P-Machine User Guide

By: Giampiero Corsbie
PID: gi501724

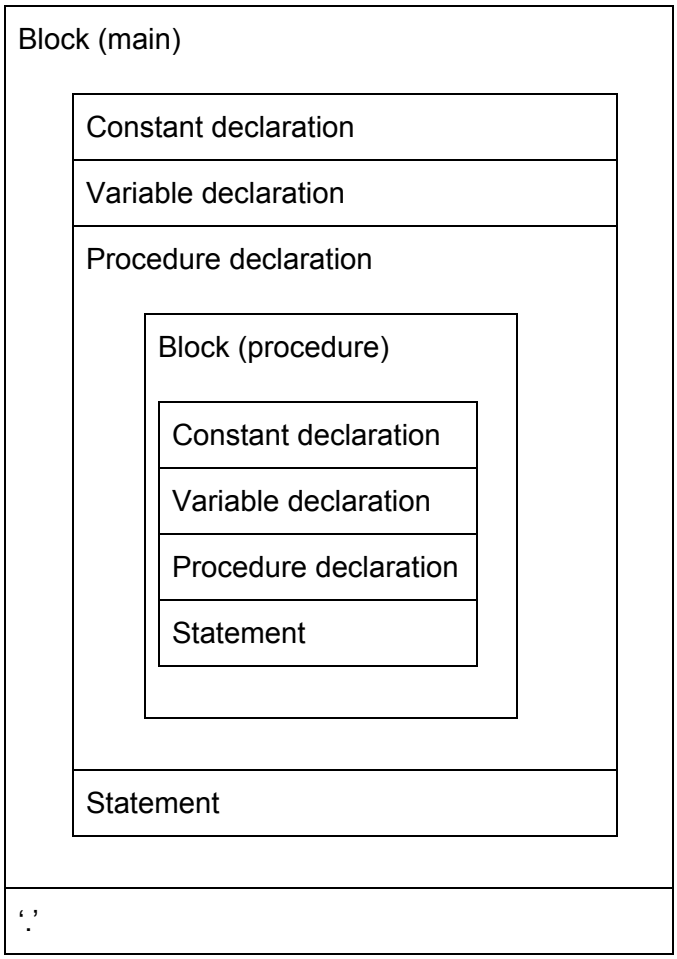
Sections:

---

1. The PL/0 Language

a. Program Structure

The structure of a PL\0 program consists of a main code block and then a period '.' to indicate the end. Inside the block; constants, variables, and procedures may be declared in that order. This is then followed by a statement. Because each one of these is optional the smallest program possible is "." by itself.



b. Identifiers

Constant, variable, and procedure symbols are accessed using identifiers. These alphanumeric identifiers have a max length of X and cannot start with a digit.

They also must not be among any of the following reserved symbols:

{ const, var, procedure, call, begin, end, if, then, else, while, do, read, write, odd }

c. Operators and precedence

Within an expression, the multiplication and division operators, ‘*’ and '/’ hold higher precedence than the addition and subtraction operators, ‘+’ and ‘-’, and a nested expression in parentheses holds the highest precedence.

d. Block Declarations

Constant declarations consist of the “const” symbol followed by an identifier, the ‘=’ symbol, and then a value assignment. The end of a constant declaration is indicated by a semicolon as shown in the example program below.

```
const a = 2;
.
```

When declaring multiple constants, place a comma between each value assignment and the next identifier, with the final one terminating on a semicolon.

```
const a = 1, b = 3;
.
```

Variable declarations that follow simply use the “var” symbol followed by an identifier and are terminated with a semicolon. Adding to the previous example, this is shown below.

```
const a = 1, b = 3;
var x;
.
```

Similar to constants, multiple variables may be declared with each identifier being separated by a comma.

```
const a = 1, b = 3;
var x, y, z;
.
```

Procedure declarations start with the symbol “procedure” followed by an identifier, parameters separated by commas (enclosed in parentheses), and semicolon. The code block for the procedure comes next. The declaration terminates with another semicolon.

```
var x, y, z;
procedure foo(u, v);
        var w;
        ;
.
```

e. Block Statements

Procedure call statement simply consists of the “call” symbol followed by an identifier.

```
const a = 1, b = 3;
procedure foo;
        var w;
        ;
call foo.
```

Variable assignment statement consists of the identifier of the variable followed by the assignment symbol “:=” and then an expression. In a procedure block, this can also be done with the implicit return variable “return”.

```
const a = 1, b = 3;
procedure foo;
        var w;
        w := 2 * a + b - 1;
call foo.
```

Read and write operations are simply either symbol “read” followed by the identifier for the variable that will store an input integer or “write” followed by an expression whose value will be printed to stdout.

```
const a = 1, b = 3;
write 2 * a + b - 1.
```

Multiple statements can be nested in one statement that begins with the “begin” symbol, then contains any number of statements, separated by semicolons, and ends with an “end” symbol.

```
var x, y;
procedure foo;
        var w;
        procedure foo2; w := y * x;
        begin
                read y;
                call foo2;
                write w;
        end;
begin
        read x;
        call foo
end.
```

An if/then/else statement begins with the “if” symbol, and is followed by a condition. A condition is just a relational operator between two expressions. After the condition comes a “then” symbol. The statement that follows is executed if the condition results in true. See example program below:

```
const n = 1;
var in;
Procedure abs;
        begin
                read in;
                If in < n - 1 then
                        in := -in;
                write in
        end;
call abs
.
```

Optionally the symbol “else” then another statement may follow, to be executed if condition results in false.

```
const z = 0;
var in;
begin
        read in;
        If in < z then
                write 1
        else
                write 0
end.
```

While loop statements begin with the “while” symbol, are followed by a condition, the symbol “do” and a statement to be executed while the condition is true.

```
var in;
begin
        read in;
        while in <> 0 do
                begin
                        in := in - 1;
                        write in
                end
end.
```

f. Scope

Whenever a PL/0 program is parsed, identifiers are added to a symbol table by name and stacked by the block level at which it was declared. Constants, variables, and procedures declared in the main block are stored with a level of zero. Any declarations within a nested block of a procedure declaration are stored with incremented level. When accessing a symbol in a statement, the symbol returned by the table has the highest level of all symbols sharing its name.
When the parser returns from a procedure block all symbols from that block level are removed. The level also represents the location of the addressed memory on the activation record stack. (See VM Memory and Execution)

This means duplicate identifiers are allowed if the new identifier is declared at a higher block level, where only that block can access it. However, the previous declaration now becomes inaccessible until the scope of the new one has ended and its symbol has been removed. The example program below demonstrates this when it prints a “10” then a “5” to the screen:

```
var x;
procedure foo;
        var x;
        begin
                x := 10;
                write x
        end;
begin
        x := 5;
        call foo;
        write x
end.
```

This also applies with procedure declarations. Note how in the following program prints “5” and “10” instead of “10” and “10”:

```
procedure a;
        write 10;
procedure b;
        procedure a;
                write 5;
        call a;
begin
        call b;
        call a
end.
```

If the scope of a symbol has ended then it is no longer accessible. The broken program below will return an “undeclared identifier error” during compilation:

```
procedure b;
        procedure a;
                write 5;
        call a;
begin
        call b;
        call a   /*ERROR*/
end.
```

Note that calling a procedure before it is has been declared will result in an error:

```
procedure b;
        call a;  /*ERROR*/
procedure a;
        write 1;
call b.
```

---

2. What is the P-Machine?

a. VM Memory and Execution

The PM/0 virtual machine takes in a compiled machine code file generated from a PL/0 program. This list of instructions is stored in a static array with a maximum instruction count of X. The position of the next instruction is stored in the program counter (pc) register. This is incremented every time an instruction is fetched, with the fetched instruction being stored in the instruction register (ir) before its execution.

When executed, each of these instructions operate on a “stack” of integer values stored in a static array. The CPU register for the stack pointer (sp) points to the top value. These values are interpreted as a stack of structures called activation records (AR), each representing data used in a procedure call. The CPU register for the base pointer (bp) points to the base of the current AR.

The data in each AR is accessed from an offset of its respective base. The first four values in an AR instance are the functional value, a pointer to the base of the parent AR (static link), a pointer to the base of previous AR (dynamic link), and the next instruction after the initial procedure call that created the AR (return address). Following that, variables are stored with an offset of 4 or more.

b. Example Assembly Structure

Relevant PL/0 code for the example assembly:

```
procedure A;
        var y;
        begin
                y := 2;
        end;
call A.
```

Example assembly:

0	JMP	0	6	Jump to main block on line 6.
1	JMP	0	2	Jump to procedure block on line 2.
2	INC	0	5	Increment sp for AR data and one variable in procedure block.
3	LIT	0	2	Place literal integer 2 on top of the stack.
4	STO	0	5	Store value at the top of stack 0 ARs down with an offset of 5.
5	OPR	0	0	Return from procedure.
6	INC	0	4	Increment sp for AR data in main block.
7	CAL	0	2	Call procedure.
8	SIO	0	3	Return from main.

3. Compiling and running PL/0 code

a. Compiling the Compiler and VM Driver

If not already in the project folder, navigate to the directory containing the file named "makefile". To compile the P-Machine driver from the terminal, use the following command:

```
make
```

This will generate object files from the source and create an executable named "driver" which will be used to compile and run PL/0 code. To remove the generated object files and executable, use the following command:

```
make clean
```

b. Running PL/0 Code

After compiling the driver, place the target PL/0 code in the same folder as the driver, with the name "in.txt". To run the driver use the following command:

```
./driver
```

This will take "in.txt" and process it into lexemes before it is parsed. While parsing, the machine code file "vminput.txt" will be created. If an error occurs, it will be recorded in an error file named "ef" which can be found in the same directory. The driver then runs the newly assembled "vminput.txt" on the VM. All output is dumped to a file called "out.txt".

Any of the following optional flags can be used to print information to the screen along with the program execution:

-l	Prints the symbolic lexeme list and lexeme list scanned in by the lexical analyzer.
-a	Print the generated assembly to the screen.
-v	Print the list of interpreted instructions and the VM stack trace across program execution.

Two examples of using of these flags on the command line:

```
./driver -l -a -v
./driver -l -v
```

c. Interpreting VM output

The first line of the VM execution strace will be the CPU registers' initial values and an empty stack. On each of the following lines, the VM will print the interpreted instruction fetched from the code memory at index provided by the previous pc.

Following that, the state of the registers and the stack following instruction execution are printed on the same line. Activation records on the stack are all separated by a '|' character.

When new AR data has been created due to a procedure call and the base pointer is just above the stack pointer, the VM will print the data of the new AR.

Example VM output:

					pc	bp	sp	stack
				Initial Values:	0	1	0	
0	jmp	0	10		10	1	0	
10	inc	0	6		11	1	6	0 0 0 0 0 0
11	lit	0	3		12	1	7	0 0 0 0 0 0 3
12	sto	0	4		13	1	6	0 0 0 0 3 0
13	lit	0	9		14	1	7	0 0 0 0 3 0 9
14	sto	0	5		15	1	6	0 0 0 0 3 9
15	cal	0	2		2	7	6	0 0 0 0 3 9 | 0 1 1 16
2	inc	0	6		3	7	12	0 0 0 0 3 9 | 0 1 1 16 0 0
3	lit	0	13		4	7	13	0 0 0 0 3 9 | 0 1 1 16 0 0 13
4	sto	0	4		5	7	12	0 0 0 0 3 9 | 0 1 1 16 13 0
5	lit	0	1		6	7	13	0 0 0 0 3 9 | 0 1 1 16 13 0 1
6	sto	1	4		7	7	12	0 0 0 0 1 9 | 0 1 1 16 13 0
7	lit	0	7		8	7	13	0 0 0 0 1 9 | 0 1 1 16 13 0 7
8	sto	0	5		9	7	12	0 0 0 0 1 9 | 0 1 1 16 13 7
9	opr	0	0		16	1	6	0 0 0 0 1 9
16	sio	0	3		0	0	0	

3. Appendix

a. PM/0 Instruction Set Architecture

OP	L	M	Description
1 LIT	0	M	Push literal M on top of the stack.
2 OPR	0	M	Perform an operation defined by M on the top of the stack.
		0 RTN	Return from call (sp=bp-1 then pc=stack[sp+4] and bp=stack[sp+3])
		1 NEG	Negate the value at the top of the stack.
		2 ADD	Pop two values off the stack and push their sum.
		3 SUB	Pop two values, subtract the top from the second, push the result.
		4 MUL	Pop two values off the stack and push their product.
		5 DIV	Pop two values, divide the second by the top, push the result.
		6 ODD	Pop a value off the stack and push 1 if it's odd, else push 0.
		7 MOD	Pop two values, push the second value modulus the top value.
		8 EQL	Pop two values and push 1 if they're equal, else push 0.
		9 NEQ	Pop two values and push 1 if they're not equal, else push 0.
		10 LSS	Pop two values and push 1 if the second < the top, else push 0.
		11 LEQ	Pop two values and push 1 if the second <= the top, else push 0.
		12 GTR	Pop two values and push 1 if the second > the top, else push 0.
		13 GEQ	Pop two values and push 1 if the second >= the top, else push 0.
3 LOD	L	M	Load the value L levels down with offset M and push it on the stack.
4 STO	L	M	Pop a value off the stack and store it L levels down at an offset of M.
5 CAL	L	M	Call the procedure that begins on line M and set values for new AR
6 INC	0	M	Allocate M locals on top of stack by adding M to the stack pointer.
7 JMP	0	M	Jump to instruction on line M by setting program counter to M.
8 JPC	0	M	Set the program counter to M if the value on top of the stack is zero.
9 SIO	0	M	System input/output instruction defined by M.
		1 Write	Pop a value off the stack and write it to stdout.
		2 Read	Read a value from the user and push it on top of the stack.
		3 Halt	Set run flag to false and set registers to null.