PL\0 User's Manual

Last updated: July 26, 2017

0.0 About PL\0

PL/0 is a simple and educational programming language similar to the general-purpose programming language Pascal. The language was originally introduced in the book, *Algorithms + Data Structures = Programs*, by Niklaus Wirth in 1976 and often serves as an example of how to construct a compiler.

The language features very limiting language constructs:

1. There are no real numbers.
2. There are few arithmetic operations and control-flow constructs.

0.1 Grammar

The syntax of the language defined by the grammar may be found in Appendix A. The rules of the grammar are defined in Extended Backus-Naur Form (EBNF) and follow a left recursive descent parser model.

1.0 Programming in PL\0

PL\0 contains a block divided into two sections: declaration and statements. The declarations consist of three types: *constants*, *variables*, and *procedures* and are strictly structured in the following way:

1. constant definitions
2. variable declarations
3. procedure declarations
    a. subroutine definition
4. statement(s)

White space and comments are ignored by the program and comments are defined by delimiters /∗ and ∗/. Any information in between these delimiters is not recognized by the language and will not execute.

```
/* This is a comment */
```

All PL/0 programs terminate with a period. This is the only instance of the period keyword.

1.1 Datatypes

PL\0 supports the following datatypes:

- constants (**`const`**)
- variables (**`var`**)
- procedures (**`procedure`**)

An identifier is used to refer to specific instances of each datatype. Identifiers must be no more than 11 characters in length, must begin with a character, may contain uppercase and lowercase letters and numbers and must not be any of the reserved keywords listed in Appendix B.

Several identifiers may be defined at a time. It is important to note that declarations are order dependent; constants must always be defined before variable and variables must always be declared before procedures.

Additionally, positive and negative number literals are permitted but must be fewer than 5 digits in length. As previously stated, real numbers are not supported in PL\0. Therefore, all constants and variables are integer types.

1.1.1 Constants

Constants are defined at the beginning of each block with the `const` keyword followed by the identifiers and their definitions. As stated previously, multiple constants may be defined at a time; however, only one `const` keyword may be used in each block. Each identifier is separated by a comma and the list is terminated with a semicolon. Below is an example of defining constants:

```
const ZERO = 0, FOO = 1
```

Constants may be used throughout the program and will be treated as their defined values. For example, the expression `FOO + ZERO` is equivalent to entering `1 + 0`.

Note that constants are immutable. This means that once they are defined, they are unable to be assigned different values during execution. Constants *must* be defined during declaration.

1.1.2 Variables

Conversely, ***variables***, are mutable and allow to be assigned values during the program's execution. However, variables are unable to be assigned values at declaration. Like constants, multiple variables may be defined at a time where only one keyword `var` is used.

```
var x, y, z
```

Variables are assigned using the "becomes" operator `:=` followed by a number literal (as shown in Figure 1) or followed by another variable or constant. All of the above may be used in the same assignment. Below is an expression that demonstrates this characteristic.

```
x := ZERO + y * 1
```

1.1.3 Procedures

Procedures are subroutines called by the main program. They act as sub programs that have the same capabilities as the main program. They are defined at the beginning of the program after variables and are terminated with a semicolon.

Unlike variables and constants, procedures are defined one at a time as shown in Figure 1. Note that the last statement in procedures are required to be terminated with a semicolon.

```
procedure foo;
      var x;
      begin
            x := 15;
            write x
      end;

procedure bar;
      var y;
            y := 42;
.
```

Figure 1

1.2 Statements

Once a program has defined its declarations, it begins to execute statements. ***Statements*** are various commands that perform different tasks. All statements except for the last statement in a block must terminate with a semicolon.

A single statement may be made in a program; however, most programs require multiple statements to perform any useful tasks. Multiple statements may be declared at once by using the `begin` and `end` keywords. Both types of declarations are demonstrated in Figure 1 by each procedure.

1.2.2 Expressions

An ***Expression*** is a mathematical phrase that can contain ordinary numbers, variables, constants, and operators (+ , - , * , /). Additionally, left and right parenthesis are permitted. Expressions follow the mathematical order of operations.

```
x := 3 * (x + 8)
```

1.2.2 Input/Output

The program requests an input from the user by utilizing the **`read`** keyword. `read` only accepts integer values and stores the values to a variable (constants and procedure identifiers are not permitted.)

```
read x
```

This command will halt the program and wait for the user to enter a value. The value is then stored in the variable `x`.

The program may output an expression by utilizing the keyword **`write`**. `write` has the ability to output any expression, literal, constant, or variable.

```
write x;
write 18;
write x + 18
```

1.2.3 Conditionals

A ***conditional*** is a statement that instructs the computer to execute a certain block of code or alter certain data if a specific condition has been met. This is achieved by using the keywords **`if`**, **`then`**, and **`else`**.

Conditionals check an expression and determines if is true or not by utilizing a relational operator (`<, >, <=, >=, =, <>`). If it is, then it will execute the following line define by `then`, otherwise it will skip that line of code.

```
if x = 3 then
     write x;
```

Optionally, an `else` can follow an `if... then` statement. This will allow only one or the other statement to execute, but never both.

```
if x = 3 then
     write x;
else
     write 0;
```

The contents of a conditional code are *statements* this means they can contain blocks of information by using `begin` and `end`. This also means they can contain `if... then... else` statements. These are called ***nested*** statements. See Figure 2 for an example of nested conditional statements.

Note that the last statement is not permitted to have a semicolon. If one is present, the compiler will produce an error.

```
var x, y;
begin
     x := 5;

     /* Is x greater than 0? */
     if x > 0 then
          if x > 3 then
               write 1
          else
               write 0

     /* Else, is x less than 0? */
     else if x < 0 then
     begin
          x := -1;
          write x
     end

end.
```

Figure 2

1.2.4 Loops

There is one type of loop defined in PL/0: the ***while*** loop. This is declared by the keywords **`while... do`** and acts in a very comparable manner to `if... then` statements. The difference being that a while loop continues to execute its contents until the conditional statement is false.

Figure 3 shows a program that counts from 1 to 5 using a while loop.

```
var i, target;
begin
     target := 5;

     while i <> target do
          i := i + 1;

     write i
end.
```

Figure 3

1.2.5 Calling Procedures

To call a procedure the keyword **`call`** is used followed by the procedure identifier.

```
call foo
```

This will interrupt the current flow of code and will execute the instructions defined in the procedure called. Like conditional statements, these calls can be nested.

Variables in procedures may be used by parent procedures. This also means that a variable within a child procedure cannot have the same name. For example, if the main procedure declares `var x` and declares a procedure `proc foo`, then `foo` is not allowed to declare any datatype named `x`.

Figure 4 shows a recursive version of the adder by utilizing parent variables and nested calls.

```
const target = 5;
var i;
procedure counter;
      if i <> target then
      begin
            i := i + 1;
            call counter;
      end;

begin
      call counter;
      write i;
end.
```

Figure 4

2.0 Compiling and Executing a Program in PL/0

The PL/0 compiler both compiles and executes the program on a virtual machine for PM/0, the machine for which the PL/0 ISA was designed.

2.1 Running PL/0 Compiler

These instructions assume you have experience using a terminal. You will need GCC and GNU Make prior to building the PL/0 compiler.

To build the compiler's executable file, do the following:

1. Obtain a copy of the source code for the PL/0 compiler.
2. Open a terminal and navigate to the project directory.
3. Compile using `gcc -std=c99 compiler.c`
    - This will output a file called a.out (or a.exe).

Once you have an executable, you are ready to run PL/0 programs.

2.2 Running PL/0 Programs with the Compiler

To begin, open a terminal and navigate the compiler's location. There are sample programs listed and the default input file is designated within. You can find and change the default file by looking at the `IN` constant found in `compiler.h`. Alternatively, the `-f <filename>` allows the user to take a custom input file.

```
flags:
------------------------------------------------------------
-f <filename> : Takes a specified input file.

-l            : Prints list of lexemes/tokens.

-a            : Prints generated assembly code.

-v            : Prints virtual machine execution trace.
```

3.0 Appendix

Appendix A: PL/0 Grammar

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**" 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**"**.**

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.

Appendix B: Symbol table

Symbol	Internal Name	Internal Value
	nulsym	1
	identsym	2
	numbersym	3
+	plussym	4
-	minussym	5
*	multsym	6
/	slashsym	7
	oddsym	8
=	eqlsym	9
<>	neqsym	10
<	lessym	11
<=	leqsym	12
>	gtrsym	13
>=	gedsym	14
(	lparentsym	15
)	rparentsym	16
,	commasym	17
;	semicolonsym	18
.	periodsym	19
:=	becomesym	20
begin	beginsym	21
end	endsym	22
if	ifsym	23
then	thensym	24
while	whilesym	25
do	dosym	26
call	callsym	27
const	constsym	28
var	varsym	29
write	writesym	30
read	readsym	31
procedure	procsym	32
else	elsesym	33

Appendix C: Instruction Set Architecture (ISA)

OP CODE	Syntax	Description
1	`LIT 0, M`	Push constant (literal) **M** onto the stack
2	`OPR 0, M`	Operation to be performed on the data at the top of the stack (See Appendix D)
3	`LOD L, M`	Load value to the top of stack from the stack location at offset **M** from **L** lexicographical levels down
4	`STO L, M`	Store value at the 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** (generates new Activation Record and pc ← **M**)
6	`INC 0, M`	Allocate **M** locals (increment sp by **M**).
7	`JMP 0, M`	Jump to instruction **M**
8	`JPC 0, M`	Jump to instruction **M** if top stack element is 0
9	`SIO 0, 1`	Write the top stack element to the screen
9	`SIO 0, 2`	Read in input from user and store it at the top of the stack
9	`SIO 0, 3`	Halt

Appendix D: OPR codes

M	Operation	Description
0	RET	Return from a procedure
1	NEG	- stack[sp] → stack[sp]
2	ADD	stack[sp] + stack[sp – 1] → stack[sp – 1]
3	SUB	stack[sp] - stack[sp – 1] → stack[sp – 1]
4	MUL	stack[sp] * stack[sp – 1] → stack[sp – 1]
5	DIV	stack[sp] / stack[sp – 1] → stack[sp – 1]
6	ODD	Replace TOS with 1 if stack[sp] is odd, otherwise replace TOS with 0
7	MOD	stack[sp] % stack[sp – 1] → stack[sp – 1]
8	EQL	If stack[sp] = stack[sp – 1], replace TOS with 1 otherwise replace TOS with 0
9	NEQ	If stack[sp] != stack[sp – 1], replace TOS with 1 otherwise replace TOS with 0
10	LSS	If stack[sp] < stack[sp – 1], replace TOS with 1 otherwise replace TOS with 0
11	LEQ	If stack[sp] <= stack[sp – 1], replace TOS with 1 otherwise replace TOS with 0
12	GTR	If stack[sp] > stack[sp – 1], replace TOS with 1 otherwise replace TOS with 0
13	GEQ	If stack[sp] >= stack[sp – 1], replace TOS with 1 otherwise replace TOS with 0

Appendix E: Compiler Errors

Code	Description
1	‘:=’ missing in statement.
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	Procedure declaration must end with a semicolon.
7	Expected ‘end’ after ‘begin’.
8	Variable <id> already exists.
9	Period expected.
10	Call must be followed by a procedure identifier.
11	Undeclared identifier.
12	Assignment to constant or procedure is not allowed.
13	Failed to execute virtual machine.
14	Call must be followed by an identifier.
16	Expected ‘then’ after ‘if’ condition.
18	Expected ‘do’ after ‘while’ condition.
20	Relational operator expected.
21	Identifier, ‘(‘, or number expected.
22	Right parenthesis missing.
25	This number is too large.
26	Unable to read parser input file. See ‘LEX_OUT’ in ‘compiler.h’.
27	Unable to read virtual machine input file. See ‘PAR_OUT’ in ‘compiler.h’.
-1	Undefined error.