Fire

File Input Reinterpretation Engine

Table of Contents

Introduction

FIRE is a 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 pipes and an array of UNIX tools, stringing together inputs and outputs in cumbersome, syntactically complex statements. 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, and aims to make the manipulation of files and text as easy as possible.

Additionally, the most common way for professional teams to share data between eachother is with a csv file. If a team receives some data and they want to quickly manipulate that data, how can we 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
- `bool` - {True or False}
- `array` - represented as associative arrays
- `file` - native file type for easily operating on files
- `func` - Treated like first class citizens, i.e. they may be passed as parameters and stored in variables

Reserved Keywords:

- all data types
- all control statements - `{if, while, for}`

- in - syntactical sugar to iterate over every element in array or every line in file stream : `for (x in numbers)`
- print - used to print data to the screen

Documentation

Syntax

Scope is bounded by `{...}` and `;` will delimit statements, ie indentation is not a syntactic enforcer.

Regular Expressions

FIRE supports regular expressions for finding, replacing, and manipulating text. For example, if you're interested in accessing elements of an array, which might be strings, a concise expression of that would be: `col = arr[r'[a-zA-Z]']`

Basic Operators

Operator	Purpose	Example
`=`	asssignment	`x=6`
`+, -, *, /`	basic arithmatic operators	`x = a {+, -, *, /} b`
`\\|`	pipe output of a function to another	`f(x) \\| g()`
`==, >, >=, <, <=, !=`	comparison operators	`if (x == y) ...`
`++, --`	{post, pre}fix increment and decrement	`x++; ++x x--; --x`
`=>`	anonymous function that can be assigned to a variable	`(param) => { body }`
`===`	matches data to regex	`if (String y === [a-zA-Z]*)`

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]`

File operators

Operator	Purpose	Example
stream	opens a stream to the file	`f = file("roster.csv"); x = f.stream()`

Control Flows

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

Code Example

```
PhoneNumbers.txt
//example file with list of phone numbers

201-445-9372
954-667-8990
312-421-0098
201-750-0911
783-444-7862
...


ColdCall.Fire
//using fire to extract NJ phone numbers and pipe into a "cold call" function

file f = file(PhoneNumbers.txt);

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


func extractRegion(func isRegion, file numbers) {
    String[] resultingNums;

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

    return resultingNums;
}



extractRegion(isNJ, f) | coldCallNumbers()
...
```

FAQ

Q. Is this AWK?

A. No, it's only the best parts.

Q. Is this awk?

A. Depends what kind of files you use

Authors

- Jason Konikow

- Frank Spano
- Graham Patterson gpp2109
- Christopher Thomas
- Ayer Chan