PL/0 User's Manual

Authors: Dave Nandlall & Barry Shevlin

This page is intentionally left blank

Table of Contents

1.0 The PL/0 Programming Language....4
1.1 PL/0 EBNF Grammar....4
1.2 Data Types....5
- 1.2.1 Constant Declarations....5
- 1.2.2 Integer Declarations....6
- 1.2.3 Procedure Declarations....7

1.3 Expressions....8
1.4 Statements....8
- 1.4.1 Blocks....8
- 1.4.2 Assignments....8
- 1.4.3 Conditional Statements and Relational Operators....9
- 1.4.4 Loops… 10
- 1.4.5 Calling Procedures....11

1.5 Advanced Procedure Methods in PL/0....12
- 1.5.1 Recursive Procedures....12
- 1.5.2 Nested Procedures....13

2.0 Compiling and Executing PL/0 Programs....14
- 2.0.1 Building the PL/0 Compiler....14
- 2.0.2 Running Programs in PL/0 with the PL/0 Compiler....15

3.0 References....17
- 3.0.1 PL/0 EBNF Grammar....17
- 3.0.2 Table of Reserved Words and Tokens....18
- 3.0.3 PL/0 Instruction Set Architecture....19

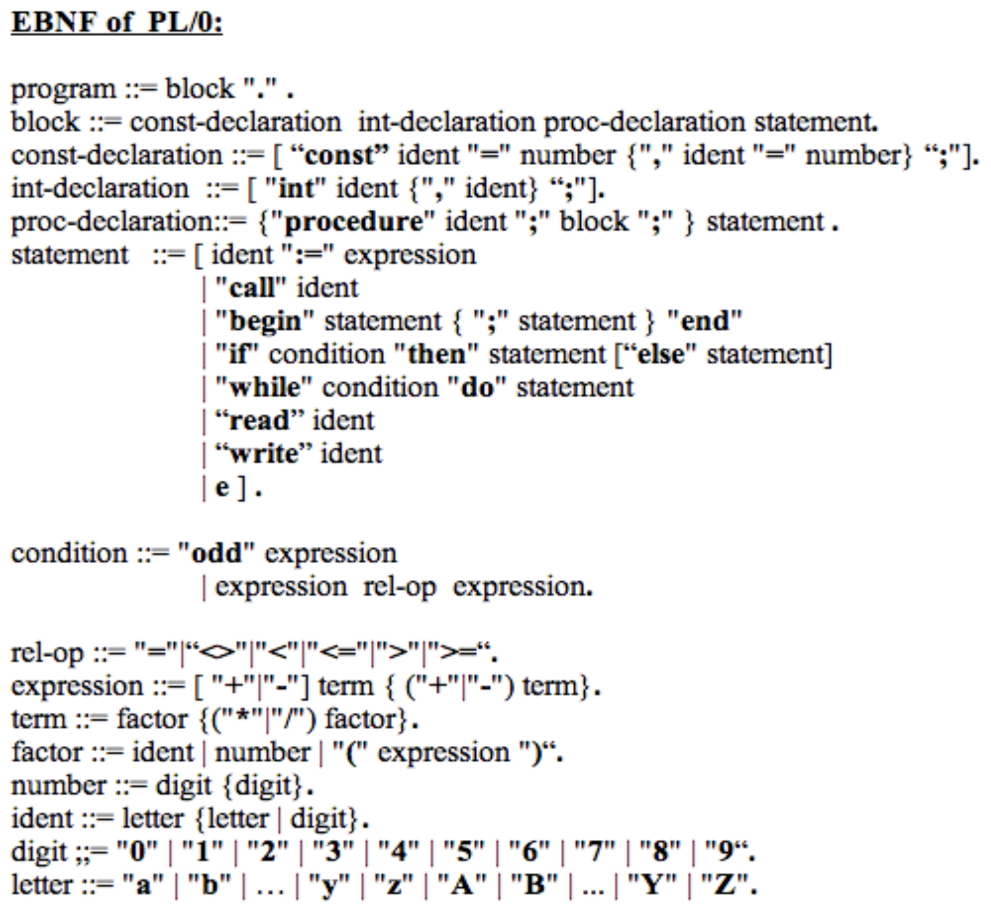
1.0 The PL/0 Programming Language

PL/0 is a fairly simple computer programming language, similar to the Pascal programming language. It serves more for educational purposes to teach students how to build a compiler. The PL/0 programming language is a fairly simple language, in that it supports very few data types, control-flow constructs, sub-routine constructs and very few arithmetic operations.

1.1 PL/0 EBNF Grammar

The PL/0 Programming language has a specific grammar that it follows in which a programmer must obey when formulating syntactically correct programs. Listed below in Figure 1, is an example of the syntactic rules of this particular programming language. The syntax notation is based on EBNF or Extended Backus-Naur Form, which is a family of syntax notations, which expresses context free grammar.

Figure 1 – Example of the PL/0 EBNF Programming Language Grammar



EBNF of PL/0:

```
program ::= block "." .
block ::= const-declaration  int-declaration proc-declaration statement.
const-declaration ::= [ "const" ident "=" number {"," ident "=" number} ";"].
int-declaration  ::= [ "int" ident {"," ident} ";"].
proc-declaration::= {"procedure" ident ";" block ";" } statement .
statement   ::= [ ident ":=" expression
                | "call" ident
                | "begin" statement { ";" statement } "end"
                | "if" condition "then" statement ["else" statement]
                | "while" condition "do" statement
                | "read" ident
                | "write" ident
                | 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".
```

1.2 Data Types

In this section, we will discuss the supported primitive data types that the PL/0 Programming Language currently supports which are of the following:

- Constant Declarations (keyword: **const**)
- Integer Declarations (keyword: **int**)
- Procedure Declarations (keyword: **procedure**)

Following these specific data types are identifiers. An identifier is a specific name, which can be no more than 11 characters in length. Identifiers cannot start with numbers, but rather the can only start with letters. For example, a valid identifier name could be x1, but could NOT be 1x. In addition, an identifiers name cannot be any of the reserved keywords, which are listed in Appendix A. In essence, identifiers act as temporary storage, and its data type reflects the type of information it contains inside of it.

1.2.1 Constant Declarations

Constant declarations are of an integer type. In a program, constants can only be declared once, and are unchangeable. Constant are defined and the value remains "constant" throughout the program, for whatever use. Below are some examples of how constants can be declared in a program.

Constants can be declared on one line:

- const X = 1, Y = 2;

Constants can be declared on separate lines:

- const X = 1;
- const Y = 2;

Each of these following const declarations define the same values, X = 1 and Y = 2. In addition, it is absolutely necessary to end a constant declaration with a semicolon, as this is a terminal symbol signifying that this completes an assignment to a constant declaration. After declaring a constant, you can use them such as:

variable := X + Y; (which is equivalent to saying -> "variable" gets the value of 1+2;)

The constants defined value will be loaded by the compiler during the compilation process and will assert the specific values to the constants names.

1.2.2 Integer Declarations

Integer declarations are changeable, in such that the programmer can define and reassign values to these variables throughout their program. Integers can be declared anywhere in a PL/0 program. Integers are declared with the keyword int, which is then followed by an identifier, which is a name to hold the specific integer value for that variable. To assign a value to an int declaration one must use the becomes symbol := which assigns the value to the specific integer identifier. Listed below is an example of how to declare integers in a PL/0 program.

Declaring an Integer on a Single Line

- int x , y, z;

Declaring an integer on Separate Lines

- int x;
- int y;
- int z;

Assigning a Value to an Integer

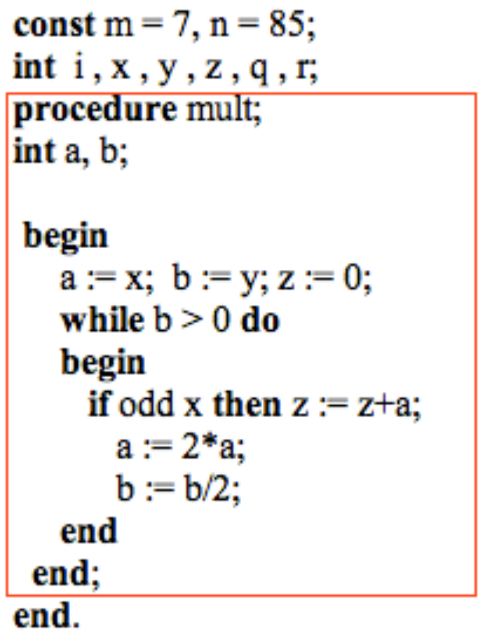
- int x, y; -> Declare integers
- x := 3; -> (Assign Values) - > " x holds the number 3 "
- y := x; -> (Assign Values) - > " y becomes x which is the value 3 "

Both of the examples above show you how to properly declare an integer value. Similarly to the constant declarations, integer declarations must end with a semicolon to complete the assignment of that value to the variable.

1.2.3 Procedure Declarations

Procedure declarations represent subroutines or sub-programs within another program. Procedures help designate different tasks to a main program and its computations. Figure 2 provides an example of a procedure in the PL/0 programming language.

Figure 2 - Example of a Procedure Declaration



```
const m = 7, n = 85;
int  i , x , y , z , q , r;
procedure mult;
int a, b;

 begin
    a := x;  b := y; z := 0;
    while b > 0 do
    begin
      if odd x then z := z+a;
        a := 2*a;
        b := b/2;
    end
  end;
end.
```

As we can see in Figure 2, the body of the PROCEDURE declaration is encapsulated within the red box. The PROCEDURE subroutine mult begins with the declaration of a procedure signified by the keyword **"procedure"**, followed by the name given to that procedure **"mult"** and finally terminates with a semicolon**.** An identifier must follow procedure names. The code within the red boundaries is the specific tasks that procedure must perform when it is used within the PL/0 program.

1.3 Expressions

Expressions in the PL/0 Programming Language are mathematical expressions, in which a value can either be returned or be represented. Expressions can be composed of constant and variable identifiers, number literals, or arithmetic symbols such as +, -, *, /, (, and). In the PL/0 Programming Language, the standard order of operations is always obeyed in the grammar. An operator of higher precedence will always be evaluated first.

1.4 Statements

Statements in a program help the program to efficiently execute its task with many multi-purpose methods.

1.4.1 Blocks

Blocks are grouped statements, which begin with the keyword “begin” and end with the keyword “end”. They are a group of particular instructions separated by a semicolon. A block can be nested inside of other statements, such as a while loop or like in figure 2.

1.4.2 Assignments

Assignments are used to assign a specific value to an integer declaration of a constant declaration. The assignment operator, which is the becomes symbol := assigns a specific value to a declared integer or constant. The assignment to a specific variable can only be used inside a statement.

1.4.3 Conditional Statements and Relational Operators

Conditionals are used in PL/0 to allow the programmer to test a specific condition, and based on that specific condition, whether the condition is true or false, execute a select group of instructions. PL/0 uses the conditionals with the keywords "if", "then", and "else". In addition, there is the keyword odd, which will return true or false based on an expression. If the expression is odd, then return true, or else return false. Furthermore, conditionals use relational operators to test a specific condition. Listed below are some examples of conditional statements in PL/0, and a list of the different relational operators used in conditional statements.

Relational Operators

- > (Greater Than)
- >= (Greater Than or Equal To)
- < (Less Than)
- <= (Less Than or Equal To)
- <> (Not Equal To)
- = (Equal)

Example of a Conditional Statement (if-then construct)

```
•  if  x = 1
     then write x;
```

Example of a Conditional Statement (if-else construct)

```
•  if x = 1
       then write x;
   else write x+1;
```

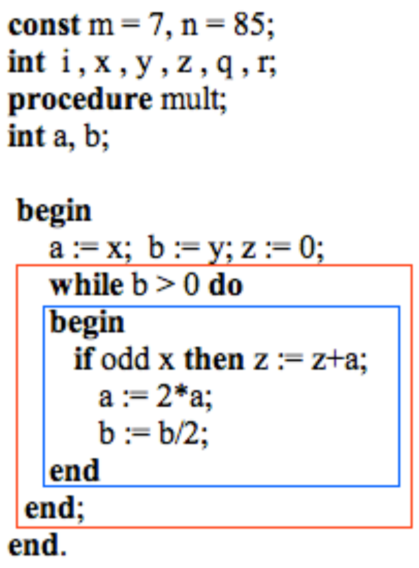
Example of Conditional Statement (Nested Conditionals)

```
•  if x = 1 then write x;
•  else if x = 2 then write x+1;
•  else if x = 3 then write x+2;
•  else write x + 3;
```

1.4.4 Loops

The loop construct is very useful in that it provides the programmer to perform a sequence of instructions until a specific condition is reached. It allows the programmer to iterate over a particular range of numbers and perform many different kinds of computations. The PL/0 programming language supports the while loop construct. Figure 3, shows an example of a while loop construct in the PL/0 Programming Language.

Figure 3. Example of a While Loop Construct



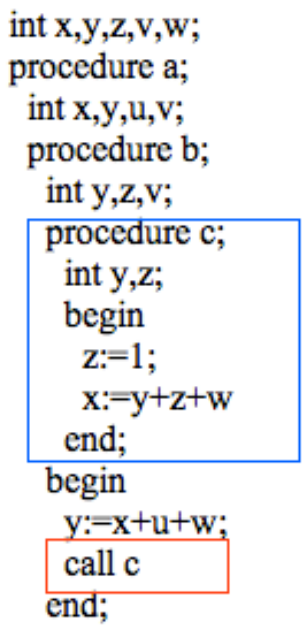
```
const m = 7, n = 85;
int  i , x , y , z , q , r;
procedure mult;
int a, b;

 begin
    a := x;  b := y; z := 0;
    while b > 0 do
    begin
      if odd x then z := z+a;
        a := 2*a;
        b := b/2;
    end
  end;
end.
```

In Figure 3, the while loop construct in the red box keeps iterating so long as the value of the identifier b is greater than (>) zero. While the value of b > 0, then keep evaluating the condition “if - then” in the blue box which states “if” b is an odd number, “then” compute the statements (z: = z + a), otherwise if b is not odd compute the statements (a := 2 * a) and (b := b / 2). If ever the value of b is less than (<) 0 when we are computing any of the statements, then the while loop will finish its execution of the code that it contains.

1.4.5 Calling Procedures

In order to invoke a specific procedure in the PL/0 Programming Language, the keyword "call" is used. The CALL keyword, calls the name of the procedure, which will then begin executing code within that specific procedure. There is no way of explicitly passing a parameter to a procedure, however the procedure has access to any variables that are declared within their specific scope. For Figure 4, we will reuse the example from Figure 2 to give an example of how procedure calls are executed.

Figure 4. Example of Calling a Procedure



```
int x,y,z,v,w;
procedure a;
  int x,y,u,v;
  procedure b;
    int y,z,v;
    procedure c;
      int y,z;
      begin
        z:=1;
        x:=y+z+w
      end;
    begin
      y:=x+u+w;
      call c
    end;
```

In Figure 4, the PROCEDURE C is highlighted within the blue box, while the red box denotes the CALL to PROCEDURE C. When the PL/0 compiler reads the CALL statement, it will perform the tasks that are given to the specific procedure name. In Figure 2, when the call to c is made, the execution of the statements denoted within the blue box will execute, and once finished will terminate.

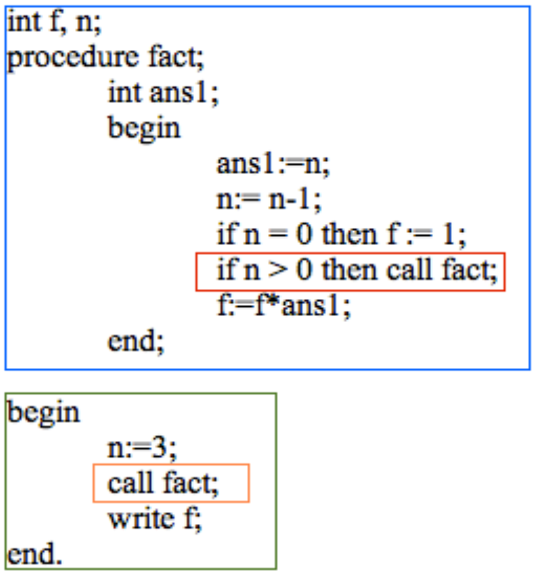
1.5 Advanced Procedure Methods in PL/0

The PL/0 Programming Language has support for nested and recursive procedures. The following sections give some examples as well as some detailed explanations on how these two types of procedures can be constructed as well as implemented.

1.5.1 Recursive Procedures

The PL/0 Programming Language also supports the use of recursive procedures. A recursive procedure is a procedure that it declares within the PL/0 program, which keeps calling itself over and over again until it cannot execute anymore. Until its execution phase is complete, the recursive procedure will terminate and return control to the caller, whom made the initial call to it. Figure 5 show an example of a recursive procedure, which calculates the factorial of a number.

Figure 5 Example of a Recursive Procedure in PL/0



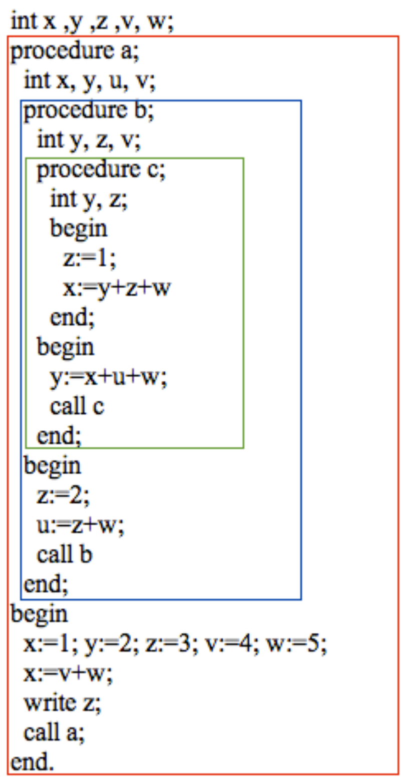
```
int f, n;
procedure fact;
        int ans1;
        begin
                ans1:=n;
                n:= n-1;
                if n = 0 then f := 1;
                if n > 0 then call fact;
                f:=f*ans1;
        end;
begin
        n:=3;
        call fact;
        write f;
end.
```

In figure 5, the blue box encapsulated the entire declaration of the procedure fact with all of the variables local to that procedure. Within the green box is where our first call to fact is made denoted within the orange box as well as setting the value for n to be “3”. When the call fact is made within this orange box the code within the blue box begins its execution phase. Within the blue box the factorial procedure makes a call to itself denoted by the red box so long as the value of n is greater than (>) 0. This call to fact within the red box will continue to occur, so long as the value of n is less than (<) 0. When the value of n is finally less than (<) 0, this recursive procedure will terminate, and will not continue to call itself.

1.5.2 – Nested Procedures

The PL/0 Programming Language allows for the use of nested procedures. The benefit of using nested procedures is that it is useful for dividing different tasks of a program into smaller sub tasks. This allows the programmer more control when creating their programs, and allows the program itself to run more efficiently. Nested procedures allow for better control flow within the program. Figure 6 gives an example of a nested procedure.

Figure 6 Example of a Nested Procedure in PL/0



```
int x ,y ,z ,v, w;
procedure a;
 int x, y, u, v;
 procedure b;
  int y, z, v;
  procedure c;
   int y, z;
   begin
    z:=1;
    x:=y+z+w
   end;
  begin
   y:=x+u+w;
   call c
  end;
 begin
  z:=2;
  u:=z+w;
  call b
 end;
begin
 x:=1; y:=2; z:=3; v:=4; w:=5;
 x:=v+w;
 write z;
 call a;
end.
```

In Figure 6, the first procedure, which gets executed, is procedure c, which is encapsulated within the green box. Upon completion of procedure c, the next procedure, namely procedure b will be executed which is denoted within the blue box. When procedure b finishes its specific tasks, the control goes to procedure a, which is denoted within the red box. Upon completion of procedure a, the program ends its final execution and displays its results. As you can see these nested procedures help the entire program complete its overall task in smaller divided sub tasks within each procedure.

2.0 Compiling and Executing PL/0 Programs

The PL/0 compiler program is a program used to compile and run any program written in the PL/0 Programming Language. This section shows how to start compiling and executing your written PL/0 Programs.

2.0.1 Building the PL/0 Compiler

In order to build the PL/0 compiler, you will need to have some experience using the command line on a UNIX like operating system. In addition, you will need GCC and GNU Make prior to building the PL/0 compiler. Provided that you do not have any experience using a UNIX like operating system, some helpful web links are posted below.

Basic Unix Commands: http://mally.stanford.edu/~sr/computing/basic-unix.html
GCC: http://gcc.gnu.org/
Make: http://www.gnu.org/software/make/

To build the PL/0 compiler, do the following:

1. Obtain a copy of the source code for the PL/0 compiler named "pl0"
2. Open a terminal window (command line) utility on your computer
3. Navigate on the terminal to the PL/0 compiler folder, the "pl0" folder
   To get to the compiler folder, use these commands on the terminal window:
   - Type `cd ~/Desktop/pl0/src` and hit enter
   - cd = change directory
   - ~ = Tilde
   - /Desktop/ = the main directory where the compiler is located
     This could vary depending on where the compiler folder is stored
     For example, if it is in a Documents folder, change /Desktop/ to /Documents/
   - /pl0/ = the name of the "pl0" compiler folder
   - /src/ = the source code files for the PL/0 Compiler

4. Now type the command:
   - `gcc –c compile.c` and hit enter

   This command will compile all of the source code for the PL/0 Compiler and build the PL/0 compiler.

5. After you compile all the source code files run the command:
   - `make` and hit enter

This command will create a main executable file for the PL/0 Compiler to run. After entering the make command you are now ready to run your PL/0 programs.

2.0.2 Running Programs in PL/0 with the PL/0 Compiler

After building the PL/0 Compiler in Section 2.0.1, you will now be able to run a PL/0 program. In order to run your PL/0 written programs, you need to enter some compiler directives, which tell the PL/0 Compiler what information about this written program you are trying to view. Listed below are some compiler directives that the PL/0 Compiler understands:

- ./compile –l (ell) instructs the PL/0 compiler to display your written PL/0 program into its primary elementary tokens. Namely your programs go through a lexical analyzer, which converts each word, identifiers, and special symbols into a simpler representation in the form of tokens.
- ./compile –a instructs the PL/0 compiler to display the generated machine code of the written PL/0 program. This process is done after the lexical analyzer phase, in which It takes each elementary token and creates machine code, so that the virtual machine can execute your written PL/0 Programs.
- ./compile –v instructs the PL/0 compiler to display the stack trace of your written PL/0 programs. This shows the final execution on the stack of your written PL/0 program.

In addition to these three different compiler directives, you can also use the compiler directives

- ./compile –l –a –v which instructs the PL/0 Compiler to display all of the contents of your written PL/0 programs which include the lexical tokens, the generated machine code as well as the program running on the virtual machine.
- ./compile without any flags tell the compiler to print everything including your written PL/0 program, the entire lexical analyzer process, the entire parser and code generation phase, as well as the entire phase of the virtual machine execution.

In the case of an unknown compiler directive, the compiler will notify the user that it does not understand the directive and prints the following

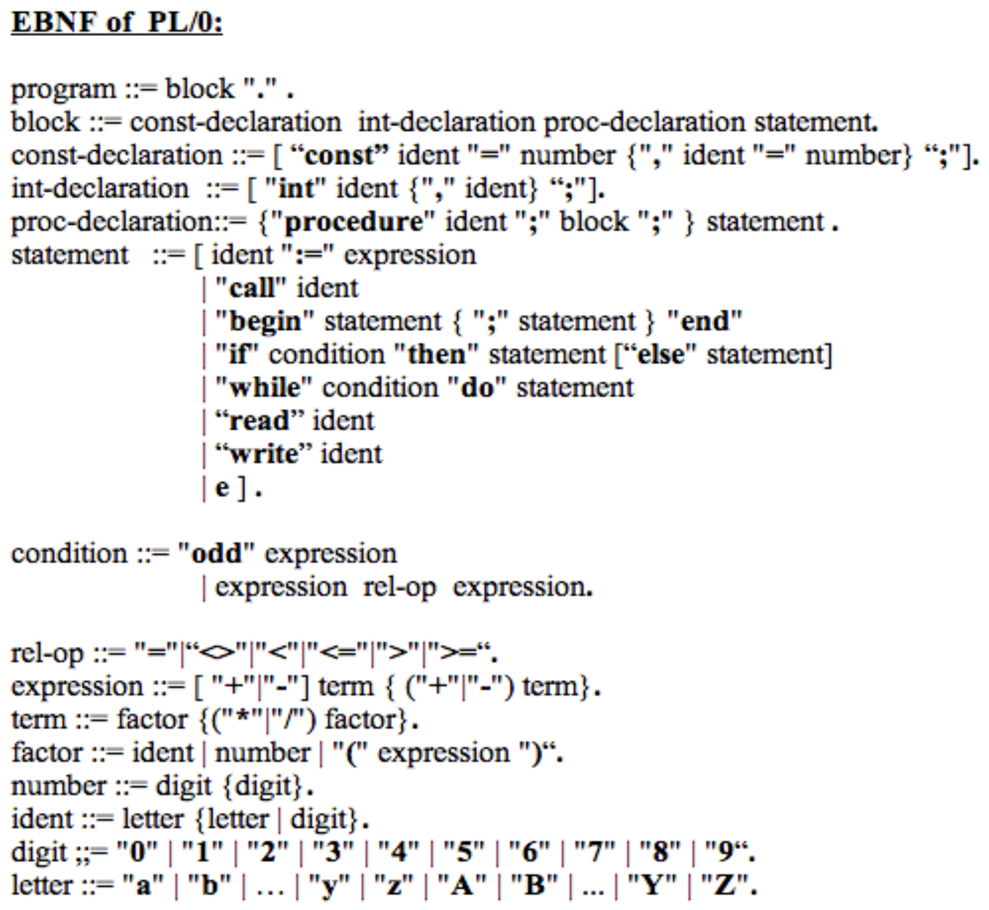
- Bad Directives –“wrong flag”

After entering any one of the valid PL/0 Compiler Directives, the compiler will now ask you to enter the name of the file, in which your PL/0 Program is called. You can now type in the name of your program file, after which the specified information will display on the console.

Furthermore, if you run into any complications running your programs, please view the README.pdf file, located in /pl0/doc/ folder which covers the building and running of the PL/0 compiler in greater detail.

3.0 References

3.0.1 PL/0 EBNF Grammar



EBNF of PL/0:

```
program ::= block "." .
block ::= const-declaration  int-declaration proc-declaration statement.
const-declaration ::= [ “const” ident "=" number {"," ident "=" number} “;"].
int-declaration  ::= [ "int" ident {"," ident} “;"].
proc-declaration::= {"procedure" ident ";" block ";" } statement .
statement   ::= [ ident ":=" expression
                | "call" ident
                | "begin" statement { ";" statement } "end"
                | "if" condition "then" statement [“else" statement]
                | "while" condition "do" statement
                | “read” ident
                | “write” ident
                | 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".
```

3.0.2 Table of Reserved Words and Tokens

Symbol	Internal Name	Internal Value	Description
	nulsym	1	reserved
	Identsym	2	Constants, Integers and Procedure Identifiers
	numbersym	3	Number literals
+	plussym	4	Addition operations in Expressions
-	minussym	5	Subtraction in Expressions
*	multsym	6	Multiplication in Expressions
/	slashsym	7	Division in Expressions
odd	oddsym	8	Determines if an expression is odd
=	eqlsym	9	Checks Equality of Two Expressions
<>	neqsym	10	Checks if two Expressions are Not Equal
<	lessym	11	Checks if Left Expression is Less than Right Expression
<=	leqsym	12	Checks if Left Expression is Less than or Equal to Right Expression
>	gtrsym	13	Checks if Left Expression is Greather Than Right Expression
>=	geqsym	14	Checks if Left Expression is Greater Than or Equal To Right Expression
(	lparentsym	15	Begins a Factor
)	rparentsym	16	Ends a Factor
,	commasym	17	Seperates constants, variable identifiers and their declarations
;	semicolonsym	18	Ends Statements
.	periodsym	19	End of Program
:=	becomessym	20	Variable Assignments
begin	beginsym	21	Begins a block of statements
end	endsym	22	Ends a block of statements
if	ifsym	23	Begins an "if" statement
then	thensym	24	Followed by "if" evaluates a condition
while	whilesym	25	Begins a while loop followed by a codition
do	dosym	26	Part of while loop eval. condition

call	callsym	27	Calls a procedure
const	constsym	28	Begins a constant declaration
int	intsym	29	Begins an integer declaration
procedure	procsym	30	Begins a procedure declaration
write	writesym	31	Outputs the value of an expression
read	readsym	32	Asks user to input a value and assign it to a variable
else	elsesym	33	Optional in that it follows the if-then construct statements

3.0.3 PL/0 Instruction Set Architecture

OP Code	Syntax	Description
1	LIT 0, M	Push a constant literal value M onto the stack
2	OPR 0, 0	Return to caller from a procedure
2	OPR 0, 1	Negatition, negative value of the top of stack
2	OPR 0, 2	Addition, pop two values add them and push back onto top of stack
2	OPR 0, 3	Subtraction, pop two values, subtract them and push back onto top of stack
2	OPR 0, 4	Multiplication, pop two values, multiply them and push back onto top of stack
2	OPR 0, 5	Division, pop two values, divide them and push back onto top of stack
2	OPR 0, 6	Divisible by 2? Pop stack push 1 if odd 0 if even
2	OPR 0, 7	Modulus, pop two values from stack divide second by first and push the remainder

2	OPR 0, 8	Equality, pop two values from stack push 1 if equal 0 if not
2	OPR 0, 9	Inequality, pop two values from stack push 1 if not equal 0 if not
2	OPR 0, 10	Less than, pop two values from stack push 1 if first is less than second 0 if not
2	OPR 0, 11	Less than Equal To, pop two values from stack push 1 if first is less than or equal to second 0 if not
2	OPR 0, 12	Greater Than, pop two values from stack push 1 if first is greater than second 0 if not
2	OPR 0, 13	Greater Than or Equal to, pop two values from stack push 1 if first is greater than or equal to second 0 if not
3	LOD 0, M	Load a value to the top of the stack from the stack location at offset M from L lexigraphical levels down
4	STO 0, M	Store a value to the top of the stack from the stack location at offset M from L lexigraphical levels down
5	CAL 0, M	Call a procedure at code index M
6	INC 0, M	Allocate M locals (increment sp by M). First four are Static Link (SL), Dynamic Link (DL), Return Address (RA) and Functional Value Return (FV)
7	JMP 0, M	Jump to instruction M
8	JPC 0, M	Jump to instruction M if top stack element is 0
9	SIO 0, 1	Write Top Stack Element to the screen

10	SIO 0, 2	Read Input from the user and store it at the top of the stack.