Fli-O: File I/O + String Processor

COMS W4115 (Fall 2018)

Matthew Chan (mac2474), Justin Gross (jg3544), Gideon Cheruiyot (gkc2112), Eyob Tefera (et2546)

Overview

Fli-O was developed to create a seamless way to for users to interact with files, especially large documents that require additional processing. To avoid the confusion around buffers and input/output, we plan to have one function for processing input and another for processing output that the user can configure to their needs. These two functions do all the background work with opening and closing files that other languages require multiple functions for. In short, we want to simplify the process of working with file I/O and change it from a pain point to a hallmark of the language.

Furthermore we want to give users the ability to process these files and do additional operations on these files while keeping the I/O process as simple as possible. Some processing that users would be able to do include spell checking, file manipulation, search and replace, merging multiple files, splitting files, as well as a whole host of string manipulation operations that would be commonly found in string libraries. We would also design operators (*, =, +, -, etc.) around strings and files as shortcuts for string and files common operations. For example, '+' would merge two files or strings, '-' would remove the contents that differ between two files. We would also include built in spell checking of documents based on a dictionary that we would provided or a user could supply their own. Our language will be written in OCaml and then compiled into LLVM code.

Features:

- Strict keywords (File != file)
- Automatic memory allocation (no need for pointers)
- Static-typing (types must be specifically declared)
- Operations with strings and other types automatically get converted to strings
- Support for control statements (if, while, for)

Syntax

Syntax will be a mix of C and Python. Arrays are declared and initialized as in C (but no pointers!). We will use C-style semicolons and braces for statement grouping.

Data Types

Keyword	**Description**
`bool`	Primitive boolean type
`char`	Primitive character type
`int, float`	Primitive numeric type, same as C
`String`	String literal type
`File`	Complex
`Dict`	Key-value pair data structure

Comments

Syntax	**Description**
`//`	Single line comment
`/*        */`	Multi-line comment

Operators

Notes:

- Arithmetic operations between an int and a float will result in a float through implicit type conversion.
- + and * operators are valid between numeric and string types, similar to Python (i.e. "3" + 5 = "35" and 4 * "cat" = "catcatcatcat")

Operator	**Type**	**Description**
`+, -, /, *`	int, float	Arithmetic operators function as

		is typical between same data types.
`+`	String, File	Concatenates two strings together. Appends two files together.
`=`	All types	Assignment operator
`==, !=`	int, float, String	Equality operators (by value)
`&&, ||`	All types	Logical AND and OR
`&,|,^,-`	String, chars	Regex operators
`[]`	Array, String, File	Array indexing operator

Keywords	**Description**
`if/else`	Basic control flow conditional expression
`for/in`	Control flow iterative loop
`while`	Iterative loop with conditional
`empty`	Indicates nothing is to be returned
`return`	Return expression

Example Code

Below are several examples of code written in our language to demonstrate syntax and code structure.

1. **File manipulation**

```
// Create a new text file
File f = open(“./newfile.txt”);

// Write a line to the file
f.write(“The quick brown fox jumped over the lazy dog.”);
```

```
// Get a word count for the file
int wc = f.count("w"); // wc = 8

// Print the first n lines of the file
print(f.head());
print(f.tail(15));

// Move and rename the file
f.move("~/newfolder/renamedfile.txt");
```

2. Looping and conditionals

```
Dict primes;
int size = 50;

// Initialize the values in a dictionary
for (n in range(0, size)) {
     primes.add(n, 1);
}

// Sieve of Eratosthenes
for (entry in primes) {
     if (entry.second == true) {
          int num = entry.first;
          for (i in range(num * num, size, num)) {
               // Remove non-primes from the dictionary
               primes.remove(num.second);
*         }
     }
}
```

3. Spell check a file

```
File f = open("./badspelling.txt");
String dictionary[50];
dictionary[0] = "Aardvark";
dictionary[1] = "Aardwolf";
// Add more entries here...
// Spell check the file using a given dictionary
Dict errors = f.spellcheck(dictionary);
if (errors.size() > 0) {
     for (error in errors) {
            print(“Misspelled word “ + error.first + “ on lines “
            + error.second.join(“,”));
            // Output: “Misspelled word asdf on lines 11,57,90”
        }
}
```