File Input Reinterpretation Engine - (FIRE) - Language Reference Manual

Graham Patterson (gpp2109)
Frank Spano (fas2154)
Ayer Chan (oc2237)
Christopher Thomas (cpt2132)
Jason Konikow (jk4057)

Table of contents

1: Introduction

File Input Reinterpretation Engine (FIRE) is a scripting language inspired by AWK, bash, and other syntacically light languages. These languages are renowed for their ability to robustly extract, pattern-match, and manipulate text files. FIRE seeks to emulate this functionality with a more attractive, C-Family inspired syntax and intuitive semantics.

FIRE is intended to be utilized with large sets of delimited data, like `csv` files. FIRE is also animated by the premise that files are first class citizens.

FIRE was built by a team of Columbia University undergraduates for Professor Stephen Edward's Programming Language and Translators course. FIRE is written in OCaml , utilizing libraries built in `C` , and leveraging the `LLVM` compiler back-end.

2: Lexical Conventions

2.1 Strings

Strings are sequences of characters surrounded by double quotes `"`. As string has the type `str`.

2.2 Identifiers

Identifiers are a sequence of characters consisting of uppercase or lowercase letters, digits, underscores or dashes.

2.3 Keywords

The following identifiers are restricted from use:

- `int`
- `str`
- `regx`
- `if`
- `else`
- `elif`
- `else`
- `for`
- `while`
- `break`
- `return`
- `func`
- `array`
- `file`
- `print`
- `return`
- `map`
- `filter`
- `true`
- `false`
- `null`
- `void`

2.5 Regular Expressions

Regular expressions are a special sequence of characters used for pattern matching surounded by single quotes `'` and preceded by the keyword `r'`. Regular expressions are of type `regx`.

2.6 Comments

FIRE only supports the use the block comments. Comments are initiated with the `/*` symbol and terminated by the `*/` symbol. Everything in between the symbols will be ignored by FIRE during compilation

3: Meaning of Identifiers

Identifiers are names that correlate to a single value, a function, or an array. The restrictions on valid identifiers are found in section 2.2.

The scope of an identifier can be either global or local. A local identifier's scope is limited to inside the braces `{ ... }` in which they are declared, whereas global identifiers are declared outside any braces.

4: Expressions

4.1 Primary Expressions

Primary expressions in FIRE can either because accessing an array type or a function call. The primary expression is followed by an expression of either asignments in the case of a function call or an accessing of the datatypes in an array.

1. `Primary Expression [ expression ]`
2. `Primary Expression ( expression )`

4.2 Assignment Operator

The assignment operators `=` returns the value of the expression that is evaluated on its right-hand side and stores it in the identifier on the left hand side. The scope of that identifier is described in section 3

4.3 Function Calls

Functions take in arguments by value except in the case of other functions which are passed by reference. Functions need to be assigned before being called but can be declared anonymously. An anonymous function's scope is within the function that it is declared.

4.4 Logical Negation

Although FIRE does not directly support a boolean type, it equates `true` expressions to `1` and false expressions to `0`. As such, the logical negation operator `!` will convert the result of the logical expression on which it is applied to the inverse of what would normally be expected.

All non-zero integers are evaluated as `true` where as `0` is false.

4.5 Logical AND Operator

The logical AND `&&` is a short circuit operator, and returns 1 if and only if the expressions on its left and right both evaluate to 1.

4.6 Logical OR Operator

The logical OR `||` operator is a short circuit operator and returns 1 if either of the expressions on its left or right return 1.

4.7 Relational Operators

The relational operators `<, >, <=, <=, ==` return `1` if the expression on the left side of the operator has the expected relation to the operator on the right-hand side.

These relationships amongst ints are determined by natural ordering. Strings can only be evaluated using the `==` operator.

4.8 Pattern Match Operator

The pattern match operator `===` returns `1` if the regular expression or string on its right side conforms to the rules laid out by the regular expression on its left side and returns `0` otherwise.

Pattern matching operator l-values must be type `regx` and r-values must be type `string`.

Example: `r'[a-z]+' === word`

4.9 String Concatination Operator

The string concatenation `^` operator returns a new string that is the concatenation of the string on its left side and the string on its right side.

4.10 Bracket Operator

The bracket operator `[]` can be used with either `string` or `array`.

When used on `array` it is supplied a key and returns the corresponding element, or `-1` if it does not exist.

When used on `string` it functions in a similar manner to character arrays in C. The r-value in brackets is an integer and returns the letter of that index, however unlike C, FIRE does not support chars so it returns a string of length 1.

```
str hello = "hello";
str o = hello[4];
```

4.11 Slice operator

The slice operator `[x:y]` is used on a string and returns a substring.

4.12 Map

The map keyword allows a programmer to apply a function to every element of an array.

Example: `map(f,arr1);`

The map keyword does not mutate the values in the provided array; it instead returns a new array with results of every element of arr1 after they are passed to func f.

4.13 Filter

The filter keyword takes any function that returns a boolean and applies it to elements of an array. This allows filter to quickly generate a new array that consists of elements that match whatever member criteria your function tests for.

Example: `filter(f,arr1);`

In the above example, arr1 contains an array of strings that are either `dog` or `cat`. func f returns true if the element is equal to `dog`. The above expression would return an array only consisting of every element in arr1 that contains `dog`.

5: Declarations

Objects are instantiated via declarations, which explicitly assign a data type to a variable. In Fire a variable cannot be declared without also being assigned to a value. Types are explicit in FIRE. The format of a declaration is as follows:

`{type} {variable name} = {value};`.

5.1 Data Types

5.1.1. `int`

Identifiers of type `int` represent the natural numbers. The upper and lower bounds for int are defined by the architechtural constraints of the computer.

Example:

```
int num = 32;
```

`int` types can also be assigned to the result of expressions:

```
int a = 34 * 2 + (2 / 1);
```

5.1.2. `str`

Identifiers of type `str` are used to represent sequences of characters, strings. Strings are immutable and can be declared in the following manner:

Example:

```
str myString = "Hello World";
```

5.1.3. `file`

Files are regarded as first-class citizens in FIRE. This is made apparent by the importance and centrality of files. A `file` type represents either an existing file or a file that is to be written. This allows the programmer to more easily perform operations on the file.

The syntax for instantiating a `file` object is as follows:

```
file f = file("filename.csv", "<mode>");
```

In the example provided above, two arguments are fed into the `file(...)` argument: *filename* for reading, writing or both, and *mode*. The *mode* argument can be `r` for read only, `w` for write only, and `rw` for both.

Example:

`file f = file("test.csv", "rw");` will open the File named test.csv in the current directory for both reading and writing.

```
file f = file("filename.csv", "<mode>", "<delim>");
```

An optional third argument *delim* may be provided to the constructor specifying a delimiter for reading. If the *delim* argument is not supplied it will default to `\n`.

Example:

`file f = file("Program.java", "rw", ";");` will open the File named `Program.Java` in the current directory for both reading and writing. Calls to `read()` will read in chunks of the file delimited by the `;` character.

5.1.4. `func`

`func` objects reference functions and are treated as first class citizens. The structure of `func` variable declarations is as follows:

```
func <return type> <name> = (<parameters>) => { <function body> };
```

Where:

- `<return type>` is the type returned by the function

- `<name>` is the variable name of the function
- `<paramters>` are the expected parameters for the function
- `<function body>` is the body of the function

Once a function has been assigined to a `func` type it becomes a "named function" that is callable using that name e.g `funcName();`

A function that does not return anything has a return type of `void`. The void return type allows for programmers to create functions that are useful for their side effects.

Paramaterization

Named functions can be passed to other functions as a parameter as follows:
`func void saySomething = () =>{ print("something"); }; func void doSomething = (func f) => { f(); }; doSomething(saySomething);`

Caveats

- Fire does not support function overloading
- Fire does not support genericity in functions

5.1.4. `array`

The `array` type is a dynamic collection of elements. Inspired by AWK's associative arrays, an `array` collection maps keys of one type to values of one type. Keys and values do not have be the same type, but all keys must share the same type and all values must share the same type.

The structure of `array` variable declarations is as follows:

`array arr[<key_type>, <value_type>];`

Example:

`array arr[int, string];`

The assignment of variables has the following structure:

`arr[<key_value>] = <element>;`

Example:

`arr[17] = "age17";`

Finally, a programmer can retrieve a value associated with a key with the below syntax:

`int element = arr[<key_value>];`

Example:

`int age = arr["age"];`

5.1.5. `regx`

Regular expressions are supported in FIRE. Via the `regx` type, which assigns an object to a regular expression. That object can then be passed as a parameter to functions that utilize regular expressions to a pattern match or extract data.

The structure of a `regx` declaration is as follows:

`regx myPattern = r'<pattern>'`

Example:

```
regx myPattern = r'[a-z]';
myFunction(someString, myPattern);
```

The syntax for the regex patterns are as follows:

- `\` escapes any of the operators for the literal character

- `^` matches only the beginning of the string
- `$` matches only the end of the string
- `.` matches any single character
- `[ ... ]` defines a character list, where the character list can also be character range. This matches any string containing these characters
- `[^ ... ]` defines a character list, but negates them. This matches any string *not* containing these characters
- `|` matches either expression `e1` or `e2`
- `( ... )` groups expressions together where `...` is some regular expression
- `*` matches the preceding character 0 or many times
- `+` matches the preceding character at least once

5.1.6. `bool`

Boolean objects contain a value of either `true` or `false`. They can be declared on their own, and are used in conditional statements.

The structure of a boolean declaration:

```
bool switch = true; // or false
```

Example:

```
int x = 0;
bool switch = true;

// infinite loop
while(switch) {
    x = x + 1;
}
```

6: Statements

6.1 Print Statement

The print statement prints a literal value, or the value returned by an expression.

`print("i will be printed to stdout");`

6.2 Conditional Statements

Conditional statements evaluate expressions and execute code based on the truth values of those expressions.

```
if (<expression>) {
    <code block>
}
elif {
    <code block>
}
else {
    <code block>
}
```

6.3 For Statements

For statements iterate over an `array` and execute a code block for every iteration. The code block can mutate the array elements, but can not add or remove elements from the array. ex:

```
for(str current : stringArray) {
    print(current);
}
```

6.4 While Statements

While statements execute a code block until its provided condition fails to be met.

```
while(<condition>) {
    <code block>
}
```

7: Code Sample

The below is an example of `FIRE` in action. In the snippet below, a `FIRE` program is used to extract phone numbers that begin with a particular area code:

```
user:~ $ cat PhoneNumbers.csv
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 is from NJ based on 201 area code
//

func string isNJ = (str phoneNumber) => {
    return phoneNumber === r'201-[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]';
};

func array[int, str] extractRegion(func isRegion, file f) {

    array njnums[int, str];

    str number = f.readLine();
    int i = 0;

    while(number != "") {
        if(isNJ(number)){
            njnums[i,number].add;
             i = i + 1;
             number = f.readLine();
        }
    }
    return njnums;
}

file f = file("PhoneNumbers.csv", "rw", ",");

print( extractRegion(isNJ, f) );

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