OpenFile Project Proposal
COMS W4115

Guanming Qiao, Xingjian Wu, Binwei Xu, Yi Zhang, Chunlin Zhu
`gq2135, xw2534, bx2157, yz3206, cz2487`

February 2, 2018

1 Description

OpenFile is a programming language that simplifies file processing for information extraction and format formalization across multiple programming languages and structured natural language texts. It eases the process of cleaning the code, extracting information about the program structures, and implementing regex-ruled text preprocessing for users. OpenFile also recognizes user-defined parameters in the form of regular expressions (regex), which extends the customizability of built-in text parsing functions. In sum, OpenFile provides an intuitive yet highly customizable file-processing interface for developers and data scientists.

2 Target Users

The main target users of OpenFile are developers and data scientists who have background knowledge in regex and the demand to extract or modify information (structural or textual) for relatively large programs.

3 Use Cases

- A user wants to delete all comments in a large set of code files.
- A user wants to extract out all print commands in a large set of code files with the line numbers and file names attached to the print commands.
- A user wants to define a specific format for Java code and apply it to all the java files he has written.
- A user wants to know the statistics of all the emails he sent to a particular person. (For example, the average length of the email.)
- A user wants to generate a skeleton of a large text file he is reading with user-defined marks as the marking points for sections.

4 Motivation

It often happens that important information lies in a huge amount of irrelevant data. For example, when developers want to contribute to an open source project, they should have a good understanding of the overall skeleton code structure where the detailed implementation is less important. However, if the project contains thousands of lines of code, which is highly possible in real-world works, developers would have a hard time targeting the key structures and functions of the project itself due to the large chunks of implementation details. It is painful and time-consuming to extract useful information from source code. Therefore, we want to develop a language with file analysis features, which can significantly lower the intellectual as well as time consumption devoted to decoupling critical information from implementation details . We would also love to enable mass application of regex-defined text processes.

5 Key Features

- **Flexibility:** Built-in functions support multiple programming language and document formats. It also contains interface for users to create new class with customized formats.
- **Versatility:** multiple built-in functionalities including bracket matching, indentation, information extraction and format manipulation designed to process program files and literal documents in various forms.
- **File Preprocessing:** transform written code into data-interchange format, JSON. Output file includes key information such as author, classes, functions and attributes.
- **Document Parsing:** parse natural language documents such as emails, legal documents and job posts, extract key information into JSON files, and label the parsed sections.

6 Syntax

6.1 Primitive Data Types

`type`	`default`	`description`
`int`	0	a `32-bit` signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$
`long`	0	a `64-bit` signed two's complement integer, which has a minimum value of -2^{63} and a maximum value of $2^{63}-1$.
`float`	0.0	a single-precision `64-bit` floating point
`double`	0.0	a double-precision `64-bit` floating point
`bool`	`false`	only two valid values: `true` and `false`

`char`	`'\u0000'`	a single `16-bit` Unicode character. It has a minimum value of `'\u0000'` (or `0`) and a maximum value of `'\uffff'` (or `65535` inclusive)
`String`	`""`	an immutable character string
`Array`	`[]`	an ordered sequence that holds a fixed number of values of a single type. Length is established when the array is created
`File`	`"\"`	a file object representing the actual file on the disk
`Regex`	`@.@`	a sequence of characters that defines a search pattern
`Directory`	`"/"`	a directory object representing the actual directory on the disk
`Null`	`null`	a null object for null values and void return types

6.2 Operators

operator	syntax	data type
Basic assignment	'a = 'b	all
Basic arithmetic	'a+'b, 'a-'b, 'a*'b, 'a/'b	int, long, float, double
Modulo	'a % 'b	int, long
Equality	'a=='b, 'a!='b	all
Comparison	'a>'b, 'a<'b, 'a<='b, 'a>='b	int, long, float, double, char
Logical AND	'a and 'b	
OR	'a or 'b	bool
NOT	not 'a	
String Concatenation	"Hello"+"World"	String
File append	'a » FileB	'a * File
File Overwrite	'a > FileB	'a * File

6.3 Control-flow

- **`if-else`** statement, used to test conditions:

```

    if(bool true){
        (* code goes here *)
    } else {
        (* code goes here *)
    }

```

- **`for`** statement, which provides a compact way to iterate over a range of values:

```

    for(int i = 0; i < n; i=i+1){
        (* code goes here *)
    }

```

- **`while`** statement continually executes a block of statements while a particular condition is `true`:

```

    while(bool true){
        (* code goes here *)
    }

```

6.4 Built-in Functions

- **`parse`** takes a file and a regular expression as parameters. It returns an array of strings. This function splits the content of the File into an array of strings according to search pattern described by Regex.

```
    parse(File file, Regex reg) -> String[]
```

- **`replace`** takes a regular expression and a string as parameters. It has no return value. This function replaces all substrings that satisfies the Regex with another replacement String.

```
    replace(Regex reg, String replace) -> null

    (* code example *)
    "abc".replace(@b.@, "pineapple")
```

- **`insert`** takes a regular expression and a string as parameters. It has no return value. This function insert an integer at the beginning of all substrings that satisfies the Regex.

```
    insert(Regex reg, int i) -> null
```

- **`append`** takes a regular expression and a string as parameters. It has no return value. This function insert an integer at the end of all substrings that satisfies the Regex.

```
    append(Regex reg, int i) -> null
```

7 Code Examples

```

    (* print hello world *)
    "hello world" >> console;

    (* create a file and write "hello world!" to it *)
    File hello_out << "hello_world.txt";
    "hello world!" > hello_out;

    (* manipulating regular files *)
    File file << "/usr/sample.txt";
    File output << "output.txt";

    (* things between the @s are regex *)
    String[] sections = parse(file, @[A-Z]+:.*\n\n\n@);
    for (int i = 0; i < sections.length; i++) {
        sections[i].replaces(@[A-Z]+:@, "title:"); (* TEST: ==> title: *)
        sections[i].inserts(@title@, i); (* title ==> 1title *)
        sections[i].append(@title@, i); (* title ==> title1 *)
        sections[i] >> output;
        sections[i] >> console;
    }

    (* manipulating a large number of files inside a directory *)
    Regex email = @abc...aa///n...@
    Directory dir << "/somedir";
    Directory newDir = createDirectory("/output");
    for (File file : dir){
        if (contains(file, email)){
          File out << "newFile";
          file >> out;
        }
    }

```