PL/0 Programming Guide

This is a guide designed to teach and assist programmers interested in using the PL/0 Language with the included PL/0 compiler

Samir Ketema
COP 3402
July 22, 2014

Table of Contents

Introduction to PL/0

What is PL/0?

PL/0 is a programming language that is simple to understand. This is a great programming language to get started to learn programming in and how some of the fundamentals of programming works. Its concise, straightforward grammar will keep you on a good track to learn some of the basics of both programming and systems software without frustration.

We included a compiler that contains three parts to it: the lexical analyzer, the code generator/parser, and a PL/0 Virtual Machine. This will ensure that your code is read syntactically, analyzed, assembled, and ran in the virtual machine. While it isn't imperative to understand how the compiler works, it will help you tremendously in making sure you're programming correctly.

However, if you aren't interested in the compiler, do not worry! The compiler will catch many errors for you! We will not let you run a syntactically incorrect PL/0 program.

Anyways, let's start with the Pl/0 basics!

Sample PL/0 Program

```
var x, y;
begin
        y := 3;
        x := y + 56;
end.
```

There is a common structure to the program. Here is the structure of this program:

1. Variables are declared at the beginning of this program
2. The program is encapsulated with a "begin" and "end."
3. Expressions within the "begin" and "end."

Although this may seem like a very arbitrary program, it is simple. Variables x and y are declared, and they are changed later on with the expressions between the begin and end. We will now go into detail how each piece of this program works.

We can execute this program by saving it in a text file named "input.txt" for the compiler. Follow the instructions below to compile the program in Eustis.

To compile the Pl/0 Compiler in Eustis (using gcc):

```
gcc -o compiler compiler.c -lm
```

To run the PL/0 Compiler in Eustis, replace the "input.txt" with any input file you wish:

```
./compiler
```

If you wish to print either the Lexemes/Symbol table, Intermediate code, or machine stack, add compiler directives –l, -a, and/or –v to the program such as shown below:

```
./compiler –l –a –b   (This will print all three)
```

To view output files (for example, output.txt) in vi:

```
vi output.txt
```

In the output.txt file, you will be able to find the resulting machine stack and code executed by the included PL/0 virtual machine. You may also print to the screen or input from it using the PL/0 language as well.

Keep this in mind: if you end up input/output in your programs, just type in what values you may need and press enter. This is how the program will continue running. Other than that, once you included your input.txt file and ran the compiler, you will be able to see any necessary output files or screen output automatically.

If you ever have any trouble executing programs or with errors, please look at the common error document included.

begin & end.

Every program is encapsulated in a “begin” and “end.”. This marks the beginning and ending of a program.

For example:

```
begin
	/* line 1 */
end.

/*line 2*/
```

If we were to have real lines of PL/0 code here, the program would not work because line 2 is outside of the program. If we removed line 2, line 1 would execute correctly. Keep in mind that you must have the period after the “end”. Without this, your program will not compile fully.

var & const

Variables and constants are imperative to programming skills. We will use a variable x here.

Example:

```
var x;
const y=4;
begin
          x := 56;
         x := y;
end.
```

Here, we declared a variable named x. We call these variables because they store values that can vary throughout the program. x can store an integer value positive or negative. We then changed the value of x to be 56.

We also declared a constant, y. Constants do not change at any point of the program, so do not try to assign any values to the constant y. In this program, y is declared to be equal to 4, so it will remain 4 throughout the program.

x is then changed to be equal to the constant y, which is 4.

if-then & relation operators

If-then statements in the PL/0 language are imperative to create good programs. We can set a specific condition to execute a line. Confused? Look below:

```
var x;
const y=4;
begin
          x:=0;
          if x = 0 then x :=y;
end.
```

In this program, we use the if statement to execute the x:= y line. We compare x to 0, and see if they are equal to each other. This is called a relation operator (rel-op in shorthand). We check if x is indeed equal to the value of zero. Since this is the case, we change x to be equal to the value of y, 4.

If we used x as a value of 1 instead of zero, then x would have remained 1 by the end of the program. This is a huge part of writing algorithms as programs. We are basically creating branches from the same node of a square/circle if we were to draw a flowchart of this program.

while-do

We can actually have a program execute many lines repeatedly (based on a condition).

```
var x;
begin
        x:=100;
        while x > 0
        do x :=x - 1;
end.
```

This program will execute x:x-1 until x becomes -1. This is very useful because you can repeatedly execute mathematical, I/O, and many other expressions/lines.

procedures

We can have procedures, which are basically sub-programs or functions of a program. These will allow us to call a block of code whenever we need to.

```
var x;
procedure A;
begin
  x := 5;
end;
begin
  x:=3;
  call A;
end.
```

This program will call the procedure declared as “A” after setting x equal to 3. Since the procedure then assigns the value 5 to x, the program will end with x being equal to 5. Note that once the procedure is declared, a begin is listed, the code for the procedure comes right after, then an “end;” follows. Do not confuse this with “end.”. “end.” is specifically for the end of the end of the entire program. When using multiple procedures, make sure to have the “begin” following immediately after it and to make sure to only recursively call programs that are statically enclosed with it. You can experiment with this since the compiler will detect incorrect references to procedures.

Appendix A: PL/0 EBNF Grammar

EBNF (Extended Backus-Naur Form) of PL/0:

program ::= block "**.**" **.**
block ::= const-declaration var-declaration procedure-declaration statement**.**
constdeclaration ::= ["**const**" ident "**=**" number {"**,**" ident "**=**" number} "**;**"]**.**
var-declaration ::= ["**var** "ident {"**,**" ident} “**;**"]**.**
procedure-declaration ::= { "**procedure**" ident "**;**" block "**;**" }
statement ::= [ident "**:=**" expression
| "**call**" ident
| "**begin**" statement { "**;**" statement } "**end**"
| "**if**" condition "**then**" statement ["**else**" statement]
| "**while**" condition "**do**" statement
| "**read**" ident
| "**write**" expression
| **e**] **.**
condition ::= "**odd**" expression
| expression rel-op expression**.**
rel-op ::= "**=**"|“**!=**"|"**<**"|"**<=**"|"**>**"|"**>=**“**.**
expression ::= ["**+**"|"**-**"] term { ("**+**"|"**-**") term}**.**
term ::= factor {("*"|"**/**") factor}**.**
factor ::= ident | number | "**(**" expression "**)**“**.**
number ::= digit {digit}**.**
ident ::= letter {letter | digit}**.**
digit ;;= "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**“**.**
letter ::= "**a**" | "**b**" | … | "**y**" | "**z**" | "**A**" | "**B**" | ... | "**Y**" | "**Z**"**.**

Based on Wirth's definition for EBNF we have the following rule:

[ ] means an optional item.

{ } means repeat 0 or more times.

Terminal symbols are enclosed in quote marks.

A period is used to indicate the end of the definition of a syntactic class.