FLOAT Language Manual

Gary T. Leavens

March 29, 2023

Abstract

This document defines the Fancy Language Offering Arithmetic Terms (FLOAT). This language is intended as a simple demonstration of some compiler techniques (for COP 3402 at UCF in Spring 2023).

1 Overview

The FLOAT compiler operates as command-line program. The following subsections specify the interface between the Unix operating system and the compiler.

1.1 Inputs and Outputs

The compiler takes a single command line argument specifying the name of a program in the FLOAT language. For example, if the file name argument is `test.flt` (and both the compiler executable, `./float`, and the file `test.flt` are in the current directory), then the following command line (given to the Unix shell)

```
./compiler test.flt
```

will run the compiler on the program in `test.flt` and put the generated FLOAT VM machine code on standard output; any error messages are printed on standard error output.

There are two optional command line arguments, both of which must appear before the file name on the command line calling the program.

The `-l` option requests that the compiler prints a table of tokens read (on stdout), and then stops. Thus the compiler only performs lexical analysis on the input. This is useful in debugging the compiler's lexical analysis phase.

The `-u` option requests that the compiler parse the program and unparse the resulting AST, with output (onto stdout), and then the compiler stops (before generating any code). This is useful for debugging (the parser or) the syntax of a program.

2 FLOAT language

This section defines the FLOAT language.

2.1 Syntax

The context-free grammar for FLOAT is defined in Figure 1 and its lexical grammar is defined in Figure 2.

There is an interesting lexical issue regarding the lexical analysis of numeric literals (⟨number⟩ in Figure 2). The lexical grammar allows an optional sign, either plus (+) or minus (−, before the first digit of a numeric literal. Since the lexer favors the longest match, an expression like:

```
3-4-5
```

has 3 tokens (`3`, `-4`, and `-5`) and no operators, so it will lead to a parse error.

If subtraction is desired, then a minus sign must be separated from the any following number numbers by whitespace, as in:

```
3 - 4 - 5
```

which has 5 tokens (`3`, `-`, `4`, `-`, and `5`) and will be parsed as an expression.

2.2 ASTs

The type for the enhanced abstract syntax trees (ASTs), which also serves as an IR, is defined in the provided files `ast.h`, with helping functions in `ast.c`.

The file `ast.h` declares a type named `AST` and a type `AST_list`. The type `AST_list` is a (linked) list of ASTs.

2.2.1 The AST Type

An `AST` is a C **`struct`** containing the following fields:

- A field named `file_loc` that gives the AST's (starting) file location. That is, it gives information about the place in a PL/0 source file where (the start of) the first token that was parsed into the AST was found: its filename, line number, and column number.

  The file location has the type `file_location`, which is a type defined in `file_location.h`. The file location is used in error messages.

- A field named `next` that is used for working with linked lists of ASTs.

- A field named `type_tag` that indicates what kind of AST is held in the `data` field (see the next item below).

  These type tags are members of the enumerated type `AST_type`, which is declared in the file `ast.h`. Each of these tags corresponds to a nonterminal symbol in the abstract syntax of PL/0 (see Figure 3).

- A field named `data` that holds a struct which has a type corresponding to the type tag (see the previous item above). This field is a C union, meaning that it can hold any one of the declared struct types corresponding to the tags of the union, which can be thought of as subfields.

2.2.2 AST Lists

In some ASTs some of the fields hold lists of other AST. The `ast.h` file declares a type `AST_list` that describes such lists and several operations that work on such lists.

⟨program⟩ ::= ⟨var-decls⟩ ⟨stmt⟩

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

⟨stmt⟩ ::= ⟨ident⟩ `=` ⟨expr⟩ **;**
| ⟨begin-stmt⟩
| **if** `(` ⟨expr⟩ `)` ⟨stmt⟩
| **read** ⟨ident⟩ **;**
| **write** ⟨expr⟩ **;**
⟨begin-stmt⟩ ::= ‘`{`’ ⟨var-decls⟩ ⟨stmt⟩ **{**⟨stmt⟩**}** ‘`}`’

⟨expr⟩ ::= ⟨lterm⟩ ⟨rel-op-lterm⟩
⟨lterm⟩ ::= `!` ⟨lfactor⟩ | ⟨lfactor⟩
⟨rel-op-lterm⟩ ::= ⟨empty⟩ | ⟨rel-op⟩ ⟨lterm⟩
⟨rel-op⟩ ::= `==` | `!=` | `<` | `<=`
⟨empty⟩ ::=
⟨lfactor⟩ ::= ⟨term⟩ **{**⟨add-sub-term⟩**}**
⟨add-sub-term⟩ ::= ⟨add-sub⟩ ⟨term⟩
⟨add-sub⟩ ::= ⟨plus⟩ | ⟨minus⟩
⟨term⟩ ::= ⟨factor⟩ **{**⟨mult-div-factor⟩**}**
⟨mult-div-factor⟩ ::= ⟨mult-div⟩ ⟨factor⟩
⟨mult-div⟩ ::= ⟨mult⟩ | ⟨div⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨number⟩ | `(` ⟨expr⟩ `)`

Figure 1: Context-free grammar for FLOAT. The grammar uses a `terminal font` for terminal symbols, and a **`bold terminal font`** for reserved words. As in EBNF, the notation **{**x**}** means an arbitrary number of (i.e., 0 or more) repetitions of x. The terminal symbols `{` and `}` are quoted (like ‘`{`’) to make clear that they are to be interpreted as terminal symbols, not as meta-notations.

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

⟨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 2: Lexical grammar of FLOAT. The grammar uses a `terminal font` for terminal symbols. Note that an underbar (_) and all ASCII letters (a-z and A-Z) are included in the production for ⟨letter⟩. Again, {x} means an arbitrary number of (i.e., 0 or more) repetitions of x. Some character classes are described in English, these are described in a Roman font between double quotation marks (" and "). Note that all characters matched by the nonterminal ⟨ignored⟩ are ignored by the lexer (except that each instance of ⟨eol⟩ ends a line).

⟨program⟩ ::= **{**⟨var-decl⟩**}** ⟨stmt⟩
⟨var-decl⟩ ::= **float** ⟨ident⟩ | **bool** ⟨ident⟩
⟨stmt⟩ ::= ⟨ident⟩ = ⟨expr⟩
  | **begin** **{**⟨var-decl⟩**}** **{**⟨stmt⟩**}**
  | **if** ⟨expr⟩ ⟨stmt⟩
  | **read** ⟨ident⟩
  | **write** ⟨expr⟩
⟨ident⟩ ::= ⟨name⟩ ⟨id-use⟩
⟨id-use⟩ ::= ⟨attrs⟩ ⟨levelsOutward⟩
⟨attrs⟩ ::= ⟨file-location⟩ ⟨var-type⟩ ⟨loc-offset⟩
⟨var-type⟩ ::= **float** | **bool**
⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | ⟨ident⟩ | ⟨number⟩ | ! ⟨expr⟩
⟨op⟩ ::= == | != | < | <= | + | − | * | /

⟨op-expr⟩ ::= ⟨op⟩ ⟨expr⟩

Figure 3: Abstract Syntax for FLOAT Again **{**x**}**, means a possibly empty list of x. Note that ⟨op-expr⟩ is intended only as an intermediate form during parsing, and is not used directly by any of the other rules. Indeed, the parser eliminates all such ASTs and builds the information in them into other types of ASTs; thus when walking the ASTs, it is not necessary to consider the ⟨op-expr⟩ type of AST.

2.3 Semantics

This subsection describes the semantics of FLOAT.

Nonterminals discussed in this subsection refer to the nonterminals in the context-free grammar defined in Figure 1.

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

All arithmetic is done on floating point numbers as in C's **`float`** type. The execution of a program declares the named variables, initializes these variables, and then it runs the statement.

It is an error if an ⟨ident⟩ is declared more than once.

2.3.1 Variable Declarations

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

Each variable declaration, of the form **`float`** ⟨ident⟩, declares that ⟨ident⟩ is a (double-precision) floating point variable that is initialized to the value 0.0. A variable declaration of the form **`bool`** ⟨ident⟩ declares that ⟨ident⟩ is a Boolean variable that is initialized to the value false (i.e., 0).

The scope of a variable declaration is: the area of the program's text if the variable is declared at the beginning of the program), or the area of a ⟨begin-stmt⟩ if the variable is declared at the beginning of that statement.

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

2.3.2 Statements

A program contains a single statement that is run when the program starts executing.

Note that the base case statements are terminated with a semicolon (**`;`**).

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 of ⟨expr⟩.

It is an error if the left hand side ⟨ident⟩ has not been declared as a variable.

It is a type error if the type declared for ⟨ident⟩ is not the same as the type of the ⟨expr⟩.

Sequential Statement A begin-statement has the form `{` $D_1 D_2 \ldots D_m S_1 S_2 \ldots S_n$ `}` (where $m \geq 0$ and $n \geq 1$) and is executed by first allocating the variables declared by $D_1 \ldots D_m$ (initialized to 0 for float variables and false for bool variables) and then executing statement S_1, then if S_1 finishes without encountering an error S_2 is executed, and so on, in sequence.

If Statement An if statement has the form **`if`** `(` E `)` S and is executed by first evaluating the expression E. When E evaluates to true, then S is executed; otherwise, if E evaluates to false then this statement does nothing.

It is a type error if E does not have type **`bool`**.

Read Statement A read statement of the form **`read`** x, where x is a declared **`float`** identifier, reads a single floating-point number (using the format of the nonterminal ⟨number⟩) from standard input and puts (an approximation of) its value into the variable x.

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

Write Statement A write statement of the form **`write`** E, first evaluates the expression E, and then writes (an approximation to) its value on standard output in decimal notation.

2.4 Expressions

An $\langle$expr$\rangle$ of the form $E_1\ o\ E_2$ first evaluates E_1 and then E_2, obtaining 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 following the semantics for C's operators.

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 stored in variable named x.

  It is an error if x has not been previously declared.
- An expression of the form `!` E, first evaluates E, which must have a Boolean value V, the result is the logical negation of V.
- An expression of the form `(`E`)` yields the value of the expression E.

3 Future Work

There are some things that the VM can do but are not expressible in FLOAT. These include the following, all of which could be considered future work to add into FLOAT:

- Integer data. The VM can deal with integers and can round a floating-point number to an integer.
- Character data. The VM can read and write individual characters.
- Procedures. The VM has instructions to support procedures, but those are not present in FLOAT.