PL/0 Manual
($Revision: 1.16 $)

Gary T. Leavens
Leavens@ucf.edu

October 15, 2023

1 The language PL/0

PL/0 is a block-structured, procedural, imperative programming language used for COP 3402 (Systems Software) at UCF.

This document describes the syntax and semantics of PL/0.

2 Syntax

The syntax of PL/0 is defined in this section. First the lexical syntax of PL/0 is defined. This is followed by the context-free syntax, which builds on the lexical syntax.

2.1 Lexical Syntax

The lexical syntax of PL/0 is defined by the lexical grammar shown in Figure 1 on the following page. Note that curly brackets are not terminal symbols in this grammar; curly brackets are only used to denote an arbitrary number of repetitions of some nonterminal. In the grammar, some character classes are described in English, these are described in a Roman font between double quotation marks (“ and ”).

All of the terminal symbols that are possible productions of ⟨punctuation⟩, ⟨reserved-word⟩, and ⟨rel-ops⟩ represent tokens in the grammar, but those nonterminals themselves are not used in the contex-free grammar of PL/0.

All characters matched by the nonterminal ⟨ignored⟩ are unused by the context-free grammar, and so should be ignored by the lexer.

2.2 Context-Free Syntax

The context-free syntax of PL/0 is defined by the context-free grammar shown in Figure 2 on page 3. The start symbol of this grammar is ⟨program⟩, which can be a ⟨block⟩ followed by a period.

⟨ident⟩ ::= ⟨letter⟩ {⟨letter-or-digit⟩}
⟨letter⟩ ::= `_` | `a` | `b` | `...` | `y` | `z` | `A` | `B` | `...` | `Y` | `Z`
⟨number⟩ ::= ⟨digit⟩ {⟨digit⟩}
⟨digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨letter-or-digit⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨plus⟩ ::= `+`
⟨minus⟩ ::= `-`
⟨mult⟩ ::= `*`
⟨div⟩ ::= `/`

⟨punctuation⟩ ::= `.` | `;` | `=` | `,` | `:=` | `(` | `)`
⟨reserved-word⟩ ::= **`const`** | **`var`** | **`procedure`** | **`call`** | **`begin`** | **`end`**
| **`if`** | **`then`** | **`else`** | **`while`** | **`do`** | **`read`** | **`write`** | **`skip`** | **`odd`**
⟨rel-ops⟩ ::= `<>` | `<` | `<=` | `>` | `>=`

⟨ignored⟩ ::= ⟨blank⟩ | ⟨tab⟩ | ⟨vt⟩ | ⟨formfeed⟩ | ⟨eol⟩ | ⟨comment⟩
⟨blank⟩ ::= "A space character (ASCII 32)"
⟨tab⟩ ::= "A horizontal tab character (ASCII 9)"
⟨vt⟩ ::= "A vertical tab character (ASCII 11)"
⟨formfeed⟩ ::= "A formfeed character (ASCII 12)"
⟨newline⟩ ::= "A newline character (ASCII 10)"
⟨cr⟩ ::= "A carriage return character (ASCII 13)"
⟨eol⟩ ::= ⟨newline⟩ | ⟨cr⟩ ⟨newline⟩
⟨comment⟩ ::= ⟨pound-sign⟩ {⟨non-nl⟩} ⟨newline⟩
⟨pound-sign⟩ ::= `#`
⟨non-nl⟩ ::= "Any character except a newline"

Figure 1: Lexical grammar of PL/0. The grammar uses a `terminal font` for terminal symbols and a **`bold terminal font`** for reserved words. Note that all ASCII letters (a-z and A-Z) are included in the production for ⟨letter⟩. Again, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x.

⟨program⟩ ::= ⟨block⟩ `.`

⟨block⟩ ::= ⟨const-decls⟩ ⟨var-decls⟩ ⟨proc-decls⟩ ⟨stmt⟩

⟨const-decls⟩ ::= { ⟨const-decl⟩ }
⟨const-decl⟩ ::= **`const`** ⟨const-defs⟩ **`;`**
⟨const-defs⟩ ::= ⟨const-def⟩ | ⟨const-defs⟩ **`,`** ⟨const-def⟩
⟨const-def⟩ ::= ⟨ident⟩ `=` ⟨number⟩

⟨var-decls⟩ ::= { ⟨var-decl⟩ }
⟨var-decl⟩ ::= **`var`** ⟨idents⟩ **`;`**
⟨idents⟩ ::= ⟨ident⟩ | ⟨idents⟩ **`,`** ⟨ident⟩

⟨proc-decls⟩ ::= { ⟨proc-decl⟩ }
⟨proc-decl⟩ ::= **`procedure`** ⟨ident⟩ **`;`** ⟨block⟩ **`;`**

⟨stmt⟩ ::= ⟨assign-stmt⟩ | ⟨call-stmt⟩ | ⟨begin-stmt⟩ | ⟨if-stmt⟩
    | ⟨while-stmt⟩ | ⟨read-stmt⟩ | ⟨write-stmt⟩ | ⟨skip-stmt⟩
⟨assign-stmt⟩ ::= ⟨ident⟩ **`:=`** ⟨expr⟩
⟨call-stmt⟩ ::= **`call`** ⟨ident⟩
⟨begin-stmt⟩ ::= **`begin`** ⟨stmts⟩ **`end`**
⟨if-stmt⟩ ::= **`if`** ⟨condition⟩ **`then`** ⟨stmt⟩ **`else`** ⟨stmt⟩
⟨while-stmt⟩ ::= **`while`** ⟨condition⟩ **`do`** ⟨stmt⟩
⟨read-stmt⟩ ::= **`read`** ⟨ident⟩
⟨write-stmt⟩ ::= **`write`** ⟨expr⟩
⟨skip-stmt⟩ ::= **`skip`**
⟨stmts⟩ ::= ⟨stmt⟩ | ⟨stmts⟩ **`;`** ⟨stmt⟩

⟨condition⟩ ::= ⟨odd-condition⟩ | ⟨rel-op-condition⟩
⟨odd-condition⟩ ::= **`odd`** ⟨expr⟩
⟨rel-op-condition⟩ ::= ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= `=` | `<>` | `<` | `<=` | `>` | `>=`

⟨expr⟩ ::= ⟨term⟩ | ⟨expr⟩ ⟨plus⟩ ⟨term⟩ | ⟨expr⟩ ⟨minus⟩ ⟨term⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨mult⟩ ⟨factor⟩ | ⟨term⟩ ⟨div⟩ ⟨factor⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨sign⟩ ⟨number⟩ | `(` ⟨expr⟩ `)`
⟨sign⟩ ::= ⟨plus⟩ | ⟨minus⟩ | ⟨empty⟩
⟨empty⟩ ::=

Figure 2: Context-free grammar for the concrete syntax of (full) PL/0. The grammar uses a `terminal font` for terminal symbols, and a **`bold terminal font`** for reserved words. As in EBNF, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Also note that an **`else`** clause is required in each if-statement.

3 Semantics

This section describes the semantics of PL/0.

Nonterminals discussed in this section refer to the nonterminals in the context-free grammar of PL/0's concrete syntax, as defined in Figure 2 on the preceding page.

A ⟨program⟩ consists of zero-or-more constant declarations (⟨const-decls⟩), zero-or-more variable declarations (⟨var-decls⟩), and zero-or-more procedure declarations (⟨proc-decls⟩), followed by a statement.

In PL/0, all constants and variables denote (32 bit) integers. The execution of a program declares the named constants, variables, and procedures, and initializes the constants and variables. Then it runs the statement.

3.1 Potential Scopes and Declaration Scopes

A (potential) *scope* is a ⟨program⟩ or ⟨block⟩; it is an area of the program's text. When it is clear that we are talking about programs or blocks, we will simply call these "scopes." A PL/0 program can contain nested scopes, as a program or block contains blocks for each of the procedures declared within it.

A *declaration scope* in PL/0 is an area of program text that extends from just after the first mention of the name being declared in a declaration form (i.e., a ⟨const-decl⟩, ⟨var-decl⟩, or ⟨proc-decl⟩) to the end of the surrounding potential scope (i.e., to the end of the surrounding ⟨program⟩ or ⟨block⟩). For example, the PL/0 program in Figure 3 has a nested scope, the ⟨block⟩ that defines the procedure `nested`, and which contains a declaration of `x` that shadows the declaration at the top-level of the program.

```
const x = 10;
procedure nested;
  const x = 3;
  write x;  # writes 3
call nested.
```

Figure 3: A PL/0 program with nested procedure scope and a declaration (of the constant `x` that shadows the declaration at the top level. When run, this procedure would print the number 3 to standard output.

Due to nesting of potential scopes, any declarations of ⟨ident⟩s in a nested block that are also declared in a surrounding potential scope causes a hole in that identifier's declaration scope that is as big as the declaration scope of the inner declaration. (That is, an ⟨ident⟩ may be declared in a nested potential scope even if it is declared in a surrounding potential scope; thus uses of a name redeclared in a nested block refer to the closest textually-surrounding declaration of that name instead of other, shadowed, declarations in surrounding potential scopes.) However, it is an error if an ⟨ident⟩ is declared more than once in a potential scope, as either a constant, a variable, or a procedure.

Since a procedure declaration mentions the name of the procedure being declared before the ⟨block⟩ that defines its meaning, recursive calls of procedures are allowed. For example the PL/0 program shown in Figure 4 on the following page has a recursive procedure `countDown`, which is has legal call statements. However, statements cannot call procedures that have not yet been declared (and thus mutual recursion is not possible).

3.2 Constant Declarations

The nonterminal ⟨const-decls⟩ specifies zero or more constant declarations.

```
var arg;
procedure countDown;
  write arg;
  if arg <= 0
  then skip
  else begin
         arg := arg - 1;
         call countDown
       end;
begin
  arg := 4;
  call countDown;
end.
```

Figure 4: A PL/0 program with a recursive procedure, `countDown`. When run this procedure would print the numbers 4, 3, 2, and then 1.

Each constant declaration, of the form ⟨ident⟩ = ⟨number⟩, declares that ⟨ident⟩ is an integer constant that is initialized to the value given by ⟨number⟩. The scope of such a constant declaration is the area of the surrounding block that follows the declaration.

It is an error for an ⟨ident⟩ to be declared as a constant more than once in a potential scope. It is an error for the program to use the a declared constant's ⟨ident⟩ on the left hand side of an assignment statement or in a read statement.

3.3 Variable Declarations

The nonterminal ⟨var-decls⟩ specifies zero or more variable declarations.

Each variable declaration, of the form ⟨ident⟩, declares that ⟨ident⟩ is an integer variable that is initialized to the value 0.

It is an error for an ⟨ident⟩ to be declared as a variable if it has already been declared as a constant or as a variable in the same potential scope.

Unlike constants, variable names may appear on the left hand side of an assignment statement or in a read statement.

3.4 Procedure Declarations

The nonterminal ⟨proc-decls⟩ specifies zero or more procedure declarations.

Each procedure declaration, of the form **`procedure`** ⟨ident⟩ **`;`** ⟨block⟩ **`;`** declares that ⟨ident⟩ is a procedure that when run executes the following ⟨block⟩; that is, it declares and initializes the constants and variables declared in the ⟨block⟩ and declares the block's procedures and then runs the statement in the ⟨block⟩. Therefore, a procedure executes as if it were a program, although it may use identifiers declared in a surrounding potential scope.

It is an error for an ⟨ident⟩ to be declared as a procedure if it has already been declared as a constant, variable, or procedure in the same potential scope.

Procedure names may not be used on the left hand side of an assignment statement nor may they be used in a read statement.

3.5 Statements

A ⟨block⟩ contains a single statement (⟨stmt⟩) that is run when the block is executed. When a program is run, its block is executed, and thus the statement in that block starts executing.

Assignment Statement An assignment statement has the form ⟨ident⟩ **:=** ⟨expr⟩. It evaluates the expression ⟨expr⟩ to obtain a value and then it assigns it to the variable named by ⟨ident⟩. Thus, immediately after the execution of this statement, the value of the variable ⟨ident⟩ is the value that was obtained for ⟨expr⟩.

It is an error if the left hand side ⟨ident⟩ has not been declared as a variable. (Note that the evaluation of the ⟨expr⟩ may produce runtime errors.)

Call Statement A call of the form **`call`** ⟨ident⟩ executes the ⟨block⟩ declared by the procedure named ⟨ident⟩. (Therefore, it allocates space for the constants and variables declared in that procedure's ⟨block⟩, initializes them, and then executes that ⟨block⟩'s statement.)

It is an error if the ⟨ident⟩ has not been declared as a procedure.

Since procedures in PL/0 do not have formal parameters and do not return results, one can only pass arguments to a procedure and return results using variables that are global to that procedure.

Begin Statement A begin statement has the form **`begin`** $S_1; S_2; \ldots; S_n$ **`end`** (where $n \geq 1$) and is executed by first executing statement S_1, then if S_1 finishes without encountering an error S_2 is executed, and so on, in sequence. Any run-time errors encountered cause the entire compound statement's execution to terminate with that error.

If-Statement An if-statement has the form **`if`** C **`then`** S_1 **`else`** S_2 and is executed by first evaluating the condition C. When C evaluates to true, then S_1 is executed; otherwise, if C evaluates to false (i.e., if it does not encounter an error), then S_2 is executed.

Note that in the concrete syntax there are no parentheses around the condition.

While Statement A while statement has the form **`while`** C **`do`** S and is executed by first evaluating the condition C. If C evaluates to false, then S is not executed and the while statement finishes its execution. When C evaluates to true, then S is executed, followed by the execution of **`while`** C **`do`** S again. Note that C is evaluated each time, not just once.

Again, in the concrete syntax there are no parentheses around the condition.

Read Statement A read statement of the form **`read`** x, where x is a declared variable identifier, reads a single character from standard input and puts its ASCII value into the variable x. The value of x will be set to -1 if an end-of-file or an error is encountered on standard input.

It is an error if x has not been previously declared as a variable.

Write Statement A write statement of the form **`write`** e, first evaluates the expression e, and then writes the decimal form of that that value to standard output (using ASCII characters) followed by a newline character. (This is the same output as would occur for the C statement `fprintf(stdout, "\%d\n", e);`, assuming that the variable `e` was an `int` variable in a C program that held the value of the expression e.)

Skip Statement A skip statement of the form **`skip`** does nothing and does not change the program's state.

3.6 Conditions

A ⟨condition⟩ is an expression that has a Boolean value: either true or false.

Odd Condition A ⟨condition⟩ of the form **`odd`** e first evaluates the expression e. If the value of e is an odd integer (i.e., it is equal to 1 modulo 2), then the value of the condition is true. If the value of e is even, then the value of the condition is false.

Relational Conditions A ⟨condition⟩ of the form $e_1\ r\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the condition as a whole encounters that error.) Then it compares v_1 to v_2 according to the relational operator r, as follows:

- if r is =, then the condition's value is true when v_1 is equal to v_2, and false otherwise.
- if r is <>, then the condition's value is true when v_1 is not equal to v_2, and false when they are equal.
- if r is <, then the condition's value is true when v_1 is strictly less than v_2, and false otherwise.
- if r is <=, then the condition's value is true when v_1 is less than or equal to v_2, and false when $v_1 > v_2$.
- if r is >, then the condition's value is true when v_1 is strictly greater than v_2, and false otherwise.
- if r is >=, then the condition's value is true when v_1 is greater than or equal to v_2, and false when $v_1 < v_2$.

3.7 Expressions

An ⟨expr⟩ of the form $e_1\, o\, e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the expression as a whole encounters that error.) Then it combines v_1 and v_2 according to the operator o, as follows:

- An expression of the form e_1+e_2 (i.e., a binary operator expression where the operator o is +) yields the value of $v_1 + v_2$, according to the semantics of the type **`int`** in C.
- An expression of the form e_1−e_2 yields the value of $v_1 - v_2$, according to the semantics of the type **`int`** in C.
- An expression of the form $e_1{*}e_2$ yields the value of $v_1 \times v_2$, according to the semantics of the type **`int`** in C.
- An expression of the form e_1/e_2 yields the value of v_1/v_2, according to the semantics of the type **`int`** in C. The expression encounters an error if v_2 is zero.

There are also a few other cases of expressions that do not involve binary operators. These have the following semantics:

- An identifier expression, of the form x, has as its value the value of the integer stored in the constant or variable named x whose declaration is found in the closest syntactically surrounding scope.

  It is an error if x has not been previously declared as a constant or variable.

- An expression of the form sn, where s is a $\langle$sign$\rangle$ and n is a $\langle$number$\rangle$ yields the value of the base 10 literal n if the sign s is + or $\langle$empty$\rangle$. However, if the sign s is −, then the value is the negated value of the base 10 literal n according to the semantics of the type **`int`** in C.

- An expression of the form `(`e`)` yields the value of the expression e.