User Manual

for PL/0 Compiler

How to use the PL/0 compiler:

Before using the compiler, make sure that the following files are in the same directory:

- "CompilerDriver.c" (the driver program)
- "LexicalAnalyzer.h" (converts code into a list of lexemes)
- "Parser.h" (converts list of lexemes into intermediate code)
- "VM.h" (converts the intermediate code into assembly and gives a trace of the code execution)

Once all files are in the same directory, create a file in that same directory labeled as "input.txt". This file will hold the starting PL/0 code before it is to be converted by the driver program.

How to compile the PL/0 compiler:

When the previous five files are in the same directory, all you need is any sort of terminal to compile and run the files. *Note* This works best in a Unix system like the Eustis server. Any terminal will work for now, but the commands may differ. Now traverse to the folder containing the fives files using the terminal (e.i using "cd <folder-name/destination>" command) and compile the driver program by typing in the following:

```
gcc –o <name-here> CompilerDriver.c
```

"<name-here>" can be any name that you want to give the executable that will now be created after compiling.

How to run the PL/0 compiler:

Once the executable is created, there are two ways the run the executable:

```
./<name-here> -l        /*This will run and print out the lexemelist
./<name-here> -a        /*This will run and print out the
                        intermediate code
```

When the program executes, it will output the information respective to the parameter that you ran with above. If any errors occurred during execution, the program will notify you about the errors and it will immediately exit; otherwise it will just print out the information above and notify you that everything went ok.

After the program has successfully exited without errors, you will notice that there are extra files in same directory. The following is a list of all the new files that you will notice and what information they hold related to the program:

- “cleaninput.txt” (the given code with comments removed)
- “lexemetable.txt” (shows the conversion of each token in code into its respective lexeme)
- “lexemelist.txt” (list of lexemes that breaks the code down into only numbers and variable names)
- “tokenOutput.txt” (shows the list of lexemes and any errors that occurred within the given code)
- “symlist.txt” (stores information of each variable declared within the given code)
- “mcode.txt” (the given code translated into intermediate code)

- “stacktrace.txt” (the given code translated into assembly code and simulates a trace through the assembly code using a stack)

How to use the PL/0 language:

The PL/0 language is very specific with its syntax. One thing to note is that the program must contain a ‘.’ at the end to signify the end of the program.

Statements:

In order to do something in a PL/0 program, we must have a statement. There are different kinds of statements, but the following examples are some assignment statements that you may see:

```
var x;      /* x is declared as of “var” type
x := 7;     /* x is equal to 7
var y;      /* y is declared as of “var” type
```

Declarations:

Any constant in a program must be declared first with the following syntax:

```
const x = 0, y = 1;
```

Declarations of variables follow constants immediately using “var”:

```
var a, b, c;
```

Declarations of procedures follow variables using the keyword “procedure”:

```
procedure Neg;
    begin
        y:= 0-x;
    end;
```

Procedures work like subprograms within a program and can have constants and variables within it, as well as loops, conditional statements, or even other procedures.

Statements:

Statements add a variety of options into your program such as I/O or conditional loops. To read in a value, you can simply use the "read" keyword; similarly to print a variable's value, you can use "write".

```
var x;
begin
    read x;
    write x;
end.
```

You can group multiple statements in code together using the "begin" and "end" keywords shown above. Loops can also be put together using the "begin" and "end" keywords as follows:

```
if x = y then
    begin
        y := y + 2;
        x := x + 1;
    end;
```

Another example is the “while-do” loop:

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

You can also choose from the following relational operators to compare variables for these conditional loops:

Relational operators	meaning
=	equal to
<	less than
>	greater than
<>	not equal to
>=	greater than or equal to
<=	less than or equal to

Expanding if-loops, you can also add an “else” portion as an extra case if the “if” statement was not executed.

```
if x < y then
    write x
else
    write y;
```

Expressions:

Expressions like “+”, “-“, “*”, and “/” allow you to use mathematical operations to manipulate data. These can be combined with parenthesis to create complex expressions or to just notify the order of operation.

```
    x := 6(y + a) / (c * z - 3);
    y := y + 1;
```

Comments:

Comments are used in programs in order to explain the actions of certain lines of code and allowing the compiler to skip these lines so that syntax is not a worry. Comments must be enclosed within /* */.

Program Example:

```
const beginningX= 3;
var facRes, facParam;
procedure Factorial;
var myFacParam;
begin
    if facParam > 0 then begin
```

```
            myFacParam:= facParam;

            facParam:= facParam*2/2 -1*1;

            call Factorial;

            if myFacParam <> facParam +1 then /*This should never be
            taken*/
            begin

                facParam:=0;

            end;

            facRes:= facRes * (facParam+1);

            facParam:= myFacParam;

        end

        else

            facRes:= 1;

end;

begin

        facParam:= beginningX; /*Just to have done something with a
constant as well*/

        call Factorial

end.
```