PL/0 User's Guide

INTRODUCTION

PL/0 is a small programming language that works with the **P-machine** (also known as **PM/0**). A **programing language** is a set of strict guidelines that form a language syntax that allows for a set of instructions to produce output on a certain machine. Bundled with this package is a PM/0 virtual machine. A **virtual machine** (VM) is a program that emulates a physical machine. Therefore, a user can write PL/0 code, pass it through the PL/0 compiler, and then execute the code on the PM/0 VM. A **compiler** is a set of computer software to transform human-written code (in this case PL/0) into a different style of code that is easily read via a machine (or virtual machine) in order to produce specific computational results and output.

A PL/0 program can be written in any standard text editor, found bundled with every major operating system. Below is an example PL/0 program:

```
var a, b, c;
begin
    read a;
    read b;
    c := a + b;
    write c;
end.
```

The above program is a very simple PL/0 example. It requests two numbers from the user, then it adds those numbers together, and finally writes the result to the computer screen. Of course this likely looks like a foreign language to someone unfamiliar with programming. Programming languages are made up of keywords, symbols, numbers, and identifiers. **Keywords** are reserved words that mean something special to the programming language; these can be used to direct the programming language. **Symbols** (such as commas, semicolons, plus sign, minus sign, et cetera) are used to separate lists, end statements, perform arithmetic, et cetera. **Numbers** can be used for mathematical computations. And, lastly, **identifiers** are programmer-defined names for either constants, variables, or procedures.

Keep in mind, a **programmer** is someone who writes the code to be passed to the compiler and eventually ran on the PM/0 virtual machine as a program. A **user** is someone who is actually running and using the program on the PM/0 virtual machine.

The PM/0 virtual machine only supports values that are integers. **Integers** are numerical values that are whole numbers, not decimals or fractions. PM/0 can support positive or negative integers. The system does not support any other data type.

KEYWORDS

Example PL/0 program:

```
var a, b, c;
begin
    read a;
    read b;
    c := a + b;
    write c;
end.
```

In the above example, various keywords are being utilized. The keyword "`begin`" starts a series of statements that either form the entire basis for the program (as in the example above) or a procedure. More details about procedures will be provided later on. After a set of statements is complete, we similarly signify the end via the keyword "`end`". Every PL/0 program will have at least one `begin` and one `end` keyword.

In the above example the keyword "`read`" is used to tell the PM/0 virtual machine to request an input from the user. The virtual machine will pause and wait for the user to input an integer, followed by the enter/return key. This will allow the program to continue executing the remaining code. The "`write`" keyword displays the specified value to the computer screen.

There is also the keyword "`var`". This keyword defines a variable. A **variable** is a programmer-defined name (identifier) given to a value that can be changed. This is one of the key building blocks to allow a program to function without having to specify every numerical digit. Each variable essentially is a placeholder for a value, just as in algebra for instance. However, the value for a variable cannot be ambiguous; it must be defined by the programmer with a starting value, assigned a value via the "`read`" keyword (therefore allowing the user to define the value), or assigned a value from other variables and/or numbers. Identifiers used to name variables must start with a letter (capital or lowercase), and can then be followed by a series of more letters (capital or lowercase), numbers, or a mix of both. Identifiers cannot be keywords (remember, keywords are reserved words for the programming language to function). For instance, the following identifiers are valid in PL/0:

```
a    b    c    alpha    Beta    gAmMa    DELTA1    a1    bB2    c3c
```

However, the following examples are not valid identifiers in PL/0:

```
1a    12345    alpha*    2-beta    gamma_delta    var    read    **
```

In our example program, `a`, `b`, and `c` are all identifiers. There is also the use of the "`+`" symbol. Just as with standard mathematics, this symbol adds two numbers together, adds the values of two identifiers, or adds a number (also known as a **literal**) to the value of an identifier. PL/0 also supports subtraction ("`-`" symbol), multiplication ("`*`" symbol), and division ("`/`" symbol).

There is also the special symbol "`:=`". This symbol assigns a value to a variable. So in our example, the symbol is used to assign the value of `a + b` to the variable `c`. After this line of code, `c` is now equal to that value until it is later changed (if ever). This symbol is very important; just as with the

`var` symbol, the `:=` symbol is a critical building block that allows programs to actually work. It is important to keep in mind that the `:=` symbol is not equivalent to the `=` symbol (which will be discussed with relational operators later).

The "`;`" (semicolon) symbol is used to signal the end of a statement. A **statement** is a set of symbols that represent one full action. The semicolon symbol is generally found at the end of most lines of code to flag the end of that statement, and that the following line will be a new statement. In PL/0, a semicolon is required to separate statements between a `begin` symbol and an `end` symbol. However, it is not strictly required to be on the last line in such a series of statements (however, it can optionally be added).

A "`.`" (period) symbol is used at the very end of a program to explicate that the program has ended. Every PL/0 will contain exactly one period symbol (no less, no more), and it will be required to be the very last symbol.

The "`,`" (comma) symbol is used to separate a list of constant or variable declarations. A **declaration** is found at the beginning of a program and at the beginning of each procedure that lists all of the constants and/or variables being used in that section of code.

RELATIONAL OPERATORS AND IF/THEN/ELSE

Let us see another simple example to introduce some more keywords:

```
const w = 10, x = 100, y = 1000;
var z;
begin
    if x > y then
        z := 1
    else
        z := w
end.
```

Here we are introduced to the keyword "`const`". This keyword is used to define a constant. A **constant** is a value that is given a programmer-defined identifier, but it cannot have its value changed. A constant must have its value defined at the beginning of the program or procedure in which it is declared. Note that constants are defined with the "`=`" symbol, not the "`:=`" symbol. This is very important to remember. It is also important to note that the "`=`" symbol being used here is not equivalent to the "`=`" symbol used later as a relational operator (even though both use the same symbol). The symbol when used during a constant declaration is used to define a value, while the symbol when used as a relational operator compares two values. Constants, as with variables, are separated by commas.

In the above example, we also use the ">" symbol. This symbol compares the two values on both sides and reads as: "x greater than y". This is a relational operator. A **relational operator** will compare two literals, variables, and/or constants. Each side of the relational operator is known as an expression. The following are the six relational operators:

`=`	"equal to"	compares if two expressions are equal
`<>`	"not equal to"	compares if two expressions are not equal
`<`	"less than"	compares if the left expression is less than the right expression
`<=`	"less than or equal to"	compares if the left exp. is less than or equal to the right exp.
`>`	"greater than"	compares if the left expression is greater than the right expression
`>=`	"greater than or equal to"	compares if the left exp. is greater than or equal to the right exp.

By themselves, relational operators do not do very much. However, they become very powerful when combined with an control-flow statement. A **control-flow statement** is a statement that allows the program to "make decisions" about which code to execute based on certain conditions. This is what allows a programmer to add logical to a program. So instead of a set of code simply executing from first line to last line, control-flow statements will allow the program to execute (or not execute) certain statements.

One such statement uses the "`if`" symbol. This symbol is combined with the "`then`" symbol to, in more plain English, consider the following: "if (this condition is true) then (execute this statement)". So, for example, in our above code the control-flow statement will only execute the statement `z := 1` if `x > y` evaluates to "true". If the statement following the `if` symbol evaluates to "false", then the statement after the `then` symbol is skipped.

There is also an optional way to execute a statement if a condition is “false”, and that is with the “`else`” symbol. This symbol expands the grammar to read like this: “if (this condition is true) then (execute this statement), otherwise (execute this statement instead)”. In our example above, if the first condition following the `then` symbol is not executed, then the statement following the `else` symbol will be executed instead. Note that the semicolon symbol is not present after each statement in between `if`, `then`, and `else` symbols. An optional semicolon may be added to the last part of an if/then or if/then/else statement, but it is not strictly required. A semicolon cannot be added in the middle of a control-flow statement. For example, the following snippet of code is valid:

```
if x > y then
    z := 1
else
    z := w;
```

But this code is not valid:

```
if x > y then
    z := 1;
else
    z := w;
```

Another operator that is similar to a relational operator is the “`odd`” symbol. What makes this symbol itself odd is that it does not relate two expressions. Instead, the condition is considered “true” if the value being evaluated is odd, or “false” if the value being evaluated is even. Like other relational operators, the `odd` symbol is most powerful when combined with control-flow statements like if/then/else statements.

It is also possible to encapsulate a statement within a “(“ symbol and a “)” symbol to make the multiple statements inside evaluate as one statement. For example:

```
if (w + x) > (x + y) then
     z := 42;
```

WHILE/DO

Consider the following code:

```
var i, x;
begin
    i := 5;
    x := 0;

    while i > 0 do
    begin
        x := x + 1;
        i := i – 1;
    end;
end.
```

The "`while`" symbol combines with the "`do`" symbol to form a different kind of control-flow statement. The condition following the `while` symbol will be evaluated; if the condition is true, then the statement following the `do` symbol will be executed. Afterwards, the condition following `while` will be evaluated again; if it remains true, then it will execute the statement following `do` again. If it is false, control will continue to after the `do` statement. In other words, the statement following the `do` symbol will repeat until the statement following `while` symbol is false.

Note that in the above example, the statement following the `do` symbol is wrapped inside of a begin/end statement. This means that the entire statement between `begin` and `end` will be executed each time the while statement evaluates to true.

PROCEDURES

An integral part of the PL/0 programming language is procedures, declared with the "`procedure`" symbol. A **procedure** is a unit of instructions that can stand alone (almost as a mini program). A procedure in PL/0 can have its own constant and variable declarations, along with even more procedure declarations. This unit of code can be called upon via the "call" symbol. It is important to note that procedures can only see variables from within itself or within a "parent" or "grandparent" procedure.

```
var a, b, c;
procedure number1
    var d, e, f;
    procedure number1b
        var x, y, z;
        begin
            x := d;
            y := e;
            z := f;
        end;
    begin
        d := a;
        e := b;
        f := c;
        call number1b;
    end;
begin
    a := 1;
    b := 2;
    c := 3;
    call number1;
end.
```

The above code shows the structure for using procedures. These are very powerful in a programming language and allow for many ways to manipulate the control of a program.

COMMENTS

In order to write "comments" in a PL/0 code file, a programmer must surround regular text between a "`/*`" symbol and a "`*/`" symbol. This will make the PM/0 machine ignore everything in between. This is useful for writing short notes about how something works, about what an output should be, et cetera.

```
Var a;
begin
    a := 0;
 /* a := 1 */
    write a;
end.
```

This code outputs the value "`0`" to the computer screen, because the second statement (written between a "`/*`" symbol and a "`*/`" symbol) is completely ignored by the PM/0 compiler and PM/0 virtual machine. These comments are for the programmer only.

FACTORIAL (EXAMPLE PROGRAM)

```
var result, arg;

procedure fac;
    var n;

    begin
        n := arg;

        if n <= 1 then

        begin
            result := 1;
        end

        else

        begin
            arg := n – 1;
            call fac;
            result := n * result;
        end;

    end;

begin
    read arg;
    call fac;
    write result;
end.
```

EXECUTE PM/0 CODE (ON PM/0 VIRTUAL MACHINE)

Executing the written code on the PM/0 virtual machine is very simple. For UNIX or UNIX-like execution, make sure that the PM/0 virtual machine program is in the same file directory as your PL/0 code. Then, execute the following command via a terminal:

```
./PM0 -i [inputFile] -o [outputFile]
```

For example, if your PL/0 code was stored in a file named “factorial.txt”, then you should run the following command via a terminal:

```
./PM0 -i factorial.txt -o factorial.out
```

This will display any output to the terminal, along with request any user input. Advanced options can be read about in the PM/0 virtual machine README file.