PL/0 User Manual

Authors: Darin Doria & Jorge Berroa

Table of Contents

The PL/0 Language

PL/0 is a fairly simple language that supports constants, integers, and procedures. All PL/0 programs have the following structure:

1. constant defintions
2. variable definitions
3. procedure declarations
    1. subroutines (same as program definition)
4. statements

Data Types

Currently, this implementation of PL/0 supports the following datatypes:

- constants (const)
- integers (int)
- procedures (procedure)

An identifier is a name used to refer to a specific instance of a datatype. Identifiers must be no more than 11 characters in length, must begin with a character, may contain uppercase and lowercase letters as well as numbers, and must not be any of the reserved keywords[1].

In addition to identifiers, number literals are used through the program for arithmetic and other operations. Number literals must be integers and must be no more than 5 digits long. Negative number literals are not currently supported.

[1] A list of keywords can be found in Appendix A

Constants

Constants are integer types. They may only be defined once per program. Constants are immutable; that is, you may not assign values to them after they have been defined. You may define more than one constant at a time by separating the identifiers by commas. Constant definitions must end with a semicolon. Constants are defined using the following syntax:

```
const FOO = 1, BAR = 2;
```

After you define a constant, you may use them throughout your program as you would an ordinary number. The compiler converts constants to their respective values upon compilation, so the following example,

```
foo := FOO + BAR;
```

is equivalent to this:

```
foo := 1 + 2;
```

Integers

Integers are mutable; meaning, you can define and reassign their values later on. Integers are declared at the top of the file after any constants (if there are any), and you may declare more than one at a time. Integers are defined with values as constants are. Instead, you must assign integers values using the assignment operator := (as seen above). Here is an example of how to declare integers in PL/0:

```
int foo, bar, baz;
```

You may assign variables values of constants or other variables, number literals, or expressions.

Procedures

They contain nearly everything that a program could contain. Example 1 shows a procedure that prints the numbers 1 through 10:

Example 1:

```
int f, n;
procedure fact;
  int ans1;
  begin
    ans1:=n;
    n:= n-1;
    if n < 0 then f := -1
    else if n = 0 then f := 1
    else call fact;
    f:=f*ans1;
end;
begin in n;
call fact;
out f; end.
```

Expressions

Expressions get their name from mathematical expressions which represent or return a value. Expressions can be composed of constant or variable identifiers, number literals, or the arithmetic symbols +, -, *, /, (, and). PL/0 follows the standard order of operations when calculating the value of an expression.

Statements

Statements are how the program gets things done. Except for the last statement in a block, statements must end with a semicolon.

Blocks

Blocks are collections of statements, each of which are separated by a semicolon. Blocks are denoted by the begin and end keywords. See section 1.1.3 (Procedures) for how a block can be nested inside of other statements.

Assignment

As mentioned before, variables can be assigned values by using constant or variable identifiers, number literals, or expressions. The assignment operator, :=, only works for variables inside of a statement. Currently, you are not able to assign variables initial values at the time of declaration.

Conditionals

To conditionally execute code, use the conditional keywords if, then, and else. These allow you to check a condition and, if true, execute some code, or else execute some other code. A conditional expression is two expressions separated by a relational operator (e.g., expression [rel-op] expression).

Valid relational operations are as follows:

- = (equal)
- < (less than)
- <= (less than or equal)
- > (greater than)
- >= (greater than or equal to)

Here is an example of an if-then without an else:

```
if 1 = 1 then out 1;
```

And with an else:

```
if 1 <> 1 then out 1 else out 0;
```

Conditionals can be nested in one another. One such use is checking multiple conditions before executing code. For example, here's a snippet from a basic four-function calculator program:

```
if op = ADD then call add
else if op = SUB then call sub
else if op = MULT then call mult
else if op = DIV then call div
else done := 1;
```

Compiling

Make sure that you have the following files in the same directory before compiling the program. They are necessary for the program to run properly. The files will work if they are blank (Keep in mind that input.txt should contain the PL0 code).

Files:

- input.txt
- mcode.txt
- stacktrace.txt

If the files are not in the directory you can create them by running following commands from the command line:

```
touch input.txt
touch mcode.txt
```

```
touch stacktrace.txt
```

To Compile

Run the following commands to compile the dependencies:

```
gcc scanner.c -o scanner
gcc parser.c -o parser
gcc vm.c -o vm
```

Now run the following command to compile the main file:

```
gcc main.c -o main
```

Running

To run the program type:

```
./main
```

Pass in any appropriate parameters as so:

```
./main –l –a –v
./main –v
```

Input

Any input being tested should be in a file named "input.txt", this file will contain the PL0 code.

It will look something like this:

```
var x, y;
begin
      y := 3;
```

```
        x := y + 56;
end.
```

Output

The output will be shown on screen depending on the command line paramaters passed into the main.c program. If there are no paramaters passed into the program, then the output from the scanner will be in "lexemelist.txt", the output from the parser will be in "mcode.txt", and the ouput from the virtual machine will be in "stacktrace.txt".

References

Appendix A:

program ::= block "." .

block ::= const-declaration var-declaration procedure-declaration statement.

constdeclaration ::= ["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" 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.

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

Special Symbols: ‘+’, ‘**-**‘, ‘*****’, ‘/’, ‘**(**‘, ‘**)**’, ‘**=**’, ’**,**’, ‘**.**’, ‘ **<**’, ‘>’, ‘**;**’ , ’:’ .

Identifiers: identsym = letter (letter | digit)*****

Numbers: numbersym = $(\text{digit})^{+}$