Complete PL/0 User Guide

A programming language is a set of commands written logically to write software applications following the syntax for a specific programming language. The computer does not do anything by itself. It only executes the logical statements written by a programmer to run any particular application.This user guide is specifically for PL/0 programming language which discusses, basic structure, syntax, virtual machine, compilation, and execution.

Here is the simple PL/0 program which has three main sections. They are constant declarations and variable declarations, procedure declarations, and statements.

```
const y = 20;
var x;
begin
        read y;
        x := y + 56;
        write x;
end.
```

The above code is a PL/0 program. Following the syntax or the grammar of this programming language, there are rules to declare constants, variables, and the main program. If we need constant values for our program, it has to be said first before declaring any variables, meaning that we can't initialize constants after using variables. We have to follow the syntax or grammar to write a program.

So, let's trace down the simple PL/0 program above. Starting from the first line of code, we have declared constant y to 20. We can notice that the word "const" is the keyword reserved in PL/0 programing language to let the computer know that we are declaring constants after this keyword. We can't use keyword "constant" or some other keyword as we wish to declare constants because these are not the reserve words for PL/0 programming language. We have to use reserved keywords and follow the grammar so that we can execute the program without any error. After this "const" keyword, we wrote down the name of the constant variable so that we can use this variable anywhere in the program. In the sample code above we have named "y" to declare the name of the constant variable, but we could have also called anything as long as the name of the variable does not start with numbers and it is not the reserved word.

We have a variable declaration in the second line of the PL/0 code. Notice that we have used the keyword "var" to declare the variable and we have named the "x." We could have also named the variable as we wish as long as the name of the variable does not start with a number and also it is not a reserved keyword.

After declaring all constants, variables, we can now begin the main program to do computations using the above constants and variables.We don't have to have constants or variables declared if

we don't need for the program, but we have to have primary function or program. Otherwise, the PL/0 program does not compile. The way we identify if the program contains the main function is by the keywords "begin" and "end." The keyword "begin" designates the start of the main function, and keyword "end" designates the end of the main function as we have in the above simple PL/0 program. We can also notice that every line of code follows semicolon except the keyword "begin" and "end." The last thing we can see is the presence of period "." after the keyword "end," which indicates the end of the PL/0 program.

Integer Data Type:

The PL/0 programming languages only support integer data types meaning that it does not support complex number or real and fractional numbers. As we have seen the declaration of constants and variables by using the keywords "var." and "const", all these support integer data types. These numbers are with within a range of {…-3,-2,-1, 0, 1, 2, 3,…..} with no fraction or decimal points. One important thing if we do any computations from addition, subtraction to multiplication and division, PL/0 will truncate all the numbers after decimal if there are decimal numbers. For example, if we do division computation like 3 /2, the mathematical answer would be in fractional number which would be 1.5. But in the PL/0 programming world, it will truncate numbers after decimal, and the result would be 1 only and this is true for all the mathematical computation

Variables and Constants
Lets look the snippet of the following PL/0 code:

```
const a1 = 5, a2 = 8;
var x1, y1,;
begin
        x1 := a1;
        y1 := a2;
end.
```

As we have discussed above the implementation of declaring constants and variables, we have to follow the order by first declaring constants followed by stating variables. We can also declare more than one constant or variable if we need more as we have shown above. We have to follow the syntax rules to declare multiple variables. The rule is that if need multiple variable declarations, we first write keyword "var" followed the variable name. If we need more variables , we will insert a comma after first variable declaration and declare the second variable. One important thing to note that we can't have duplicate variables or constant and also the name of the variable or constant has to start with lowercase letters only. We have correct implementation of the PL/0 code above.

Assignment and Operators:

We have different mathematical operators used in the PL/0 program in the expression. When we are initializing constants, we use this assignment operator "=" in the PL/0 program. On the other hand, if we are initializing variables, we use this assignment operator ":=" followed by any numbers, mathematical operators, and semicolon.

Relational Operators

Relational operators are comparisons between two numbers. They consist of the following symbols

Symbol	Feature
<	tests that left expression is less than right expression
<=	tests that left expression is less than or equal to right expression
>	test that left expression is greater than right expression
>=	test that left expression is greater than or equal to right expression
=	check if two expressions are equal
<>	check if two expressions are not equal
Odd	test if an expression is odd

Mathematical Operators:

Mathematical operations like addition, subtraction, multiplication, division are supported by PL/0 programming language. We can use something like this.
value1 = (2 + 3) / (5 * 2);
Assuming that the variable "value1" is already have been declared at the beginning of the program. This expression would the same way in PL/0 program as it would in Algebra

Expressions:

Mathematical Expression has been defined as a series of numbers, variables, and mathematical operators in the appropriate order. For example,
Using only numbers and operators "5 * (3+2) " is an expression.
Using only variables and operators "(a + b) * (a – b) is an expression
Using all together " (2*a + b) / (3 * b -2) is an expression.

One important thing to note that PL/0 programming language does not understand the convention that "10(a + b)" and "2a + 3b" is equivalent to "10 *(a + b)" and "2 *a + 3 * b" and will not result compiling the program. We have to include an operator between every variable and number.

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.

Assignment Statements

Let's take a look at the following snippet of the PL/0 code here

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

As we can see that variable "x" is declared in the beginning before using that in the main program. We have an assignment statement in the main program. We can use any number of assignment statements in the main program.

Read and Write Statements

Let's look at the following snippet of PL/0 code

```
var a, b;
begin
        a  :=  5;
        b  :=  10;
        read a;
        read b;
        write a;
        write b;
end.
```

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

If, Then, Else Statements

Let's look the following PL/0 code snippet

```
var x;
begin
        if x > 2 then x = x * 2;
end.
```

Some we have to do computations only if the only certain condition is true. If the condition is false, it will not execute the statement. We can see the PL/0 code above that we are multiplying "x" by "2" and storing the result in the same variable only if 'x" is greater than 2. If the condition is false, it will not execute if statement.

Let's look at the little complex example PL/0 code snippet

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

If we have more computations and statements in the conditional statement, we can't write everything in a line. We should write in multiple lines to make easier to read the program. We have to include "begin" keyword to indicate the starting of the statements and "end" keyword to indicate the end of the if statement

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

For while loop, we must present a condition using conditional operators. If this is evaluated as false, the program will end the loop
Let's the PL/0 code snippet

```
var x;
begin
        x := 0;
        while x < 10 do x :=  + 1;
end.
```

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.

Let's look the little complicated PL/0 code snippet

```
Var x, y;
begin
        x := 0;
        while x < 10 do
        begin
        x := x *2;
        y : = y + 1;
        end;
end.If we wish to execute more than one statement after the do, we have to put them in a begin end statement
```

Procedures

Procedure Declarations
To declare procedures, we must first designate the name. This is done using procedure name; followed by the code that you want your procedure to contain.
Let's look at the following PL/0 code snippet:

```
Procedure add;
const x = 5;
var y;
begin
        y := 10;
        y := y + x;
end;
```

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.

Calling Procedures
To call a procedure, we simply use the statement "call" procedure name. This can be used anywhere in the code, except for nested procedures. Procedures can call themselves and main can call any procedure at the lowest level.

Recursion Procedures
When a procedure calls itself, it practices what is called recursion.
Let's look at the following PL/0 code snippet:

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

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 we start removing blocks. We then keep removing the blocks until we are back in main.

This procedure calls itself if a, a variable from main, is less than 15. In this instance, the procedure adds one to a 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 we should do something within the recursive procedure to change something. If we do not change anything, then our procedure will never return, and keep calling itself until it runs out of memory.

Nested Procedures

Nested procedures are extremely similar to normal procedures, except that they operate differently when being called. Here we 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;
```

They work just as normal procedures do, where we declare the name, then the variables or constants we 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.

Let's look at the some following PL/0 programs containing errors and see if we can identity errors to fully get the idea as to how program works

```
var z,a, b;
begin
    a := 0;
    b:= 8;
    z:=0;
    while b > 0 do
    begin
        if odd b then
            z := z + a;
        a := 2*a;
        b := b/2;
    end

end;
```

If we are to analyze the code, we could see that keyword "end" of the main function is followed by semicolon which is wrong. The "end" keyword has to be followed by period to be able to compile this program.

Let's look at another PL/0 code.

```
var w,q, w, r, y;

read x;
read r;
read y;

begin
  r := 0;
  q := 1;
  w : 1;

    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.
```

As we trace down the code, we could see at line 10 that instead of using ":=", it has ":". We have to fix the code so that it compiles successfully

Let’s look at another example PL/0 code

```
const a = 1 , c = 3;
var ;
begin
  b := 2 + a;
  write b;
end.
```

As we can see at line 2, we are missing identifier after keyword “var”. Keywords “const”, “var”, and “procedure” has to be followed by an identifier.

Lets look at another code

```
const n = ;
var i,h;
   procedure sub;
     const k = 7;
     var j,h;
    begin
      j:=n;
      i:=1;
      h:=k;
    end;
begin
       i:=3;
       h:=0;
       call sub;
end.
```

As we can see at line 1 that constant is declared without a number. Number is expected after operator “=”. The constant has to be initialized to be able to compile this program successfully

Lets look at the last example code

```
var x, w;

begin
 x := 4;
 read w;

     w > x then
       write w;
     if x > w then
      write x;
end.
```

We can easily see at line 7 that we are missing keyword “if” before the using conditional operator. We have to insert “if” keyword to successfully compile this program.

Building a Compiler

It is assumed that you are using Linux-based terminal and also you have GCC compiler already installed in your Linux machine.
1 . Make sure all files are in the same directory.
2 . Open up your terminal
3. Run the command “gcc CompilerDriver.c -o CompilerDriver”
Now you should have a file called “CompilerDriver.0” in you same directory, and you are ready to execute

Executing the Compiler

There are different options available to run this program after it has been compiled.
There has to be at least 2 command lines arguments passed in to execute the program, one being the executable file and the second one being the ".txt" file. For example,
To compile the program
"gcc CompilerDriver.c -0 CompilerDriver"
To run or execute the program
"./CompilerDriver SouceCode.txt"
I have included a sample "SouceCode.txt" file which has PL/0 source code. I have shown with a sample "SouceCode.txt" file which has PL/0 source code. There are other".txt" test files that can be used to test the working of PL/0 compiler

There are also other commands like "-l", "-a", "-v" available to be passed in as a command line argument to print out desired output to the console.
One thing to note is that if the grammar is incorrect, the possible errors will be printed to the console for the errors to be fixed. It does not print out anything to the corresponding output files if the PL/0 grammar is incorrect. Once the PL/0 is correct, the results will be written to the corresponding output files.

Reference:

PL/0 EBNF Grammer

```
program ::= block "." .
block ::= const-declaration var-declaration procedure-declaration statement
.
const-declaration ::= ["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" .
```

Figure 7: PL/0 EBNF Grammar

Complete List of reserved words and tokens

Symbol	Internal Name	Internal Value	Usage
	nulsym	1	reserved
	identsym	2	constant, variable, and procedure identifiers
	numbersym	3	number literals
+	plussym	4	addition in expressions
-	minussym	5	subtraction in expressions
*	multsym	6	multiplication in expressions
/	slashsym	7	division in expressions
odd	oddsym	8	determining if an expression is odd
=	eqlsym	9	constant definitions, checking the equality of two expressions
<>	neqsym	10	checking that two expressions are not equal
<	lessym	11	checking that the left expression is less than the right expression
<=	leqsym	12	checking that the left expression is less than or equal to the right expression
>	gtrsym	13	checking that the left expression is greater than the right expression
>=	geqsym	14	checking that the left expression is greater than or equal to the right expression
(	lparentsym	15	begin a factor
)	rparentsym	16	end a factor
,	commasym	17	separate constant, variable identifiers in their respective declarations
;	semicolomsym	18	end statements
.	periodsym	19	end of program
:=	becomessym	20	variable assignments
begin	beginsym	21	begin a block of statements
end	endsym	22	end a block of statements
if	ifsym	23	begin an if-then statement, followed by a condition
then	thensym	24	part of if-then, followed by a statement
while	whilesym	25	begin while loop, followed by a condition
do	dosym	26	part of while loop, followed by a statement
call	callsym	27	calls a procedure
const	constsym	28	begin constant declarations
int	intsym	29	begin integer declarations
procedure	procsym	30	begin a procedure declaration
out	outsym	31	output the value of an expression
in	insym	32	ask the user to input a value and assign it to a variable
else	elsesym	33	optionally follows if-then statements

PL/0 Instruction set Architecture

All PL/0 instructions are of the form **OP L, M** where **OP** is the op code, **L** is the lexicographical level, and **M** is an address, data, or an ALU operation.

Op Code	Syntax	Description
1	LIT 0, M	Push constant value **(literal) M** onto the stack
2	OPR 0, M	**Operation** to be performed on the data at the top of the stack
	OPR 0, 0	**Return**; used to return to the caller from a procedure.
	OPR 0, 1	**Negation**; pop the stack and return the negative of the value
	OPR 0, 2	**Addition**; pop two values from the stack, add and push the sum
	OPR 0, 3	**Subtraction**; pop two values from the stack, subtract second from first and push the difference
	OPR 0, 4	**Multiplication**; pop two values from the stack, multiply and push the product
	OPR 0, 5	**Division**; pop two values from the stack, divide second by first and push the quotient
	OPR 0, 6	Is **odd**? (divisible by two); pop the stack and push 1 if odd, 0 if even
	OPR 0, 7	**Modulus**; pop two values from the stack, divide second by first and push the remainder
	OPR 0, 8	**Equality**; pop two values from the stack and push 1 if equal, 0 if not
	OPR 0, 9	**Inequality**; pop two values from the stack and push 0 if equal, 1 if not
	OPR 0, 10	**Less than**; pop two values from the stack and push 1 if first is less than second, 0 if not
	OPR 0, 11	**Less than or equal to**; pop two values from the stack and push 1 if first is less than or equal second, 0 if not
	OPR 0, 12	**Greater than**; pop two values from the stack and push 1 if first is greater than second, 0 if not
	OPR 0, 13	**Greater than or equal to**; pop two values from the stack and push 1 if first is greater than or equal second, 0 if not
3	LOD L, M	**Load** value to top of stack from the stack location at offset **M** from **L** lexicographical levels down
4	STO L, M	**Store** value at top of stack in the stack location at offset **M** from **L** lexicographical levels down
5	CAL L, M	**Call** procedure at code index **M**
6	INC 0, M	**Increment** the stack pointer by **M** (allocate **M** locals); by convention, this is used as the first instruction of a procedure and will allocate space for the **Static Link (SL)**, **Dynamic Link (DL)**, and **Return Address (RA)** of an activation record
7	JMP 0, M	**Jump** to instruction **M**
8	JPC 0, M	Pop the top of the stack and **jump** to instruction **M** if it is equal to zero
9	SIO 0, 1	**Start I/O**; pop the top of the stack and output the value
10	SIO 0, 2	**Start I/O**; read input and push it onto the stack