FFBB Reference Manual

Bowen Chen, Joseph Yang, Jianan Yao, Xiaosheng Chen

{bc2916, zy2431, jy3022, xc2561}@columbia.edu

February 24, 2021

1 Introduction

The FFBB programming language is an imperative language mainly based on the C programming language, with some other features inspired by Java.
It is a general-purpose programming language and even users with non-technical background will be able to study FFBB easily. FFBB will finish syntax-checking during compile time so that programmers won't waste too much time on syntax problem.
The general syntax and language features would be similar to those of the C programming language, with some other operators and features from Java (e.g. type declaration, comment, `void` keyword). Also, FFBB programming language accepts some functional programming features like higher-order functions. We hope that our language could combine the advantages of C and the flexibility of Python to some extent, with certain acceptable trade-off.

Our goals are:

- C-style language design with safe explicit type and easy compilation.
- Support of recursion and higher-order functions.
- Built-in data structures of list (array), dictionary and set like in Python. Implemented using hashtable to achieve high efficiency.

2 Lexical conventions

FFBB will include the following types of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. General blanks will be ignored and FFBB needs at least one blank to separate adjacent identifiers, constants, and certain operator-pairs

2.1 Comments

2.1.1 Multi-line Comments

The characters /* introduce a comment, which terminates with the characters */.

2.1.2 Single-line Comments

The characters // introduce a comment until the end of line.

2.2 Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore "_" counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

$$if, else, for, while, return, int, bool, float, int, void, true, false, in, List, Set, Dict, def, rec$$

2.3.1 Integer constants

An integer constant is a sequence of digits. And integer should not have leading zero.

2.3.2 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing.

2.3.3 String constants

A string constant is a sequence of characters surrounded by double quotes ”

3 Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `gothic`. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript ”opt,” so that

$$\{expression_{opt}\}$$

would indicate an optional expression in braces.

4 What’s in a Name?

FFBB bases the interpretation of an identifier upon its *type*. The type determines the meaning of the values found in the identifier’s storage.
There are four fundamental types of objects: strings, integers, floating-point numbers, and booleans.

- Strings (declared, and hereinafter called, `string`) is an immutable data structure that contains a variable-length sequence of characters. Each character can be accessed in constant time through its index.
- Integers (`int`) are represented in 16-bit 2’s complement notation.
- Single precision floating point (`float`) quantities have magnitude in the range approximately 10^{38} or 0; their precision is 24 bits or about seven decimal digits.
- Booleans (`bool`) are represented by `true` or `false`.

Besides the four fundamental types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *lists* of objects of a given type;
- *sets* of objects of a given type;
- *dictionaries* of objects of two given types;

5 Conversions

Some operators, like the arithmetic operators, may perform automatic type conversions on the values of specific operands. We list the rules of such conversions in this section.

5.1 Arithmetic conversions

There are two pre-defined arithmetic value types in FFBB language: `int` and `float`. We allow their auto conversion.

5.2 Booleans and integers

The types `bool` and `int` are used interchangeably in the FFBB language. Specifically, the `True` value in `bool` type corresponds to the `1` value in `int` type, whereas the `False` value in `bool` type corresponds to the `0` value in `int` type.

6 Expressions

The precedence of expression operators follow the conventions of order of operations. Within each subsection, the operators have the same precedence.

6.1 Primary expressions

Primary expressions involve only function calls and group left to right.

6.2 identifier

An identifier is one of the most primitive expression, and it will be used to identify an unique object or function in FFBB

6.3 constant

Constant is one of the most fundamental expression in FFBB. The value of a constant is fixed and remains the same during the entire execution of the program.

6.4 expression

Expression in FFBB are usually linked by different operands. An expression can be a variable, constant or some other expressions.

6.4.1 expression * expression

The binary * operator indicates multiplication. If both operands are float/int, the result are float/int; If operands are float and int, the type of results will be float.

6.4.2 expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

6.4.3 expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int or char, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

6.4.4 expression + expression

The binary + operator indicates addition. If both operands are float/int, the result are float/int; If operands are float and int, the type of results will be float. Other type combinations might be discussed later .

6.4.5 expression - expression

The binary - operator indicates subtraction. The same type considerations as for addition apply.

6.5 Relational operators

6.5.1 expression<expression

6.5.2 expression>expression

6.5.3 expression<=expression

6.5.4 expression>=expression

The relational operators group left-to-right. and all of the operators are following the mathematical conventions: `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to). 1 indicates true while 0 will be considered as false.

6.6 Equality operators

6.6.1 expression==expression

6.6.2 expression!=expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence.

6.7 expression && expression

The `&&` operator returns 1 if both its operands are non-zero, 0 otherwise. The second operand is not evaluated if the first operand is 0.

6.8 expression || expression

The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. The second operand is not evaluated if the value of the first operand is non-zero.

6.9 !expression

The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is int. This operator is applicable only to ints or booleans.

6.9.1 identifier++

Increment the value of identifier by 1. This is applicable only to ints.

6.9.2 identifier–

Decrement the value of identifier by -1. This is applicable only to ints.

7 Declarations

In function/variable definitions, declarations are used to specify the interpretation which FFBB gives to each identifier. There are two types of declarations in the FFBB language.

7.1 Type specifiers

The type-specifiers are

type-specifier :

```
bool
int
float
string
List
Set
Dict
```

These specifiers explicitly define the type of

- function return value, if explicitly used in function declarations. In this case, the defined function must return a value with the specified type. Otherwise compile error will be raised.
- variable stored value, if explicitly used in variable declarations. In this case, the defined variable must always store values with the specified type. If ever try to assign a value with other types to the variable, compile error will be raised.

7.2 Return specifiers

There is only one return-specifier

type-specifier :

`void`

This return-specifier can only be used in function declarations. Also, it cannot be used together with type specifiers. If this return-specifier is used in function declarations, that means the defined function cannot return anything as output. An analogy in Python would be: the defined function works like a procedure, instead of a function.

8 Statements

Except as indicated, statements are executed in sequence.

8.1 Expression statement

Most statements are expression statements, which have the form

$$expression;$$

Usually expression statements are assignments or function calls.

8.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

compound-statement :

{*statement-list*}

statement-statement :

statement

statement statement-list

8.3 Conditional statement

The two forms of the conditional statement are

`if` (*expression*) *statement*

`if` (*expression*) *statement* `else` *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ”else” ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

8.4 While statement

The while statement has the form

`while` (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

8.5 For statement

The `for` statement has the form

`for` (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$;) *statement*

This statement is equivalent to

```
expression-1;
while ( expression-2 ){
        statement;
        expression-3;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied `while` clause equivalent to "while(1)"; other missing expressions are simply dropped from the expansion above.

8.5.1 For-in statement

The `for-in` statement has the form

`for` (*identifier-1* `in` *identifier-2*) *statement*

This statement is equivalent to

```
int i = 0;
while (i < list.length){
    identifier-1 = identifier-2[i];
    statement;
    i++;
}
```

Thus the *identifier-1* is the iterator of the list *identifier-2*.

8.6 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

```
return ;
return ( expression );
```

9 Functions

Function definitions have the form

function-definition :

`def` `rec`$_{opt}$ *type-specifier function-declarator function-body*

Note `rec` is an optional keyword indicating whether the function is recursive or not.

function-declarator :

declarator (parameter-list$_{opt}$)

parameter-list :

identifier

identifier, parameter-list

The function-body has the form

function-body :

type-decl-list function-statement

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here. The function-statement is just a compound statement which may have declarations at the start.

function-statement :

{ *statement-list*}

10 Lexical scoperules

The lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; FFBB is not a block-structured language; this may fairly be considered a defect. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading thestatements constituting the function itself) is the body of the function. It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

11 Constant expressions

In several places FFBB requires expressions that have to evaluate to a constant: in array bounds, the expression can only involve integer constants, possibly connected by binary arithmetic operators; in control flow condition statements, the expression has to evaluate to booleans.

12 Examples

Here are some example codes to demonstrate FFBB language.

```
def void swap(int[] A, int i, int j) {
    int t = A[i]; A[i] = A[j]; A[j] = t;
}

def int partition(int[] A, int p, int r) {
  int x = A[r];
  int i = p - 1;
  for (j in range(p, r + 1)) {
      if (A[j] <= x) {
          i++;
          swap(A, i, j);
      }
  }
  swap(A, i + 1, r);
  return i + 1;
}

// Recursive function to sort list A using quick-sort
def rec void quicksort(int[] A, int p, int r) {
  if (p < r) {
      int q = partition(A, p, r);
      quicksort(A, p, q-1);
      quicksort(A, q+1, r);
  }
}

def void main () {
    // Fibonacci number: Compute Nth value
    int n = 10;
    int[] f = ListCreate<int>(n);
    f[0] = 0;
    f[1] = 1;
    for (i in range(2, n)) {
        f[i] = f[i - 1] + f[i - 2];
    }
    print(f[n-1]);

    // Using quicksort
    int[] A = [4, 2, 7, 3, 1, 9, 6, 10, 5, 8];
    quicksort(A, 0, ListLength(A) - 1);
    for (a in A) {
        print(a);
    }
}
```