PL/0 and Complier User Guide

PL/0 is a simplified programming language that still allows users to do some powerful operations. This user guide will explain various aspects of PL/0, how to write full programs, how to compile and run programs, and show some examples.

Variables

Variables are used to store values in the PL/0 programming language. Variables are split into two different categories, constants and vars. Constants are permanent and are not meant to change throughout the program execution. For example if you find you will need the number 3 a lot then you might save it as a constant. Vars are like normal variables, they can be modified throughout the program with different values. All constants and vars in PL/0 only hold integer values meaning numbers such as 0,1,2,3…ect but no decimal values or fractions.

Declaring Variables

In order to use a variable, it must first be declared at the start of your PL/0 program. Since there are two types of variables, they are declared in different ways. Constants are always declared first. This example shows a constant called x being declared with the value 5.

const x = 5 ;

Any number of constants can be declared in a row as this next example shows.

const x = 5, y = 6, z = 7 ;

Vars are declared in a similar way but they use a different keyword and don't specify a value when declared. Here is an example of a var declaration.

var x ;

Vars can also have any number of them declared in the same way that constants can.

var x, y, z ;

After you have declared your constants then your vars you can use them in other parts of the program. An important note is that any name can be used for variables, except keywords that are reserved in PL/0. These words are bolded throughout the guide.

Statements

Statements are the building blocks of your program and how it completes a task. There are many different types of a statement and they are explained in subsections below.

Variable Assignment

Now that you know how to declare variables from the previous section its important to know how to give them values. Note this only applies to vars because constants already have a value and it is not allowed to be changed. In this example I am giving the var x that was declared in the previous section the value 5.

```
x := 5
```

The symbol := is used for assigning a var a value. Be careful not to confuse this with the symbol = which is used to tell if one value is equal to another.

Variables can also be assigned to expressions which is similar to a math equation you have probably seen before. Here are several forms of an expression:

```
x + y

x * y

(x + y) / 6
```

*, /, (), and + can all be used in an expression to assign different values to a variable.

Begin / End

This statement type is a good way to chain multiple statements together in longer programs. It follows the form:

begin statement **end**

In order to add additional statements you follow the first statement with ; statement. He is an example of three statements in begin end:

begin statement ; statement ; statement **end**

Remember statements can be anything in this section or they can be empty which just means blank. This allows you to chain begin end statements together if you choose to, although it could get confusing. Here is an example replacing the statement with a variable assignment, which is one option for a statement.

begin x := 5 **end**

If – Statements

If statements are very powerful and allow the program to make decisions on which sections of code to execute and which to ignore. If statements come in two forms. The first is the basic if then statement which looks like this:

if condition **then** statement

A condition is a logical decision. Conditions can be one of two things. The first is the key word odd followed by a variable or a number as follows:

odd 5

The second is a variable or a number then a relational operator and then another variable or number as follows:

5 > 6

The relational operators allowed in PL/0 are <, <=, >, >=. To test whether two variables or numbers are equal the operator = is used. For not equal <> is used.

Putting this all together you can use if then statements to do things based on different results for example:

if 6>y **then** x := 5

This code checks the condition and sees if 6 is bigger than some variable y. If it is then x will be assigned the value 5 and if not then x is left alone.

The basic if then statement can be expanded to an if then else statement which allows for a different branch if the condition is false. For example using the code above:

if 6>y **then** x:=5 **else** x:=6

This code sets x to 5 if 6 is bigger than y and if not it sets x to 6. Else is similar to then in that it can be followed by any statement which means if statements can be chained together and nested depending on the needs of your program.

While Loops

Sometimes it is necessary to repeat a task multiple times. Instead of typing a line of code you want repeated several times, a while loop can be used in PL/0. Here is the form of a while loop:

while condition **do** statement

Condition works the same way it did in the if statement and as always statement can be one of anything in this section. Here is an example of a while loop using a condition and a statement:

while x<100 **do** x := x+y

This code assigns x the value of x+y until x is bigger than 100, or while x being less than 100 is true. Since the while loop executes a statement it can of course be combined with other while loops however too many while loops will slow your program down by a lot so you must be careful with nested while loops.

Read / Write

It is perfectly acceptable to have the program do calculations by itself but you may want to input different values without going and changing the code every time or you may want to see what value an equation calculates. This is where the read and write statements are used.

Read is used to take a value from the user and assign it to a var in your program. Here is an example of read:

read x

This code asks a value from the user and then stores it in x. Remember x must be a var and must have been declared prior to using it in read or else an error will stop your program.

Write is similar but instead of just vars, you can write constants or entire expressions. Write will output whatever follows it to the screen for the user to see. Here are several write examples.

write x

write 6

write (x+8)*6

Note that all variables or constants must be declared before using them just like in any statement.

Call

The last type of statement is used to call procedures. A procedure is a subprogram that executes a section of code. They are explained in detail in the section. Here is an example of call:

```
call one
```

Like all names the procedure “one” must have been declared first, explained in the procedure section, to be used. An error will happen if call is used on a name that is not a procedure, for example a variable, so you must be careful.

Procedures

Procedures are sub-functions that are used to complete a specific task in the program. They are useful for organizing your code and for controlling where entire tasks are done. Procedures are declared after the variable declarations of your main program and have all the aspects of a main program. That means they can have constants, vars, procedures, and statements of their own. Here is an example of a program which has one procedure called “one”:

```
const x, y ;

var z;

procedure one ;

        const d ;

        var f ;

;
```

Notice that the word procedure starts the declaration. It is followed by a name for the procedure, then a semicolon, then the code of the procedure which works like a normal program, and then a final semicolon to signify the end of the procedure.

Since a procedure works the same way as a regular program you can have procedures inside of procedures as follows:

```
procedure one ;
```

```
        var y ;
        procedure two ;
                const x = 6;
                y := x + 4
        ;
        begin
                call  two ;
                write y
        end
;
call one.
```

This program will print the value 10 to the screen.

Full Programs

Creating programs for PL/O is simple using a text editor. All thats needed is to type the program, save it as a .txt file, and then run it through the compiler/virtual machine which will be explained in the next section.

In order to create a program using a text editor, the proper form of a PL/O program must be used. Programs have constant declaration, var declaration, procedure declaration (all of which are optional), and then a statement (also optional) followed by a period. Here is an example of a simple full program:

```
var x ;
begin
        x:=5 ;
        write x
end.
```

This program assigns the value 5 to x and then prints x to the screen.

Programs can be more advanced as well. Here are some examples of some more advanced programs and descriptions of what they do.

```
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
    arg := 6;

    call fac;
    write result;
end.
```

This program calculates 6 factorial using a procedure. The variable arg can be changed to different values in order to calculate different numbers.

```
var input, ans;
```

```
begin
    read input;
    if input > 5 then
        ans := 5
    else if input > 4 then
        ans := 4
    else if input > 3 then
        ans := 3
    else
        ans := 0;


    write ans;
end.
```

This program asks the user for the number and then uses nested if then else statements to produce different results based on the user's input into the program.

Compiling

In order to run the program it must use the compiler/virtual machine program on a unix machine. The system used for the explanations in this section will be the UCF Eustis unix system.

The first step once you have written your program in the .txt file is to load the file onto the Eustis server. In the same location as that folder the compiler needs to be loaded. The compiler comes with a .c file and then 3 .h files and all of them need to be in the same location as the input program. Once both of those are in the same location, the compiler needs to be compiled with this command in the terminal:

```
gcc -o <runfile> main.c
```

The <runfile> can be any name you want it to be. This command will create a .exe file in the directory that will be used to run the program. Next use the command:

```
./run input.txt
```

Run is the name of the runfile that you used in the previous command and input.txt is the name of the input file of the program you coded. This command will run the program you inputed and then output any output that is demanded in the program.

The compiler also have several advanced commands that can be used to customize and produce additional output. These commands are -l, which outputs the parsed tokens of your program, -a, which outputs the intermediate assembly, and -v which will output the stack trace of your program in the virtual machine. The commands are added on top the base command used previously. Here is an example of a command to print the virtual machine stack and the generated assembly.

```
./run input.txt -v -a
```

If the program that was inputed into the compiler is not syntactically correct, which means it is missing a symbol or something else is out of place, an error will be printed to the screen with a message saying what is wrong with the program. There are many types of errors from forgetting to end the program with a period or using a variable before declaring it. Here is a list of the possible errors and example code to show how they occur:

List of Compiler Errors

period expected

```
var a, b, answer;
```

const/var must be followed by an identifier

```
var .
```

identifier must be followed by an =

```
const a 4;.
```

semicolon or comma missing

```
var a, b, answer.
```

undelcared identifier

```
var a, b;
begin
    read a;
    read b;
```

```
        answer := a-b;
    end.
```

assignment to constant not allowed

```
    const c = 4 ;
    var a, b, answer;
    begin
        read a;
        read b;
        c  := a-b;
    end.
```

assignment operator expected

```
    var a, b, answer;
    begin
        read a;
        read b;
        answer  a-b;
        write answer;
    end.
```

then expected

```
    var a, b, answer;
    begin
        read a;
        read b;
        if (a = 4)
        answer := a-b;
        write answer;
    end.
```

right parenthesis missing

```
    var a, b, answer;
    begin
        read a;
        read b;
        answer := (a-b;
        write answer;
    end.
```

variable expected

```
    const c = 4
    var a, b, answer;
    begin
        read c;
        read b;
        answer := a-b;
        write answer;
    end.
```