PL\0 User’s Manual

Last updated: July 26, 2017

# 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*



Source: <https://en.wikipedia.org/wiki/PL/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
   1. subroutine definition
4. statement(s)

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

All statements are terminated with a semicolon, except for the last statement in the program. All PL/0 programs terminate with a period. This is the only instance of the period keyword.

Figure 1 below demonstrates a simple PL/0 program with comments.

Figure 1

var f;  
begin  
 /\* This is a comment \*/  
 f := 3;  
  
 /\* Note the last statement  
 in the block does not   
 a semi-colon.  
 \*/  
  
 f := 10  
end.

# 1.1 *Datatypes*

PL\0 supports the following datatypes:

* constants
* variables
* procedures

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 keyword followed by the identifiers and their definitions. As stated previously, multiple constants may be defined at a time; however, only one 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:

Constants may be used throughout the program and will be treated as their defined values. For example, the expression is equivalent to entering .

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

# 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 is used.

Variables are assigned using the “becomes” operator followed by a number literal (as shown in Figure 1) or followed by another variable or constant. Below is an expression that demonstrates this characteristic.

# 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. Unlike variables and constants, procedures are defined one at a time as shown in Figure 2. Note that keywords in procedures are terminated with a semicolon rather than a period.

Figure 2

Procedure foo  
 var x;  
 begin  
 x := 15;  
 end;  
  
Procedure bar  
 var y;  
 begin  
 y := 42;  
 end;

# 3.0 Appendix

Appendix A: PL/0 Grammar



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 |