Fli-o: File Manipulation Language

Language Reference Manual

Matthew Chan (mac2474)
Justin Gross (jg3544)
Gideon Cheruiyot (gkc2112)
Eyob Tefera (et2546)

I Introduction

Fli-O was developed to create a seamless way to for users to interact with files, especially large documents that require file or directory manipulation. To avoid the confusion around buffers and input/output, we plan to allow users to open documents and then handle the closing and handling of the file ourselves. This should greatly increase the ease of interacting with files as a user just had to open them before proceeding to work with the file, no longer does the user of our language have to do any file memory management aside for indicating which file they want to work with in the first place. In short, we want to simplify the process of working with file I/O and change it from a pain point to a hallmark of the language.

Furthermore we want to give users the ability to process these files and do additional file management operations on these files while keeping the I/O process as simple as possible. Some processing that users would be able to do include directory deletion, file manipulation, search and replace, merging multiple files, splitting files, and easily appending to files. We also plan to include several file management functions that users can build from in order to create custom file management processes that simplify a user's workflow. Our language will be written in OCaml and then compiled into LLVM code.

2 Lexical Conventions

2.1 Tokens

There are five classes of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, and comments as described below (i.e. white space) are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token. (Liberally borrowing from K&R)

2.2 Comments

Comments are signified by two forward slash characters '//' after which the comment is terminated by the new line.

2.3 Identifiers

Identifiers are a sequence of digits, letters, and the underscore, where the first element in the sequence must be a letter (upper or lower case). The explicit range of available ASCII characters that define a 'letter', 'digit', and the 'underscore' are:

‘0’-’9’ {digit}
‘a’-’z’|’A’-’Z’ {letter}
‘_’ {underscore}

2.4 Keywords

The following case-sensitive identifiers are reserved for use as keywords in Fli-o and may not be used otherwise:

int	string	file	dir	if	elif	for	foreach	in
and	or	not	def	return				

2.5 Literals

2.5.1 Int Literals

An integer constant consisting of a sequence of digits that is expressed strictly in decimal notation and 64 bit in size with the initial bit indicating sign.

2.5.2 String Literals

A string literal, also called a string constant, is a sequence of characters surrounded by single quotes as in ‘...’

2.6 Separators

There are three sets of types of separators, {}, () and [], which are used to denote separation between other tokens and group them in specific ways.

3 Types

3.1 Primitive Types

int: 63 bit signed integer. The first bit represents the sign, the others represent the value.

```
int i=3;
```

3.2 Derived Types

string: String literal. A sequence of ASCII characters surrounded by single quotes. Immutable, any attempts to reassign a string variable’s value result in the creation of a new string.

```
string s=’hello’;
```

file: A file handle for file manipulation as well as reading and writing. Contains metadata about the filepath (string), size of file in bytes (int), and a list of permissions (array). Initialized using the fopen function.

```
file f;
f = fopen("c\mydoc\dict.txt");
```

dir: A type that exemplifies a file directory. It contains metadata with the filepath of the directory itself (string), a list of permissions (array), two indexed lists (arrays) that contain strings of file paths: one for its subdirectories and the other for the files within the directory. They are sorted in lexicographical order.

```
dir d;
d = dopen("c\mydoc\");
```

Arrays: `type id assignment`

Arrays are 0-indexed mutable lists of same-typed values. They are instantiated and semantically indicated by a type followed by an identifier and then two braces encasing an integer that initializes the array size. Individual elements from the array can later be accessed by specifying its index into the array enclosed by square brackets at the end of the name of the identifier. Arrays may be list of more arrays, allowing for unbounded multidimensional arrays. Arrays have an a property of size that indicates total members of the array (array).

```
string[5] arr;
arr = ['hello','my','name','is','Eyob'];
```

4 Expressions

Expressions are sequences of one or more tokens that have a value. Identifiers and literals are also expressions by this definition. The precedence of these expressions beyond the base literals and identifiers is listed in order below from highest to least except where explicitly noted. Operators described in the same section are closely related and share the same precedence

4.1 Parenthetical Expression

```
(expression)
```

A simple expression that takes the value of the interior of the parentheses. Used to express the highest precedence amongst expressions.

4.2 Function Call

```
identifier(arguments)
```

A function call is consists of an identifier, which is the name of the function, followed by a sequence of zero or more arguments enclosed by parentheses. Function calls pass their arguments by value. The value of the function call itself is of the return type specified in the function declaration with the value of the expression within the return statement. Function calls are left to right associative.

4.3 Array Access

```
identifier[expression]
```

Array access is an expression that accesses a member of the array. The expression must be of an integer type (which is the desired index), while the identifier must be of one of the derived array types. The value of the array access expression is the value at the specified index. Array access is left to right associative.

4.4 Property Access

```
expression.identifier
```

The dot operator allows for access of a property of a type. The identifier is the name of the property, which must be specified in the type definition. The expression must ultimately be an identifier of the type that corresponds to the identifier for the property. The value of this expression is the value of the property for that explicit identifier. Property access is left to right associative.

4.5 Logical Not

```
NOT expression
```

The not operator expresses logical negation using the keyword `not`. The expression must of integer type. If the value is not 0, the resulting value from the expression is 0, and if the value is 0, the resulting value is 1. The operator is right to left associative.

4.6 Multiplicative/Division Operators

```
expression * expression
expression / expression
```

These are the two basic multiplicative operators. The two expressions on both sides must be of the integer type. The times operator * denotes multiplication and the evaluation of the expression is the integer that results from multiplying the two. Similarly the divide operator / denotes division and the evaluation of the expression gives the resulting value of the truncated integer quotient. If the second operand in the division operation is 0, the result is undefined. The operators are left to right associative.

4.7 Additive/Subtraction Operators

```
expression + expression
expression - expression
```

These are the two basic additive operators. The two expressions on both sides must be of the integer type. The plus operator + denotes addition and the evaluation of the expression is the integer that sum of adding the two. Similarly the minus operator - denotes subtraction and the evaluation of the expression results in the integer difference. The operators are left to right associative.

4.8 Relational Operators

```
expression < expression
expression > expression
expression == expression
expression != expression
```

These are the two basic relational operators greater than >, less than <, equals to ==, and not equals !=. The two expressions on both sides must be of the integer type. If the specified relation is true, the resulting value is an integer, 1, and 0, if false. The operators are left to right associative.

4.9 Logical And

```
expression AND expression
```

The and operator expresses logical and using the keyword `and`. The expressions must be integers. If the values of both expressions are not equal to zero, the resulting value from the expression is 1, and if the one of the values is zero, the resulting value is 0. The operation is left to right associative and if the left hand side is evaluated to 0, it does not evaluate the right hand side already knowing the result is 0.

4.10 Logical Or

```
expression OR expression
```

The or operator expresses logical or using the keyword `or`. The expressions must be integers. If the values of one of the expressions are not equal to zero, the resulting value from the expression is 1, and only if both of the values are zero, is the resulting value 0. The operation is left to right associative and if the left hand side is evaluated to 1, it does not evaluate the right hand side already knowing the result is 1.

5 Statements

These statements are executed in sequence and contain no inherent value. The SEQUENCING token refers to using the semi-colon to terminate some statements. They are described below with some minor categorization due to similarity.

5.1 Expression Statement

```
expression;
```

The basic statement of an expression that is terminated. Frequently function calls.

5.2 Variable Declaration Statement

```
type identifier;
type identifier=expression;
type identifier=[arguments];
```

These statements indicate the declaration of variables. The first form indicates declaration without initializing where the

5.3 Return Statements

```
return expression;
return ;
```

These statements return to the call of the function. The expression must match the return type specified in the function declaration statement. If there is a type, it must be same, and if there type, any return statements must be of the second form above. The value of the expression is returned to function call expression and is the function call expression's value.

5.4 Block Statement

```
{statement_list}
```

This statement indicates grouping and limiting of scope. It is a series of 1 or more statements enclosed by {}. Any identifiers declared within a statement like this can not be used in statements outside of the block statement.

5.5 Control Flow Statements

```
if (expression) statement
if (expression) statement else statement
if (expression) statement elif_list
if (expression) statement elif_list else statement
elif (expression) statement
```

The control flow statements listed above allow for different statements to be optionally executed. If the first expression, which must be an integer, evaluates to not 0, the first statement executes in all forms. The other forms indicate differently other statements to take and what conditions must be true to take them. `elif_list` indicates a series of 1 or more elif statements, which provide secondary conditional expressions (i.e. else if) that are checked if the the preceding if and elif statements evaluate to 0. If those expressions are again non-0, the following statement is executed and the rest are not considered. If none of the preceding if or elif (if there are any) expressions are evaluated to be non-0, then if there is an else, the following final statement is executed.

5.5 Loops

```
for (expressionOpt; expressionOpt; expressionOpt)  statement
foreach expression in expression statement
```

These are two forms of iterative statements:

(largely from K&R):
In the for statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must be integer type, it is evaluated before each iteration, and if it becomes equal to 0, the for is terminated. The third expression is evaluated after each iteration, and thus specifies a reinitialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. Any of the three expressions may be dropped. A missing second expression makes the implied test equivalent to non-0 and thus always true.

In the foreach statement, the first expression is an identifier that functions as a member of an array and the second expression is the identifier that indicates the desired array. Thus, the statement is executed for each element in the array. The element in the array that is currently being iterated over can be accessed within the loop by using the first identifier.

5.6 Piping Statement

```
expression PIPE expression
```

The piping operator, '|>' allows Fli-o to utilize outside utilities and programs in subprocesses. It pipes the output of the utility on the left to the input of the utility on the right. Each of the expressions must be of the string type and correspond to a utility listed in the imports at the beginning the program with the import statement. The program will be checked at runtime to ensure that the utilities listed in the import statement exist The string for each indicates the execution of that program and contains the program path of the desire utility.

5.7 Assignment Statement

```
identifier=expression;
identifier=[array_literals];
identifier=[expression];
```

These are three different ways to assign values to identifiers. The first form is used for all non-array types. The expression must correspond to the type of the identifier. Strings are literal values, so each time they are changed, new string is created. Files and dicts are references to larger structured types and so merely assign the identifier the old reference value. In particular, files and directories must be opened by fopen and dopen, respectively, though the structure referred to by a specific identifier may freely change. The next two forms indicate two different ways to assign values to an array. The first form simply replaces whatever values had previously been in the array with the new series of identifiers/literals within the brackets. The third form is used to change the size of the array, where the expression is an integer and can only increase the array size.

6 Functions

6.1 Function Declaration

```
def identifier (params) type_opt {statement_list}
```

A function can be declared anywhere within a function (besides within another function or statement block). It is indicated that this is a function and the first declaration of such by the def (define) keyword. It is then followed by the identifier that names the function and then a series

of comma-delimited parameters (0 or more) that are enclosed by parentheses. Each parameter consists of a type and a local identifier to be used in the function. After the parameters, there may be a type. This type is the return type of the function. Finally, there are braces that surround the series of statements that make up the function itself.

6.2 Function Structure

Each function is passed its arguments by value (though the value of directories and files are references). The arguments included in the function call must match the types and order of the parameters listed in the function declaration. The return type of the function indicates if there need be return statements; a return type requires that a return statement with an expression of the correct type end all possible functions. Alternatively, lack of a return type means that there need not be a return statement in the function, though any return types must be of the second form listed in 5.3.

7 Built-in Functions

```
fopen(string filePath) file{}
```
- Takes a string of the filepath, creates the file object and returns the file.

```
dopen(string filePath) dir{}
```
- Takes a string of the filepath, creates the dir object and returns the dir.

```
copy(file target, string dest) file{}
```
- Takes a string of the filepath, creates the file object and returns the file, or it returns an empty newly created file if the target is invalid

```
move(file target, string dest) string{}
```
- Takes a string of the filepath, moves the file object and returns the files new location, if it fails it returns the current location

```
delete(file target) int{}
```
- Returns 0 or 1 depending upon if it successfully deletes anything

```
rmdir(dir target) int{}
```
- Remove a directory
- Returns 0 or 1 depending upon if it successfully deletes anything

```
concat(string s1, string s2) string{}
```
- Doesn't modify s1 or s2, just returns the concatenated string

```
read(file target, int num_chars) string{}
```
- Returns a string with the next num_chars of characters from the file

```
readLine(file target) string{}
```
- Returns a string up to the first \n character encountered from the file

```
write(file target, string contents) file{}
```
- Overwrites the target file with the contents of string
- If the target file does not exist, create the file

```
appendString(file target, string contents) int {}
```
- Appends a string to the end of the target file, returns 1 if successful, 0 if unsuccessful

```
appendFile(file target, file other) int{}
```
- Appends the contents of other file to the target file

```
print(anytype) string{}
```
- For int prints the integer value
- For string prints the string contents
- For file prints the file name
- For dir prints the dir name

8 Scope

Local scope is defined to be any series of statements enclosed by {}, whether it be in a function or a general block statement. Any string and integer type variables are limited to their local scope, if it exists, and may not be referred to outside of their scope. In particular, files and directory types are not closed even if the identifiers are local to the scope of a function and persist until the program quits, wherein these types are closed. Strings and ints do not persist outside of their scope. In general, scoping is otherwise very similar to that of C.

9 Utility Processing

```
import expression;
```
At the beginning of each program, there will be a series of import statements that relate to outside programs that can be run by this Fli-o program. They are checked at runtime to ensure

that the utility does exist at the indicated path. The expression must be a string type that is the filepath of the desired utility.

10 Grammar

program:
    Declaration EOF

Declaration:
    // empty declaration are valid
    Declaration Import
    Declaration Function-Declaration
    Declaration Statement

Import:
    IMPORT StringLiteral

Function-Declaration:
    Def ID (params) type_opt {statement_list}

Params:
    // empty params is valid
    paramlist

Paramlist:
    type ID
    paramlist,type ID

Variable-Declaration-Statement:
    type ID Sequencing(;)
    type ID Assignment expression Sequencing(;)
    type ID Assignment ArrayLiteral Sequencing(;)

ArrayLiteral:
    [arguments]

Arguments:
    // empty arguments are allowed
    arglist

Arglist:
    expression

arglist, expression

StatementList:
// empty statement list are allowed
statementList statement

StatementOpt:
SEQUENCING(;)
Statement

Statement:
Expression Sequencing
Variable-Declaration-Statement
ID Assignment Expression Sequencing(;)
ID Assignment ArrayLiteral Sequencing(;)
ID Expression Assignment expression Sequencing(;)
Return expression sequencing
Return Sequencing
{StatementList}
FOR (expressionOpt Sequencing expressionOpt Sequencing expressionOpt) Statement
FOREACH expression IN expression statement
IF (expression) statement
IF (expression) statement ELSE statement
IF (expression) statement elifList
IF (expression) statement elifList ELSE statement

ElifList:
elif
elif-list elif

Elif:
ELIF (expression) statement

ExpressionOpt:
// empty expression_opt is valid
expression

Expression:
expression PLUS expression
expression MINUS expression
expression TIMES expression
expression DIVIDE expression
expression PIPE expression
expression LESSTHAN expression

expression GREATERTHAN expression
expression EQUAL expression
expression NOTEQUAL expression
expression AND expression
expression OR expression
NOT expression
expression DOT ID
INTLIT
STRINGLIT
ID
ID [ARGUMENT]
ID [EXPRESSION]

typeOpt:
// empty typeOpt is valid
type

Type:
int
String
File
Dir
type [IntLiteral]