**User Guide For PL/0**

By Philip Rodriguez & Steven Chen

ph644520 & st140537

**Table of Contents**

How to use the PL/0 language ……………………………………………………………………………………………………………1 - 6

Compile and Run …………………………………………………………………………………………………………………………………….7

How to use the PL/0 compiler once it is running ………………….…………………………………………………………..8 - 12

Reference ………………………………………………………………………………………………………………….…………………..13 - 14

**How to use the PL/0 Programming Language**

**The PL/0 Programming Language**

PL/0 is a simple language that is used to teach how compilers work. The language supports the use of constants, variables, and procedures. A PL/0 program contains, in the most basic sense, the following structure:

1. Constant declarations
2. Variable declarations
3. Procedure declarations
4. Statements(s)

All PL/0 programs must end with a period.

**White Space**

In PL/0, whitespace is ignored and thus serves no functional purpose other than code legibility. Anything between comment delimiters, e.g. /\* and \*/, will also be ignored by the lexical analyzer.

**Data in PL/0**

This implementation of PL/0 supports the following for data storage:

* Constants (const)
* Variables (var)

Both constants and variables can hold only integer number values. There is no other type of data supported.

Both constants and variables must be associated with an identifier. An identifier is used to identify the program. In the current implementation, an identifier cannot be more than 11 characters in length. A valid identifier must begin with a letter. It may contain uppercase or lower case letters as well as numbers.

Number literals are used in PL/0 for arithmetic and logical operations, as well as for data assignment. Number literals cannot be more than 5 digits long.

**Constants**

A constant contains an identifier that references a stored number. Constants are immutable (you cannot assign values to them after they have been declared). In the PL/0 programing language, constants are defined after a block. You can define 0 or as many constants as you would like per block. Defining multiple constants can be done by separating the value and the next constant’s identifier with commas. See the example below for this. All constant declarations must end in a semicolon. Constants are declared with the reserved word *const.* Below are examples of how constants can be declared:

const x = 1; /\* single constant declaration \*/

const x = 1, y = 2; /\* multiple constant declarations \*/

After a constant is declared, you will be able to use them throughout the program. They will be converted to the values you assigned upon compilation. Thus, the following two statements are functionally equal assuming we declared constants x and y to be 1 and 2 respectively:

foo := x + y;

foo := 1 + 2;

Constant declaration and assignment must happen at the beginning of a block and before variable declaration. You cannot declare or assign constants via statements. Below is an example of a program with valid constant declaration:

const x = 1, y = 2, z = 3;

var a, b;

begin

write x+y+z;

end.

Notice how the constants are declared before variables are declared, and that constants are also assigned upon declaration as well. You cannot re-assign constants later in the execution of the program, and you must assign them at declaration. Note that you are not required to have any constants. Below is an example of a program that has no constants but is still valid:

var a, b;

begin

a := 5;

b := 4;

write a+b;

end.

See PL/0 EBNF Grammar on page 7 for more on the grammar of a constant declaration.

**Variables**

Unlike constants, variables are mutable. You can declare variables and reassign their values later in program execution as much as you want. Variables must be declared right after constant declarations (constant declarations are not required) and before the statement(s) of the block. Like constants, variables are assigned an identifier that is used as a reference. You can declare 0 or as many variables as you want. Multiple variable declarations are separated by commas. Variable declarations must end with a semicolon. Variables can be declared with the reserved word *var*.

Unlike constants, variables can be assigned values as many times as you want after being declared. However, unlike constants, variables cannot be assigned right when they are being declared. Variables can be assigned values by using the assignment operator :=. Below are examples of how variables can be declared:

var foo; /\* single variable declaration \*/

var foo1, foo2; /\* multiple variable declarations \*/

A variable may be assigned values as seen above (foo := x + y;). Below are more examples of variable assignment statements:

foo := 1 + x;

foo := 2 + y; /\* the foo variable is being defined again \*/

Although literals can only be up to 5 digits in length, and therefore have a maximum value of 99999, variables can hold all integer values that fit in a 32-bit signed 2’s complement number. This means that variables are capable of holding values as low as -231 and as high as 231-1. Below is a simple example program showing proper variable declaration and usage:

var a, b, c;

begin

a := 5;

b := 6;

c := a + b;

write c;

end.

The result of compiling and running the above program is the answer 11.

See PL/0 EBNF Grammar on page 7 for the grammar of a variable declaration.

**Procedures**

Procedures can be viewed as a sub-program. They contain the essentials that a program contains: a block. You may define a procedure at the beginning of a program after the variable declarations (variable declarations are not required). Like constants and variables, procedures are assigned an identifier that is used as a reference for calling the procedure later in the program. You can declare 0 or as many procedures as you want. Multiple procedure declarations are separated by semicolons, and thus a procedure declaration is essentially the keyword procedure followed by an identifier, followed by a semicolon, followed by a block, followed by a semicolon. Procedures are declared by using the reserved word *procedure.* Below is an example of a short procedure declaration:

procedure proc1;

var a;

a := 1;

Below is an example of a full PL/0 program that has a procedure that declares its own constants and variables:

const a = 1, b = 2;

var c;

procedure hello;

const d = 5;

var e;

begin

e := 17;

c := a + b + d + e;

end;

begin

call hello;

write c;

end.

If you compile and run that program, you will see that it prints 25. This is because upon calling hello, the procedure hello puts into variable c the value of 1 + 2 + 5 + 17. For more information about the call statement, see the call section in this guide.

See PL/0 EBNF Grammar on page 7 for the grammar of a procedure declaration.

**Blocks**

Blocks contain variable declarations, then constant declarations, then procedure declarations, and finally one statement. However, the one statement could be a statement containing multiple statements. Below is an example of a program, which is a block followed by a period, that computes and prints the value of 5+15:

const a = 5;

var b;

begin

b := 15;

write a + b;

end.

See PL/0 EBNF Grammar on page 7 for more on the grammar of a block.

**Expressions**

Expressions represent a value. Their name comes from mathematical expressions. They are constructed with variable identifiers, number literals, or the arithmetic symbols +, -, \*, /, \*, (, and ). The PL/0 language follows the standard order of operations. In our current implementation of an expression, an expression contains multiple parts (term and factors). Below is an example of an expression that represents the value 53:

5+(6\*8)

Expressions are thus the workhorse for arithmetic computation in PL/0. Expressions allow you to add, subtract, multiply, and divide integers. Not that division is integer division, and thus the result is always an integer. If you try to divide a number by a number that is not a divisor of it, then the result will be truncated into an integer. For example, the result of 5/2 will be 2, since 2.5 gets truncated to 2 in integer division.

See PL/0 EBNF Grammar on page 7 for the full grammar of an expression.

**Statements**

Statements are sections of a program that are executed by the program. A statement is the last required part of a block. Statements in PL/0 begin with an identifier, or the following reserved words: *begin*, *if*, *while*, *read*, *write*. Below are statements our program implements:

* Assignment Statement: Assign a value to an identifier. Only variable identifiers can be assigned expressions.
  + Examples
    - a := 5
    - b := 16+c
* Call Statement: Executes a previously declared procedure.
  + Examples
    - call hello
    - call sum
    - call theProc
* Begin End Statement: Wraps up zero or more statements. It can be used to run multiple from one statement. This statement consists of the keyword begin followed by 0 or more semicolon separated statements followed by the keyword end.
  + Examples
    - begin

a:= 5;

b := 6;

write a+b

end

* If Statement: Executes a statement if a condition is true. If the condition is false, the statement will be skipped and ignored.
  + Examples
    - if x < 16 then

call hello

* If Else Statement: Executes the first statement if a condition is true. Executes the second statement if the condition is false.
  + Examples
    - if x < 16 then

call hello

else

write x

* While Do Statement: Executes a statement repeatedly until the condition is false.
  + Examples
    - while x < 16 do

x := x + 1

* Read Statement: Assigns the user’s input value to an identifier. The program will wait until the user provides integer input upon hitting the execution of this statement. Values can only be assigned to variable identifiers.
  + Examples
    - read x
    - read inputVal
* Write Statement: Outputs an expression value to the terminal console.
  + Examples
    - write x
    - write sum

See PL/0 EBNF Grammar on page 7 for the grammar of a statement.

**Assignments**

Variables can be assigned values by using constant identifiers, variable identifiers, number liters, or expressions. The assignment operator, :=, can only be used inside a statement (see Statements). Assignments to undeclared identifiers are not allowed. In this implementation, variable assignments cannot be assigned during a variable declaration. Below is an example of an assignment statement:

x := 1; /\* x is assigned the value 1 assuming x is a variable that is previously declared \*/

See PL/0 EBNF Grammar on page 7 for the grammar of an assignment statement.

**Call**

The call statement can be constructed with the reserved word *call* followed by an identifier. Only procedure identifiers can be called. Below is an example program with two call statements:

procedure proc1;

var x;

begin

read x;

write x

end;

begin

call proc1; /\* executes procedure proc1 twice \*/

call proc1

end.

**Begin End**

The begin end statement can be used to group 0 or more statements. The statement begins with the reserved word *begin* followed by zero or more semicolon-separated statements and then the reserved word *end*. Below is an example of a begin end statement:

begin

write 1;

write 2

end;

See PL/0 EBNF Grammar on page 7 for the grammar of an begin end statement.

**Conditionals**

A code segment can be conditionally executed with the use of the reserve words *if*, *then*, *else*. The else is an optional. There are two ways of using a condition.

The first utilizes the reserve word *odd* followed by an expression. This will execute code if the value of the expression is odd. Below is an example of an odd conditional statement:

odd x + y /\* will be true if the expression *x + y* is an odd value \*/

The second utilizes the relational operators (rel-op) to compare expressions. They include:

* equal: =
* not equal: <>
* less than: <
* less than or equal: <=
* greater than: >
* greater than or equal: >=

Below is an example of the *­*if then statement:

If 0 = 0 then

write 1

Below is an example of the if then else statement:

if 0 = 0 then

write 1

else

write 0

Conditionals can be nested. This can be used to check multiple conditions before executing a statement. Below is an example:

if x > 0 then

write 1

else if x < 0 then

write -1

else if x = 0 then

write 0;

See PL/0 EBNF Grammar on page 7 for the grammar of a condition.

**Loops**

Loops can be constructed by using the reserved words *while* and *do*. Traditionally, loops iterate through a range of numbers until a condition is false. Below is an example of a while loop:

while 10 > i do /\* while loop that outputs 0 to 9 assuming i was assigned 0 \*/

begin

write i;

i = i + 1

end

See PL/0 EBNF Grammar on page 7 for the grammar of while statements.

**Input and Output**

The input is handled by using the reserved word *read* followed by an identifier. In our implementation, reading into a constant identifier is not permitted. The output is handled by using the reserved word *write* followed by a variable identifier or constant identifier. The input and output can be entered in the terminal with the keyboard. Below are examples of input and output in PL/0:

read x;

This will assign any valid value the user inputs to the variable *x*.

write 1; /\* outputs 1 on the terminal console \*/

write x; /\* outputs the value of x on the terminal console \*/

See PL/0 EBNF Grammar on page 7 for the grammar of read and write statements.

**How to Compile and Run the PL/0 Compiler**

To build and compile an executable file:

1. Write or obtain a source code (our implementation use the .txt format) in PL/0 for the PL/0 compiler. Place the files in the same directory as the compile executable. You must name the source code file as “input.txt” and it must reside in the same directory hat the program compile is in.
2. Open terminal and navigate to the directory where the source code is located.
3. To compile the program, enter:

gcc -o compile compile.c

1. To run the PL/0 compiler, enter:

./compile -l -a -v

1. The “-l”, “-a”, and “-v” are command line arguments. They are optional.

-l Prints the output from the lexical analyzer to the screen.

-a Prints the generated assembly code to the screen.

-v Prints the virtual machine output to the screen after it runs.

**How to Use Compiler While it is Running**

**Error Handling**

There are many types of errors that may arise depending on what input is given to the program. These errors come from either the lexical analyzer or the parser. When an error occurs, it will be printed out and then the program will try to continue parsing.

**List of Errors**

|  |  |
| --- | --- |
| Error Numbers | Error Meaning |
| 1 | Use of = instead of :=.\ |
| 2 | = must be followed by a number. |
| 3 | Identifier must be followed by =. |
| 4 | const, var, procedure must be followed by identifier. |
| 5 | Semicolon or comma missing. |
| 6 | Incorrect symbol after procedure declaration. |
| 7 | Statement expected. |
| 8 | Incorrect symbol after statement part in block. |
| 9 | Period expected. |
| 10 | Semicolon between statements missing. |
| 11 | Undeclared identifier. |
| 12 | Assignment to constant or procedure is not allowed. |
| 13 | Assignment operator expected. |
| 14 | Call must be followed by an identifier. |
| 15 | Call of a constant or a variable is meaningless. |
| 16 | then expected. |
| 17 | Semicolon expected. |
| 18 | do expected. |
| 19 | Incorrect symbol following statement. |
| 20 | Relational operator expected. |
| 21 | Expression must not contain a procedure identifier. |
| 22 | Right parenthesis missing. |
| 23 | The preceding factor cannot begin with this symbol. |
| 24 | An expression cannot begin with this symbol. |
| 25 | This number is too large. |
| 26 | The input file is too large. |
| 27 | Call must be followed by an identifier for a procedure. |
| 28 | Expected a factor, did not find one! |
| 29 | Too many symbols, or a conflicting symbol was found! |
| 30 | Read and write require an identifier after them! |
| 31 | You can only read into a variable! |
| 32 | You can only write a variable or constant! |
| 33 | Use of unassigned variable! |
| 34 | Program requires too many registers to run on the vm! |
| 35 | Semicolon required after procedure declaration. |
| 36 | Semicolon required after procedure block. |

**When it says “Running VM…”**

When the Virtual Machine says “Running VM…” it is waiting for an input from the user.

**Reference**

**PL/0 EBNF Grammar**

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 ["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" .

\*\* [] 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

**List of Reserved Words and Tokens**

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbol** | **Symbol Name** | **Symbol Number** | **Usage** |
|  | nulsym | 1 | reserved word |
|  | identsym | 2 | constants, variables, and procedure identifiers |
|  | numbersym | 3 | number literals |
| + | plussym | 4 | adding expressions |
| - | minussym | 5 | subtracting expressions |
| \* | multsym | 6 | multiplying expressions |
| / | slashsym | 7 | dividing expressions |
| odd | oddsym | 8 | check if an expression is odd |
| = | eqlsym | 9 | assign values to constants in constant declarations or check if two expressions are equal |
| <> | neqsym | 10 | check if two expressions are not equal |
| < | lessym | 11 | check if the left expression is less than right expression |
| <= | leqsym | 12 | check if the left expression is less than or equal to right expression |
| > | gtrsym | 13 | check if left expression is greater than right expression |
| >= | geqsym | 14 | check if left expression is greater than or equal to right expression |
| ( | lparentsym | 15 | begin a factor to define a precedence |
| ) | rparentsym | 16 | end a factor to define a precedence |
| , | commasym | 17 | separate multiple constants and variable identifiers during declaration |
| ; | semicolonsym | 18 | end of declarations and statements |
| . | periodsym | 19 | end of a program |
| := | becomesym | 20 | assign expressions to variables |
| begin | beginsym | 21 | begins a block of statement |
| end | endsym | 22 | ends a block of statements |
| if | ifsym | 23 | begins an if-then statement |
| then | thensym | 24 | ends an if-then statement |
| while | whilesym | 25 | begins a while-do loop |
| do | dosym | 26 | ends a while-do loop |
| call | callsym | 27 | calls a procedure |
| const | constsym | 28 | begin constant declarations |
| var | varsym | 29 | begin constant declarations |
| procedure | procsym | 30 | begins a procedure declaration |
| write | writesym | 31 | outputs the value of an expression to the screen |
| read | readsym | 32 | asks the user for an input and assigns the value to a variable |
| else | elsesym | 33 | optional part of if-then statement |