FIRE

File Input Reinterpretation Engine

Team Members

- Language Guru Christopher Thomas cpt2132
- System Architect Jason Konikow jk4057
- System Architect Frank Spano fas2154
- Tester Ayer Chan oc2237
- Manager Graham Patterson gpp2109

Table of Contents

Introduction

FIRE is a statically typed programming language designed for implementing algorithms which extract, mutate, process, and report text and structured data. FIRE is meant to be used in conjunction with large sets of structured and delimited data, like CSV's. At the core of the language is the motivation to intuitively iterate over, manipulate, and map functions to large sets of structured data.

Motivation

Many programmers who use UNIX-based, command-line interfaces prefer to do their text manipulation with an array of UNIX tools. Stringing together inputs and outputs with syntactically complex statements is cumbersome and confusing. Our language aspires to streamline and simplify text manipulation tasks by making files first-class citizens. FIRE is a scripting language, inspired by AWK and other languages.

Additionally, the most common way for professional teams to share data between each other is with a CSV file.

If a team receives some data and they want to quickly manipulate that data, how can they avoid the overhead of importing it into a relational database, then querying that database for the desired manipulation? FIRE allows for such a manipulation.

Features

Primitive Data Types:

- `int` - Integer
- `float` - A floating point number
- `string` - A sequence of characters
- `file` - Native file type for easily operating on files
- `func` - Function type, are first class citizens
- `array` - An associative array

Reserved Keywords:

- all data types
- all control statements - `{if, while, for}`
- `in` - syntactic sugar to iterate over every element in array or every line in file stream : `for (x in numbers)`
- `print` - used to print data to `stdout`
- `return` - used to return value from `func`
- `map` - `array` operator keyword
- `stream` - `file` type operator keyword
- `extract` - `file` type operator keyword
- `stdin` - standard input
- `stdout` - standard output
- `stderr` - standard error

Documentation

Syntax

Statements are delimited by `;` and scope is bounded by `{...}`.

Regular Expressions

FIRE supports regular expressions for finding, replacing, and manipulating text. For example, if you're interested in accessing elements of an array whose indices take a particular form you can use a regular expression : `col = arr[r'[a-zA-Z]']` . This would return an array of alphabetical characters in an existing array.

Arrays

FIRE supports the use of associative arrays, similar to the awk implementation. The major difference between associative arrays and indexed arrays is that in an associative array the indices are converted to strings under the hood allowing for any valid string (including numbers) to be used as an index. Additionally, arrays are not stored in any particular order. The order in which elements are produced in an expression `for (element in array)` is undefined.

The associative array allows FIRE to pair information in a way that “associates” the key to a value so that the array is more flexible and intuitive than traditional indexing. You can simulate indexed arrays in FIRE by simply using sequential numbers as your indices, but the keys are being stringified which means its possible to have numbers such as -1 or 2.55 as indices.

Arrays are declared using the `array <variable name>` keyword and do not require an initial size.

Example:

```
array arr;
arr["fireIsCool"] = 1;
arr[-987] = "two";
arr[66.876] = 3;

for (e in arr) {
    print e;
}
```

Basic Operators

Operator	Purpose	Example
`=`	assignment	`x=6`
`+, -, *, /`	basic arithmetic operators	`x = a {+, -, *, /} b`
`==, >, >=, <, <=, !=`	comparison operators	`if (x == y) ...`
`++, --`	{post, pre}fix increment and decrement	`x++; ++x` `x--; --x`
`=>`	anonymous function	`(param) => { body }`
`===`	matches data to regex	`if (String y === [a-zA-Z]*)`

Logical Operators

Operator	Purpose	Example
`||`	logical or	`if (x||y)`
`&&`	logical and	`if (x&&y)`
`!`	logical not	`if (!x)`

Array Operators

Operator	Purpose	Example
`[::]`	slicing operators on arrays	`x = arr[3:5:];`
`del <arr>[<item>]`	delete operator on an item in an array	`del arr[3];`
`map <arr>(<func>)`	the passed func is called on each element in the array	`map numbers(doubleFunction);`

File operators

Operator	Purpose	Example
`stream`	returns an array of the file delimited by `\n`	`f = file("roster.csv"); f.stream();`
`extract`	returns an array of all matching fields in the file	`array x = file.extract("/w{5}");`

Control Flows

FIRE provides the following set of control flow operators: `if`, `while`, and `for`.

Comments

Comments are represented with two forward slashes, e.g:

```
// This is a comment!
```

Code Example

```
user:~ $ cat PhoneNumbers.txt
Dennis 201-445-9372
Kenneth 954-667-8990
Richie 312-421-0098
Thomas 201-750-0911
Albert 783-444-7862

user:~ $ cat nj_numbers.fire
//
// Program that determines if a number if from NJ based on 201 area code
//

func isNJ = (String phoneNumber) => {
    return phoneNumber === "201-/d{3}-/d{4}";
};

func extractRegion(func isRegion, file numbers) {

    array resultingNums;
    for(number in numbers.stream()) {
        if(isRegion(number)){
            resultingNums[number] = number;
        }
    }
    return resultingNums;
}

file in = file(stdin);
print extractRegion(isNJ, in);

user:~ $ cut -d' ' -f2 | fire nj_numbers.fire
201-445-9372
201-750-0911
```