The VIOLET Programming Language

Last updated 07/24/14

1 History

VIOLET was developed for the Buttech CAI-1 "Blue Box", and served as the default ROM for the CAI-1. Licensing a BASIC interpreter was too expensive for the project's budget, and the programmer in charge of the requested PILOT implementation quit in protest, leaving a quickly promoted intern student from Butte County Community College with the task of quickly developing a suitable language.

VIOLET (short for Verbose Interpreted Operating Language for Educational Terminals) was the result. A gross mish-mash of PILOT, BASIC, FORTH, and even LISP, VIOLET nonetheless remained in use in the Butte County school system well into the 1990s, and some have blamed the county's poor turnout for quality software engineers directly on VIOLET's continued use and influence on impressionable young minds.

2 The Editor

VIOLET does not follow a typical interactive REPL such as many other interpreted languages, but instead begins operation in its editor mode. This is a simple line editor, with support for file save and load.

Its commands (or 'controls' in VIOLET terminology) are listed below, with arguments in lowercase. All arguments for editor controls are mandatory.

Control of the VIOLET editor switches between two modes: control mode, and entry mode. In control mode, the prompt '> ' appears, and controls may be entered. Some controls will begin entry mode (indicated with a *), in which the user will be instead prompted with numbered prompts, into which a new code line can be entered. Entry mode can be exited by entering an empty line.

2.1 Editor Controls

BEGIN*: Begins a new program and enters entry mode.

APPEND*: Enters entry mode, adding new lines to the end of resident program in memory. Returns an error if no program is present.

INSERT *line**:* Enters entry mode, inserting new lines at the point designated with the line argument. Returns error if line is not a number, is not in the range of current program length, or there is no program resident in memory.

DELETE *startline endline*: Removes lines from the program from startline (inclusive) to endline (exclusive). Returns an error if arguments are not numbers, if they exceed range of current program length, or no program is in memory.

LIST: Lists the present program, preceded by line numbers. Returns an error if no program is resident.

SAVE *filename*: Saves the contents of the current program to a file called (filename)

LOAD *filename*: Loads the contents of a file called filename, or returns an error if it doesn't exist.

DIR: Lists the contents of the working directory (note: this is not a 100% authentic handling, but has been implemented for usability's sake.)

RUN: Sends the current program in memory to the interpreter and attempts to execute it.

EXIT: Exits the interpreter. (Note: On the original hardware this would quit to the built in monitor; here it just exits the program)

3 Basic Structure

Each line in a VIOLET program consists of an "operator", and its arguments. The operator must begin the line, and if the interpreter fails to find a valid operator, it will quit.

Every operator essentially acts as a function, and thus returns a value in addition to any other effects, which is stored in a built-in variable called &LAST. This is used both internally to handle conditional statements, and can be called upon to apply the result of previous mathematical operations to a variable. For example:

```
ADD #X 45
SET #X %LAST
```

is a statement roughly equivalent to "x = x + 45" as might be found in other languages.

All programs MUST begin with the declarative operator PROGRAM, taking START as its argument, and end with PROGRAM STOP. This is especially necessary because the VIOLET editor operates in plain text and VIOLET code

is not tokenized like other contemporary languages, thus requiring a clear statement of initialization to distinguish a program file from arbitrary text documents.

Operators are always given in upper case, and will not be recognized otherwise. Variables can be upper or lower case, however easier typing meant convention tended to be to keep them upper case.

4 Variable Conventions

Variables in VIOLET consist of three types, indicated by their preceding symbol:

```
#VAR (integer)
%VAR (float)
$VAR (string)
&VAR (dump, reserved for %LAST)
```

Integers and floats are numeric values. An integer declaration will fail if a decimal point is found, and a float will fail if no decimal point is found (declare whole number floats with trailing 0 decimal, like *4.0*).

String values are sequences of ASCII characters enclosed in quotation marks.

The "dump" variable is a special type, reserved solely for the &LAST value, and may at any given time contain either a string or a number. Care should thus be taken when calling &LAST to ensure it contains the right value.

Note also that unlike some languages, bare declarations of variables are NOT allowed in VIOLET, and must always begin with the operator SET. SET takes two arguments, a variable name, and an appropriate value to the variable type. SET will fail on a mismatch, or on attempt to assign a new dump or to &LAST.

All variables are global, as VIOLET does not possess any form of scoping, much like old-fashioned BASIC.

5 Flow Control

VIOLET's flow control methods are fairly primitive, being limited largely to the IF/IFY/IFN operator set, and a WHILE/WHEND loop, as well as the usual GOTO statement found in many of these languages.

5.1 IF

The IF operator takes a test, and evaluates it, returning 1 to &LAST if the test is true, and 0 if the test is false. The IF statement can then be followed with the IFY operator (which executes the given operator and its arguments if &LAST is 1, returning 0 if not) or the IFN operator (which executes the given operator and its arguments if &LAST is 0, returning 1 if not). Not that either statement can be skipped, and because they rely on &LAST, they do not even especially require the presence of a preceding IF operator.

The IF test syntax is simple, taking exactly three arguments:

IF TESTOPERATOR first_value second_value

The test operators are as follows:

EQUALS Tests if the two values are equal to each other.

NOTEQ Tests if the two values are equal to each other.

GREATER Returns true if *first_value* is greater than *second_value.*

LESSER Returns true if *first_value* is less than *second_value.*

AND Returns true if both values are greater than 0, or false if not.

OR Returns true if either value is greater than 0.

A sample usage of IF looks like the following:

```
IF EQUALS #VAR 45
IFY SET #VAR 0
IFN SET #VAR 100
```

5.2 WHILE

The WHILE operator syntax is similar to the IF operator, but rather than executing a different bit of code depending on the test, it instead continually executes everything between the WHILE and WHEND operators while the test remains true. For example:

```
SET #X 99
WHILE NOTEQ #X 0
PRINT #X "BOTTLES OF BEER ON THE WALL"
SUB #X 1
SET #X &LAST
WHEND
```

5.3 GOTO

VIOLET contains the dreaded GOTO command, and its use is perhaps even more problematic here than in other languages, as the line number of a given line cannot be guaranteed. GOTO *line* jumps the current execution pointer to *line,* where line is indexed from 0 based on the number of lines in the current program. Thus, if the program is edited and the number of lines changes, the old GOTO command will now go god knows where in the program.

6 Operator Reference

This section contains a list of the available VIOLET operators and their syntax. It is useful to note a convention here: operators are always given word names rather than symbols (thus giving meaning to the "Verbose" portion of the acronym). This was a requirement set by the Board of Education, who believed that use of full word commands would be 'easier to understand.' Minutes of previous meetings indicate that even allowing the type prefix on variables required considerable argumentation.

6.1 Arithmetic

ADD **args:* Adds the value list given and assigns it to &LAST.

SUB *first_value second_value:* Subtracts *second_value* from *first_value* and assigns it to &LAST.

MULT **args:* Multiplies the value list given in order and assigns the total to &LAST.

DIV *first_value second_value:* Divides *first_value* by *second_value,* and assigns the result to &LAST. Note that if both values are integers, it will return an integer result, dropping the remainder. If one value is a float, it will return a float. *Be very careful therefore when assigning &LAST back to a variable!*

6.2 Flow Control

IF *test first_value second_value:* Tests *first_value* against *second_value* using operator *test,* returning 1 to &LAST if true, or 0 if false.

IFY *operator *args:* If &LAST is 1, executes *operator* with its *args.* Otherwise, sets &LAST to 0.

IFN *operator *args:* If &LAST is 0, executes *operator* with its *args.* Otherwise, sets &LAST to 1.

WHILE *test first_value second_value:* Tests *first_value* against *second_value* using operator *test.* If the test is true, it stores the pointer and executes the next line of code, if not, it seeks the WHEND operator and sets the pointer to the line following it.

WHEND Sets the pointer to the value stored by WHILE.

GOTO *line:* Jumps the pointer to the *line* specified.

6.3 SET

SET *name value:* Sets the variable *name* to *value*, creating *name* if it does not exist. Name **must** be preceded by one of the following symbols, designating its type, and if the type and the value mismatch, the interpreter will produce an error and exit. SET returns *value* to &LAST.

- # *integer:* A whole number value. Will create a mismatch if a decimal point is found.
- % *float:* A floating-point numeric value. Will create a mismatch if no trailing decimal is found. Create whole number values by trailing with *.0*.
- $ *string:* A sequence of ASCII characters, which must be enclosed in quotation marks

6.4 Input/Output

PRINT **args:* Prints each argument, seperated by spaces.

PROMPT *type [prompt]:* Accepts input from the terminal, returning an error if it does not match type, and stores the result to &LAST. The optional argument *prompt* takes a string which will replace the standard "> " prompt.

6.5 Additional Operators

IGNORE **args:* The IGNORE operator tells the interpreter to skip this line. This can be used for code commenting, allowing arbitrary information to follow it.