Plum User Manual

Written by Tiger Sachse

2018

What is Plum?

Plum is an interpreter designed to run PL/0 code. It is modal, and is capable of parsing, scanning, compiling, assembling, and executing programs. Compiled programs are executed on a provided stack-based virtual machine. A description of each of Plum's modes, as well as additional options, is presented later in this document.

Plum is derived from PLVM, an acronym for PL/0 Virtual Machine. The name is a bit of a misnomer, as Plum is no longer only a virtual machine... but the name is cool, so it stays!

What is PL/0?

PL/0 is a simple, educational programming language meant to assist in teaching compiler design. It was invented in 1976 by Niklaus Wirth and it stands for *Programming Language Zero*. It's limited features make it an impractical language to write real programs in, however its small size allows student-designed compilers to remain small and *relatively* simple.

Plum Installation

Plum is designed to be easy to install and use! It can be found at this link, where it can be downloaded from manually. If you'd prefer, you can also clone the repository using *git* like so:

```
$ git clone https://www.github.com/tgsachse/plum.git
```

To build, use the provided build script like this:

```
$ ./build.sh
```

This command will produce an executable program named *plum* in the same folder as the build script. This executable is the entire interpreter and can be used to run your PL/0 files!

Flags/Options

Plum supports a good deal of flags/options to aid developers in finding bugs in their code. What follows is a list of Plum's flags and a brief description of what each flag does.

- **--skip-errors**

  Skip any detected errors in scan mode.

- **--print-source**

  Print the provided source code.

- **--print-lexeme-table**

  Print a table of lexemes and their values.

- **-l / --print-lexeme-list**

  Print the list of lexemes produced by the scan mode.

- **--print-symbol-table**

  Print the symbol table created during parsing.

- **-a / --print-assembly**

  Print the machine language bytecode provided/generated.

- **--print-all**

  Print everything listed above (i.e. include all the print flags).

- **--trace-cpu**

  Trace the status of the CPU of the virtual machine for each instruction.

- **--trace-records**

  Trace the activation record stack of the virtual machine for each instruction.

- **--trace-registers**

  Trace the registers of the virtual machine for each instruction.

- **-v / --trace-all**

  Trace everything listed above (i.e. include all of the trace flags).

- **-o *filename* / --outfile *filename***

  Set the output of the machine to a new location. The name of your desired location replaces the word *filename*.

Modes of Operation

Plum is a modal program, so it can be used in several different ways. What follows is a list of Plum's modes, as well as brief descriptions of what each mode does.

- *RUN*

  This mode takes a PL/0 source program as input, scans it, parses it, and then executes it on the virtual machine. This is the primary command of the interpreter that "does it all."

- *SCAN*

  This mode takes a PL/0 source program as input and produces a list of lexemes (tokenizations of the source code) as output. These lexemes can be translated into executable bytecode in the parse mode.

- *PARSE*

  This mode takes a list of PL/0 lexemes as input and produces executable bytecode for the virtual machine. This bytecode is "machine language," a sequence of numbers that the virtual machine understands as instructions.

- *COMPILE*

  This mode takes a PL/0 source program as input and produces executable bytecode as output. It is functionally the same as passing a source program into the scan mode, and then parsing the results.

- *EXECUTE*

  This mode takes PL/0 bytecode as input and executes that bytecode on the virtual machine.

Example Usage

Plum's syntax looks like this:

```
$ ./plum <mode> <filename> <--options>
```

The mode and filename are required. 0 or more options can be included but must trail the filename and mode as shown above. Here are a few example commands to get you started:

```
$ ./plum run program1.plo
$ ./plum run program2.plo --print-symbol-table --trace-cpu
$ ./plum compile program3.plo --print-assembly --print-lexeme-table
$ ./plum execute program4.plc --trace-all
$ ./plum scan program5.plo --outfile lexemes.txt
```

PL/0 Syntax

The syntax for PL/0 is described using Extended Backus-Naur Form (EBNF), which is just a fancy way to display syntax. EBNF follows these rules:

- [items in brackets] are optional
- {items in braces} are repeated zero or more times
- "literal symbols/words" are enclosed in quotation marks
- ranges are defined with an ellipsis...
- (items in parentheses) are grouped together
- vertical bars indicate either this piece | or that piece
- a period is used to indicate the end of a class.

With these rules in mind, here is the EBNF for PL/0:

```
PROGRAM -> BLOCK ".".
BLOCK -> CONSTANT VARIABLE STATEMENT.
CONSTANT -> ["const" IDENTIFIER "=" NUMBER
             {"," IDENTIFIER "=" NUMBER} ";"].
VARIABLE -> ["var" IDENTIFIER {"," IDENTIFIER} ";"].
STATEMENT -> [IDENTIFIER ":=" EXPRESSION
              | "begin" STATEMENT {";" STATEMENT} "end"
              | "if" CONDITION "then" STATEMENT
              | "while" CONDITION "do" STATEMENT
              | "read" IDENTIFIER
              | "write" EXPRESSION
              | ϵ].
CONDITION -> EXPRESSION RELATION EXPRESSION.
RELATION -> "=" | "<>" | "<" | "<=" | ">" | ">=".
EXPRESSION -> ["+" | "-"] TERM {("+" | "-") TERM}.
TERM -> FACTOR {("*" | "/") FACTOR}.
FACTOR -> IDENTIFIER | NUMBER | "(" EXPRESSION ")".
NUMBER -> DIGIT{DIGIT}.
IDENTIFIER -> LETTER{LETTER | DIGIT}.
DIGIT -> "0"..."9"
LETTER -> "a"..."z" | "A"..."Z"
```

Writing in PL/0

This section covers writing PL/0 programs and includes all implemented features of the language, from variables to loops!

Variables and Constants

In programming, numbers must be stored in memory so they can be manipulated by the program. These numbers can be stored as either variables (which can change as the program executes) or constants (which cannot change during execution). All variables and constants must be declared at the top of every block of code (e.g. at the beginning of the program).

Here's a program that defines a variable named x and a constant named y (example 1):

```
const y = 100;
var x;
.
```

Notice that constants must be assigned a value when they are declared, but variables cannot be assigned at declaration! Also, every valid program in PL/0 needs a period at the end, but not every program needs statements. Constants must also be declared above variables.

In this program, constant a, constant b, and variable c are declared. The variable c is then assigned a value in a *statement* (example 2).

```
const a = 500, b = 300;
var c;
c := a + b
.
```

The statement below the variable declaration does the heavy lifting of assigning the result of an expression to the variable c. All typical operators are supported (subtraction, multiplication, etc) and the order of operations is followed. Parentheses can be used to enforce order in expressions. Here's a more complicated program (example 3):

```
const a = 100, b = 200;
var c;
c := (b + 2) * 99 / (15 - a)
.
```

Input/Output

Programs can take input and produce output during execution. To produce output, the *write* keyword is used. To accept input, the *read* keyword comes in handy.

The following program reads in two values from the user. The first is assigned to *user1* and the second is assigned to *user2*. These values are multiplied together and the *result* is printed to the screen (example 4):

```
var user1, user2, result;
begin
    read user1;
    read user2;
    result := user1 * user2;
    write result;
end.
```

Notice that this program uses the *begin* and *end* keywords. These must be used to "wrap" multi-statement blocks. Each statement inside of a begin-end wrap must be followed by a semicolon. Indentation, though excellent for increased readability, is not necessary.

Boolean Logic and Conditions

PL/0 supports if statements with this general form:

```
if condition then
    statement
```

In the above psuedocode, if the *condition* is true, then *statement* will execute. Here's an example of boolean logic and conditions in action (example 5):

```
var x, y;
begin
    x := 25;
    y := 30;
    if x < 27 then
        x := x * 1000;

    if y < 27 then
        y := y * 1000;
end.
```

The code on the previous page will multiply x by 1000, but will not multiply y by 1000. Note that begin-end can be used in if statements, and if statements can be nested. Here's a more complicated program (example 6):

```
var x, y;
begin
    x := 100;
    y := 200;

    if x > 0 then
    begin
        x := x / 10;
        y := y / 10;
        if y <> 0 then
        begin
            y := y / 10;
        end;
    end;
end.
```

While Loops

Often, you may find yourself wanting to execute a statement several times. For example, if you wanted to print every number from 1 to 100, you'd need an awful lot of *write* statements. Loops solve this problem by allowing statements to be executed over and over again until some *condition* becomes false. The psuedocode for loops in PL/0, called *while* loops, is as follows:

```
while condition do
    statement
```

The following program will keep a variable, *count*, and print that variable to the screen. Each time *count* is printed, it will be incremented (increased by 1). This will continue until *count* is greater than or equal to 20 (example 7):

```
var count;
begin
    count := 1;
    while count < 20 do
    begin
        write count;
        count := count + 1;
    end;
end.
```

Complete Examples

In this last section I've included some larger, more complete programs. These programs are available in the program repository on GitHub.

/* Comments are included in these programs using this syntax. */

Factorial

```
var count, running, target;
begin
    read target;

    running := 1;
    count := 1;

    while count < target do
    begin
        count := count + 1;
        running := running * count;

        write running;
    end;

    if target <= 0 then
    begin
        running := 0;
        write running;
    end;
end.
```

FizzBuzz

```
/* Because I can't print strings, numbers will have to do.
   I will unfortunately have to keep counters for 3's and 5's
   as well because this implementation has no modulus operator. */
const fizz = 10000, buzz = 20000;
var num, threes, fives;
begin
    num := 0;
    fives := 0;
    threes := 0;

    /* Run FizzBuzz from 1 to 100. */
    while num < 100 do
    begin

        /* Increment all the counters. */
        num := num + 1;
        fives := fives + 1;
        threes := threes + 1;

        write num;

        /* If the number is divisible by 3, print fizz. */
        if threes = 3 then
        begin
            write fizz;
            threes := 0;
        end;

        /* If the number is divisible by 5, print buzz. */
        if fives = 5 then
        begin
            write buzz;
            fives := 0;
        end;
    end;
end.
```

Nested Conditions

```
const A = 100, B = 200;
var x, y, z, yy, zz;
begin
    x := 1;
    y := 2;
    z := 3;
    yy := 4;
    zz := 5;
    if A < B then
    begin
        x := (A + B) / 10;
        if x > 100 then
            y := B * 4;
        if x < 100 then
        begin
            z := (A + B) * 70;
            if z > 10000 then
            begin
                zz := 10000;
                if zz <> 10000 then
                begin
                    zz := 88;
                end;
                if zz = 10000 then
                    zz := 999;
                if zz < 10 then
                begin
                    zz := 99;
                end;
            end;
            if y < 10000 then
            begin
                yy := 1000;
                zz := 42;
            end;
        end;
    end;
    zz := 25;
end.
```