OpenFile Project Language Reference Manual
COMS W4115

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

February 27, 2018

1 Introduction

OpenFile is a programming language that simplifies file processing for information extraction and format formalization for 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. Inspired by Awk, 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 Lexical Conventions

2.1 Comments

OpenFile contains comments block enclosed by characters `"/*"` and `"*/"`. Any character wrapped within the starting and ending flags is treated as comments.

2.2 Identifiers

Identifiers are a sequence of letters, digits and underscores. The starting character must be an alphabetic letter. Upper and lower cases are treated as different characters.

2.3 Keywords

The keywords contain the following ones: `if , elif, else, for, while, return, true, false, and, or, not` The operators include all of the following: `=, +, -, *, /, %, @, ==, !=, », «, >, <, <=, >=, ;, „ (, ), {, }, [, ], ", /*`

2.4 Integer Constants

Integers are in base 10, written as a sequence of digits from 0 to 9. An integer can also be negative, which is denoted by `'-'` at the front.

2.5 Float Constants

Floats are consist of a sequence of digits that follow the integer pattern followed by a dot and more digits. Either the integer component or the fraction component could be omitted.In any case, the decimal point must not be omitted.

2.6 String Literals

A string consists of a sequence of characters and is wrapped by double quotes (`"..."`). Strings offer several escape sequences that are represented by chars within the string: Newline (`\n`), tab (`\t`), double quote (`\"`), single quote (`\'`), backslash (`\\`)

2.7 Regex Constants

A regex starts and ends with the `'@'` character, and is designed for pattern matching. A range pattern is defined by [X1 -X2], such that X1 and X2 are defined at either uppercase letters, lowercase letters, or digits. X1 must have a lower value than X2 on the ASCII table.

3 Syntax Notation

The syntax notation of this reference manual for all code, including words, characters, regular expressions and keywords, will be typed in the `teletype` font.

4 Meaning of Identifiers

Identifiers are names that refer to corresponding variables or functions. Refer to section 2.2 for a precise definition and format of identifiers. Variable and function identifiers don't have inferred types. It means the programmer needs to make explicit type declarations together with an identifier.

4.1 Storage Scope

The identifiers are splitted into global or local storage scope. Global identifiers are initialized outside the local blocks such as helper functions, for-loops or if-else statements, enclosed by the curly brackets and are accessible to the entire program. Local identifiers are only accessible within the block they are declared.

5 Conversion

OpenFile will support string-to-integer conversion as well as integer-to-string conversions. However, string-to-float and float-to-string will not be supported. Users need to implement this kind of conversion themselves.

5.1 String to Integer Conversion

Only string with the right integer format will be successfully converted to an integer type, otherwise, a "wrong format" error will be thrown. For example, `string "1143"` will be converted to `int 1143`. However, `"32.45"` will not be able to be converted to an integer and a "wrong format" error will be thrown.

5.2 Integer to String Conversion

A proper integer type can be converted to string. For example, `int 1123` will be converted to `string "1123"`

5.3 Arithmetic Conversion

Only two numeric data types are supported: int and float. For arithmetic operations, if one operand is a float, the other operand will be converted to a float as well. The result will be converted to whatever type specified. For example, `int a = 5 + 6.0`. First, `5` will be treated as `5.0` and then be added to `6.0`, getting `11.0`. Secondly, the result `11.0` will be converted to `int 11` since the type of the result is specified as `int`.

6 Expressions and Operators

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values. Here are some example:

```
1
1 + 1
a = 1 + 1
```

6.1 Assignment Operators

Assignment operators store values in variables. The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand (commonly referred to as the "lvalue") cannot be a literal or constant value.

```
int x = 10
float y = 45.12 + 2.0
int z = (2 * (3 + function () ))
```

6.2 Incrementing and Decrementing

The increment operator `++` adds 1 to its operand. The operand must be a variable of one of the primitive data types. You can apply the increment operator either before or after the operand. Here are some examples:

```
int x = 5
x++ /* x is now 6. */
char w = '1'
++w; /* w is now the character '2' (not the value 2). */
```

A prefix increment adds 1 before the operand is evaluated. A postfix increment adds 1 after the operand is evaluated. Code examples are following:

```
int x = 5
print (x++)  //it will print 5
print (++x) //it will print 6
```

6.3 Array Subscript

You can access array elements by specifying the name of the array, and the array subscript (or index, or element number) enclosed in brackets. Here is an example, supposing an integer array called `myArray`:

```
myArray[0] = 5
```

6.4 Array membership

Array membership operator includes "in". If the key is in the array, it would return 1, otherwise return 0. It associates from left to right.

```
array = ["a", "b", "c"]
"b" in array
"a" in ["a", "b", "c"]
3 in [1, 2, 3]
5.1 in [1.1, 2.2, 3.3]
```

6.5 Multiplicative Operators

Multiplicative Operators include *, /, %, and associate from left to right. Modulo (%) takes two integers and results in integer. Division (/) takes two numbers, either integer or float, and results in float. Multiplication (*) takes two numbers, either integer or float. If two numbers are integers, then it results in integer, otherwise it results in float.
Expressions here must be numeric.

```
float x = 10 / 2.1
int y = 10 / 2
int m = 10 % 3
int a = 10 * 2
float b = 10 * 3.12
```

6.6 Additive Operators

Additive Operators include +, -, and associate from left to right. Addition (+) and subtraction (-) take two numbers, either integer or float. If two numbers are integers, then it results in integer, otherwise it results in float.
Expressions must be numeric.

```
int a = 10 + 2
int b = 10 - 3
float c = 10 - 2.6
float d = 10.4 + 2.7
```

6.7 String Concatenation

String Concatenation uses the operator +, and associates from right to left. It takes two strings, and results in a string.
Expressions should be strings. If expression is integer or float, it will be implicitly converted into string.

```
string h = "Hello" + " " + "World"
string b = "I " + "like " + "Ocaml"
```

6.8 Relational and Equality Operators

Relational and equality operators include >, >=, <, <=, ==, !=, and associate from left to right.
Relational and equality operators take either two strings or two numeric values. It returns 1 if true, and 0 if false. When taking two strings, strings are compared character by character with lexicographical order. When taking two integers or two floats, numbers are compared mathematically. When taking one integer and one float, integer is converted into float first, and then compared as two floats.

```
2 > 10
2.3 >= 10.3
10.1 < 2
10 <= 2.8
5 == 7
2.6 != 3.7
"Hello" == "Hello"
"Hello" > "World"
```

6.9 Matching Operators

Matching operators include ~, !~, and associate from left to right. When using ~, string matching regex would return 1, otherwise 0. When using !~, string matching regex would return 0, otherwise 1.
Both operators take a regex and a string.

```
@a|b@ ~ "a"
@a*|b*@ !~ "apple"
```

6.10 Array membership

Array membership operator includes "in". If the key is in the array, it would return 1, otherwise return 0. It associates from left to right.

```
"a" in ["a", "b", "c"]
3 in [1, 2, 3]
5.1 in [1.1, 2.2, 3.3]
```

6.11 Logical AND Operators

Logical AND operator includes &&, and associates from left to right. && would evaluate left-side expression first, and would only evaluate right-side expression if left-side expression is true. && would return 1 if both expressions are true, otherwise it would return 0.
It should take two boolean values and return a boolean value.

```
true && false
(3 > 0) && ("a" == "a")
```

6.12 Logical OR Operator

Logical OR operator includes ||, and associates from left to right.
|| would evaluate left-side expression first, and would only evaluate right-side expression if left-side expression is false. || would return 1 if at least one expression is true, otherwise if both expressions are false, it would return 0.

```
false || true
(3 > 0) || ("a" != "a")
```

6.13 Assignment Expression

Assignment operator includes =, and associates from left to right.
For example, x = y = 8 would have the same effect as x = (y = 8).

```
int x = 3
int y = 5
x = y = 8
float c = 5.5
String s = "apple"
```

6.14 Print Operator

Print operator includes print, and associates from right to left.
It would print the given expression into the terminal. Multiple expressions can be printed with comma separated.

```
print (5)
print (3.5)
print ("Hello World")
```

6.15 Precedence Order

When there are multiple operators in one expression, operators would be evaluated according to precedence order. An intuitive rule is multiplication has a higher order than addition. Here is the list of order from high to low:

- Function calls, array subscripting, and membership access operator expressions.
- Unary operators. When several unary operators are consecutive, the later ones are nested within the earlier ones: !-x means !(-x).
- Multiplication, division, and modulo.
- Addition and subtraction.
- Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to.
- Equal-to and not-equal-to expressions.
- Logical AND expressions.
- Logical OR expressions.
- All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.
- Comma.

7 Declarations

7.1 Primitive Data Types

type	declaration	description
int	int a = 0	a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$
float	float a = 0.0	a single-precision 64-bit floating point
char	char a = 'a'	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	String a = "PLT"	an immutable character string
Array	int[] = {1,2,3,4}	an ordered sequence that holds a fixed number of values of a single type. Length is established when the array is created
Regex	Re a = @.z*	a sequence of characters that defines a search pattern

7.2 Functions

You write a function declaration to specify the name of a function, a list of parameters, and the function's return type. A function declaration ends with a semicolon. Here is the general form:

```
return-type function-name (parameter-list);
```

return-type indicates the data type of the value returned by the function. You can declare a function that doesnâĂŹt return anything by using the return type void.

8 Statements

In OpenFile, statements are executed in order from top to bottom and from left to right.

8.1 Assignment Statements

Assignment statements specify an identifier and its equality, ending in semicolon.

```
Id = value
```

8.2 Conditional Statements

Conditional statements evaluate expressions and execute other statements based on the truth of the expression evaluation.

```
if ( expression )  statement
if ( expression )  statement  else statement
if ( expression )  statement  elseif ( expression ) statement else statement
```

8.3 For Statements

For statements execute a specified statement *stmt* based on a condition. The statement will be executed in a loop as long as the condition is true. *exp*1 is used as initial state, *exp*2 is used as increment, and *exp*3 is used as final condition.

```
for ( exp1 ; exp2 ; exp3 )  stmt
```

8.4 Return Statements

A return statement will return nothing or an expression from a function.

```
return
return expr
```

9 Scope

9.1 Global Variable Scope

All variables declared in OpenFile outside of function scopes are global. Declared and initialized variables are accessed globally, and their values can change globally.

```
/**Results:
  *var is 5
  *var now is 6**/

    int var = 5;
    void changeGlobal(){
      var = 6;
    }
    print ("var is: " + var);
    changeGlobal();
    print ("var is now: " + var);
```

9.2 Local Variable Scope

OpenFile does not have local variables outside of function scopes. All variables first declared in a function is local to the scope of the function.

```
/**Results:
  *Error, local not found in scope.**/

    void changeGlobal(){
        int local = 6;
    }
    changeGlobal();
    print ("local is: " + local);
```

10 Grammar

- Basic-unit:
  Declaration
- Declaration:
  Function-definition
  Variable-definition
- Function-definition:
  declaration
  declaration-list
  statement
  parameter-list
- Variable-definition:
  declaration
  declaration Assignment Statement
- Statement:
  print-expression
  Assignment Statement
  Conditional Statement
  For statement
  Return statements
  Break Statements
  Continue Statements
- print-expression:
  expression
- Conditional-expression:
  logical-or-expression
  logical-or-expression ? expression : conditional-expression
- logical-or-expression:
  logical-and-expression
  logical-or-expression `||` logical-and-expression
- logical-and-expression:
  Array-member-expression
  logical-and-expression `&&` array-member-expression
- array-member-expression:
  matching-expression
  string-concat-expression in primary-expression

- matching-expression:
  relational-expression
  matching-expression
  string-concat-expression
  matching-expression `!` string-concat-expression

- relational-expression:
  string-concat-expression
  relational-expression
  relational-operator string-concat-expression

- relational-operator:
  one of `<, <=, ==, =>, >, !=`

- string-concat-expression:
  additive-expression
  string-concat-expression string-concat-expression

- Additive-expression:
  Multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression

- Multiplicative-expression:
  Unary-expression
  multiplicative-expression `*` unary-expression
  multiplicative-expression `/` unary-expression
  multiplicative-expression `%` unary-expression

- unary-expression:
  exponentiation-expression
  unary-operator unary- expression

- unary-operator:
  one of `+ - !`
  prefix-inc-expression:
  postfix-expression `++`
  prefix-inc-expression `--`
  prefix-inc-expression

- postfix-expression:
  prefix-field-expression
  postfix-expression[expression]
  postfix-expression(argument-expression-list)
  postfix-expression`++`
  postfix-expression`--`

- primary-expressions:
  identifier
  (expression)
- argument-expression-list:
  expression
  argument-expression-list
- expression Identifier:
  Constant
- Function-definition Constant:
  integer-constant
  float-constant
  Regex-constant
  string constant

11 Code Examples

11.1 Username Pattern Matching Example

```
1
2     //print hello world
3     print("hello world");
4
5     //regex types support regular expression operators
6     //pattern = (char + space* + num*) - (not c || not g)*
7     Regex pattern = @\([a-Z]\) + [[:space:]]* + [0-9]*@ \- @([^c]|[^g])*@;
8
9     //arrays support all primitive OpenFile types
10    String[] sections = replace(file, \$1 , pattern , "hello");
11    for (int i = 0; i < sections.length; i++) {
12        print(sections[i]);
13    }
14
```

11.2 Text File Editing Example

```
1  /**     Input file:
2    *       1. (021)-123-1234  \$200,000     Peter
3    *       2. (010)-123-1234  \$30,000      Brian
4    *       3. (010)-123-1234  \$350,000     Stewie
5    *       4. (010)-123-1234  \$96,000      Mag
6    *       5. (0351)-123-1234 \$1,000       Quagmire
7    *       6. (0351)-123-1234 \$100,000     Bonnie
8    *       7. (0351)-123-1234 \$2222,000    Mia
9    **/
10     int highEarners=0;
11     int lowEarners=0;
12
```

```
13      //Delete \$ and () in phone number and income columns
14      replace(file , \$3, /\$|,/,"");
15      replace(file , \$2, /\(|\)|-/,"");
16
17      string[] personID=\$1;
18      string[] phoneNum=\$2;
19      string[] salary=\$3;
20      string[] name=\$4;
21
22      for (int i = 0; i < salary.length(); i++){
23          if (salary >= 100000) {
24              highEarners=highEarners+1;
25          } else {
26              lowEarners=lowEarners+1;
27          }
28      }
29
30      int count[50];
31      for (int i = 0; i < phoneNum.length(); i++){
32          areaCode = substr(phoneNum, 0, 3);
33          count[areaCode]++;
34      }
35
36      print ("High Earners: " + highEarners + '\n');
37      print ("Low Earners: " + lowEarners + '\n');
38      for (int i = 0; i < phoneNum.length(); i++){
39          areaCode = substr(phoneNum, 0, 3);
40          print("Area code: " + areaCode + " has " + count[areaCode]" + people."
        + '\n');
41      }
42
43  /**
44    *      Output:
45    *      High Earners: 4
46    *      Low Earners: 3
47    *      Area code 021 has 1 people
48    *      Area code 010 has 3 people
49    *      Area code 0351 has 3 people
50    **/
```