condition
then	thensym	24	follows then, followed by statement
while	whilesym	25	begins while loop, followed by condition
do	dosym	26	follows while, followed by statement
call	callsym	27	calls a procedure
const	constsym	28	begins a constant declaration
var	varsym	29	begins a variable declaration
procedure	procsym	30	begins a procedure declaration
write	writesym	31	prints a value to the screen
read	readsym	32	asks the user to enter a value
else	elsesym	33	may follow if statements

Table 5: Tokens and Reserved Words of PL/0

5.0: Machine Code Instructions in PL/0

5.1.1 The Stack

This will be a basic explanation of the stack, which appears both in the output file `stacktrace.txt` and if you enter the flag "`-v`" when executing the compiler. A stack is utilized internally in order to properly execute the input code.

A **stack** is defined as a set of data sections in which data can be stored during execution. Everything is stored in it, including variable values and various results of mathematical operations.

The **lexical level** (referred to as "level") is the number of activation records minus 1, since it starts at zero. The first is level 0, the second is level 1, and so on.

An **activation record**, or **AR**, is defined as a section of the stack used for the main code of a block. Each procedure gains its own activation record when called, and though they cannot be shared, they can be accessed by any level, at any level.

```
                    pc  bp  sp  stack
10  SIO  0   1      11   1   7  0 0 0 0 0 4 2
11  STO  0   4      12   1   6  0 0 0 0 2 4
```

Figure 11: An Example of a Stack

We are going to focus on the section of numbers at the end, the representation of the stack, after the 7 and 6 for both lines respectively. However, here are some quick definitions for the other numbers, by position:

1. Line of code being executed
2. The operation being executed, see section 5.1.2 for meanings
3. The level for the operation being executed
4. The value or line for the operation being executed
5. The line number of the next line to be executed, or *program counter* (pc)
6. The position of the bottom of the topmost activation record, or *base pointer* (bp)
7. The number of elements in all activation records, or *stack pointer* (sp)

The first four spaces (shown above as three zeros) are reserved, and their meanings are:

1. Reserved.
2. The *static link*, or the positon of the activation record that is its most senior parent. Unless a special case, this will most likely have a value of one in any record after the first.
3. The *dynamic link*, or the position of the previous activation record.
4. The *return address*. This is the line number of the code to execute after a procedure is returned.

Every position after these zeroes are used for variables, and since constants cannot be changed and procedure names do not hold values, they do not get a position (though they *are* acknowledged during execution). Variables are stored in the activation record where they were declared. We see that there are two positions after the zeros in the figure, and also a mysterious 2.

In the first line we see the result of a read, the 2, and in the second line we see the read's value being stored into whatever variable was asked to be read, position 1. The 4 is already stored in another variable's value, in position 2.

```
                  pc  bp  sp  stack
21  INC  0   5    22  13  17  0 0 0 0 6 9 | 0 1 1 191 2 1 | 0 1 7 111 0
22  SIO  0   1    23  13  18  0 0 0 0 6 9 | 0 1 1 191 2 1 | 0 1 7 111 0 2
```

Figure 12: A Stack with Three Activation Records

Here we can see these things in action. Each "|" denotes a separation between activation records. This is not used internally, and is only displayed in order to make stacks easier to read.

the second record is pointing to the first with its values of "1 1 191" and has two new variables, "2 1". It will execute line number 191 when returned. The third record is pointing to the second with its values of "1 7 111", with one new variable, "0". It will execute line number 111 when returned.

5.1.2 Machine Code Meanings

The machine code instructions are in the form "**operation L M**," where **operation** is the operation code (op code), **L** is the level, and **M** is the address, data value, or mathematical operation.

Here are two definitions to make the following table easier to read:

To **push** means to put a number onto the top of the stack.

To **pop** means to take the number on the top of the stack and remove it or use it for something.

Op Code	Value	Explanation
1	`LIT 0 M`	Push a number, M, onto the stack
2	`OPR 0 0`	Return from a function
	`OPR 0 1`	Multiply the number on the top of the stack by -1
	`OPR 0 2`	Pop the top two numbers on the stack, add them, and push their result
	`OPR 0 3`	Pop the top two numbers on the stack, subtract them, and push their result
	`OPR 0 4`	Pop the top two numbers on the stack, multiply them, and push their result
	`OPR 0 5`	Pop the top two numbers on the stack, divide them, and push their result
	`OPR 0 6`	Test if the final number on the stack is odd. If true, it pushes 1. If false, it pushes 0
	`OPR 0 7`	Not implemented
	`OPR 0 8`	Pop the top two numbers on the stack, and test if they are equal. If true, it pushes 1. If false, it pushes 0.
	`OPR 0 9`	Pop the top two numbers on the stack, and test if they are not equal. If true, it pushes 1. If false, it pushes 0.

	OPR 0 10	Pop the top two numbers on the stack, and test if the first is less than the second. If true, it pushes 1. If false, it pushes 0.
	OPR 0 11	Pop the top two numbers on the stack, and test if the first is less than or equal to the second. If true, it pushes 1. If false, it pushes 0.
	OPR 0 12	Pop the top two numbers on the stack, and test if the first is greater than the second. If true, it pushes 1. If false, it pushes 0.
	OPR 0 13	Pop the top two numbers on the stack, and test if the first is greater than or equal to the second. If true, it pushes 1. If false, it pushes 0.
3	LOD L M	Push the value contained L levels down, at position M
4	STO L M	Store the value on the top of the stack L levels down, at position M
5	CAL L M	Call a procedure, and return to line M when it returns.
6	INC 0 M	Push the value M onto the top of the stack
7	JMP 0 M	Jump to line M
8	JPC 0 M	If the top of the stack is equal to zero, jump to line M
9	SIO 0 0	Display the top of the stack to the screen
	SIO 0 1	Read a value from the screen
	SIO 0 2	Halt

Table 6: Machine Code Instructions for PL/0

6.0: Error Codes

Error	Error Message	Explanation
0	Program is syntactically correct.	n/a
1	Invalid file input	Make sure the input file is named correctly or edit header.h's filename definitions
2	Use "=" not ":="	Don't use := when you're not assigning a value to a variable.
3	Use ":=" not "="	Always use := when assigning a value to a variable.
4	"=" expected after const declaration	When declaring a constant, it must be given a value.
5	Number expected after "=" with const	Constant declarations must be followed by =.
6	"then" expected after "if"	"then" must follow after an "if" statement.
7	"do" expected after "while"	"do" must follow after a "while" statement.
8	const, var, and procedure must be followed by an identifier	You must give a name to every constant, varable, and procedure when declaring it.
9	":=" expected after identifier	Missing ":=" after a variable name.
10	Ident expected after "call"	The proper syntax is "call name;"
11	Relational operator expected	Missing a relational test.
12	Assignment to constants and procedures not allowed	You cannot assign a value to procedure names or constants. They are static.
13	Semicolon needed between statements	Missing a semicolon after a statement.
14	Cannot begin statement with this symbol	Check your program's syntax.
15	Undeclared variable detected	Declare the variable before using it.
16	Unclosed parenthesis detected	All parenthesis must be closed.
17	Invalid operator	See operators section for correct implementation.

18	Invalid symbol	The symbol is not supported by PL/0.
19	";" expected	Missed a “;” in code
20	Number too long	A number is too long. Make it shorter.
21	Identifier too long	The name of a variable is too long. Make it shorter.
22	Generated code too long	Your program is too complicated for the compiler. Shorten or simplify your code.
23	Compiler has run out of memory	n/a
24	Period expected.	A period must be at the end of the program.
25	Var or const detected more than once	You can only declare variables and constants once per level. Merge the declarations into one.
26	Ident name declared twice	Rename the variable or constant.

Table 7: Error Codes