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 Newman dln2111@columbia.edu

December 20, 2013

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 to manage and program is now easy and fast. Users can write programs that organize their file systems by conveniently copying/moving files and directories to different locations through the use of special operators, and conveniently accessing specific attributes of the files. Users can loop through subdirectories and files contained within a chosen directory, with a template that can be utilized to browse the file/directory tree stemming from that directory. Files/Directories can be organized by the built-in attributes such as last modified date and names. The built-in list data structure allows users to conveniently store and access groups of files/directories.

Furthermore, by implementing the “path” type to hold a valid file or directory path, FDL simplifies how files are manipulated, giving users the ability to define a path with a string, then use predefined operators on the files to move or copy them within the filesystem, leaving the code clean and succinct. One is able to reorganize a file system in a way that is simple to code, and simple for others interpret.

2. Language Tutorial

Structure:

An FDL program has the following structure:

a. Declaration/Initialization of global variables.
b. Definition of functions
    i. Each program must contain a 'main' function, which will be executed from the command line when the program is run
    ii. Within each function, variables declarations/initializations come first, followed by statements.
    iii. All functions, loops and if blocks are terminated with the keyword 'end', and spacing is left to the user to organize code by preference (such as indenting blocks or keeping code dense)

Paths in FDL:

Paths are central to FDL. A 'path' datatype allows users to create variables using the relative/absolute paths of files/directories as follows:

```
path file1
path file2 = './Documents/foo.txt'
```

To iterate through files in a directory, FDL provides a unique for-loop:

```
for ( filename in dirname )
        /* statements */
end
```

A special operators are provided to copy/move files from one directory to another:

```
destDir <- file1 /*copies file1 to destDir */
destDir <<- file2 /* moves file2 to destDir */
```

Every path variable, comes built in with 'attributes' which can be used to obtain useful information about paths:

```
pathvar.kind : returns 0 for invalid, 1 for file, 2 for directory
pathvar.name : extracts the name of the file/directory from the path
pathvar.type: return "." followed by the file extension, like ".ml" for "fdl.ml"
```

Lists in FDL:

Lists are a useful data structure in FDL. List variables can be declared/initialized as follows:

```
list l1 = [1,2,3]
list l2 = []
```

Lists can hold elements of different fdl types:

```
l = ['a', 1, file1]
```

Finally, FDL supports the following list operators:

```
l.add(a)        /* adds a variable a to the list l */
l.remove(b)     /* removes an item matching variable b from l */
```

A special if-in construct helps check if an item exists in a list:

```
if file1 in list1 then print file1.name
```

A simple FDL program:

The following program copies a file from one specified location to a destination directory:

```
def int main()
        path src = "./sample_dir/sample_file.pdf"
        path dest =  "./test"
        dest <- src
        return 0
end
```

Within the main method, the path variable, 'src', is initialized to the file path of a file that we wish to copy. The file path of the directory into which we wish to copy 'src' is stored in the path variable 'dest'. The copy operator, '<-' is then called, and a copy of the src file will now exist in both the src location of the file system, as well as in the dest location.

One step further:

If we wish to do more than copy just one file, we can place the copy operation into a loop that iterates through a full directory, moving all files in the source directory to a target directory, as follows:

```
def int main()
      path src = "./sample_dir"
      path dest = "./test"
      path f
      for ( f in dir )
         print "file path "
         print f
         if (f.kind == 0) then
            print f
            dest <- f
         end
      end
      return 1
end
```

In this example src is set to the directory filled with files we wish to move rather than setting it to one file within the directory, ‘dir’. An enhanced for loop, which acts on all files in the specified directory, executes the for loop’s definition, with each subsequent file as the program iterates through the directory. Along the way print statements were specified, in order to keep track of what is happening in the console as the file system is manipulated behind the scenes.

In this particular case, we check that the files we are going to copy are of type ‘file’ rather than ‘directory’, before copying them, exhibiting yet another feature of FDL. Specific file attributes can be accessed by calling them from ‘path’ type variables.

3. FDL Language Reference Manual

3.1 Data Types

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

3.1.2 **`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.

3.1.3 **`string:`** A sequence of characters surrounded by double quotes.

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

`name:` Field that holds the name of the file or directory, at the end of path

`type:` Field that holds the extension of the file, valid only in case of directories.

`kind:` Field that holds the kind of the path. It will return 1 for file, 2 for directory and 0 if the path is invalid.

3.1.5 **`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.

3.2 Lexical Conventions

3.2.1 **Identifiers**

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

3.2.2 **Comments**

Comments are specified like a block comment in C using the open “/*” and close “*/” reserved symbols.

3.2.3 **End of Statement**

A newline “/n” specifies the end of a statement and a tab “\t” specifies the scope

3.2.4 **Keywords**

Keywords are special identifiers reserved as part of FDL itself. Here is the list of keywords recognized by FDL:

`path, bool, string, list, int, void, if, else, then, while, for, in, true, false, return, def, main, print`

3.2.5 **Constants**

FDL has string constants called **`paths.`** They specify the location of a file or directory in memory. FDL also stores the following escape sequences as constants:

Newline “\n”, Tab “\t”, Double Quotation “\””

3.3 Functions

3.3.1 **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

3.4 Expressions and Operators

3.4.1 **Primary Expressions**

3.4.1.1 `identifier`

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

3.4.1.2 `constant`

An integer is a primary expression of type int.

3.4.1.3 `bool`

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

3.4.1.4 `string`

A string is a primary expression composed of ASCII characters.

3.4.1.5 `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 file-system.

3.4.1.6 `(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.

3.4.1.7 `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

“...””

3.4.1.8 **`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 = []
```

3.4.2 Multiplicative Operators

3.4.2.1 **`expression * expression`**

The binary * operator indicates multiplication.

3.4.2.2 **`expression / expression`**

The binary / operator indicates division.

3.4.3 Additive Operators

The additive operators + and - group left to right.

3.4.3.1 **`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.

3.4.3.2 **`expression - expression`**

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

3.4.4 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)

3.4.4.1 **`expression < expression`**

3.4.4.2 **`expression > expression`**

3.4.4.3 **`expression <= expression`**

3.4.4.4 **`expression >= expression`**

3.4.4.5 **`expression == expression`**

3.4.4.6 **`expression != expression`**

3.4.4.7 **`expression && expression`**

The && operator returns 1 if both its operands are non-zero, 0 otherwise.

3.4.4.8 **`expression || expression`**

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

3.4.5 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.

3.3.5.1 **`lvalue = expression`**

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

3.4.6 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

3.3.6.1 **`path_dest <<- path_src`**

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

3.3.6.2 **`path_dest <- path_src`**

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

3.4.7 Comma Operator

It is used to separate function arguments,and list arguments.

3.5 Declarations

3.5.1 Variable Declarations

Variables must be declared before they are used in the program, including the ones that are used as “iterators” in for loops. All variable must be declared at the start of a function before any other statements are entered. A variable declaration has the following form:

`var_type` `var_name`

The **`var_type`** can be **`int, bool, list, 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.

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.

3.5.2 Function Declarations

A function declaration has the following format:

```
def return_type function_name ( <arg_type arg_name> )
```

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

3.6 Statements

3.6.1 Statement

A statement is composed of expressions, which can be grouped by operators. We use newline to separate one statement from the next. There is a limitation in our language that all declarations of variables must be done at the start of the function.

```
string str1
str1 = “ hello “
```

The above code snippet has 2 statements that are separated by the newline character (‘\n’).

3.6.2 If Statement

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

```
if ( expression ) then
     statement
end

if ( expression ) then
     statement1
else
     statement2
end
```

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.

3.6.3 **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.

```
while ( i < 10 )
    i = i + 1
    print i
end
```

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.

3.6.4 **For In Statement**

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

```
for (file in path_variable )
    statement
end
```

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

3.6.4 **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
```

3.7 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.

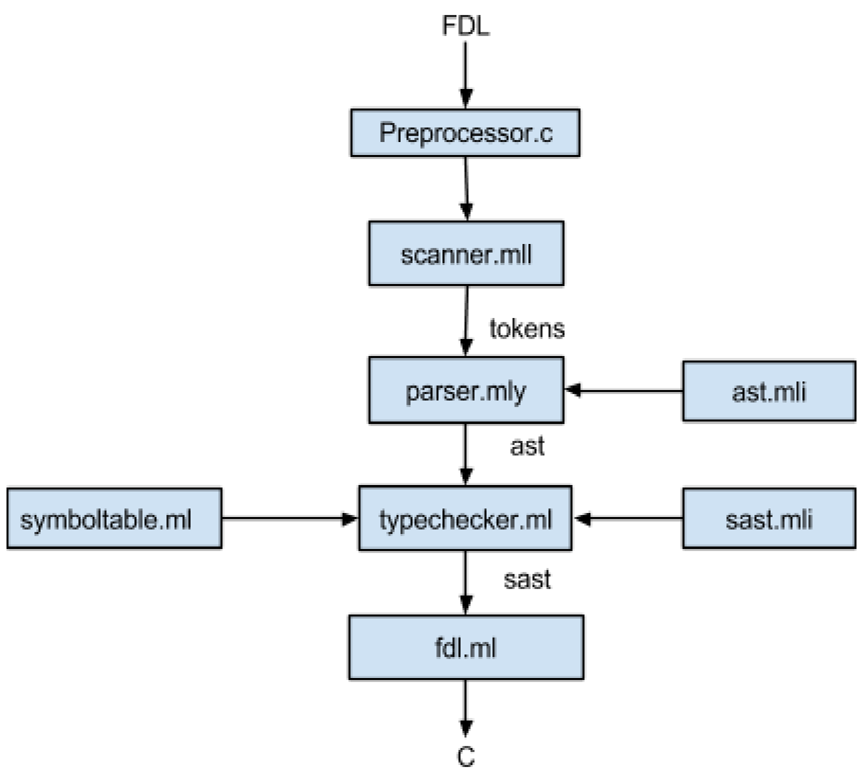
Scope of variables is within the code blocks they are declared, similar to the code block scoping rules in C. Functions are of global scope from the position they are defined till the end of code. Function calls are possible as long as the target function has been defined before the current position.

4. Architecture

The FDL compiler reads a program written in FDL and translates it into C code. The compiler itself is written in O'Caml and consists of the following main components:

1. **Preprocessor** - reads a program in fdl and adds syntactic details such as braces and semicolons
2. **Scanner** - reads the preprocessed fdl program and produces valid tokens
3. **Parser** - performs the syntactic analysis of the tokens and produces an Abstract Syntax Tree (AST)
4. **AST** - contains the definitions for the nodes of the abstract syntax tree
5. **Type/Scope checker** - recursively traverses AST, performs semantic checks, produces the Semantic AST
6. **Symbol table API** - an interface for managing the environments for local/global variables and function names
7. **SAST** - similar to the AST, but definitions contain additional semantic details useful for code generation
8. **Code generator** - recursively traverses the SAST and builds a string of code in the destination language C.

The block diagram below describes the overall control flow -

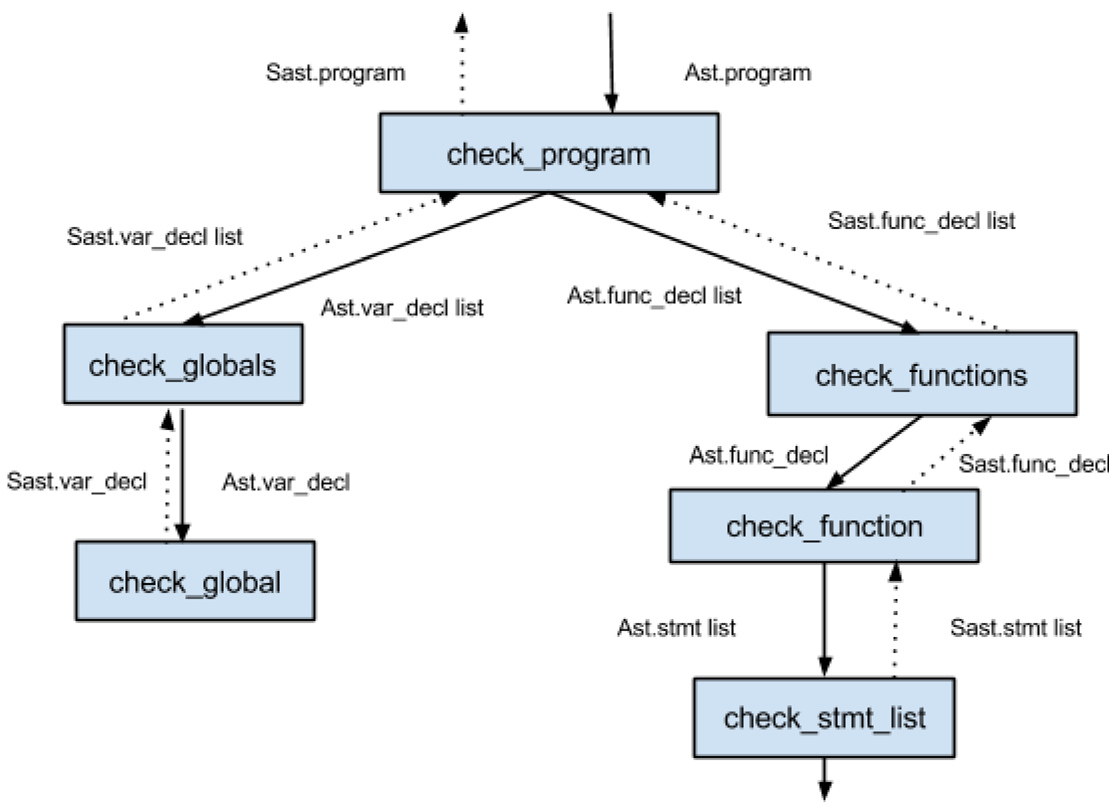


The entry point of the compiler is in fdl.ml, which handles not only the code generation, but also handles the control flow between the various components of the compiler, as shown in the extract below:

```
1. let input = open_in fname in
2.        let lexbuf = Lexing.from_channel input in
3.        let program = Parser.program Scanner.token lexbuf in
4.        let program_t = Typecheck.check_program program in
5.        let listing = string_of_program program_t in
6.        print_string listing
```

The ‘program’ in line 3 is the ast produced by the parser. The typechecker reads ‘program’ and produces ‘program_t’, the sast. Finally, the function string_of_program takes program_t as input in line 5 and prints out the generated c code in line 6.

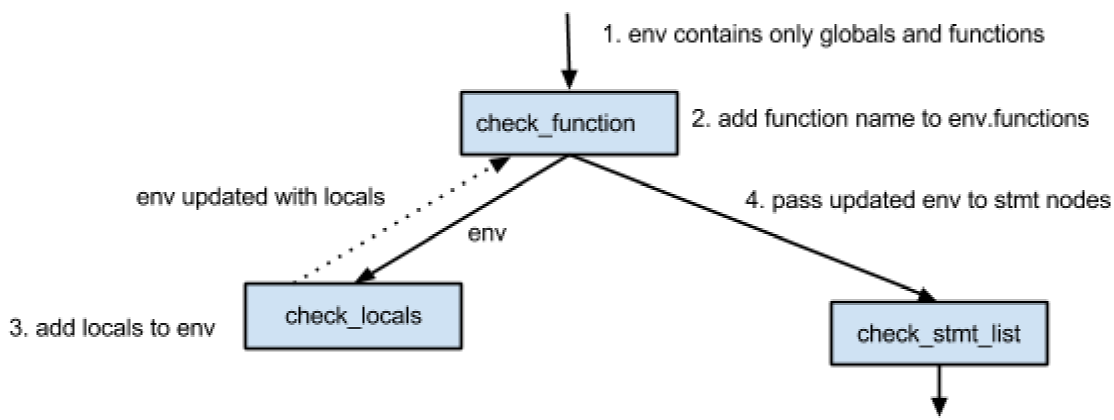
The typechecker recursively traverses the abstract syntax tree by invoking the functions corresponding to each node in the ast. The following diagram provides an overview of the control flow inside typechecker -



At each node of the ast, the typecheck performs scope and type checks before returning the corresponding node of the sast. For example, in order to keep track of the scope, the typechecker maintains an environment variable of type env as described below:

```
type env = {
     locals:          string StringMap.t;
     globals:         string StringMap.t;
     functions:       string list StringMap.t;
}
```

As the typechecker traverses a node in the ast, it updates the env variable and passes it to the node in the level below it. Functions initially have env with empty locals, but might contain globals and other function names. This env is then passed to nodes below functions, such as var_decl or stmt. The following block diagram describes how this is achieved:



The symbol table provides the interface for maintaining the env variable. It contains the following functions:

add_global: makes a global variable visible in the scope of the entire program.
add_function: makes a function name visible in the scope of the entire program.
add_local: adds a local variable to the current scope only.
find_function: used to check if a particular function name is visible in the current scope.
find_variable: used to check if a particular variable id is visible in the current scope

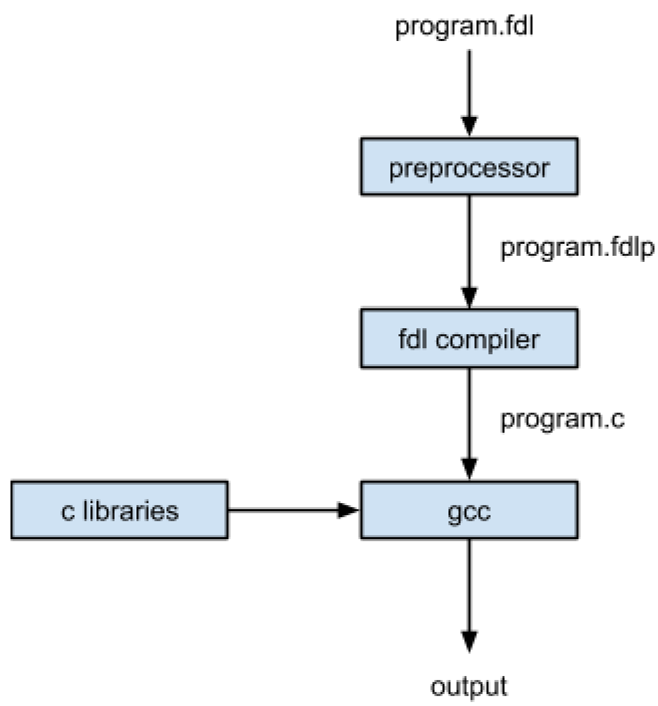
The code generator (string_of_program inside fdl.ml) performs a similar traversal on the sast, but this time, it builds the string of c code.

5. Compilation

To execute an fdl program the user needs to run the following script:

```
$ ./runfdl.sh path/to/fdl/file.fdl
```

which will produce an executable C-language file. It does so by running the .fdl file through the preprocessor, outputting a .fdlp file, then compiling that file into a .c file with the fdl compiler, and using a shell script to produce the executable. Once the executable is created, it will be automatically executed by the script.



6. Testing

The goal of our testing plan is to cover all basic functionalities we deemed critical to the FDL language. While our tests cannot catch every bug, we aim to cover as much functionality as possible. All of our tests are automated with provided scripts.

Phase I:

The first stance of testing occurred at early stages of development. When possible, we aimed to use Test Driven Development (TDD). This means that as we thought of functionality we wanted FDL to have, we wrote tests that tested the desired functionality and then implemented the functionality in FDL.

Phase II:

The second stage of testing occurs later on in the development cycle. While we aim to catch most errors early on with TDD, in order to ensure our testing plan is robust, we need to implement some tests later on in the development cycle to make sure no critical functionalities are left untested. Phase II of testing is specifically aimed at small functionalities that are added later on in the development cycle to make larger functionalities work. While their importance seems secondary to larger functionalities, they must be treated with the same importance as the original small functionalities that are tested in Phase I. Phase II of testing ensures all basic functionalities are working as expected.

Phase III:

The final stage of testing occurs at the end of the development cycle. The purpose of this stage is to ensure that all of our larger programs and more complicated functionalities work as expected. This requires confirming the robustness of our regression tests and making sure that all of our small tests work well together, not just independently.

7. Project Plan

We came up with many ideas for languages that we wanted to implement, that tried to either make an existing task easier or languages for many problems for which languages didn't exist. We finally decided on implementing a file manipulation language, since we felt from all our ideas it would be the most useful and challenging one. We had a lot of ideas of what are language could do and the minimum it should do. And had discussions to prioritize these ideas, so that the most important features got implemented first.

7.1 Planning

We discussed strategies on how to go about building the language and the compiler and one of first things we decided was that we wanted to build the language iteratively i.e. starting from a small program, that prints "hello world", we build on all the stages of the compiler as we keep adding functionalities. This minimizes error and we can keep adding test cases for the functionalities as we keep implementing them. We also decided on C to be our target language since we found it to be flexible enough for our varied needs, especially the use of pointers.

After we had divided up the initial work, we met regularly once a week to discuss our progress, solve each others problems and then divide up the work for the next week. We had a list of features we wanted to implement and we could keep track of our progress based on the number of features we had implemented.

7.2 Specification

After working on the LRM we were clear about the expectations we had from our language, though we were unsure whether we would be able to implement all of them. We went to build a basic skeleton of a compiler involving all the stages of scanning, parsing, the AST, typechecking, the SAST, and then the translation to C code. Post that we divided up the features into smaller units to implement and worked on them parallely. We kept updating the LRM when we felt it was necessary to make changes because our assumption were incorrect, or we found a better way to do things.

7.3 Development

Since we decided on the iterative approach to building a compiler it took us some time before we had the basic skeleton working. After this period, we were all able to work parallely, implementing various features simultaneously. Continuous integration through gitHub insured that our code got merged on a regular basis and we were not building conflicting code bases. In the latter stages we focused much more on the "typechecker" since there were a lot of invalid rules and boundary cases to be taken care of as the grammar for the language grew.

7.4 Testing

Iteratively developing test cases ensured that we could keep adding test cases as soon as we were done with implementing the feature. And using the test script, we could perform regression testing post each change and immediately recognize in case our changes broke anything. We can remember many instances where testing helped us identify missing cases in our implementation, and in turn let to a robust implementation and strong regression test suite.

7.5 Programming Style Guide

One of the “cool” aspects of our language is that we use no semicolons or curly braces, that make the code obtruse to read and grasp easily, since it absolves the program writer of the responsibility of indenting the code properly. It is much easier to understand code without these special characters, if it is indented properly. We have tried to follow this philosophy while writing our compiler as well, though the OCaml language has pretty good editing and formatting style inbuilt. The scanner, parser, ast, sast are written pretty much like typical implementations, though we have tried to arrange the scanner in such a way that it is easier to read, with multiple statements in a single line, and the statements are together if they belong to the same class in the scanner (various braces{}[](), logical operators, arithmetic operators, special symbols). The typechecker and the translator (fdl.ml) are our two largest files and we have tried to add comments wherever possible so that it is easy to recall and understand the working of the code. The typechecker actually begins at the bottom of the file, where it start dividing the whole program into smaller chunks and then moves on to checking the individual parts. And having a separate symbol table file helps remove a lot of repeated code from the typechecker.
All of our C functions are written in camel casing, and we have ensured none of our functions are larger that a single screen size, which is a good measure of the level of their modularity. In general we have tried to write only single statement per line in both the OCaml code and the C library code.

7.6 Software Development Environment

The fdl compiler was primarily developed for Apple's Mac OS platform. However, the compiler was also tested successfully on an Ubuntu system. Following are the details of the languages used for the various modules:

Module	**Language/Version**
FDL Compiler	Ocaml 4.01
Preprocessor	C (gcc 4.2.1)
Helper Libraries	C (gcc 4.2.1)
Runfdl	Shell script
Testall	Shell script
Cleanall	Shell script

Editor: Sublime Text 2.0.2
Version control: git 1.8.3.4
Repository: github.com
Online Document Collaboration: Google Docs

7.7 Project Timeline

09-09-2013	Team formed and Language idea developed
09-25-2013	Language proposal submitted
10-21-2013	Basic Code skeleton (Scanner, Parser, Translator)
10-28-2013	Language Reference Manual (Hello World done)
11-11-2013	Typechecker and Code generator finished
11-25-2013	Initial Testing Phase Completed
12-02-2013	Compiler fixes done
12-10-2013	Second Testing Phase Completed
12-20-2013	Project Report created and submitted

7.8 Project Log

09-09-2013	Team formed, different ideas discussed and FDL chosen
09-18-2013	Basic scanner and parser made
09-25-2013	Language whitepaper submitted, test cases decided
10-21-2013	Basic Code skeleton (Scanner, Parser, Translator)
10-28-2013	Language Reference Manual, “Hello World” program works, Created run and testsuites
11-04-2013	Typechecker added, Move, Copy implemented
11-11-2013	Typechecker done, Lists created
11-25-2013	Testing finished
12-20-2013	Project Report created and submitted

8. Lessons Learned

(and advice for future teams)

Pranav Bhalla:

One of the things with trying to program iteratively is that it takes some time to get the initial code base up, and during that time it is not possible to have multiple people work on it. We felt we lost a lot of time during the initial implementation, because once we had our skeleton up and running we were able to create features much faster than anticipated.

Since we were implementing different features by ourselves, the person implementing ended up writing the test cases for the feature as well, which was not a good practice. We found a lot more bugs when one person tried to use the functionality the other had implemented and in hindsight it might have been better to have another team member writing test cases for your features.

Daniel Newman:

I learned a lot about the complexities of building a language, but what made my learning experience unique was that I was working on a Columbia UNIX clic machine, running on Ubuntu, whereas the rest of my teammates worked on their Macbook OS machines. Initially we had not anticipated that this would result in any significant differences when running the code, but later on in the project I had trouble compiling, because errors that came up on my machine were not errors when running on a Mac. One example is the shell used for our runfdl.sh script (a component of our testall script, separated for purposes of testing individual tet cases whose functionality is being worked on. I had to change the way the shebang at the beginning of the script defined the shell to use, in order to get rid of Syntax errors that were not coming up for my peers.

Additionally, a function that we had intended to use to retrieve the date a file was created (rather than just 'last modified') as a File attribute, does not work when running on Linux machines because I learned that such machines do not store that data, and so trying to access that attribute from my machine in the code, results in compiling errors for the small C library that we created to accompany our program.

An additional problem encountered was using execl to implement the UNIX cp and mv commands for the unique move and copy operators of our program. execl executes and then terminates the process, so keeping such a function would likely have required us to complicate things and add a bit of code to fork and wait, and deal with multiple processes wherever execl is called, and so we switched to using the 'system()' function call.

Daniel Garzon:

By working on FDL I was able to realize the importance of good software engineering practices. Because of the nature of the assignment, we had to learn and put to practice techniques that are used in the industry today. After we had designed our language, it was time to start working on the implementation. As with anything the learning curve was really steep, but once we figured out a way to get “Hello World!” working the curve began to shallow. Because our language tries to imitate python, by implementing the preprocessor, and helping with the implementation of lists and paths I am much more aware of how python and many other languages work under the hood.

I also realized the power that Shell Scripts, C and Makefile have. By translating FDL to C we were able to do system calls and get real low access to the file system. This allowed us to implement the desired functionality of our language. Also, we kept out code modular by having each have its own custom library, so that it is available to all the .fdl files. Makefiles were extremely useful to get this libraries working. With just a single command, we were able to compile and link the libraries into .a objects. Once we had all these working, it seemed like a good idea to make a shell script to compile and run the FDL source, the libraries, and the .fdl program into a C executable. With a few lines of code, we were running FDL. Because we were using github to store our repository, and it is bad practice to leave executables in a shared repository, a clean script was also included to clean all executables.

In general, for future students my advice would be to start early, design a language that would really have an influence in your live and after try to always maintain a friendly relationship between your peers. In my opinion, one of the reasons we were able to accomplish what we did is because of the enjoyable and positive environment in which we all worked. By always helping each other, and collaborating the whole project which, at first might be a little threatening, will become an experience you will never forget.

Cara Borenstein:

Working on the typechecker gave me a much better understanding of and appreciation for all of the type-checking that languages I use support. I enjoyed switching off between the two roles of the programmer, who thinks of cases to implement, and the "devious" user, who tries to break the code. After developing FDL, I have a much better understanding of the difference between writing a program that seems to work and writing a full robust language with extensive error handling. For a robust program, the "devious" user becomes the programmer and this mindset makes programming both more fun and rewarding.

I also learned about the importance of modularity with testing. If you use larger tests, the functionality you are testing may work but a different functionality may cause the program to fail. For example, when testing binary operators, it seemed at first that the binary operator was not parsing correctly when it was actually the order of the declarations and initializations that caused the program to fail. We split this test into two tests: one for declaring and initializing variables and another for the binary operator.

For future groups, I would recommend becoming a git (version control) expert before beginning the coding process. We had recurring issues with merge conflicts, especially when the bulk of the code was finished and we were all working on the same files simultaneously. Ultimately, we all had the opportunity to master github but learning the ins and outs of github earlier on would have been useful.

Also, I would recommend meeting frequently with your group. We met at least once a week (usually twice) to check in and code together. Since we are building off and testing each other's code, I found that coding together was more efficient because any questions that came up could be answered immediately and efficiently. Coding together is also more enjoyable because of the shared sense of excitement (and prior frustration) when a new functionality is implemented correctly.

Rupayan Basu:
By working on the compiler for fdl, I gained a better understanding of the design decisions behind some of the languages that I usually use. For example, while working on implementing fdl lists, I found I could make informed (and often valid) guesses about how lists are implemented in other languages, like python. After working on fdl, what seemed idiosyncratic in many languages earlier, makes much more sense to me now.
My advice to future teams would be to meet frequently to ensure that everybody in the team is always on the same page, and is fully aware of each others' progress, using short knowledge transfer sessions at the start of every meeting. We found this approach to be very constructive, as this helped us develop meaningful test cases and understand how every team member's contribution to the compiler fit together.

8. Individual Work Breakdown

Pranav Bhalla:
I worked on the initial skeleton of the code, making the basic scanner, parser and translator. Also worked on the run and test shell scripts. I created the library for the path functions and the path attributes. And later on worked on the typechecker rules and the symbol table.

Daniel Newman:
Implemented functionality unique to FDL, particularly Copy and Move Operators. Wrote code in Ast, Sast, fdl.ml, typecheck, to make these operators work properly. Contributed to testing. Contributed through pair programming throughout the project, coming up with multiple bug fixes. Contributed the Expressions and Operators section of the LRM.

Daniel Garzon:
Implemented the preprocessor, shell scripts, Makefiles, and back-end code in C for lists and paths. Also helped with the lexical analyzer, scanner, and the code generation/conversion to C. Performed and implemented different test scenarios to check for vulnerabilities in the preprocessor, list and path libraries with Pranav and Rup.

Cara Borenstein:
Implemented tests for multiplicative, additive, relational, equality, and assignment operators and modified typecheck, ast, sast, and fdl as needed. Implemented while loop and contributed to implementation of other control flow statements and corresponding tests through pair programming.

Rupayan Basu:
Implemented fdl lists, including all list operations, ‘if in’ constructs for lists, for loops for paths, scope for functions, variable initialization. Fixed parsing/type check issues. Customized C list implementation using union/enums for fdl types. Added test programs, modified scripts.

9. Appendix

1 O'Caml

1.1 Scanner

```
{ open Parser }

let letter = ['a' - 'z' 'A' - 'Z']
let digit = ['0' - '9']
let quote = '"'

rule token = parse
  [' ' '\r' '\n' '\t']  { token lexbuf }  | "/*"    { comment lexbuf }
  | '('     { LPAREN }    | ')'     { RPAREN }
  | '{'     { LBRACE }    | '}'     { RBRACE }    | ','     { COMMA }
  | '+'     { PLUS }    | '-'     { MINUS }
  | '*'     { TIMES }   | '/'     { DIVIDE }
  | '='     { ASSIGN }    | ';'     { SEMI }
  | "<<-"     { MOVE }
  | "<-"      { COPY }
  | "=="      { EQ }      | "!="      { NEQ }
  | '<'     { LT }      | "<="      { LEQ }
  | '>'     { GT }      | ">="      { GEQ }
  | '['     { LBRACK }    | ']'     { RBRACK }
  | "&&"      { AND }     | "||"      { OR }
  | '!'     { NOT }     | ".name"   { PATHNAME }
  | "def"     { DEF }     | ".created_at" { PATHCREATED }
  | "int"     { INT }     | ".kind"   { PATHKIND }
  | "path"    { PATH }
  | "string"    { STR }     | "list"    { LIST }
  | "if"      { IF }      | "else"    { ELSE }
  | "then"    { THEN }    | "print"   { PRINT }
  | "for"     { FOR }     | "in"      { IN }
  | "do"      { DO }      | "bool"    { BOOL }
  | "while"   { WHILE }   | "return"    { RETURN }
  | "void"    { VOID }    | ".add"    { ADD }
  | "true"    { TRUE }    | ".remove"   { REMOVE }
  | "false"   { FALSE }   | ".type"   { PATHEXT }
  | "trash"   { TRASH }
  | eof     { EOF }     (* do as microC *)
  | digit+ as lit         { LIT_INT(int_of_string lit) }
  | quote [^'"']* quote as lit  { LIT_STR(lit) }
  | letter | (letter | digit | '_')* as id    { ID(id) }
  | _ as char     { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/"      { token lexbuf }
  | _       { comment lexbuf}
```

1.2 Parser

```
%{ open Ast %}

%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA TAB SEMI
%token PLUS MINUS TIMES DIVIDE ASSIGN MOVE COPY
%token EQ NEQ LT LEQ GT GEQ NOT
%token AND OR
%token RETURN IF THEN ELSE FOR IN WHILE DO
%token DEF VOID INT STR LIST PATH BOOL TRASH TRUE FALSE PRINT
%token PATHNAME PATHCREATED PATHKIND PATHEXT ADD REMOVE
%token <int> LIT_INT
%token <string> LIT_STR
%token <bool> LIT_BOOL
%token <string> ID
%token IN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN MOVE COPY NOT

%left AND OR
%left EQ NEQ
%left LT GT LEQ GEQ
%left IN
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
    { [], [] }
    | program vdecl { ($2 :: fst $1), snd $1 }
    | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
    DEF return_type ID LPAREN formals_opt RPAREN LBRACE vdecl_opt stmt_list RBRACE
       {{
        return = $2;
        fname = $3;
        formals = $5;
        fnlocals = List.rev $8;
        body = List.rev $9 }}
```

```
return_type:
      VOID      { VoidType }
    | INT       { IntType }
    | BOOL      { BoolType }
    | PATH      { PathType }
    | STR       { StrType }
    | LIST      { ListType }

formals_opt:
    { [] }
    | formal_list   { List.rev $1 }

formal_list:
    formal                     { [$1] }
    | formal_list COMMA formal { $3 :: $1 }

formal:
    INT ID       { { vtype = IntType;  vname = $2; vexpr = Noexpr; } }
    | BOOL ID    { { vtype = BoolType; vname = $2; vexpr = Noexpr; } }
    | PATH ID    { { vtype = PathType; vname = $2; vexpr = Noexpr; } }
    | STR ID     { { vtype = StrType;  vname = $2; vexpr = Noexpr; } }
    | LIST ID    { { vtype = ListType; vname = $2; vexpr = Noexpr; } }

vdecl_opt:
    { [] }
    | vdecl_list    { List.rev $1 }

vdecl_list:
    vdecl             { [$1] }
    | vdecl_list vdecl { $2 :: $1 }

vdecl:
      vdecl_type ID SEMI    { { vtype = $1;  vname = $2; vexpr = Noexpr } }
    | vdecl_type ID ASSIGN expr SEMI  { { vtype = $1;  vname = $2; vexpr = $4 } }

vdecl_type:
    VOID          { VoidType }
    | INT         { IntType }
    | BOOL        { BoolType }
    | STR         { StrType }
    | PATH        { PathType }
    | LIST        { ListType }

stmt_list:
    { [] }
    | stmt_list stmt { $2 :: $1 }

rev_stmt_list:
    stmt_list         { List.rev $1 }

stmt:
    expr SEMI                                         { Expr($1) }
    | RETURN expr_opt SEMI                            { Return($2) }
    | IF LPAREN expr RPAREN THEN stmt %prec NOELSE    { If($3, $6, Block([])) }
    | IF LPAREN expr RPAREN THEN stmt ELSE stmt       { If($3, $6, $8) }
    | PRINT expr SEMI                                 { Print($2) }
    | WHILE LPAREN expr RPAREN stmt                   { While($3, $5) }
    | FOR LPAREN for_expr IN for_expr RPAREN stmt     { For($3, $5, $7 ) }
    | IF list_expr IN list_expr THEN stmt %prec NOELSE  { Ifin($2, $4, $6, Block([])) }
    | IF list_expr IN list_expr THEN stmt ELSE stmt   { Ifin($2, $4, $6, $8) }
    | LBRACE rev_stmt_list RBRACE                     { Block($2) }


for_expr:
    ID                              { Forid($1) }

list_expr:
    ID                             { ListId($1) }
    | LIT_INT                      { ListItemInt($1) }
    | LIT_STR                      { ListItemStr($1) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr         { $1 }
```

```
expr:
    | LIT_INT                     { LitInt($1) }
    | TRUE                        { LitInt(1) }
    | FALSE                       { LitInt(0) }
    | LIT_STR                     { LitStr($1) }
    | LBRACK list_items RBRACK    { List($2) }
    | ID                          { Id($1) }
    | expr PLUS   expr            { Binop($1, Add,      $3) }
    | expr MINUS  expr            { Binop($1, Sub,      $3) }
    | expr TIMES  expr            { Binop($1, Mult,     $3) }
    | expr DIVIDE expr            { Binop($1, Div,      $3) }
    | expr EQ     expr            { Binop($1, Equal,    $3) }
    | expr NEQ    expr            { Binop($1, Neq,      $3) }
    | expr LT     expr            { Binop($1, Less,     $3) }
    | expr LEQ    expr            { Binop($1, Leq,      $3) }
    | expr GT     expr            { Binop($1, Greater,  $3) }
    | expr GEQ    expr            { Binop($1, Geq,      $3) }
    | expr AND expr               { Binop($1, And,      $3) }
    | expr OR expr                { Binop($1, Or,       $3) }
    | ID ASSIGN expr              { Assign($1, $3) }
    | expr COPY expr                { Copy($1,   $3) }
    | expr MOVE expr                { Move($1,  $3) }
    | ID LPAREN actuals_opt RPAREN { Call($1,   $3) }
    | ID pathattributes           { Pathattr($1, $2) }
    | ID ADD LPAREN list_expr RPAREN { ListAppend($1, $4) }
    | ID REMOVE LPAREN list_expr RPAREN { ListRemove($1, $4) }

pathattributes:
    | PATHNAME                    { Pathname }
    | PATHCREATED                 { Pathcreated }
    | PATHKIND                    { Pathkind }
    | PATHEXT                     { Pathext }

list_items:
    { Noitem }
    |  expr                        { Item($1) }
    | expr COMMA list_items       { Seq($1, Comma, $3) }


actuals_opt:
    /* nothing */   { [] }
    | actuals_list  { List.rev $1 }

actuals_list:
    expr                      { [$1] }
    | actuals_list COMMA expr { $3 :: $1 }
```

1.3 AST

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Or

type sep = Comma

type data_type = PathType | StrType | IntType | BoolType | VoidType | ListType

type pathattr_type = Pathname | Pathcreated | Pathkind | Pathext

type list_expr =
    ListId of string
  | ListItemInt of int
  | ListItemStr of string

type items =
    Item of expr
  | Seq of expr * sep * items
  | Noitem
and expr =
    LitInt of int
  | LitStr of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Copy of expr * expr
  | Move of expr * expr
  | List of items
  | ListAppend of string * list_expr
  | ListRemove of string * list_expr
  | Pathattr of string * pathattr_type
  | Noexpr


type for_expr =
    Forid of string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of for_expr * for_expr * stmt
  (* | For of expr * expr * stmt*)
  | While of expr * stmt
  | Print of expr
  | Ifin of list_expr * list_expr * stmt * stmt

type var_decl = {
  vtype : data_type;
  vname : string;
  vexpr : expr;
}

type func_decl = {
    return : data_type;
    fname : string;
    formals : var_decl list;
    fnlocals : var_decl list;
    body : stmt list;
  }


type program = var_decl list * func_decl list
```

1.4 SAST

```
type op_t = Add | Sub | Mult | Div | Equal | Neq | Less | Leq
          | Greater | Geq | And | Or | StrEqual | StrNeq | StrAdd

type sep_t = Comma

type data_type_t = PathType | StrType | IntType | BoolType | VoidType | ListType

type pathattr_type_t = Pathname | Pathcreated | Pathkind | Pathext

type list_expr_t =
    ListId of string * string
  | ListItemInt of int
  | ListItemStr of string

type items_t =
    Item of expr_t
  | Seq of expr_t * sep_t * items_t
  | Noitem
and expr_t =
    LitInt of int
  | LitStr of string
  | Id of string
  | Binop of expr_t * op_t * expr_t
  | Assign of string * expr_t
  | Call of string * expr_t list
  | Copy of expr_t * expr_t
  | Move of expr_t * expr_t
  | List of items_t
  | ListAppend of string * list_expr_t
  | ListRemove of string * list_expr_t
  | Pathattr of string * pathattr_type_t
  | Noexpr

type for_expr_t =
    Forid of string

type stmt_t =
    Block of stmt_t list
  | Expr of expr_t
  | Return of expr_t
  | If of expr_t * stmt_t * stmt_t
  (*| For of expr_t * expr_t * stmt_t *)
  | For of for_expr_t * for_expr_t * stmt_t
  | While of expr_t * stmt_t
  | Print of expr_t * string
  | Ifin of list_expr_t * list_expr_t * stmt_t * stmt_t

type var_decl_t = {
  vtype : data_type_t;
  vname : string;
  vexpr : expr_t;
}

type func_decl_t = {
    return : data_type_t;
    fname : string;
    formals : var_decl_t list;
    fnlocals : var_decl_t list;
    body : stmt_t list;
  }

type program_t = var_decl_t list * func_decl_t list
```

1.5 Type Check

```
open Ast
open Symboltable

module StringMap = Map.Make(String)

let string_of_vtype = function
  VoidType -> "void"
  | IntType -> "int"
  | StrType -> "string"
  | BoolType -> "bool"
  | PathType -> "path"
  | ListType -> "list"

let get_sast_type = function
  Ast.PathType -> Sast.PathType
  | Ast.StrType -> Sast.StrType
  | Ast.IntType -> Sast.IntType
  | Ast.BoolType -> Sast.BoolType
  | Ast.VoidType -> Sast.VoidType
  | Ast.ListType -> Sast.ListType

let get_sast_pathattrtype = function
  Ast.Pathname -> Sast.Pathname, "string"
  | Ast.Pathcreated -> Sast.Pathcreated, "int"
  | Ast.Pathkind -> Sast.Pathkind, "int"
  | Ast.Pathext -> Sast.Pathext, "string"

let get_vtype env id =
  (* find_variable method is from the symbol table *)
  let t = find_variable id env in
  if t = "" then raise (Failure ("undefined variable: " ^ id)) else t

let get_expr_type t1 t2 =
  if t1 = "void" || t2 = "void" then raise (Failure ("cannot use void type inside expression")) else
  if t1 = "string" || t2 = "string" then "string" else
  if t1 = "int" && t2 = "int" then "int" else
  if t1 = "bool" && t2 = "bool" then "bool" else
  if t1 = "int" && t2 = "bool" then "int" else
  if t1 = "bool" && t2 = "int" then "int" else
  raise (Failure ("type error"))
```

```
let check_listexpr env = function
  | Ast.ListId(id) ->
    Sast.ListId(id, get_vtype env id), get_vtype env id
  | Ast.ListItemInt(i) -> Sast.ListItemInt(i), "int"
  | Ast.ListItemStr(s) -> Sast.ListItemStr(s), "string"


let match_oper e1 op e2 =
  let expr_t = get_expr_type (snd e1) (snd e2) in
  (match op with
     Ast.Add -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Add, fst e2), "int") else
          if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrAdd, fst e2), "string") else
          raise (Failure ("type error"))
   | Ast.Sub -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Sub, fst e2), "int") else
      raise (Failure ("type error"))
   | Ast.Mult -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Mult, fst e2), "int") else
       raise (Failure ("type error"))
   | Ast.Div -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Div, fst e2), "int") else
      raise (Failure ("type error"))
      (* equal and not equal have special case for string comparison
          we may need to add SAST and Eqs and Neqs *)
   | Ast.Equal -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Equal, fst e2), "bool") else
          if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrEqual, fst e2), "bool") else
                  raise (Failure ("type error in == "))
   | Ast.Neq -> if expr_t = "int" then (Sast.Binop(fst e1, Sast.Neq, fst e2), "bool") else
          if expr_t = "string" then (Sast.Binop(fst e1, Sast.StrNeq, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Less ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Less, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Leq ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Leq, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Greater ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Greater, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.Geq ->if expr_t = "int" then (Sast.Binop(fst e1, Sast.Geq, fst e2), "bool") else
                  raise (Failure ("type error"))
   | Ast.And ->if expr_t = "bool" then (Sast.Binop(fst e1, Sast.And, fst e2), "bool") else
            raise (Failure ("type error in and"))
   | Ast.Or ->if expr_t = "bool" then (Sast.Binop(fst e1, Sast.Or, fst e2), "bool") else
            raise (Failure ("type error in or"))
  )
```

```
let rec check_expr env = function
  Ast.LitInt(i) -> Sast.LitInt(i), "int"
  | Ast.LitStr(s) -> Sast.LitStr(s), "string"

  | Ast.Id(id) ->
    Sast.Id(id), (get_vtype env id)

  | Ast.Binop(e1, op, e2) ->
    match_oper (check_expr env e1) op (check_expr env e2)

  | Ast.Assign(id, e) ->
    let t = get_vtype env id in
    Sast.Assign(id, (get_expr_with_type env e t)), "void"

  | Ast.Call(func, el) ->
    let args = find_function func env in  (* return & arguments type list from definition *)
    ( match args with
      [] -> raise (Failure ("undefined function " ^ func))
      | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map (check_expr env) el) tl
               with Invalid_argument "arg" -> raise(Failure("unmatched argument list"))
            in Sast.Call(func, List.rev new_list ), hd )

    | Ast.Move(e1, e2) ->
    let e_t1 = check_expr env e1 in
    let e_t2 = check_expr env e2 in
    if snd e_t1 = "path" && snd e_t2 = "path"
      then Sast.Move(fst e_t1, fst e_t2), "void"
    else
      raise(Failure("cannot use path function on non-path variables"))
    | Ast.Copy(e1, e2) ->
    let e_t1 = check_expr env e1 in
                let e_t2 = check_expr env e2 in
                if snd e_t1 = "path" && snd e_t2 = "path"
                        then Sast.Copy(fst e_t1, fst e_t2), "void"
                else
                        raise(Failure("cannot use path function on non-path variables"))
  | Ast.List(items) -> Sast.List(check_list_items env items), "list"
  | Ast.ListAppend(id, item) -> let t1 = get_vtype env id in
                  let t2 = check_listexpr env item in
                if not (t1 = "list")
                  then raise(Failure("Can append only to id of type list."))
                else if ((snd t2) = "list")
                  then raise(Failure("Cannot append list to list."))
                else
                  Sast.ListAppend( id, (fst t2)), "void"
    | Ast.ListRemove(id, item) -> let t1 = get_vtype env id in
                  let t2 = check_listexpr env item in
                  if not (t1 = "list")
                    then raise(Failure("Can call remove only on type list."))
                  else if ((snd t2) = "list")
                    then raise(Failure("Cannot remove a list from list."))
                  else
                    Sast.ListRemove(id, (fst t2)), "void"
    | Ast.Pathattr(id, e) ->
      if not ((get_vtype env id) = "path")
        then raise(Failure("cannot use path attributes on non-path variable " ^ id))
      else
        Sast.Pathattr(id, fst (get_sast_pathattrtype e)), snd (get_sast_pathattrtype e)
    | Ast.Noexpr -> Sast.Noexpr, "void"

  and check_list_items env = function
      Ast.Item(e) ->let i,t = check_expr env e in
              Sast.Item(i)
    | Ast.Seq(e1, sep, e2) -> Sast.Seq(fst (check_expr env e1), Sast.Comma, (check_list_items env e2))
    | Ast.Noitem -> Sast.Noitem

  and get_expr_with_type env expr t =
    let e = check_expr env expr in
    if ((snd e) = "string" && t = "path") then (fst e)
    else if ((snd e) = "int" && t = "bool") then (fst e)
    else if not((snd e) = t) then raise (Failure ("type error")) else (fst e)

  let check_forexpr env = function
    Ast.Forid(id) -> Sast.Forid(id), get_vtype env id
```

```
let rec check_stmt env func = function
    Ast.Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
  | Ast.Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
  | Ast.Return(expr) -> let e = check_expr env expr in
            if not(snd e = string_of_vtype func.return) then raise (Failure ("The return type doesn't match!"))
            else (Sast.Return(fst e)), env
  | Ast.If(expr, stmt1, stmt2) -> let e = check_expr env expr in
                if not(snd e = "bool") then raise (Failure ("The type of the condition in If statement must be boolean!"))
                else (Sast.If(fst e, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
  | Ast.Ifin(lexpr1, lexpr2, stmt1, stmt2) -> let e1 = check_listexpr env lexpr1 in
              if (snd e1 = "list") then raise (Failure ("Cannot have list in list!"))
              else let e2 = check_listexpr env lexpr2 in
              if not(snd e2 = "list") then raise (Failure ("\'in\' operator works with list type expression only!"))
            else (Sast.Ifin(fst e1, fst e2, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
  | Ast.While(expr, stmt) -> let e = check_expr env expr in
               if not (snd e = "bool") then raise (Failure ("The type of the condition in While statement must be boolean!"))
               else (Sast.While(fst e, fst (check_stmt env func stmt))), env        (* while() {} *)
   | Ast.For(expr1, expr2, stmt) -> let e1 = check_forexpr env expr1 in let e2 = check_forexpr env expr2 in
               if not (snd e1 = "path" && snd e2 = "path" ) then raise
                                      (Failure("The type of the expression in a For statement must be path"))
               else (Sast.For(fst e1, fst e2, fst (check_stmt env func stmt))), env
  | Ast.Print(expr) -> let (expr, expr_type) = check_expr env expr in
              (Sast.Print(expr , expr_type)), env

let rec check_expr env = function
   Ast.LitInt(i) -> Sast.LitInt(i), "int"
  | Ast.LitStr(s) -> Sast.LitStr(s), "string"

  | Ast.Id(id) ->
    Sast.Id(id), (get_vtype env id)

  | Ast.Binop(e1, op, e2) ->
    match_oper (check_expr env e1) op (check_expr env e2)

  | Ast.Assign(id, e) ->
    let t = get_vtype env id in
    Sast.Assign(id, (get_expr_with_type env e t)), "void"

  | Ast.Call(func, el) ->
    let args = find_function func env in  (* return & arguments type list from definition *)
    ( match args with
      [] -> raise (Failure ("undefined function " ^ func))
      | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map (check_expr env) el) tl
               with Invalid_argument "arg" -> raise(Failure("unmatched argument list"))
            in Sast.Call(func, List.rev new_list ), hd )
    (* Need to add type checking for Move and Copy *)
      | Ast.Move(e1, e2) ->
    let e_t1 = check_expr env e1 in
    let e_t2 = check_expr env e2 in
    if snd e_t1 = "path" && snd e_t2 = "path"
      then Sast.Move(fst e_t1, fst e_t2), "void"
    else
      raise(Failure("cannot use path function on non-path variables"))
      | Ast.Copy(e1, e2) ->
    let e_t1 = check_expr env e1 in
                let e_t2 = check_expr env e2 in
                if snd e_t1 = "path" && snd e_t2 = "path"
                        then Sast.Copy(fst e_t1, fst e_t2), "void"
                else
                        raise(Failure("cannot use path function on non-path variables"))
```

```
  | Ast.List(items) -> Sast.List(check_list_items env items), "list"
  | Ast.ListAppend(id, item) -> let t1 = get_vtype env id in
                  let t2 = check_listexpr env item in
                if not (t1 = "list")
                  then raise(Failure("Can append only to id of type list."))
                else if ((snd t2) = "list")
                  then raise(Failure("Cannot append list to list."))
                else
                  Sast.ListAppend( id, (fst t2)), "void"
  | Ast.ListRemove(id, item) -> let t1 = get_vtype env id in
                  let t2 = check_listexpr env item in
                if not (t1 = "list")
                  then raise(Failure("Can call remove only on type list."))
                else if ((snd t2) = "list")
                  then raise(Failure("Cannot remove a list from list."))
                else
                  Sast.ListRemove(id, (fst t2)), "void"
  | Ast.Pathattr(id, e) ->
    if not ((get_vtype env id) = "path")
      then raise(Failure("cannot use path attributes on non-path variable " ^ id))
    else
    (* return type is string assuming path attributes will be treated that way *)
      Sast.Pathattr(id, fst (get_sast_pathattrtype e)), snd (get_sast_pathattrtype e)
  | Ast.Noexpr -> Sast.Noexpr, "void"

and check_list_items env = function
    Ast.Item(e) ->let i,t = check_expr env e in
            Sast.Item(i)
  | Ast.Seq(e1, sep, e2) -> Sast.Seq(fst (check_expr env e1), Sast.Comma, (check_list_items env e2))
  | Ast.Noitem -> Sast.Noitem

and get_expr_with_type env expr t =
  let e = check_expr env expr in
  (* added special case for the path variable *)
  if ((snd e) = "string" && t = "path") then (fst e)
  else if ((snd e) = "int" && t = "bool") then (fst e)
  else if not((snd e) = t) then raise (Failure ("type error")) else (fst e)


let check_forexpr env = function
  Ast.Forid(id) -> Sast.Forid(id), get_vtype env id
```

```
let rec check_stmt env func = function
    Ast.Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
  | Ast.Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
  | Ast.Return(expr) -> let e = check_expr env expr in
            if not(snd e = string_of_vtype func.return) then raise (Failure ("The return type doesn't match!"))
            else (Sast.Return(fst e)), env
  | Ast.If(expr, stmt1, stmt2) -> let e = check_expr env expr in
                if not(snd e = "bool") then raise (Failure ("The type of the condition in If statement must be boolean!"))
                else (Sast.If(fst e, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env  (* if() {} else{} *)
  | Ast.Ifin(lexpr1, lexpr2, stmt1, stmt2) -> let e1 = check_listexpr env lexpr1 in
                if (snd e1 = "list") then raise (Failure ("Cannot have list in list!"))
              else let e2 = check_listexpr env lexpr2 in
              if not(snd e2 = "list") then raise (Failure ("\'in\' operator works with list type expression only!"))
            else (Sast.Ifin(fst e1, fst e2, fst (check_stmt env func stmt1), fst (check_stmt env func stmt2))), env
  | Ast.While(expr, stmt) -> let e = check_expr env expr in
               if not (snd e = "bool") then raise (Failure ("The type of the condition in While statement must be boolean!"))
               else (Sast.While(fst e, fst (check_stmt env func stmt))), env       (* while() {} *)
   | Ast.For(expr1, expr2, stmt) -> let e1 = check_forexpr env expr1 in let e2 = check_forexpr env expr2 in
               if not (snd e1 = "path" && snd e2 = "path" ) then raise (Failure("The type of the expression in a For statement must be path")
               else (Sast.For(fst e1, fst e2, fst (check_stmt env func stmt))), env
  | Ast.Print(expr) -> let (expr, expr_type) = check_expr env expr in
              (Sast.Print(expr , expr_type)), env

and check_stmt_list env func = function
    [] -> []
  | hd::tl -> let s,e = (check_stmt env func hd) in s::(check_stmt_list e func tl)

let convert_to_sast_type x env =
  let t = get_vtype env x.vname in
    let s_expr =
    if not (x.vexpr = Ast.Noexpr) then
      get_expr_with_type env x.vexpr t
    else Sast.Noexpr
    in
  {
    Sast.vtype = get_sast_type x.vtype;
    Sast.vname = x.vname;
    Sast.vexpr = s_expr;
  }

let check_formal env formal =
  let ret = add_local formal.vname formal.vtype env in
  if (string_of_vtype formal.vtype) = "void" then raise (Failure("cannot use void as variable type")) else
  if StringMap.is_empty ret then raise (Failure ("local variable " ^ formal.vname ^ " is already defined"))
  else let env = {locals = ret; globals = env.globals; functions = env.functions } in
  convert_to_sast_type formal env, env

let rec check_formals env formals =
  match formals with
    [] -> []
  | hd::tl -> let f, e = (check_formal env hd) in (f, e)::(check_formals e tl)

let check_local env local =
  let ret = add_local local.vname local.vtype env in
  if (string_of_vtype local.vtype) = "void" then raise (Failure("cannot use void as variable type")) else
  if StringMap.is_empty ret then raise (Failure ("local variable " ^ local.vname ^ " is already defined"))
  else let env = {locals = ret; globals = env.globals; functions = env.functions } in
  convert_to_sast_type local env, env
```

```
let rec check_locals env locals =
  match locals with
    [] -> []
  | hd::tl -> let l, e = (check_local env hd) in (l, e)::(check_locals e tl)

let check_function env func =
  match List.hd (List.rev func.body) with
  Return(_) ->
      let env = {locals = StringMap.empty; globals = env.globals; functions = env.functions } in
      (*  ret is new env *)
    let ret = add_function func.fname func.return func.formals env in
    if StringMap.is_empty ret then raise (Failure ("function " ^ func.fname ^ " is already defined"))
    else let env = {locals = env.locals; globals = env.globals; functions = ret } in
    let f = check_formals env func.formals in
    let formals = List.map (fun formal -> fst formal) f in

    (* get the final env from the last formal *)
    let l, env =
    (match f with
        [] -> let l = check_locals env func.fnlocals in
           l, env
      | _ ->  let env = snd (List.hd (List.rev f)) in
          let l = check_locals env func.fnlocals in
          l, env
    ) in
    let fnlocals = List.map (fun fnlocal -> fst fnlocal) l in
     (match l with
             [] -> let body = check_stmt_list env func func.body in
                 { Sast.return = get_sast_type func.return;
                   Sast.fname = func.fname;
                   Sast.formals = formals;
                   Sast.fnlocals = fnlocals;
                   Sast.body = body
                 }, env
             | _ -> let e = snd (List.hd (List.rev l)) in
                    let body = check_stmt_list e func func.body in
                   { Sast.return = get_sast_type func.return;
                     Sast.fname = func.fname;
                     Sast.formals = formals;
                     Sast.fnlocals = fnlocals;
                     Sast.body = body
                   }, e
          )
  | _ -> raise (Failure ("The last statement must be return statement"))

let rec check_functions env funcs =
  match funcs with
    [] -> []
  | hd::tl -> let f, e = (check_function env hd) in f::(check_functions e tl)

let check_global env global =
  if (string_of_vtype global.vtype) = "void" then raise (Failure("cannot use void as variable type"))
  else let ret = add_global global.vname global.vtype env in
  if StringMap.is_empty ret then raise (Failure ("global variable " ^ global.vname ^ " is already defined"))
  (* update the env with globals from ret *)
  else let env = {locals = env.locals; globals = ret; functions = env.functions } in
  convert_to_sast_type global env, env

let rec check_globals env globals =
  match globals with
    [] -> []
  | hd::tl -> let g, e = (check_global env hd) in (g, e)::(check_globals e tl)

let check_program (globals, funcs) =
  let env = { locals = StringMap.empty;
        globals = StringMap.empty;
        functions = StringMap.empty }
  in
  let g = check_globals env globals in
  let globals = List.map (fun global -> fst global) g in
  match g with
   [] -> (globals, (check_functions env (List.rev funcs)))
  | _ -> let e = snd (List.hd (List.rev g)) in (globals, (check_functions e (List.rev funcs)))
```

1.6 Symbol Table

```
open Ast

module StringMap = Map.Make(String)

type env = {
    locals:         string StringMap.t;
  globals:        string StringMap.t;
  functions:      string list StringMap.t;
}

let string_of_vtype = function
  VoidType -> "void"
  | IntType -> "int"
  | StrType -> "string"
  | BoolType -> "bool"
  | PathType -> "path"
  | ListType -> "list"

let find_variable name env =
  try StringMap.find name env.locals
  with Not_found -> try StringMap.find name env.globals
  with Not_found -> ""
  (*raise (Failure ("undefined variable " ^ name)) *)

let find_function name env =
  try StringMap.find name env.functions
  with Not_found -> []
  (*raise (Failure ("undefined function " ^ name)) *)

let add_local name v_type env =
  if StringMap.mem name env.locals then StringMap.empty
  else StringMap.add name (string_of_vtype v_type) env.locals

let add_global name v_type env =
  if StringMap.mem name env.globals then StringMap.empty
  else StringMap.add name (string_of_vtype v_type) env.globals

(* from the ast *)
let get_arg_type = function
  v -> string_of_vtype v.vtype

let add_function name return_type formals env =
  if StringMap.mem name env.functions then StringMap.empty
  else let f = List.map get_arg_type formals in
  StringMap.add name (string_of_vtype (return_type)::f) env.functions
```

2 Shell Scripts

2.1 Test All Script

```
#!/bin/sh

if [ ! -f ”c/libraries/liblist.a” ] || [ ! -f ”c/libraries/libpath.a” ] ; then
    cd c/libraries
    make >> lib_msgs.txt
    cd ../..
fi

if [ ! -f ”preprocessor/./preprocessor” ]; then
    cd preprocessor
    make >> preproc_msgs.txt
    cd ..
fi

if [ ! -f ”./fdl” ]; then
    make >> compiler_msgs.txt
fi

FDL=”./fdl”
PRE=”preprocessor/./preprocessor”

Compare() {
  difference=$(diff -b $1 $2)
  echo $difference
  if [ ”$difference” != ”” ]; then
    echo $difference > $3
  fi
}

function compile() {
  basename=‘echo $1 | sed ’s/.*\\///
                             s/.fdl//’‘
  reffile=‘echo $1 | sed ’s/.fdl$//’‘
    prepfile=${reffile}’.fdlp’
    basedir=”‘echo $1 | sed ’s/\/[ˆ\/]*$//’‘/”

  testoutput=‘echo ${basedir}test_outputs/$basename.c.out‘

    echo ”Preprocessing ’$1’...”
    $PRE $1 $prepfile && echo ”Preprocessor for $1 succeeded”

  echo ”Compiling $prepfile ...”
    $FDL $prepfile > ”${reffile}.c” && echo ”Ocaml to C of $1 succeeded”

    if [ -f ”${reffile}.c” ]; then
      gcc -Ic/libraries -Lc/libraries -llist -lpath ”${reffile}.c” -o ”${reffile}” && echo ”COMPILATION of ${reffile}.c succeeded”
    else
      echo ”Ocaml to C of $1 failed”
      return
    fi

    rm -rf $prepfile

    if [ -f ”${reffile}” ]; then
      eval ${reffile} > ${reffile}.generated.out
      Compare ${testoutput} ${reffile}.generated.out ${reffile}.c.diff
        rm -rf ${reffile}.generated.out
        rm -rf ${reffile}.c
        rm -rf ${reffile}
    else
      echo ”C to binary of ${reffile}.c failed”
    fi
}

files=sample_program/*.fdl

for file in $files
do
  compile $file
done
```

2.2 Run FDL Script

```
#!/bin/sh

if [ ! -f "c/libraries/liblist.a" ] || [ ! -f "c/libraries/libpath.a" ] ; then
    cd c/libraries
    make >> lib_msgs.txt
    cd ../..
fi

if [ ! -f "preprocessor/./preprocessor" ]; then
    cd preprocessor
    make >> preproc_msgs.txt
    cd ..
fi

if [ ! -f "./fdl" ]; then
    make >> compiler_msgs.txt
fi

# fdl exectutable
FDL="./fdl"
# preprocessor executable
PRE="./preprocessor/preprocessor"

function compileAndRun() {
  basename=`echo $1 | sed 's/.*\\///
                             s/.fdl//'`
  reffile=`echo $1 | sed 's/.fdl$//'`
    prepfile=$reffile'.fdlp'
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/"


  $PRE $1 $prepfile

  if [ ! -f $prepfile ]; then
    echo "$prepfile does not exist"
        return
  fi

    $FDL $prepfile > "${reffile}.c"

    if [ -f "${reffile}.c" ]; then
      gcc -Ic/libraries -Lc/libraries -llist -lpath -w -o "${reffile}" "${reffile}.c"
    else
      echo "Ocaml to C of $1 failed"
      return
    fi

    if [ -f "${reffile}" ]; then
        eval ${reffile}
        rm -rf ${reffile}.fdlp
        rm -rf ${reffile}.c
        rm -rf ${reffile}
    else
        echo "C to binary of ${reffile}.c failed"
    fi
}

if [ -f $1 ]; then
  compileAndRun $1
else
  echo "$1 doesnt exist"
fi
```

2.3 Clean All Script

```
#!/bin/sh

if [ -f "c/libraries/liblist.a" ] || [ -f "c/libraries/libpath.a" ] ; then
    cd c/libraries
    make clean
    cd ../..
fi

if [ -f "preprocessor/./preprocessor" ]; then
    cd preprocessor
    make clean
    cd ..
fi

if [ -f "./fdl" ]; then
    make clean
fi
```

3 Preprocessor

3.1 Makefile

```
CC  = gcc
CXX = g++

INCLUDES =
CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

LDFLAGS =
LDLIBS =

.PHONY: default
default: preprocessor

# header dependency
preprocessor: preprocessor.o

.PHONY: clean
clean:
	rm -f *.o *.txt *~ a.out core preprocessor

.PHONY: all
all: clean default
```

3.2 Preprocessor

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
#include <ctype.h>  /* For isspace(). */
#include <stddef.h>  /* For size_t. */

#define MAX_BUFFER 4096

static void die(const char *message)
{
    perror(message);
    exit(1);
}

const char *getFileExtension(const char *fileName) {
    const char *dot = strrchr(fileName, '.');
    if(!dot || dot == fileName) return "";
    return dot + 1;
}

void remove_whitespace(char *str) {
    char *p;
    size_t len = strlen(str);

    for(p = str; *p; p ++, len --) {
        while(isspace(*p)) memmove(p, p+1, len--);
    }
}

int is_empty(const char *s) {
  while (*s != '\0') {
    if (!isspace(*s))
      return 0;
    s++;
  }
  return 1;
}


int main(int argc, char const *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "%s\n", "usage: ./preprocessor <fdl file> <fdlp file>");
        exit(1);
    }
    char *fileName = (char *) argv[1];
    char *outputFileName = (char *) argv[2];

    if (strcmp("fdl", getFileExtension(fileName)) != 0)
    {
        die("file extension must be fdl");
    }
    if (strcmp("fdlp", getFileExtension(outputFileName)) != 0)
    {
        die("output file extension must be fdlp");
    }
    FILE *input;
    if ((input = fopen(fileName, "r")) == NULL) {
        die("fpen() failed");
    }
    FILE *output;
    if ((output = fopen(outputFileName, "w")) == NULL) {
        die("fpen() failed");
    }

    char buffer[MAX_BUFFER];
```

```
    while (fgets(buffer, sizeof(buffer), input) != NULL) {
        size_t len = strlen(buffer) - 1;
        if (buffer[len] == '\n') {
            buffer[len] = '\0';
        }
        if (strstr(buffer, "*/") != NULL) {
            fprintf(output, "%s\n", buffer);
        }
        else if (strstr(buffer, "/*") != NULL) {
            fprintf(output, "%s\n", buffer);
        }
        else if (strstr(buffer, "def ") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
        else if (strstr(buffer, "int ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "path ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "dict ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "list ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "string ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "bool ") != NULL) {
            fprintf(output, "%s;\n", buffer);
        }
        else if (strstr(buffer, "for ") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
        else if ((strstr(buffer, "if (") != NULL || strstr(buffer, "if(") != NULL)
                                              && (strstr(buffer, "then") != NULL)) {
            fprintf(output, "%s {\n", buffer);
        }
        else if ((strstr(buffer, "if (") != NULL || strstr(buffer, "if(") != NULL)
                                              && (strstr(buffer, "then") == NULL)) {
            fprintf(output, "%s\n", buffer);
        }
        else if (strstr(buffer, "then") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
```

```
 else if (strstr(buffer, "else") != NULL) {
            int i;
            int counter = 0;
            for (i = 0; i < strlen(buffer); ++i)
            {
                if (buffer[i] == ' ') {
                    fprintf(output, "%c", buffer[i]);
                    counter++;
                }
            }
            fprintf(output, "} %s {\n", buffer + counter);
        }
        else if (strstr(buffer, "while (") != NULL || strstr(buffer, "while(") != NULL) {
            fprintf(output, "%s {\n", buffer);
        }
        else if (strstr(buffer, "end") != NULL) {
            int i;
            for (i = 0; i < strlen(buffer); i++){
                if (buffer[i] == 'e') {
                    buffer[i] = '}';
                } else if (buffer[i] == 'n') {
                    buffer[i] = '\n';
                } else if (buffer[i] == 'd') {
                    buffer[i] = '\0';
                } else {

                }
            }
            fprintf(output, "%s", buffer);
        }
        else {
            if (is_empty(buffer)) {
                remove_whitespace(buffer);
                fprintf(output, "\n");
            } else {
                fprintf(output, "%s;\n", buffer);
            }
        }
    }
    fclose(input);
    fclose(output);
    return 0;
}
```

4 Libraries in C

4.1 Makefile

```
CC  = gcc
CXX = g++

INCLUDES = -I libraries/

CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

LDFLAGS = -g -L libraries/

LDLIBS = -llist -lpath

stat_calls: stat_calls.o

stat_calls.o: stat_calls.c

.PHONY: clean
clean:
	rm -f *.o *.txt a.out core stat_calls

.PHONY: all
all: clean stat_calls
```

4.2 Lists

4.2.1 List Header

```c
#ifndef _LIST_H_
#define _LIST_H_

enum fdl_type { fdl_str, fdl_path, fdl_int, fdl_bool };

struct Node {
  enum fdl_type type;
    union {
        int int_item;
        int bool_item;
        char *string_item;
        char *path_item;
    };
    struct Node *next;
};

struct List {
    struct Node *head;
};

struct Node *createIntNode(int data, enum fdl_type type);
struct Node *createStrNode(char *data, enum fdl_type type);

static inline void initList(struct List *list)
{
    list->head = 0;
}

static inline int isEmptyList(struct List *list)
{
    return (list->head == 0);
}

void addFront(struct List *list, struct Node *node);
void traverseList(struct List *list, void (*f)(struct Node *));
void printNode(struct Node *node);
int findNode(struct List *list, struct Node *node1);
void removeNode(struct List *list, struct Node *node1);
struct Node popFront(struct List *list);
void removeAllNodes(struct List *list);
void addAfter(struct List *list, struct Node *prevNode, struct Node *newNode);
void reverseList(struct List *list);
void addBack(struct List *list, struct Node *newNode);
void loadDirectoryToList(char *path, struct List *subPath);

#endif
```

4.2.2 List Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"
#include "dirent.h"

void loadDirectoryToList(char *path, struct List *subPath){
    char *buffer;
    DIR *dir;
    struct dirent *ent;
    int len;
    if ((dir = opendir (path)) != NULL) {
        /* print all the files and directories within directory */
        while ((ent = readdir (dir)) != NULL) {
            len = strlen(path) + strlen(ent->d_name) + 2;
            buffer = (char *)malloc(sizeof(char)*len);
            //printf("%s\n",ent->d_name);
            strcpy(buffer, path);
            strcat(buffer, "/");
            strcat(buffer, ent->d_name);
            struct Node * node = createStrNode(buffer, fdl_path);
            addBack(subPath, node);
            //buffer = "\0";
        }
        closedir (dir);
    } else {
        /* could not open directory */
        perror ("");
        exit(0);
    }
}

struct Node *createIntNode(int data, enum fdl_type type) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if(newNode == NULL){
        printf("Could not create new node!\n");
        exit(1);
    }

    newNode->type = type;
    newNode->next = NULL;
    switch(newNode->type){
        case fdl_int: newNode->int_item = data; break;
        case fdl_bool: newNode->bool_item = data; break;
        default: break;
    }
    return newNode;
}

struct Node *createStrNode(char *data, enum fdl_type type) {
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    if(newNode == NULL){
        printf("Could not create new node!\n");
        exit(1);
    }

    newNode->type = type;
    newNode->next = NULL;
    switch(newNode->type){
        case fdl_str: newNode->string_item = data; break;
        case fdl_path: newNode->path_item = data; break;
        default: break;
    }
    return newNode;
}

void addFront(struct List *list, struct Node *node)
{
    node->next = list->head;
    list->head = node;
}
```

```
void traverseList(struct List *list, void (*f)(struct Node *))
{
    struct Node *node = list->head;
    while (node) {
    f(node);
    node = node->next;
    }
}

void printNode(struct Node *node)
{
    switch(node->type){
        case fdl_int: printf("%d\n",node->int_item); break;
        case fdl_bool: if(node->bool_item == 1) printf("True\n");
                        else printf("False\n"); break;
        case fdl_str: printf("%s\n",node->string_item); break;
        case fdl_path: printf("%s\n",node->path_item); break;
    }
}

int findNode(struct List *list, struct Node *node1) {
    struct Node *node2 = list->head;
    while (node2) {
        if(node1->type == node2->type){
            switch(node1->type){
                case fdl_int: if (node1->int_item == node2->int_item) return 0; else break;
                case fdl_str: if (strcmp(node1->string_item, node2->string_item) == 0) return 0; else break;
                case fdl_bool: if (node1->bool_item == node2->bool_item) return 0; else break;
                case fdl_path: if (strcmp(node1->path_item, node2->path_item) == 0) return 0; else break;
                default: return 1;
            }
        }
        node2 = node2->next;
    }
    return 1;
}

void removeNode(struct List *list, struct Node *node1) {
    struct Node *node2 = list->head;
    int del = 0;
    struct Node *prev = list->head;
    while (node2) {
        if(node1->type == node2->type){
            switch(node1->type){
                case fdl_int: if (node1->int_item == node2->int_item) { del = 1; break; } else break;
                case fdl_str: if (strcmp(node1->string_item, node2->string_item) == 0) { del = 1; break; } else break;
                case fdl_bool: if (node1->bool_item == node2->bool_item) { del = 1; break; } else break;
                case fdl_path: if (strcmp(node1->path_item, node2->path_item) == 0) { del = 1; break; } else break;
                default: del = 0;
            }
        }
        if(del == 0){
            prev = node2;
            node2 = node2->next;
        }
        else break;
    }
    if(del == 1){
        if(node2 == list->head)
            list->head = node2->next;
        else
            prev->next = node2->next;
        free(node2);
    }
    else {
        printf("List item not found.\n");
    }
}

struct Node popFront(struct List *list) {
    struct Node *oldHead = list->head;
    struct Node node = *oldHead;
    list->head = oldHead->next;
    free(oldHead);
    return node;
}
```

```
void removeAllNodes(struct List *list)
{
    while (!isEmptyList(list))
    popFront(list);
}

void addAfter(struct List *list,
    struct Node *prevNode, struct Node *newNode)
{
    if (prevNode == NULL)
       addFront(list, newNode);

    newNode->next = prevNode->next;
    prevNode->next = newNode;
}

void reverseList(struct List *list)
{
    struct Node *prv = NULL;
    struct Node *cur = list->head;
    struct Node *nxt;

    while (cur) {
    nxt = cur->next;
    cur->next = prv;
    prv = cur;
    cur = nxt;
    }

    list->head = prv;
}

void addBack(struct List *list, struct Node *newNode)
{
    newNode->next = NULL;

    if (list->head == NULL) {
       list->head = newNode;
       return;
    }

    struct Node *end = list->head;
    while (end->next != NULL)
       end = end->next;

    end->next = newNode;
}
```

4.3 Paths

4.3.1 Path Header

```
#ifndef _PATH_H_
#define _PATH_H_

char* getName(char *path, char *output);
int checkValid(char *path);
int getCreatedAt(char *path);
int getPathType(char *path);
int isDir(char *path);
char* getPathName(char *path);
int copyFile(char* src, char *dest);
int moveFile(char* src, char *dest);
char* getExtension(char *path);
char* stringConcat(char *str1, char *str2);

#endif
```

4.3.2 Path Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sys/stat.h"
#include "time.h"
#include<libgen.h>

// test function
char* getName(char *path, char *output){
  char *dirc, *basec, *bname, *dname;

  dirc = strdup(path);
  basec = strdup(path);
  dname = dirname(dirc);
  bname = basename(basec);
  //printf("dirname=%s, basename=%s\n", dname, bname);
  strcpy(output, dname);
  return output;
}

int checkValid(char *path){
  /* testing the stat sys call for files and directories */
  struct stat info;
  if (stat(path, &info) != 0)
    return 0;
  else
    // can be valid directory or file
    return S_ISDIR(info.st_mode) ? 1 : S_ISREG(info.st_mode);
}

// returns -1 in case of invalid path
int getCreatedAt(char *path){
  if(checkValid(path)){
    struct stat info;
    stat(path, &info);

    return (int) info.st_birthtime;
  }else
    return -1;
}

// Directory 1, File 0, invalid path -1
int getPathType(char *path){
  if(checkValid(path)){
    struct stat info;
    stat(path, &info);

    return S_ISDIR(info.st_mode);
  }else
    return -1;
}

int isDir(char *path){
  if(checkValid(path)){
    struct stat info;
    stat(path, &info);

    return S_ISDIR(info.st_mode);
  }else
    return -1;
}

// get the last directory or filename
char* getPathName(char* path){
  if(checkValid(path)){
        char *basec = strdup(path);
        char *bname = basename(basec);
        return bname;
    }else
        return NULL;

}
```

```
int copyFile(char* src, char *dest){
  char copycommand[1000];
  if (checkValid(dest) == 0) {
    char temp[1000] = "mkdir -p ";
    strcat(temp, dest);
    system(temp);
  }
  sprintf(copycommand, "/bin/cp %s %s", src, dest);
  return system(copycommand);
}

int moveFile(char* src, char *dest){
  char movecommand[1000];

  if (checkValid(dest) == 0) {
    char temp[1000] = "mkdir -p ";
    strcat(temp, dest);
    system(temp);
  }
  sprintf(movecommand, "/bin/mv %s %s", src, dest);
  return system(movecommand);
}

char* getExtension(char *path){
  char *ptr = rindex(path, '.');
  return strdup(ptr);
}

char* stringConcat(char *str1, char *str2){
  char *strdup1 = strdup(str1);
  char *strdup2 = strdup(str2);
  strcat(strdup1, strdup2);
  return strdup1;
}
```

5 FDL Demos

5.1 HTML

```
def int main()
    path src = ”./demo/site”
    path js = ”./demo/site/js”
    path css = ”./demo/site/css”
    path imgs = ”./demo/site/images”
    path html = ”./demo/site/html”

    path f
    for ( f in src )
        if (f.type == ”.js”) then
            js <<- f
        end
        if (f.type == ”.css”) then
            css <<- f
        end
        if (f.type == ”.jpeg”) then
            imgs <<- f
        end
        if (f.type == ”.html” && f.name != ”index.html”) then
            html <<- f
        end
    end

    return 0
end
```

5.2 Duplicates

```
def int main()
  path dir1 = "./demo/duplicates/fdl_copy"
  path dir2 = "./demo/duplicates/fdl"
  path trash = "~/.Trash"
  path file1
  path file2
  int check = 0
  string a
  string b
  list l

  l = []

  for (file1 in dir2)
    a = file1.name
    l.add(a)
  end

  l.remove(".")
  l.remove("..")
  l.remove(".DS_Store")

  for (file2 in dir1)
    b = file2.name
    if b in l then
      print "Duplicate Found"
      print b
      trash <<- file2
      check = 1
    end
  end

  if (check != 1) then
    print "No duplicates found"
  end

  return 0
end
```

5.3 Same File Type Different Directories

```
def int main()
  path d1 = "."
  path d2 = "./c"
  path f1
  path f2
  list l
  string a
  string b
  int check = 0
  l = []

  for ( f1 in d1 )
    if (f1.kind == 1) then
      a = f1.type
      l.add(a)
    end
  end

  l.remove(".")
  l.remove("..")
  l.remove(".DS_Store")

  for ( f2 in d2 )
    b = f2.type
    if b in l then
      print "type " + b
      print f2.name
      check = 1
    end
  end

  if (check != 1) then
    print "No files of same type found"
  end

  return 0
end
```

6 FDL Tests

```
int a = 1 + 2
def int main()
  int b = a + 2
  print a + b
  return 1
end
```

```
  int a
  def int main()
    int b
    a = 2
    b = a+1
    print b
    return 0
  end
```

```
int a
def int main()
  a = 0
  a == 0
  print a==0
  return 0
end
```

```
bool b
def int main()
  b = 0
  print b
  b = false
  print b
  return 0
end
```

```
def int main()
  bool a = true
  print a
  a = 1==1
  if(1 < 2 == 1 == 0) then
    print true
  end
  return 0
end
```

```
def int main()
  bool a
  bool b
  bool c
  a = 1==1
  b = 1<1
  c = a || b
  print a
  print b
  print c
  return 0
end
```

```
int a
def int main()
  int b
  print 1/2
  return 1
end
```

```
def int main()
  print 1==2
  print 1==1
  return 1
end
```

```
def int main()
  path file
  path file2
  path dir
  file = "/Users/cjborenstein/Desktop/file.txt"
  file2 = "/Users/cjborenstein/Desktop/file2.txt"
  dir = "/Users/cjborenstein/Desktop/"
  return 1
end
```

```
def void findAndCopy()
  path loc1 = "./sample_program/copy.txt"
  path loc1dest = "."
  path loc2 = "./copy.txt"
  path loc2dest = "./sample_program"

  /* move the file out if it exists*/
  if (loc1.kind != 0) then
    loc1dest <<- loc1
    print "moved"
  end

  /* copy the file back*/
  loc2dest <- loc2
  print "copied"

  return
end

def int main()
  findAndCopy()
  return 0
end
```

```
def void fun(int a)
    print a
    return
end

def int main()
    int a
    a=5
    fun(5)
    return 0
end
```

```
def void findAndMove()
  path loc1 = "./sample_program/move.txt"
  path loc1dest = "."
  path loc2 = "./move.txt"
  path loc2dest = "./sample_program"

  if (loc1.kind != 0) then
    loc1dest <<- loc1
    print "moved to"
    print loc1dest
  else
    loc2dest <<- loc2
    print "moved to"
    print loc2dest
  end
  return
end

def int main()
  findAndMove()
  return 0
end
```

```
def int main()
    path f
    path dir = "."
    int count = 0
  for ( f in dir )
        print count
        print f
    count = count + 1
  end

  return 1
end
```

```
def int main()
    path f
    path dir
    path dest

    dir = "./test/src"
    dest = "./test/dest"

    print "destination "
    print dest.kind
    /* moving to destination */
  for ( f in dir )
        if (f.kind == 0) then
            print f
            dest <<- f
        end
  end

    /* moving back */
    for ( f in dest)
        if (f.kind == 0) then
            dir <<- f
        end
    end

  return 1
end
```

```
def int gcd(int a, int b)
  while (a != b)
    if(a>b) then
      a = a-b
    else
      b = b-a
    end
  end
  return a
end

def int main()
  int g = gcd(15, 30)
  print g
  return 0
end
```

```
def int main()
  print 1>=2
  print 1>=1
  print 2>=1
  return 1
end
```

```
def int main()
  print 1>2
  print 2>1
  return 1
end
```

```
int a
def int main()
    int b

    a = 0
    b = 3

    if ( a == 0 ) then
    print b
  end
  return 0
end
```

```
def int main()
  int a
  int b
  bool c
  a = 1
  b = 2
  c = a < b
  if ( a < b ) then
    print b
  end
  return 1
end
```

```
def int main()

  list l
  l = ["a","b",1,2,3]
  if 1 in l then print l
  return 0
end
```

```
def int main()
  print 1<=0
  print 1<=1
  print 1<=2
  return 1
end
```

```
def int main()
  print 1<1
  print 1<2
  return 1
end
```

```
def int main()
  list l
  l = [1,2]
  l.add(3)
  l.remove(1)
  print l
  return 0
end
```

```
def int main()
  list l
  l = [3]
  l.remove(1)
  print l
  return 0
end
```

```
def int main()
  list l
  l = []
  l.add(3)
  l.remove(1)
  print l
  return 0
end
```

```
def int main()
  list l
  l = []
  return 0
end
```

```
def int main()
  list l
  l = [”a”,”b”,1,2,3]

  return 0
end
```

```
/* using local var b */
int a

def int main()
  int b
  a = 2
  b = a + 1
  print b
  return 0
end
```

```
int a

def int main()
  path home
  home = "./sample_program/sample_path.fdl"

  /*print home
  print home.name*/

  print home.type
  print "\n"
  if(home.kind == 2) then
    print "we have a directory here"
  else
    print "we have a file here here"
  end

  return 1
end
```

```
int a

def int main()
  int a
  a = 2
  print a
  return 0
end
```

```
def int main()
  int b
  print "home/"+"files"
  return 1
end
```

```
int a
def int main()
  int b
  print 1-2
  return 1
end
```

```
int a

def int main()
  int b
  bool c
  a = 0
  b = 3
  c = a < b
  while ( a < b )
    b=a
  end
  return 1
end
```

```
int a
def int main()
  int b
  print 1*2
  return 1
end
```