# File and Directory Manipulation Language (FDL)

Rupayan Basu rb3034@columbia.edu
Pranav Bhalla pb2538@columbia.edu
Cara Borenstein cjb2182@columbia.edu
Daniel Garzon dg2796@columbia.edu
Daniel L. Newman dln2111@columbia.edu

October 28, 2013

# Table of Contents

# 1. Introduction

File and Directory Manipulation Language (FDL, pronounced "fiddle") provides a simple and intuitive syntax for managing file systems. By providing the user with new data types, and an extensive list of mathematical and logical operators, what used to be tedious and time consuming will now be easy and fast. Users can write programs that organize their file systems by conveniently copying/moving files and directories to different locations, and removing files and directories from specific file paths, through the use of special operators. Users can loop through subdirectories and files contained within a chosen directory, with a template to browse the file/directory tree stemming from that directory. Files/Directories can be organized by the built-in attributes spanning from last modified date to size. Built-in data structures like list and dict allow users to conveniently store and access groups of files/directories.

# 2. Data Types

2.1. int: The set of all integers in the range of -2^31 - 1 to +2^31 + 1.

2.2. bool: A binary variable having two values, 1 for true and 0 for false. Used in conditional statements, such as if and while. Can be used to compare paths, lists, dictionaries and integers.

2.3. string: A sequence of characters surrounded by double quotes.

2.4. path: String that specifies a valid location of a file or directory in the file system for which the following attributes are defined:

- 2.4.1. name: Field that holds the name of the file or directory.
- 2.4.2. created_at: Field that holds the date when a file or directory was created.
- 2.4.3. modified_at: Field that holds the date of the last time a file or directory was modified.
- 2.4.4. kind: Field that holds the kind of the path. It will return file or directory.

2.5. dict: A dictionary is an associative array containing a collection of primitives, indexed by a key that is a singleton integer, or a string containing a path.

2.6. list: A list is an unordered collection of primitives. It can contain zero or more elements that are indexed by an integer value that gets incremented every time an element is appended.

# 3. Lexical Conventions

3.1. Identifier

An identifier is a sequence of lowercase and uppercase letters, digits (0-9) and underline "_". Each identifier begins with a lowercase letter or underscore.

3.2. Comments

Comments are specified by a double slash "//" and can take up only one line.

```
//this is an example comment
```

3.3. End of Statement

A newline '/n' specifies the end of a statement and a tab '\t' specifies the scope

3.4. Keywords

Keywords are special identifiers reserved as part of FDL itself. Here is the list of keywords recognized by FDL:
***path***, ***bool***, ***dict***, ***list***, ***int***, ***if***, ***else***, ***then***, ***while***, ***for***, ***in***, ***do***, ***true***, ***false***, ***return***, ***break***, ***continue***, ***def***, ***trash***, ***main***

### 3.5. Constants

FDL has string constants called ***paths***. They specify the location of a file or directory in memory. FDL has a special path called ***trash*** which stores the path a file is moved to for deletion. FDL also stores the following escape sequences as constants:
Newline "\n", Tab "\t", Double Quotation "\""

## 4. Functions

### 4.1. *Function Definitions*

-A function definition in FDL begins with the keyword 'def', followed by the return type, function name and a parenthesized list of input parameters, with each parameter preceded by the type. The statements that form the body of the function begin on the next line, indented by a tab. The 'return' keyword is used to return values to the calling statement.
-Every valid FDL program must have a 'main' function which is always executed first. The 'main' keyword is reserved.
-All user defined functions must be defined before the main function, at the top of the program.
-No statements can exist outside function definitions

Example:

```
//Function to return all files in a directory and its sub-directories up to a
//given depth
def list listAllFiles(path dirPath, int depth)
        list fileList = []
        dict dirDepth = {dirPath = 1}
        int currentDepth

        for Dir in (keys(dirDepth)) do
                if dirDepth{Dir} <=depth then
                        currentDepth = dirDepth{Dir}
                        for item in Dir do
                                if isFile(item) then
                                        fileList = append(fileList, item)
                                else
                                        dirDepth{item} = currentDepth + 1
        return fileList
```

### 4.2. *Build-in Functions*

4.2.1. len - takes a path/string/dict/list variable as input and returns the 'length' based on the following definitions:

path: length = depth of the directory/file at the end of the path with respect to the directory at the beginning of the path
string: length = number of characters in the string
dict: length = number of key-value pairs in the dictionary
list: length = number of items in the list

4.2.2. keys - takes a dictionary type variable as input and returns a list of all the keys found in a dictionary. If the dictionary is empty it returns NULL.

4.2.3. values: takes a dictionary type variable as input and returns a list of all the values found in a dictionary. If the dictionary is empty it returns NULL.

4.2.4. delete - takes a dictionary type variable, and a key, as input and deletes an entry of a dictionary. If the key is invalid, the function will return NULL, else it deletes the entry and returns TRUE.

4.2.5. append - takes a list variable and a variable to be appended to the list, and returns a new list with new variable appended at the end.

4.2.6. print - takes a list of arguments separated by commas, converts them to a string and displays them in the console.

## 5. Expressions and Operators

### 5.1. **Primary Expressions**

5.1.1. identifier

An identifier is a primary expression, declared with a type, that can be assigned a value of that type, to which it refers

5.1.2. constant

An integer is a primary expression of type int.

5.1.3. bool

A bool is an int, storing the value 0 or the value 1.

5.1.4. string

A string is a primary expression composed of ASCII characters.

5.1.5. path

A path is a primary expression, in the format of a string. It refers to a valid path of a file or directory from the current directory of the program or originating in the home directory of filesystem

5.1.6. ( expression )

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. Parenthesis are used to indicate precedence, to compute the values inside the parentheses before handling the rest of the associate expressions from left to right.

5.1.7. def primary-expression ( expression-list )

"A function call is a primary expression preceded by the reserved word "def" and followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type ''function returning . . .'', and the result of the function call is of type ''. . . ''"

5.1.8. dict { key }

The curly brackets "{" "}" are used to access the dictionary elements where the variable before the starting bracket is the dictionary name and the variable inside the brackets is the key. It can be used to create and update elements in the dictionary

The curly brackets are also used to add and create dictionary elements.

***dict*** dirDepth = {dirPath = 1}

dirDepth { "/home" } = 0

5.1.9. list [ index ]

The square brackets "[" "]" are used to access list elements, where the variable before the starting bracket is the list variable and the variable inside the brackets is the index of the element.

***list*** fileList = []

5.2. Multiplicative operators

The multiplicative operators *, /, and % group left to right, and are used only for mathematical operations with ints

5.2.1. expression * expression

The binary * operator indicates multiplication.

5.2.2. expression / expression

The binary / operator indicates division.

5.2.3. expression % expression

The binary % operator yields the remainder from the division of the first expression by the second.

5.3. Additive operators

The additive operators + and - group left to right.

5.3.1. expression + expression

The result is the sum of the expressions. If both operands are int, the result is int. If one of the expressions is a string, the result is a string, in the form of the second expression concatenated to the end of the first expression.

5.3.2. expression1 - expression2

The result is the difference of the operands. Both operands must be int and the result is int.

5.4. Relational and Equality operators

The relational operators group left to right, and return the boolean pertaining to the truth of the expression (1 if true, 0 if false)

5.4.1. expression < expression

5.4.2. expression > expression

5.4.3. expression <= expression

5.4.4. expression >= expression

5.4.5. expression == expression

5.4.6. expression != expression

5.5. expression && expression

The && operator returns 1 if both its operands are nonzero, 0 otherwise. .

5.6. expression || expression

The || operator returns 1 if either of its operands is nonzero, and 0 otherwise.

5.7. Assignment operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

5.8. lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue.

5.9. Move and Copy operators

The <<- and <- operators group left to right, and are used to move or copy the file/directory of path_scr to the directory path_dest on the left of the operator

5.9.1. path_dest <<- path_src

The file/directory in path_src is moved into the path_dest directory.

The below code moves the path to trash

***trash*** <<- "home/fdl.pdf"

5.9.2. path_dest <- path_src

The file/directory in path_src is copied into the path_dest directory.

5.10. Comma Operator

It is used to separate function arguments.

# 6. Declarations

## 6.1. Variable Declarations

Variables must be declared before they are used in the program, except for the ones that are used as "iterators" in for loops. A variable declaration has the following form:

*var_type var_name*

The *var_type* can be ***int***, ***bool***, ***list, dict, string*** *or* ***path***. The *var_name* can be any valid identifier which is letter followed by any number of letter or digits. If a variable is declared, in the following assignment, value assigned to the variable must have exactly the same type as declared. Variables can also be initialized during the declaration. A declaration with initialization has the following form:

*var_type var_name = expression*

The expression must have exactly the same type as *var_type*.

***path*** variables are declared like other variables with the ***path*** keyword before the identifier. A ***string*** can be assigned to the ***path*** variable and interpreted as a "path"

to a directory or file in the file system.
Also we allow ***path*** variables to be used as "associative arrays" in for loops. That is to say we can write

***for*** (file ***in*** path_variable)

Here, ***for*** and ***in*** are keywords and *file* and *path_variable* are variables. The *path_variable* must be declared whereas the file variable need not be declared beforehand. It is automatically interpreted as a variable of type ***path*** that loops through all the sub-paths in *path_variable*.

### 6.2. Function Declaration

A function declaration has the following format:

***def*** *return_type* function_name ( *arg_type* arg1_name, *arg_type* arg2_name, ….)

We use the keyword ***def*** to identify that what follows is either a function declaration or definition. return_type and *arg_type* are one of the predefined types ***int***, ***bool***, ***list, dict, string*** *or* ***path.*** *function_name* and *arg1_name* and the other arguments can be any valid identifiers.

## 7. Statements

### 7.1. Statement

A statement is composed of expressions, which can be grouped by operators. We use newline to separate one statement from the next.

```
string str1
string str2
str1 = str2 + " hello "
```

The above code snippet has 3 statements that are separated by the newline character ('\n').

7.2. If Statement

If statement consists of keywords ***if, then*** and ***else***. It has the following two varieties:

```
if ( expression ) then
        statement

if ( expression ) then
        statement1
else
        statement2
```

The expression must be of ***bool***. To ensure scope the statements must be indented inside the if using the tab. In the first case, if the expression is evaluated to true, then statement is executed. Otherwise statements after the if statement is executed. In the second case, if the expression is evaluated to true, then statement1 is executed, otherwise statement2 is executed.

7.3. While Statement

While statements consists of keyword ***while*** and it allows a statement to be executed for any number of times, until the expression evaluates to false. It has the following format:

```
while ( expression )
        statement
```

The expression must be of type ***bool***. To ensure scope the statements must be indented inside the while using tab. The expression is evaluated before the execution of the statement and statement will be executed until the expression is evaluated to false.

7.4. Break Statement

Break statement consists of keyword ***break***. It's used to jump out of the while or for loop and hence must always be used inside one of these two. The following is an example of using break

```
statement
while ( true )
        break
```

7.5. Continue statement

Continue statement consists of the keyword ***continue.*** It's used to pass control to the next iteration of a while or for loop and hence must always be used inside one of these two.

```
while ( true )
        i = i + 1
        if ( i < 10 ) then
                continue
        else
                break
```

7.6. For Loop Statement

for loops are used to iterate through a list of paths or a path to a directory, we interpret the variable given as an associative array and we iterate through their sub-paths one at a time. ***for, in*** and ***do*** are the keywords that are used to define the for loop.

```
for file in path_variable do
        statement

for file in list_variable do
        statement
```

the statement that needs be run over repeatedly needs to be indented inside the for statement.

7.7. Return Statement

Return statement consists of keyword ***return***. A function must have a return statement to return its value to its caller. It can return an expression that is evaluated to type ***path***, ***int***, ***bool*** or ***string***, or it can return nothing when the function uses void as its return type.

```
return expression
return
```

# 8. Scoping and Indentation

Our language is modeled on the python rules for indentation and scope, where whitespace is used to delimit program blocks. It does away with the requirement of putting braces("{ }") around code blocks, but we require some extra symbols to detect the end of if, for and while expressions which has already been explained in the previous sections.

We compare two code in our language and in C for better understanding. The following code is C.

```
void foo(int x)
{
   if (x == 0) {
      bar();
   } else {
      foo(x - 1);
   }
}
```

Now the same program in our language

```
def void foo(int x)
   if (x == 0)  then
      bar()
   else
      foo(x - 1)
```

Scope of variables is within the code blocks they are declared, similar to the code block scoping rules in C. Functions are of global scope from the position they are defined till the end of code. Function calls are possible as long as the target function has been defined before the current position.

# 9. References


9.1. Ritchie, D. M., "C Language Reference Manual", Bell Telephone Laboratories, http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf

9.2. Python Software Foundation, "The Python Language Reference — Python v2.7.5 documentation", http://docs.python.org/2/reference/index.html#reference-index