**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

Reference ………………………………………………………………………………………………………………….…………………………...9

**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 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. Anything between comment delimiters, e.g. /\* and \*/, will be ignored by the lexical analyzer.

**Datatypes**

This implementation of PL/0 supports the following datatypes:

* Constants (const)
* Variables (var)
* Procedures (procedure)

Each of these datatypes are 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 by the program for arithmetic and logical operations. Numbers 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. Defining multiple constants can be done by separating the value and the next constant’s identifier with commas. All constants 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 two statements are equal:

foo := x + y;

foo := 1 + 2;

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

**Variables**

Unlike constants, variables are mutable. You can define a variable and reassign their values. Variables are declared right after constant declarations (constant declarations are not required). 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. 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:

foo := 1 + x;

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

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. You must 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. You can declare 0 or as many procedures as you want. Multiple procedure declarations are separated by semicolons. Procedures are declared by using the reserved word *procedure.* Below is an example of a procedure declaration:

procedure proc1;

var a;

a := 1;

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

**Blocks**

Blocks contain variable declarations, constant declarations, procedure declarations, and statements. There can be 0 or multiple statements, each separated by a semicolon. See PL/0 EBNF Grammar on page 7 for the grammar of a block.

**Expressions**

Expressions represent or return 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). 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. Statements are part of a block. There can be 0 or more statements. The last statement must end with a semicolon. 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.
* Call Statement: Executes a previously declared procedure.
* Begin End Statement: Executes a single statement or multiple statements. It can be used to nest statements in a block.
* If Statement: Executes a statement if a condition is true. If the condition is false, the statement will be skipped and ignored.
* If Else Statement: Executes the first statement if a condition is true. Executes the second statement if the condition is false.
* While Do Statement: Executes a statement repeatedly until the condition is false.
* Read Statement: Assigns the user’s input value to an identifier. Values can only be assigned to variable identifiers.
* Write Statement: Outputs the identifier’s value to the terminal console.

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 of a call statement:

procedure proc1;

begin

read x; /\* assumes x is a variable that is already declared \*/

write x;

end;

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

call proc1;

**Begin End**

The begin end statement can be used to group statements. The statement begins with the reserved word *begin* followed by a statement and a semicolon and statement as many times as you want for multiple statements and a 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 execute code if the expression *x + y* is an odd value \*/

The second utilizes the relational operators (rel-op). 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 *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 and 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????**

**Reference**

**PL/0 EBNF Grammar**

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" .

\*\* [] 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** | **Internal Name** | **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 |