Fli/o: File Manipulation Language

Final Project Report

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

1 Introduction **5**
1.1 Overview 5
1.2 Features 5

2 Language Tutorial **6**
2.1 Environment Setup 6
2.2 Compiling 6
2.3 Sample Program 6

3 Language Reference Manual **7**
3.1 Lexical Conventions 7
3.1.1 Tokens 7
3.1.2 Comments 7
3.1.3 Identifiers 7
3.1.4 Keywords 8
3.1.5 Literals 9
3.1.5.1 Integer Literal 9
3.1.5.2 String Literal 9
3.1.6 Separators 9
3.1.7 Operators 9
3.2 Types 10
3.3 Expressions 10
3.3.1 Parenthetical Expression 10
3.3.2 Function Call 10
3.3.3 Logical Not 11
3.3.4 Multiplication/Division Operators 11
3.3.5 Addition/Subtraction Operators 11
3.3.6 Relational Operators 12
3.3.7 Logical And 12
3.3.8 Logical Or 12
3.4 Statements 12
3.4.1 Expression Statement 13
3.4.2 Variable Declaration Statement 13
3.4.3 Return Statements 13
3.4.4 Block Statement 13
3.4.5 Control Flow Statements 13
3.4.5 Loops 14
3.4.6 Assignment Statement 14
3.5 Functions 14
3.5.1 Function Declaration 14

3.5.2 Function Structure 15
3.6 Built-in Functions 15
3.6.1 File-related 15
3.6.2 String-related 16
3.6.3 Miscellaneous 17
3.7 Scope 17
3.8 Grammar 17
4 Project Plan 19
4.1 Scheduling and Organization 19
4.2 Overview of Process 20
4.3 Style Guide 20
4.3.1 Indentation 20
4.3.2 Line width 21
4.3.3 Scoping (Curly Braces) 21
4.3.4 Spacing 21
4.3.5 Naming Conventions 22
4.4 Project Timeline 22
4.5 Team Roles 22
4.6 Development Environment 23
4.7 Project Log 23
5 Language Evolution 23
6 Architecture Design 25
6.1 Diagram 25
6.2 Compiler 25
6.2.1 your_program 25
6.2.2 Scanner 26
6.2.3 Parser 26
6.2.4 Abstract syntax tree (AST) 26
6.2.5 Static-semantic checker 26
6.2.6 Code Generation 26
6.2.7 Assembly 27
6.2.9 Linking 27
6.2.9 Executable 27
6.3 Contribution 27
7 Test Plan 27
7.1 Overview 27
7.2 Script Testing 28

7.3 Manual Testing 29
7.4 Responsibilities 30

8 Lessons Learned 30
8.1 General Conclusion 30
8.2 Personal Conclusions 31
8.2.1 Matthew’s Conclusion 31
8.2.2 Justin’s Conclusion 31
8.2.3 Gideon’s Conclusion 32
8.2.4 Eyob’s Conclusion 32

9 Appendix 32
9.1 Scanner 32
9.2 Parser 34
9.3 Abstract Syntax Tree 37
9.4 Static-semantic checker 40
9.5 Code Generation 45
9.6 Fli/o 54
9.7 Standard Library 55
9.8 Tests 63
9.8.1 Fail Tests 63
9.8.2 Success Tests 65
9.9 Demo 70
9.10 Miscellaneous 71
9.11 Git Log 74

1 Introduction

1.1 Overview

Fli/o was developed to create a seamless way to for users to interact with files, especially large documents that require file or directory manipulation. To avoid the confusion around buffers and input/output, we plan to allow users to open documents without having to worry about managing file pointers and remembering to closing file streams. This design should increase the ease with which users interact with files as a user just had to open them before proceeding to work with the file, no longer does the user of our language have to do any file memory management aside for indicating which file they want to work with in the first place. In short, we want to simplify the process of working with file I/O and change it from a pain point to a hallmark of the language.

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

1.2 Features

Some key features of the language that we believe are worthy of highlighting have been listed below. These are not all of the things that Fli/o supports but are only a few of some of the more interesting features.

- **Built-in file and directory types.** In order to make the task of interacting with files easier, Fli/o offers built-in file and directory types. These types are equivalent to the file and directory pointers returned by google the C Standard Library fopen and opendir functions. By providing these types, users can more easily and more intuitively interact with files via Fli/o's standard library of built-in functions.

- **Built-in library for file operations.** Fli/o provides numerous built-in functions to simplify the process of working with files. Some of the functionality provided include: creating files, moving files, copying files, deleting files, and reading and writing to and from files. The idea behind providing these built-in functions is to easily allow users to implement bash-like functions without having to worry about calling fork and exec (like in C).

2 Language Tutorial

2.1 Environment Setup

The Fli/o compiler requires an installation of OCaml (and dependencies like ocamlbuild/ocamlfind), a C compiler (e.g. clang, gcc, cc) and LLVM to build the project.

2.2 Compiling

To compile and create a Fli/o program first you need to download the Fli/o tar folder, unzip it and then open the directory that contains the unzipped files.

Once you are in the project root's src directory, you can run *make.* This will build the source code for the compiler and produce the executable flio.native.

To compile a program written in Fli/o, run the following commands from the src directory:

1. Output the LLVM IR to a .ll file
   a. `flio.native < your_program.f > your_program.ll`
2. Compile it to assembly with llc
   a. `llc your_program.ll`
3. Use clang to build the exe
   a. `clang your_program.s stdlib.o -o your_program`
4. Run the executable file
   a. `./your_program`

2.3 Sample Program

Before we begin the language manual and detail all of the possibilities in the language I have included a sample program demonstrates some of the file manipulation that is possible in Fli/o.

This program opens a file pointer to a file named "myfile.txt" in the current working directory and prints out the contents of the entire file, line by line.

```
string filename = 'myfile.txt';
// Open the file for reading / writing
```

```
file f = fopen(filename);
string line = readLine(f);
// Read and print all lines from the line
for(; strcmp(line, ‘’) != 0;;) {
	prints(line);
	line = readLine(f);
}
```

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Tokens

Tokens in our language can be categorized into five classes: identifiers, keywords, literals, operators, and other separators.

Note: Individual tokens must be separated by whitespace (i.e. spaces, tabs or newlines).

3.1.2 Comments

The token for single-line comments consists of two forward slashes (“//”), and the comment terminates with a newline character.

Please note that multi-line comments are not supported.

3.1.3 Identifiers

The characters which make up variable or function names, hence referred to as “identifiers”, must be in the set of alphanumeric characters or must be an underscore character. Furthermore, identifiers must begin with a letter (either uppercase or lowercase).

The regular expression which formally expresses these constraints is as follows:

```
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

3.1.4 Keywords

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

```
int, string file, dir, if, else, for, and, or, def, return
```

Keyword	Description
int	32-bit signed integer
string	string datatype
file	file datatype Note: Under the hood, this is represented by a file pointer.
dir	directory datatype
if	if-statement
else	
for	for-loop Note: The syntax for for-loops differs slightly from the standard C-style syntax. The third clause of a for-loop must also be followed by a semi-colon. Here is an example: `for(int i = 0; i < 10; i = i + 1;) {}`
and	A logical and, evaluates to true only if all operands for the and are true
or	A logical or, evaluates to true if any operand for the or is true
not	A logical not, negates the operand that it is attached to
def	A keyword used to defining a function
return	Keyword that defines the returned value of a function

3.1.5 Literals

3.1.5.1 Integer Literal

An integer constant consisting of a sequence of digits that is expressed strictly in decimal notation and 32 bit in size with the initial bit indicating sign.

3.1.5.2 String Literal

String literals must be surrounded by single-quotes, as in ‘hello world’.

3.1.6 Separators

As with C-style syntax, instructions are separated by semicolons. Furthermore, curly braces are used to define scope and parentheses are used either to explicitly specify order of operations or invoke a function call.

3.1.7 Operators

Below is a list of the operators provided in Fli/o. Please note that these operators are not overloaded and hence only work on integer types.

Operator	**Description**
+	Addition operator
-	Subtraction operator
*	Multiplication operator
/	Division operator
>	Greater than operator
<	Less than operator
==	Equality operator
!=	Non-equality operator

3.2 Types

Keyword	Description
int	32-bit signed integer
string	string datatype String literals must be surrounded by single-quotes. The underlying data is represented as an array of characters.
file	file datatype Files are represented by a file pointer.
dir	directory datatype Directories are represented by a directory pointer.

3.3 Expressions

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

3.3.1 Parenthetical Expression

Notation: (expression)

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

3.3.2 Function Call

Notation: identifier(arguments)

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

3.3.3 Logical Not

Notation: NOT expression

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

3.3.4 Multiplication/Division Operators

Notation:

expression * expression
expression / expression

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

3.3.5 Addition/Subtraction Operators

Notation:

expression + expression
expression - expression

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

3.3.6 Relational Operators

Notation:

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

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

3.3.7 Logical And

Notation: expression AND expression

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

3.3.8 Logical Or

Notation: expression OR expression

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

3.4 Statements

These statements are executed in sequence and contain no inherent value. Most statements are terminated using a semicolon. The following subsections describe how different statements can be expressed.

3.4.1 Expression Statement

Notation: expression;

The most basic statement is a standalone expression.

3.4.2 Variable Declaration Statement

Notation:

type identifier;
type identifier=expression;

These statements indicate the declaration of variables. The first form represents declaration without initializing whereas the second represents declaration and initialization.

3.4.3 Return Statements

Notation:

return expression;
return ;

Return statements must be placed inside of a function and must match the return type as specified in the function declaration.

3.4.4 Block Statement

Notation: {statement_list}

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

3.4.5 Control Flow Statements

Notation:

if (expression) statement
if (expression) statement else statement

The control flow statements listed above allow for different statements to be optionally executed. If the first expression, which must be an integer, evaluates to not 0, the first statement executes in all forms. The other forms indicate differently other statements to take and what conditions must be true to take them. If the preceding if expression is not evaluated to be non-0, then, if there is an else, the following final statement is executed.

3.4.5 Loops

Notation: for (statement_opt; expression_opt; statement_opt;) statement

These is one form of iterative statements:

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

3.4.6 Assignment Statement

Notation:

identifier=expression;

The expression must match the type of the identifier. Strings are literal values, so each time they are changed, new string is created. Files and dicts are references to larger structured types and so merely assign the identifier the old reference value. In particular, files and directories must be opened by fopen and dopen, respectively, though the structure referred to by a specific identifier may freely change.

3.5 Functions

3.5.1 Function Declaration

Notation: def identifier (params) type_opt {statement_list}

A function can be declared anywhere within a function (besides within another function or statement block). It is indicated that this is a function and the first declaration of such by the def (define) keyword. It is then followed by the identifier that names the function and then a series of comma-delimited parameters (0 or more) that are enclosed by parentheses. Each parameter consists of a type and a local identifier to be used in the function. After the parameters, there may be a type. This type is the return

type of the function. Finally, there are braces that surround the series of statements that make up the function itself.

3.5.2 Function Structure

Each function is passed its arguments by value (though the value of directories and files are references). The arguments included in the function call must match the types and order of the parameters listed in the function declaration. The return type of the function indicates if there need be return statements; a return type requires that a return statement with an expression of the correct type end all possible functions. Alternatively, lack of a return type means that there need not be a return statement in the function, though any return types must be of the second form listed in 3.4.3.

3.6 Built-in Functions

3.6.1 File-related

Below is a list of built-in functions related to manipulating files.

Name	**Function Signature**	**Description**
fopen	file fopen(string fpath)	Open a file pointer to the file at file path *fpath*
create	int create(string fname)	Create a new empty file with name *fname* Returns a negative value if the write failed, otherwise returns a non-negative value.
move	int move(string fp1, string fp2)	Move the file at file path *fp1* to the location *fp2* Returns a negative value if the write failed, otherwise returns a non-negative value.
copy	int copy(string f1, string f2)	Copy the file at file path *fp1* to the location *fp2* Returns a negative value if the write failed, otherwise returns a non-negative value.

delete	int delete(string fname)	Delete the file *fname* Returns a negative value if the write failed, otherwise returns a non-negative value.
write	int write(file f, string buf)	Write the string *buf* to the file *f*. Returns a negative value if the write failed, otherwise returns a non-negative value.
read	string read(file f, int len)	Read and return *len* characters from *f*
readLine	string readLine(file f)	Read up until a newline character is encountered in *f* and return a string containing that line
appendString	int appendString(file f, string buf)	Append *buf* to the end of *f* Returns a negative value if the write failed, otherwise returns a non-negative value.

3.6.2 String-related

Below is a list of built-in functions related to strings:

Name	Function Signature	Description
prints	void prints(string s)	Print *s* to stdout
concat	string concat(string s1, string s2)	Concatenate two strings, returning the concatenated version
strcmp	int strcmp(string s1, string s2)	Compares *s1* to *s2* Returns a negative value if *s1* < *s2* Returns a positive value if *s1* > *s2* Returns zero if *s1* == *s2*

intToStr	string intToStr(int i)	Returns *i* as a string

3.6.3 Miscellaneous

Name	**Function Signature**	**Description**
print	void print(int i)	Print *i* to stdout
dopen	dir dopen(string path)	Open a directory pointer to *path*
rmdir	int rmdir(string path)	Remove the directory located at *path* Returns a negative value if the write failed, otherwise returns a non-negative value.

3.7 Scope

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

3.8 Grammar

The grammar of Fli/o, as defined by our parser, is as follows:

```
program:
  decls EOF

decls:
  ε
| decls fdecl
| decls stmt
```

```
/* Function declaration / definition */
fdecl:
  DEF ID(params) typ_opt {stmt_list}

params:
  ε
| paramlist

paramlist:
  typ ID
| paramlist, typ ID

args:
  ε
| arglist
arglist:
  expr
| arglist COMMA expr

/* Statements */
stmt_opt:
  SEQUENCING
| stmt

stmt_list:
  ε
| stmt_list stmt

stmt:
  expr SEQUENCING
| vdecl_stmt
| ID = expr;
| RETURN expr;
| RETURN;
| [stmt_list]
| FOR(stmt_opt expr_opt ; stmt_opt) stmt
| IF(expr) stmt %prec NOELSE
| IF(expr) stmt ELSE stmt

vdecl_stmt:
  typ ID;
| typ ID = expr;
```

```
/* Expressions */
expr_opt:
  ε
| expr

expr:
| INTLIT
| STRINGLIT
| ID
| ID(args) %prec CALL
| expr + expr
| expr - expr
| expr * expr
| expr / expr
| expr < expr
| expr > expr
| expr == expr
| expr != expr
| expr AND expr
| expr OR expr
| NOT expr
| -expr %prec NEG

typ_opt:
  ε
| typ

typ:
  INT
| STRING
| FILE
| DIR
```

4 Project Plan

4.1 Scheduling and Organization

Throughout the semester we had a group Facebook message that we used as our primary mode of communication. Whenever there was some complication or confusion we quickly settled the issue

through this group chat. We met 1-2 times a week throughout the semester once with our IA John at minimum unless we were on break or had a clear idea of what had to be done next in the project. During the beginning of the project we would meet multiple times a week to iron out the design of our language near deadlines we would also meet more often. However some weeks where we still be continuing on the things we were working on the previous week we would not have a meeting.

To do the planning and to complete the written portions of the final project we used Google Docs so we could all work on the same documentations and easily and seamlessly collaborate with one another. This worked exceptionally well because we could start a messenger group call, work on the same document together and then quickly complete the task.

We also used GitHub as tool for version control, we had a master branch and a development branch. To ensure that the master branch was always working any changes to the master branch must be verified by another member in the group. So someone would submit a pull request with their personal branch to the master and then have it be approved by one other person and then it would become the new master branch. The development branch was to used a working copy where people would send broken or partially functional code where it could get a second look at for testing or for other developmental purposes.

4.2 Overview of Process

To complete the project we followed the schedule of deliverables as outlined at the beginning of the class. We first focused on the proposal, then the LRM and parser, then the Hello World Demo, and then completing the project and creating the final report. We found this to be the easiest way to complete the project as it gave us a clear timeline on when each deliverable should be completed. We began by first assigning the four roles as seen in section 4.2 to each of the members in the group. We then brain stormed several ideas for our language before settling on the idea of a file manipulation idea which became the language Fli/o as we developed it over time. From there we would start working on new parts of the project in accordance to people's roles and what people's areas of interests are. Which again is is displayed in the chart in 4.2 of team responsibilities. Once we complete the Language Reference Manual and parser work became less collaborative as people were all working on their own separate sections of the project, which is how we worked until the programming language was actually completed.

4.3 Style Guide

4.3.1 Indentation

Similar to Linux conventions, tabs should be 8 characters wide. Having this spacing makes identifying scope from indentation less difficult.

When doing pattern matching in OCaml, use indentation to line up the return values. This makes the code more readable.

4.3.2 Line width

Inspired again by Linux kernel conventions, lines should not run over 80 characters in width so as to make source code more readable. Having lines run over 80 characters often requires that the text wrap onto another line, which is not ideal.

4.3.3 Scoping (Curly Braces)

Function definitions should start on a new line from the function declaration. That is to say, after declaring a function, the opening curly brace indicating the start of the function definition should fall on the next line.

```
def myfunc(int a) int
{
	// Your code here…
}
```

However, for statements like for-loops and if-statements, the opening curly brace should fall on the same line. For example,

```
for (int i = 0; i < 10; i = i + 1;) {
	// Do stuff…
}
```

4.3.4 Spacing

In most cases, spaces should follow keywords and variable names. Additionally, spaces should surround binary operators such as +, -, /, *.

However, when performing function calls, there should be no padding inside the parentheses, nor should there be spacing between the function name and the opening parenthesis.

```
(* Do not do this *)
String.concat “” ( String.uppercase “hello” )

(* Do this *)
String.concat “” (String.uppercase “hello”);
```

4.3.5 Naming Conventions

Use camel case rather than underscores when naming variables.

Additionally, variable names should begin with a lowercase letter. However, for constant global variables, please use all caps.

Lastly, when possible, try to use shorter variable names. For example, use cntr over numberCounter.

4.4 Project Timeline

Deliverable	Notes	Deadline
Project Proposal	Met up to allocate team roles and brainstorm project ideas	September 19
Language Reference Manual Scanner, Parser	Completed scanner, parser and the abstract syntax tree before this deadline	October 15
Hello World	Implemented static semantic checking, testing, and the framework for code generation in time for this deadline	November 14
Project Presentation	Built a slide deck and presentation	December 17
Final Project	All components of the project are complete	December 19

4.5 Team Roles

Member	Role	Responsibilities
Eyob Tefera	Language Guru	Codegen, final report, presentation, testing
Justin Gross	System Architect	LRM, language design, Final Report
Gideon Cheruiyot	Tester	Testing, LRM, Final Report
Matthew Chan	Manager	Scanner, parser, AST, static-semantic checker, codegen, built-in library, tests, Makefiles & scripts, LRM, final report, presentation

4.6 Development Environment

To ensure that the project worked for every we used the Linux virtual machine that John, our TA, set up for us and the class. The VM included all possible languages, compilers, and tools that we could possibly need such as GCC, ocamlyacc, ocammlex, git, vim, LLVM, menhir, ocamlfmt, and more. This allowed us to all have access to the same tools and environments and avoid problems that could stem from different environments (Windows vs Mac) which would shift our focus away from our implementation of the language, so the singular native platform helped avoid this issue.

4.7 Project Log

The git log for this project is located at the very end of the Appendix (Section 9.11).

5 Language Evolution

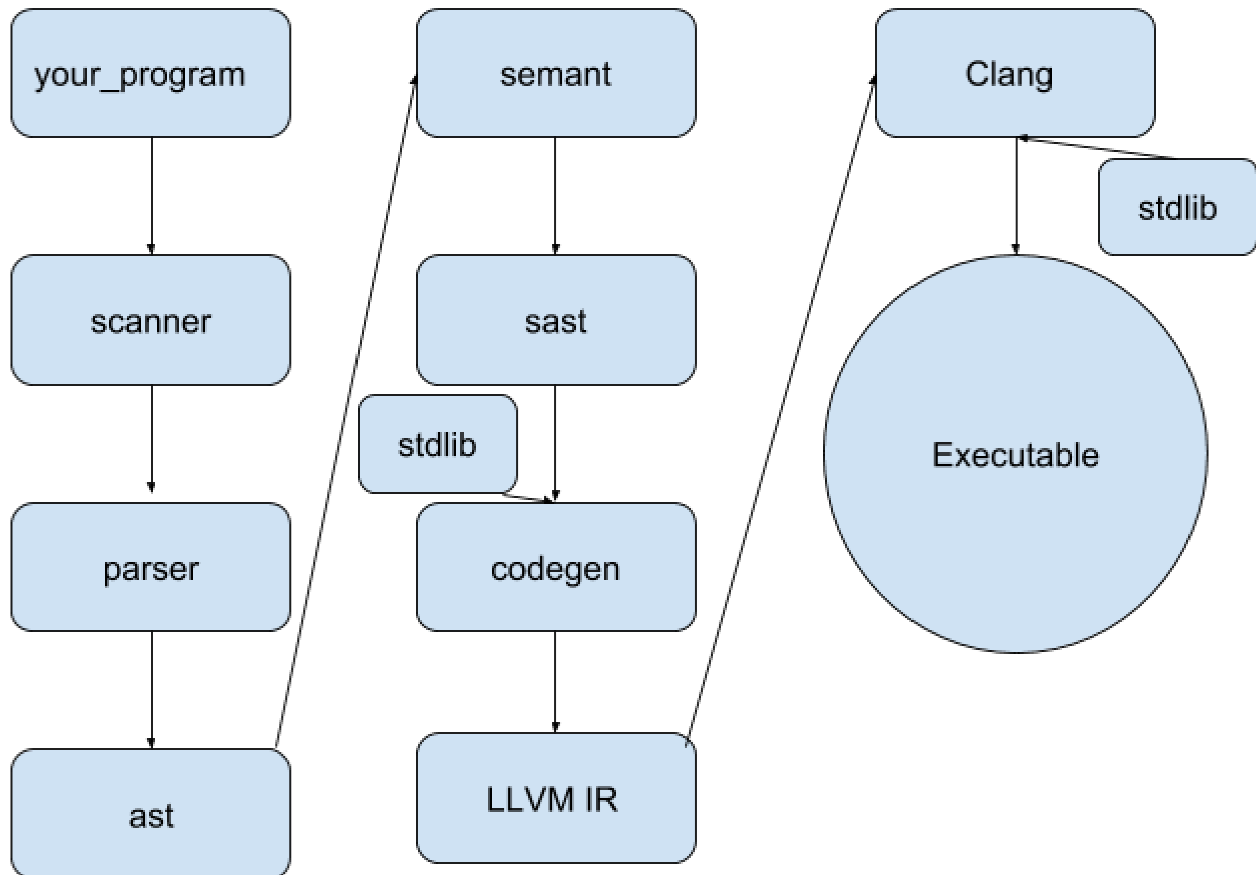
During the early stage of the project our group quickly selected upon two different languages, one that is a string manipulation language and another language that is centered around file management. We at first decided to combine these ideas because we felt that both of these ideas were fairly interesting and were something that we believe to be fairly interlinked. As a result on our first iteration of our project proposal we wrote both of these ideas into the language and meant then both equal focuses of our language.

However after receiving feedback from the TAs and Professor we released that having a language that focused on both would not only be difficult to implement in the time span that we were given, but we would be better served from focusing on one of these subjects rather than both. Furthermore, we were also given that insight that it would be easy to create or link in a library for doing some of the string processing that we had envisioned and it wasn't necessarily necessary to develop a language complete around string processing. So we settled on the idea of a file manipulation language. For the second proposal feedback we removed some of the ideas about string processing and decided that if that was direction we wanted to go in the future we would work with some external library to accomplish that.

In terms of our focus as a file manipulation language we decided to add more built-in methods for file manipulation and we decided to take from cues from command lines languages such as bash by adding in the pipe operator to mimic some of the already existing command file manipulation actions. Furthermore we decided to remove the ability to import libraries because we felt that it distracted from our new primary purpose as a file manipulation language. Thus through this iterative design process the language known as Fli/o was born.

6 Architecture Design

6.1 Diagram



6.2 Compiler

6.2.1 your_program

In the pipeline depicted above, your_program is meant to represent the program one would write in Fli/o. Furthermore, this represents the file that someone is attempting to compile when they run -c on the top level.

6.2.2 Scanner

The scanner reads *your_program* and performs the lexical analysis. It treats *your_program* as a large string of characters and then tokenizes it using the tokens previously mentioned in Section 3.1.1.

If *your_program* has illegal characters, the compilation will fail at this point. Otherwise, all the tokens are passed into the parser.

6.2.3 Parser

The parser uses the tokens passed in by the scanner and tries to construct an abstract syntax tree. The goal of the parser is to make sure that there are no grammatical errors present in the program. If any are detected, then the parser will fail at this stage. Otherwise, the parser successfully generates the AST.

6.2.4 Abstract syntax tree (AST)

The AST is the abstract syntax tree created by the parser. Once the Ast is created it is then passed in to the semantic checker where the AST is to be determined semantically valid or not.

The abstract syntax tree is a concise way of representing the source code which will be traversed by the static-semantic checker in the next phase.

6.2.5 Static-semantic checker

The static-semantic checker walks through the abstract syntax tree, checking for potential syntax errors such as type and scope errors. For instance, the semantic checker looks for whether variables exist when they are referenced.

If the semantic checker identifies any syntax errors, then the compilation fails here. Otherwise, the now semantically checked AST is ready for the next stage of the compilation process.

6.2.6 Code Generation

The code generation script traverses the AST to generate LLVM code, which is a register-based intermediate representation that looks similar to Assembly. We are compiling to LLVM for this project because it can easily be compiled into code that will run on any system architecture.

Note: Code generation should not fail, as all parser and syntax errors will be caught before this stage.

6.2.7 Assembly

Now that we have the LLVM intermediate representation of *your_program*, we can convert this into Assembly using LLC, LLVM's static compiler.

6.2.9 Linking

At this point, all we need is to convert our Assembly source code into an executable, while also remembering to link in Fli/o's standard library of built-in functions. For this purpose, you can use compilers like clang or gcc.

6.2.9 Executable

Compilation is now complete and the generated executable is ready to be run.

6.3 Contribution

All source code relating to the front-end and back-end of the compiler (i.e. scanner, parser, abstract syntax tree, static-semantic checker, code generator, standard library) was written by Matthew Chan.

7 Test Plan

7.1 Overview

We employed two different methods for testing the compiler. The first of which was creating a script that runs a bunch of unit tests on the various aspects of the language. This would ensure that as features were developed and added to the language that the features we already created were not broken by a subsequent change to the language. These tests ranged from extremely basic tests of a couple lines that merely did simple addition to longer tests that integrated multiple aspects of the language. We also did manual testing by creating executables and ensuring that the output of a program matched the output that we expected. This ensured that features not only continued to work but also worked as intended.

Here are two examples of the outputted Assembly code that is generated from a source Fli/o script. The first script demonstrates looping and conditionals, while the second script demonstrates file operations via built-in functions.

```
// test-if.f
// Author: Matthew Chan
for (int i = 0; i < 10; i = i + 1;) {
    if (i > 5 or i == 5) {
      prints('i is less or equal to 5');
    }
    else {
      prints('i is more than 5');
    }
}
```

```
        .text
        .file   "flio"
        .globl  main                    # -- Begin function main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:                                # %entry
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        pushq   %r15
        pushq   %r14
        pushq   %r13
        pushq   %r12
        pushq   %rbx
        pushq   %rax
        .cfi_offset %rbx, -56
        .cfi_offset %r12, -48
        .cfi_offset %r13, -40
        .cfi_offset %r14, -32
        .cfi_offset %r15, -24
        movq    %rsp, %rax
        leaq    -16(%rax), %rbx
        movq    %rbx, %rsp
        movl    $0, -16(%rax)
        leaq    .Lfmt(%rip), %r14
        leaq    .Lstrptr(%rip), %r15
        leaq    .Lfmt.2(%rip), %r12
        leaq    .Lstrptr.1(%rip), %r13
        cmpl    $9, (%rbx)
        jle     .LBB0_2
        jmp     .LBB0_7
        .p2align        4, 0x90
.LBB0_5:                                # %for
                                        #   in Loop: Header=BB0_2 Depth=1
        callq   printf@PLT
        incl    (%rbx)
        cmpl    $9, (%rbx)
        jg      .LBB0_7
.LBB0_2:                                # %for_body
                                        # =>This Inner Loop Header: Depth=1
        cmpl    $5, (%rbx)
        jg      .LBB0_4
# %bb.3:                                # %for_body
                                        #   in Loop: Header=BB0_2 Depth=1
        je      .LBB0_4
# %bb.6:                                # %else
                                        #   in Loop: Header=BB0_2 Depth=1
        xorl    %eax, %eax
        movq    %r12, %rdi
        movq    %r13, %rsi
        jmp     .LBB0_5
        .p2align        4, 0x90
```

```
// test-copyfile.f
// Author: Matthew Chan
string filename = 'myfile.txt';
create(filename);
string copyname = 'copyfile.txt';
file f = fopen(filename);

copy(filename, copyname);

delete(filename);
delete(copyname);
```

```
        .text
        .file   "flio"
        .globl  main                    # -- Begin function main
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:                                # %entry
        subq    $24, %rsp
        .cfi_def_cfa_offset 32
        leaq    .Lstrptr(%rip), %rdi
        movq    %rdi, (%rsp)
        callq   create@PLT
        leaq    .Lstrptr.1(%rip), %rax
        movq    %rax, 8(%rsp)
        movq    (%rsp), %rdi
        leaq    .Lmode(%rip), %rsi
        xorl    %eax, %eax
        callq   fopen@PLT
        movq    %rax, 16(%rsp)
        movq    8(%rsp), %rsi
        movq    (%rsp), %rdi
        callq   copy@PLT
        movq    (%rsp), %rdi
        xorl    %eax, %eax
        callq   remove@PLT
        movq    8(%rsp), %rdi
        xorl    %eax, %eax
        callq   remove@PLT
        xorl    %eax, %eax
        addq    $24, %rsp
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .type   .Lstrptr,@object        # @strptr
        .section        .rodata.str1.1,"aMS",@progbits,1
.Lstrptr:
        .asciz  "myfile.txt"
        .size   .Lstrptr, 11

        .type   .Lstrptr.1,@object      # @strptr.1
.Lstrptr.1:
        .asciz  "copyfile.txt"
        .size   .Lstrptr.1, 13

        .type   .Lmode,@object          # @mode
.Lmode:
        .asciz  "r+"
        .size   .Lmode, 3


        .section        ".note.GNU-stack","",@progbits
```

7.2 Script Testing

This is an example of us calling the test-suite during development and seeing tests pass and fail, this alerted us to what our changes broke and what needs to be fixed. In this case all of our tests passed once

we have made a change so know that we can continue working on new features because we did not damage the existing code.

The test cases for automated tested were selected primarily to test features of the language while they were being added. So for instance, there are tests corresponding to each built-in function of the language, as well as tests for keywords of the language.

Additionally, there were tests checking for syntactic errors including scoping and typing errors.

```
al@numel:~/plt/flio/src$ ./testall.sh
test-binop...OK
test-concat...OK
test-copyfile...OK
test-createfile...OK
test-decl2...OK
test-fcall2...OK
test-fcall...OK
test-fdecl2...OK
test-fdecl...OK
test-for...OK
test-hello2...OK
test-hello...OK
test-if...OK
test-logic...OK
test-movefile...OK
test-number...OK
test-readfile...test-rmdir...OK
test-string...OK
test-void...OK
fail-assign...OK
fail-fdecl2...OK
fail-fdecl3...OK
fail-fdecl...OK
fail-for...OK
fail-hello...OK
fail-if...OK
fail-return...OK
fail-scope...OK
fail-types...OK
```

7.3 Manual Testing

Manual testing occurred mostly when testing new features as they were implemented as whenever a new feature worked we wanted to ensure that a new feature not only worked but could interact with other features of the language without breaking them. We would manually test something to verify that it works before adding a test for a feature in the test suite. The test suite test would either be the same or a slightly modified version of the manually tested version. Below is an example of making sure that standard library functions work with string and file types. We would also change inputs and values to ensure that the code is valid not just for one input by for any input.

```
string filename = 'myfile.txt';
string newname = 'renamedfile.txt';
file f = filename;

move(filename, newname);

delete(newname);
~
~
~
~
```

7.4 Responsibilities

The automated testing script (testall.sh) as well the majority of the test scripts were written by Matthew Chan.

Additional test scripts were provided by Gideon Cheruiyot.

8 Lessons Learned

8.1 General Conclusion

Of the general sentiments that the group had there were quite a few that the group held in common. In terms of the project itself and the experience we had while working on it, every viewed it as extremely rewarding and insightful. We all have experiences with multiple different programming languages but none of us have either actually put one together so being able to break down what a programming language consists of and build one ourselves was again an amazing experience. In hindsight we may have chosen to focus on a single idea earlier rather than trying to combine multiple ideas in the early stages of the project the ideas at times got jumbled together and ran into one another. Also instead of everyone being assigned a portion of the compiler (ast, semant, codegen, etc..) we should have split the assignments into functionality rather than part of the compiler, some parts of the compiler were far more complicated than others and required more work. At first we didn't realize this so the workload was rather uneven but we tried to even it out over time. In hindsight work would should be assigned over time in small digestible portions than assigning entire portions of the project fool hardily. Aside from that I think we are all happy with the way that the language turned out implemented everything we had originally planned to do with the exception of pipe statements and imports. Building a language from scratch is rather difficult but it builds a lot of insight into how a language is written which in the future should

hopefully make us better software architectures since we now have a strong understanding of how the programming languages we use work, which should hopefully translate into the other languages we use.

8.2 Personal Conclusions

While the general conclusion was the section where we shared our thoughts about the project that we held in common this section is for our personal thoughts.

8.2.1 Matthew's Conclusion

After working on this project, I feel that I have gained a strong and intimate understanding of how programming languages and compilers work, both in theory and in practice. Seeing as how programming languages are the platform on which we build software, I find this knowledge crucial to my education in Computer Science.

As advice to other groups, I would suggest explicitly defining the core features of your language during the initial brainstorming phase and focus solely on implementing that feature. When I say this, I mean that all other features should be ignored and your language should be stripped down to its barebones in favor of implementing the core feature. In our case, the direction and purpose of our language got muddled during the planning phase due to over-ambition in what features we could implement.

As a last point of advice, I would also suggest concretely dividing up roles and assignments. Each member should take part in developing all portions of the compiler source code, so it would be a good idea to convey this notion early on. Otherwise one teammate is likely to end up writing the entire compiler and carrying most of the onus to meet the project deadlines.

8.2.2 Justin's Conclusion

I did enjoy working on the project and I certainly have a much better understanding of programming languages and how they work. It also gave me an appreciation for functional languages and how they can be used and why OCaml works well for making compilers.

I do feel that communication could have been a lot better in our group. While there was communication about what needed to happen to meet deadlines and who was working on what, communication outside of that was very limited. Communication on whether people needed help on certain parts and communication on whether communication on project direction outside of meetings were both somewhat neglected. It ultimately resulted in certain members claiming parts of the project and left much less room for collaboration. This made integration of parts more difficult and certain features were unable to be implemented in the final project. More meetings or, preferably, more regular communication about what was being done and why is really necessary to make the project run more smoothly.

8.2.3 Gideon's Conclusion

All in all, I feel like the long semester project was a great learning experience about how languages work. Starting off, the project idea we had was really optimistic and i think our TA really helped us narrow down on how to make Flio unique and better.

Personally, I feel like I would have reached out to our TA John regarding some difficulties I had while writing some components of the project like the testall.sh. More so, due to the bulk of the project, a great relationship with teammates early on is definitely helpful so as to understand each other's progress and to ask for help or clarification.

The challenging part of the project was collaboration since we could have done much better to help each other fix errors in our parts rather than overlapping in terms of work delegation.

8.2.4 Eyob's Conclusion

Overall, I would have to agree with the general consensus that we had around the project. I felt that the project went rather well, it had most of everything that we wanted to include in it. Yet if I had the opportunity to take the class again or in the future to design another programming language I think I might have chosen a mathematics based language, perhaps something similar to matlab. While working on the codegen section of the project there were a lot of interesting LLVM constructs that would lend themselves to being applied in a mathematical language. I think that creating built in matrix types and building a language that has the base functionality that the module numpy provides for python would have been really interesting as it would've given us the ability to use a lot of those LLVM constructs. I think there would have been a lot of conversion and interesting instances of typing matching or type conversion as well as size and array dimension conversions that would have made some type of numerical programming language interesting. Aside from maybe trying to build a different programming language, which is more of a retrospective than deeply held belief, I am extremely satisfied with the project and how it turned out.

9 Appendix

9.1 Scanner

src/scanner.mll

```
01: (*
02:   scanner.mll
```

```
03:   Author: Matthew Chan
04: *)
05:
06: { open Parser }
07:
08: rule token = parse
09:   [' ' '\t' '\r' '\n']       { token lexbuf }
10: (* Types *)
11: | "int"                      { INT }
12: | "string"                   { STRING }
13: | "file"                     { FILE }
14: | "dir"                      { DIR }
15: (* Function keywords *)
16: | "def"                      { DEF }
17: | "return"                   { RETURN }
18: (* Loops and conditionals *)
19: | "for"                      { FOR }
20: | "in"                       { IN }
21: | "if"                       { IF }
22: | "else"                     { ELSE }
23: (* Misc *)
24: | "//"                       { comment lexbuf }
25: | '('                        { LPAREN }
26: | ')'                        { RPAREN }
27: | '{'                        { LBRACE }
28: | '}'                        { RBRACE }
29: | '['                        { LBRACK }
30: | ']'                        { RBRACK }
31: | ','                        { COMMA }
32: | ';'                        { SEQUENCING }
33: (* Operators *)
34: | '+'                        { PLUS }
35: | '-'                        { MINUS }
36: | '*'                        { TIMES }
37: | '/'                        { DIVIDE }
38: | '>'                        { GT }
39: | '<'                        { LT }
40: | "=="                       { EQ }
41: | "!="                       { NEQ }
42: | "and"                      { AND }
43: | "or"                       { OR }
44: | '='                        { ASSIGNMENT }
45: | "not"                      { NOT }
```

```
46: (* Literals*)
47: | ['0'-'9']+ as lit          { INTLIT(int_of_string lit) }
48: | '\''([^'\'']* as string_lit)'\''      { STRINGLIT(string_lit) }
49: | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id { ID(id) }
50:
51: | eof                       { EOF }
52:
53: and comment = parse
54:   '\n'                      { token lexbuf }
55: | _                         { comment lexbuf }
```

9.2 Parser

src/parser.mll

```
001: /*
002:   parser.mly
003:   Author: Matthew Chan
004: */
005:
006: %{ open Ast %}
007:
008: %token EOF LBRACE RBRACE LPAREN RPAREN LBRACK RBRACK COMMA
SEQUENCING
009: %token INT STRING FILE DIR
010: %token PLUS MINUS TIMES DIVIDE ASSIGNMENT
011: %token GT LT EQ NEQ NOT AND OR
012: %token DEF RETURN
013: %token FOR IN IF ELSE
014: %token <int> INTLIT
015: %token <string> STRINGLIT
016: %token <string> ID
017:
018: %nonassoc NOELSE
019: %nonassoc ELSE
020: %right ASSIGNMENT
021: %left CALL
022: %left OR
023: %left AND
024: // %left SEQUENCING
025: %left EQ NEQ
026: %left LT GT
```

```
027: %left PLUS MINUS
028: %left TIMES DIVIDE
029: %right NOT NEG
030:
031: %start program
032: %type <Ast.program> program
033:
034: %%
035:
036: /* { funcs: [<fdecl>]; stmts: [<stmt>] } */
037: program:-
038:        decls EOF { {funcs = $1.funcs; stmts = List.rev $1.stmts}
}
039:
040: decls:
041:             { {funcs = []; stmts = []} }
042: | decls fdecl    { {funcs = ($2 :: $1.funcs); stmts = $1.stmts} }
043: | decls stmt     { {funcs = $1.funcs; stmts = ($2 :: $1.stmts)} }
044:
045:
046: /* Function declaration / definition */
047: fdecl:
048:   DEF ID LPAREN params RPAREN typ_opt LBRACE stmt_list RBRACE
{ {typ = $6; fname = $2; params = $4; body = List.rev $8} }
049:
050: params:
051:             { [] }
052: | paramlist{ List.rev $1 }
053:
054: paramlist:
055:   typ ID                   { [($1, $2)] }
056: | paramlist COMMA typ ID   { ($3, $4) :: $1}
057:
058:
059: args:
060:             { [] }
061: | arglist  { List.rev $1 }
062:
063: arglist:
064:   expr                 { [$1] }
065: | arglist COMMA expr  { $3 :: $1 }
066:
067: /* Statements */
```

```
068: stmt_opt:
069:   SEQUENCING          { Nostmt }
070: | stmt                { $1 }
071:
072: stmt_list:
073:             { [] }
074: | stmt_list stmt      { $2 :: $1 }
075:
076: stmt:
077:   expr SEQUENCING                      { Expr($1) }
078: | vdecl_stmt                           { $1 }
079: | asn_stmt                             { $1 }
080: | RETURN expr SEQUENCING               { Return($2) }
081: | RETURN SEQUENCING                    { Return(Noexpr) }
082: | LBRACE stmt_list RBRACE              { Block(List.rev $2) }
083: | FOR LPAREN stmt_opt expr_opt SEQUENCING stmt_opt RPAREN stmt
          { For($3, $4, $6, $8) }
084: | IF LPAREN expr RPAREN stmt %prec NOELSE
{ If($3, $5, Block([])) }
085: | IF LPAREN expr RPAREN stmt ELSE stmt      { If($3, $5, $7) }
086:
087: vdecl_stmt:
088:   typ ID SEQUENCING                    { VarDecl($1, $2) }
089: | typ ID ASSIGNMENT expr SEQUENCING    { VarDeclAsn($1, $2, $4) }
090:
091:
092: asn_stmt:
093:   ID ASSIGNMENT expr SEQUENCING
{ Asn($1, $3) }
094: | ID LBRACK expr RBRACK ASSIGNMENT expr SEQUENCING
{ Asn($1, $3) }
095:
096: /* Expressions */
097: expr_opt:
098:                 { Noexpr }
099: | expr          { $1 }
100:
101: expr:
102: | INTLIT                          { IntLit($1) }
103: | STRINGLIT                       { StringLit($1) }
104: | ID                              { Id($1) }
105: | ID LPAREN args RPAREN %prec CALL     { FuncCall($1, $3) }
106: | expr PLUS  expr                 { Binop($1, Add, $3) }
```

```
107: | expr MINUS expr                { Binop($1, Sub, $3) }
108: | expr TIMES expr                { Binop($1, Mul, $3) }
109: | expr DIVIDE expr               { Binop($1, Div, $3) }
110: | expr LT  expr                  { Binop($1, Lt, $3) }
111: | expr GT  expr                  { Binop($1, Gt, $3) }
112: | expr EQ  expr                  { Binop($1, Eq, $3) }
113: | expr NEQ  expr                 { Binop($1, Neq, $3) }
114: | expr AND  expr                 { Binop($1, And, $3) }
115: | expr OR  expr                  { Binop($1, Or, $3) }
116: | NOT expr                       { Uop(Not, $2) }
117: // | MINUS expr %prec NEG         { Uop(Neg, $2) }
118:
119: /* Types */
120: typ_opt:
121:  { Void }
122: | typ { $1 }
123:
124: typ:
125:   INT      { Int }
126: | STRING   { String }
127: | FILE     { File }
128: | DIR      { Dir }
```

9.3 Abstract Syntax Tree

src/ast.ml

```
001: (*
002:   ast.ml
003:   Author: Matthew Chan
004: *)
005:
006: type operator = Add | Sub | Mul | Div | Gt | Lt | Eq | Neq | And
| Or
007:
008: type uoperator = Neg | Not
009:
010: type typ = Int | String | File | Dir | Void
011:
012: (* Statements can be expressions or local var declarations *)
013: type param = typ * string
014:
```

```
015: (* Expressions are assignment and basic operations *)
016: type expr =
017:   Noexpr
018: | Binop of expr * operator * expr
019: | Uop of uoperator * expr
020: | IntLit of int
021: | StringLit of string
022: | Id of string
023: | FuncCall of string * expr list
024:
025: type stmt =
026:   Nostmt
027: | Block of stmt list
028: | Expr of expr
029: | VarDecl of typ * string
030: | VarDeclAsn of typ * string * expr
031: | Asn of string * expr
032: | Return of expr
033: | For of  stmt * expr * stmt * stmt
034: | If of expr * stmt * stmt
035:
036:
037: (* Functions have a return type, name, argument list, and body of
statements *)
038: type fdecl = {
039:  typ: typ;
040:  fname: string;
041:  params: param list;
042:  body: stmt list;
043: }
044:
045: (* Program is composed of functions and statements *)
046: type program = {
047:  funcs: fdecl list;
048:  stmts: stmt list;
049: }
050:
051: (* Pretty printing functions *)
052: let string_of_op = function
053:   Add -> "+"
054: | Sub -> "-"
055: | Mul -> "*"
056: | Div -> "/"
```

```
057: | Eq -> "=="
058: | Neq -> "!="
059: | Lt -> "<"
060: | Gt -> ">"
061: | And -> "and"
062: | Or -> "or"
063:
064: let string_of_uop = function
065:   Neg -> "-"
066: | Not -> "!"
067:
068: let string_of_typ = function
069:   Int -> "int"
070: | Void -> "void"
071: | String -> "string"
072: | File -> "file"
073: | Dir -> "dir"
074:
075: let rec string_of_expr = function
076:   IntLit(l) -> string_of_int l
077: | StringLit(l) -> l
078: | Id(s) -> s
079: | Binop(e1, op, e2) -> let lhs = string_of_expr e1 and rhs =
string_of_expr e2 in
080:  (lhs ^ " " ^ string_of_op op ^ " " ^ rhs)
081: | Uop(op, e) -> string_of_uop op ^ string_of_expr e
082: | FuncCall(f, args) -> f ^ "(" ^ String.concat ", " (List.map
string_of_expr args) ^ ")"
083: | Noexpr -> ""
084:
085: let rec string_of_stmt = function
086:   Block(stmts) ->
087:  "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
088: | Expr(expr) -> string_of_expr expr ^ ";\n";
089: | VarDecl(t, id) -> (string_of_typ t) ^ " " ^ id ^ ";\n"
090: | VarDeclAsn(t, id, e) -> (string_of_typ t) ^ " " ^ id ^ " = " ^
(string_of_expr e) ^ ";\n"
091: | Asn(id, e) -> id ^ " = " ^ (string_of_expr e) ^ ";\n"
092: | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
093: | For(s1, e, s2, s3) ->
094:       "for (" ^ string_of_stmt s1  ^ " ; " ^ string_of_expr e ^ "
; " ^
095:       string_of_stmt s2  ^ ";) " ^ string_of_stmt s3
```

```
096: | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
097: | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
098:         string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
099: | Nostmt -> ""
100:
101: let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
102:
103: let string_of_fdecl fdecl =
104:   string_of_typ fdecl.typ ^ " " ^
105:   fdecl.fname ^ "(" ^ String.concat ", " (List.map snd
fdecl.params) ^
106:   ")\n{\n" ^
107:   String.concat "" (List.map string_of_stmt fdecl.body) ^
108:   "}\n"
109:
110:
111: let string_of_program program =
112:   String.concat "\n" (List.map string_of_fdecl program.funcs) ^
"\n" ^
113:   String.concat "\n" (List.map string_of_stmt program.stmts)
```

9.4 Static-semantic checker

src/semant.ml

```
001: (*
002:  * semant.ml
003:  * Author: Matthew Chan
004:  *)
005: open Ast
006:
007: module StringMap = Map.Make (String)
008:
009: let check ast =
010:
011:        (* Raise an exception if the given list has a duplicate *)
012:        let report_duplicate exceptf list =
013:                let rec helper = function
014:                        n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
015:                      | _ :: t -> helper t
```

```
016:                    | [] -> ()
017:              in helper (List.sort compare list)
018:        in
019:
020:        (* Raise an exception if a given binding is to a void type *)
021:        let check_not_void exceptf = function
022:                (Void, n) -> raise (Failure (exceptf n))
023:              | _ -> ()
024:        in
025:
026:        (* Raise an exception of the given rvalue type cannot be assigned to
027:        the given lvalue type *)
028:        let check_assign lvaluet rvaluet err =
029:              if lvaluet == rvaluet then lvaluet else raise err
030:        in
031:
032:
033:        (* Print function cannot be redefined *)
034:        if List.mem "print" (List.map (fun fd -> fd.fname) ast.funcs)
035:        then raise (Failure ("function print may not be defined")) else ();
036:
037:
038:        (* Duplicate function names not permitted *)
039:        report_duplicate (fun n -> "duplicate function " ^ n)
040:        (List.map (fun fd -> fd.fname) ast.funcs);
041:
042:        let built_in_decls =  StringMap.add "print"
043:              { typ = Void; fname = "print"; params = [(Int, "x")];
044:              body = [] } (StringMap.add "prints" { typ = Void; fname = "prints"; params =
[(String, "x")];
045:              body = []} (StringMap.add "fopen" { typ = File; fname = "fopen"; params =
[(String, "f")];
046:              body = []} (StringMap.add "delete" { typ = Int; fname = "delete"; params =
[(String, "x")];
047:              body = []} (StringMap.add "copy" { typ = Int; fname = "copy"; params = [(String,
"src") ; (String, "dest")];
048:              body = []} (StringMap.add "move" { typ = Int; fname = "move"; params = [(String,
"src") ; (String, "dest")];
049:              body = []} (StringMap.add "write" { typ = Int; fname = "write"; params = [(File, "f") ;
(String, "buf")];
050:              body = []} (StringMap.add "read" { typ = String; fname = "read"; params = [(File,
"f") ; (Int, "length")];
```

```
051:            body = []} (StringMap.add "readLine" { typ = String; fname = "readLine"; params
= [(File, "f")];
052:            body = []} (StringMap.add "appendString" { typ = Int; fname = "readLine"; params
= [(String, "f") ; (String, "buf")];
053:            body = []} (StringMap.add "dopen" { typ = Dir; fname = "dopen"; params =
[(String, "d")];
054:            body = []} (StringMap.add "rmdir" { typ = Int; fname = "rmdir"; params = [(String,
"d")];
055:            body = []} (StringMap.add "concat" { typ = String; fname = "rmdir"; params =
[(String, "s1") ; (String, "s2")];
056:            body = []} (StringMap.add "strcmp" { typ = Int; fname = "strcmp"; params =
[(String, "s1") ; (String, "s2")];
057:            body = []} (StringMap.add "intToStr" { typ = String; fname = "intToStr"; params =
[(Int, "i")];
058:            body = []} (StringMap.add "create" { typ = Int; fname = "create"; params =
[(String, "filename")];
059:            body = []} StringMap.empty)))))))))))))))
060:        in
061:
062:        (* Keep track of function declarations *)
063:        let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
064:        built_in_decls ast.funcs
065:        in
066:
067:        let function_decl s = try StringMap.find s function_decls
068:        with Not_found -> raise (Failure ("unrecognized function " ^ s))
069:        in
070:
071:
072:        let type_of_identifier s map =
073:            try
074:                StringMap.find s map
075:            with Not_found -> raise (Failure ("undeclared identifier " ^ s))
076:        in
077:
078:        let rec expr map = function
079:              IntLit _ -> Int
080:            | StringLit _ -> String
081:            | Id s -> let t = type_of_identifier s map in t
082:            | Noexpr -> Void
083:            | Uop(op, e) as ex -> let t = expr map e in
084:                (match op with
085:                  Neg when t = Int -> Int
```

```
086:                | Not when t = Int -> Int
087:                | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
088:                    string_of_typ t ^ " in " ^ string_of_expr ex))
089:                )
090:            | FuncCall(f, args) as call -> let fd = function_decl f in
091:                if List.length args != List.length fd.params then
092:                    raise (Failure ("expecting " ^ string_of_int
093:                    (List.length fd.params) ^ " arguments in " ^ string_of_expr call))
094:                else
095:                    List.iter2 (fun (ft, _) e -> let et = expr map e in
096:                    ignore (check_assign ft et (Failure ("illegal actual argument found " ^
097:                    string_of_typ et ^ " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr
e))))
098:                    fd.params args ; fd.typ
099:            | Binop(e1, op, e2) as e -> let t1 = expr map e1 and t2 = expr map e2 in
100:                (match op with
101:                  Add | Sub | Mul | Div when t1 = Int && t2 = Int -> Int
102:                | Eq | Neq when t1 = t2 -> Int
103:                | Lt | Gt when t1 = Int && t2 = Int -> Int
104:                | And | Or when t1 = Int && t2 = Int -> Int
105:                | _ -> raise (Failure ("illegal binary operator " ^
106:                    string_of_typ t1 ^ " " ^ string_of_op op ^
107:                    " " ^ string_of_typ t2 ^ " in " ^ string_of_expr e))
108:                )
109:
110:        in
111:
112:        let check_bool_expr map e = if expr map e != Int
113:            then raise (Failure ("expected Boolean expression in "
114:                ^ string_of_expr e))
115:            else ()
116:        in
117:
118:        (**** Check Global Scope ****)
119:        let check_stmt s =
120:
121:            (* Type of each variable (global, formal, or local *)
122:            let symbols = StringMap.empty
123:
124:            in
125:
126:            let rec stmt map = function
127:                Block sl -> let rec check_block m = function
```

```
128:                         [Return _ as s] -> stmt m s
129:                        | Return _ :: _ -> raise (Failure "nothing may follow a return")
130:                        | Block sl :: ss -> check_block m (sl @ ss)
131:                        | s :: ss -> check_block (stmt m s) ss
132:                        | [] -> m
133:                        in check_block map sl
134:                  | VarDecl(t, n) -> (StringMap.add n t map)
135:                  | VarDeclAsn(t, n ,e) -> ignore(expr map e); (StringMap.add n t map)
136:                  | Asn(n, e) as ex -> let lt = type_of_identifier n map and rt = expr map e in
137:                        ignore(check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ
lt ^
138:                        " = " ^ string_of_typ rt ^ " in " ^ string_of_stmt ex))) ; map
139:                  | Expr e -> ignore(expr map e) ; map
140:                  | Return e -> ignore(expr map e) ; raise (Failure ("returns not allowed
outside of function scope"))
141:                  | For(s1, e, s2, s3) -> let m = stmt map s1 in
142:                        ignore(expr m e); ignore(stmt (stmt m s2) s3) ; map
143:                  | If(e, s1, s2) -> check_bool_expr map e; ignore(stmt map s1); ignore(stmt
map s2); map
144:                  | Nostmt -> map
145:            in stmt symbols (Block s)
146:
147:      in
148:
149:      (**** Check Functions ****)
150:      let check_function global_map func =
151:
152:            (* Params cannot have void type *)
153:            List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
154:            " in " ^ func.fname)) func.params;
155:
156:            (* Params cannot have duplicate names *)
157:            report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
158:            (List.map snd func.params);
159:
160:            (* Type of each variable (global, formal, or local *)
161:            let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m)
162:            global_map (func.params)
163:            in
164:
165:            (* Verify a statement or throw an exception *)
166:            let rec stmt map = function
167:                  Block sl -> let rec check_block m = function
```

```
168:                              [Return _ as s] -> stmt m s
169:                            | Return _ :: _ -> raise (Failure "nothing may follow a return")
170:                            | Block sl :: ss -> check_block m (sl @ ss)
171:                            | s :: ss -> check_block (stmt m s) ss
172:                            | [] -> m
173:                            in check_block map sl
174:                   | VarDecl(t, n) -> (StringMap.add n t map)
175:                   | VarDeclAsn(t, n , _) -> (StringMap.add n t map)
176:                   | Asn(n, e) as ex -> let lt = type_of_identifier n map and rt = expr map e in
177:                            ignore(check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ
lt ^
178:                            " = " ^ string_of_typ rt ^ " in " ^ string_of_stmt ex))) ; map
179:                   | Expr e -> ignore(expr map e) ; map
180:                   | Return e -> let t = expr map e in
181:                            if t = func.typ then map
182:                            else raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
183:                                     string_of_typ func.typ ^ " in " ^ string_of_expr e))
184:                   | For(s1, e, s2, s3) -> ignore(expr map e); ignore(stmt (stmt (stmt map s1)
s2) s3) ; map
185:                   | If(e, s1, s2) -> check_bool_expr map e; ignore(stmt map s1); ignore(stmt
map s2); map
186:                   | Nostmt -> map
187:
188:            in ignore(stmt symbols (Block func.body))
189:
190:      in List.iter (check_function (check_stmt ast.stmts)) ast.funcs
```

9.5 Code Generation

src/codegen.ml

```
001: (*
002:  * codegen.ml
003:  * Author: Matthew Chan
004:  *)
005: module L = Llvm
006: module A = Ast
007:
008: module StringMap = Map.Make(String)
009:
010: let translate program =
011:
```

```
012:  (* Setup LLVM environment vars *)
013:  let context = L.global_context () in
014:    let the_module = L.create_module context "flio"
015:    and i32_t = L.i32_type context
016:    and i8_t = L.i8_type context
017:    and str_ptr_t = L.pointer_type (L.i8_type context)
018:    and void_t = L.void_type context
019:  in
020:
021:  (* Pattern match AST types to LLVM types *)
022:  let ltype_of_typ = function
023:        A.Int -> i32_t
024:      | A.String -> str_ptr_t
025:      | A.Void -> void_t
026:      | A.File -> str_ptr_t
027:      | A.Dir -> str_ptr_t
028:  in
029:
030:  (* Utility function for getting a val from a map, given a key *)
031:  let lookup n m = StringMap.find n m
032:  in
033:
034:
035:  (* Utility function to build a return block *)
036:  let add_terminal builder f =
037:    match L.block_terminator (L.insertion_block builder) with
038:        Some _ -> ()
039:      | None -> ignore (f builder) in
040:
041:  (* Declare built-in functions *)
042:  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
043:  let printf_func = L.declare_function "printf" printf_t the_module in
044:
045:  let concat_t = L.function_type (L.pointer_type i8_t) [| L.pointer_type i8_t ; L.pointer_type
i8_t |] in
046:  let concat_func = L.declare_function "concat" concat_t the_module in
047:
048:  let intToStr_t = L.function_type (L.pointer_type i8_t) [| i32_t |] in
049:  let intToStr_func = L.declare_function "intToStr" intToStr_t the_module in
050:
051:  let strcmp_t = L.function_type i32_t [| L.pointer_type i8_t ; L.pointer_type i8_t |] in
052:  let strcmp_func = L.declare_function "strcmp" strcmp_t the_module in
053:
```

```
054:  let create_t = L.function_type i32_t [| L.pointer_type i8_t |] in
055:  let create_func = L.declare_function "create" create_t the_module in
056:
057:  let fopen_t = L.var_arg_function_type (L.pointer_type i8_t) [| L.pointer_type i8_t |] in
058:  let fopen_func = L.declare_function "fopen" fopen_t the_module in
059:
060:  let dopen_t = L.function_type (L.pointer_type i8_t) [| L.pointer_type i8_t |] in
061:  let dopen_func = L.declare_function "opendir" dopen_t the_module in
062:
063:  let delete_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
064:  let delete_func = L.declare_function "remove" delete_t the_module in
065:
066:  let rmdir_t = L.function_type i32_t [| L.pointer_type i8_t |] in
067:  let rmdir_func = L.declare_function "rmdir" rmdir_t the_module in
068:
069:  let copy_t = L.function_type i32_t [| L.pointer_type i8_t ; L.pointer_type i8_t |] in
070:  let copy_func = L.declare_function "copy" copy_t the_module in
071:
072:  let move_t = L.function_type i32_t [| L.pointer_type i8_t ; L.pointer_type i8_t |] in
073:  let move_func = L.declare_function "move" move_t the_module in
074:
075:  let write_t = L.function_type i32_t [| L.pointer_type i8_t ; L.pointer_type i8_t |] in
076:  let write_func = L.declare_function "bwrite" write_t the_module in
077:
078:  let appendstr_t = L.function_type i32_t [| L.pointer_type i8_t ; L.pointer_type i8_t |] in
079:  let appendstr_func = L.declare_function "appendString" appendstr_t the_module in
080:
081:  let read_t = L.function_type (L.pointer_type i8_t) [| L.pointer_type i8_t ; i32_t |] in
082:  let read_func = L.declare_function "bread" read_t the_module in
083:
084:  let readline_t = L.function_type (L.pointer_type i8_t) [| L.pointer_type i8_t |] in
085:  let readline_func = L.declare_function "readLine" readline_t the_module in
086:
087:  (* Build a map of function declarations *)
088:  let function_decls =
089:    let function_decl m fdecl =
090:      let name = fdecl.A.fname
091:      and formal_types =
092:        Array.of_list (List.map (fun (t, _) -> ltype_of_typ t) fdecl.A.params)
093:      in let ftype = L.function_type (ltype_of_typ fdecl.A.typ) formal_types in
094:      StringMap.add name (L.define_function name ftype the_module, fdecl) m in
095:    List.fold_left function_decl StringMap.empty program.A.funcs in
096:
```

```
097:   (* Default format strings *)
098:   let int_format_str builder = L.build_global_stringptr "%d\n" "fmt" builder in
099:   let str_format_str builder = L.build_global_stringptr "%s\n" "fmt" builder in
100:   let fopen_mode builder = L.build_global_stringptr "r+" "mode" builder in
101:
102:   (* Construct code for an expression; return its value *)
103:   let rec expr map builder = function
104:         A.IntLit i -> L.const_int i32_t i
105:       | A.Noexpr -> L.const_int i32_t 0
106:       | A.FuncCall ("print", [e]) ->
107:               L.build_call printf_func [| (int_format_str builder) ;
108:               (expr map builder e) |] "printf" builder
109:       | A.FuncCall ("concat", [s1 ; s2]) ->
110:               L.build_call concat_func [| (expr map builder s1) ;
111:               (expr map builder s2) |] "concat" builder
112:       | A.FuncCall ("create", [f]) ->
113:               L.build_call create_func [| (expr map builder f) |]
114:               "create" builder
115:       | A.FuncCall ("intToStr", [e]) ->
116:               L.build_call intToStr_func [| (expr map builder e) |]
117:               "intToStr" builder
118:       | A.FuncCall ("strcmp", [s1 ; s2]) ->
119:               L.build_call strcmp_func [| (expr map builder s1) ;
120:               (expr map builder s2) |] "strcmp" builder
121:       | A.FuncCall ("fopen", [e]) ->
122:                     L.build_call fopen_func [| (expr map builder e) ; (fopen_mode builder)|]
"fopen" builder
123:       | A.FuncCall ("dopen", [e]) ->
124:                     L.build_call dopen_func [| (expr map builder e) |] "dopen" builder
125:       | A.FuncCall ("delete", [e]) ->
126:       L.build_call delete_func [| (expr map builder e) |] "delete" builder
127:       | A.FuncCall ("rmdir", [e]) ->
128:       L.build_call rmdir_func [| (expr map builder e) |] "rmdir" builder
129:       | A.FuncCall ("copy", [e1 ; e2]) ->
130:                     L.build_call copy_func [| (expr map builder e1) ; (expr map builder e2)|]
"copy" builder
131:       | A.FuncCall ("write", [e1 ; e2]) ->
132:                     L.build_call write_func [| (expr map builder e1) ; (expr map builder e2)|]
"write" builder
133:       | A.FuncCall ("appendString", [e1 ; e2]) ->
134:                     L.build_call appendstr_func [| (expr map builder e1) ; (expr map builder e2)|]
"appendString" builder
135:       | A.FuncCall ("read", [e1 ; e2]) ->
```

```
136:                    L.build_call read_func [| (expr map builder e1) ; (expr map builder e2)|] "read"
builder
137:        | A.FuncCall ("readLine", [e1]) ->
138:                    L.build_call readline_func [| (expr map builder e1) |] "readLine" builder
139:        | A.FuncCall ("move", [e1 ; e2]) ->
140:                    L.build_call move_func [| (expr map builder e1) ; (expr map builder e2)|]
"move" builder
141:        | A.FuncCall ("prints", [e]) ->
142:               L.build_call printf_func [| (str_format_str builder);
143:               (expr map builder e) |] "printf" builder
144:        | A.Binop (e1, op, e2) ->
145:        let e1' = expr map builder e1
146:        and e2' = expr map builder e2 in
147:        (match op with
148:          A.Add     -> L.build_add
149:        | A.Sub     -> L.build_sub
150:        | A.Mul    -> L.build_mul
151:            | A.Div     -> L.build_sdiv
152:        | A.And     -> L.build_and
153:        | A.Or      -> L.build_or
154:        | A.Eq   -> L.build_icmp L.Icmp.Eq
155:        | A.Neq     -> L.build_icmp L.Icmp.Ne
156:        | A.Lt    -> L.build_icmp L.Icmp.Slt
157:        | A.Gt -> L.build_icmp L.Icmp.Sgt
158:        ) e1' e2' "tmp" builder
159:        | A.Uop (_, _) -> raise (Failure ("not implemented yet"))
160:        | A.StringLit s ->
161:               L.build_global_stringptr s "strptr" builder
162:        | A.Id s -> L.build_load (lookup s map) s builder
163:        | A.FuncCall(f, args) ->
164:                let (fdef, fdecl) = StringMap.find f function_decls in
165:                let actuals = List.rev (List.map (expr map builder) (List.rev args)) in
166:                let result = (match fdecl.A.typ with
167:                  A.Void -> ""
168:                  | _ -> f ^ "_result") in
169:                L.build_call fdef (Array.of_list actuals) result builder
170:      in
171:
172:    (* Build function bodies *)
173:    let build_function_body global_map fdecl =
174:      (* Find the function in our map *)
175:      let (fdef, _) = StringMap.find fdecl.A.fname function_decls in
176:      (* Move the builder to the entry point of that function *)
```

```
177:    let builder = L.builder_at_end context (L.entry_block fdef) in
178:
179:
180:    (* The function's local variables are initially the function args *)
181:    let add_formal m (t, n) p = L.set_value_name n p;
182:      let local = L.build_alloca (ltype_of_typ t) n builder in
183:      ignore (L.build_store p local builder);
184:      StringMap.add n local m in
185:    let local_vars = List.fold_left2 add_formal global_map
186:      fdecl.A.params (Array.to_list (L.params fdef)) in
187:
188:    let rec fstmt mb = function
189:        A.Block sl -> (List.fold_left fstmt mb sl)
190:      | A.Expr e -> ignore (expr (fst mb) (snd mb) e); mb
191:      | A.Nostmt -> mb
192:      | A.For (s1, e, s2, body) ->
193:              (* Construct for basic block *)
194:              let init_bb = L.append_block context "init" fdef in
195:              ignore (L.build_br init_bb (snd mb));
196:
197:              let pred_bb = L.append_block context "for" fdef in
198:              ignore(pred_bb);
199:
200:              let init = fstmt (fst mb, L.builder_at_end context init_bb) s1 in
201:              add_terminal (snd init) (L.build_br pred_bb);
202:
203:              (* Construct body basic block, and add s2 at the tail *)
204:              let body_bb = L.append_block context "for_body" fdef in
205:              let b = (fstmt (fst mb, L.builder_at_end context body_bb) body) in
206:              add_terminal (snd (fstmt b s2)) (L.build_br pred_bb);
207:
208:              (* Do initialization before checking the predicate e *)
209:              let pred_builder = L.builder_at_end context pred_bb in
210:              let bool_val = expr (fst init) pred_builder e in
211:
212:              (* Construct merge basic block *)
213:              let merge_bb = L.append_block context "merge" fdef in
214:              ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
215:              (fst mb, L.builder_at_end context merge_bb)
216:      | A.If (p, then_stmt, else_stmt) ->
217:         let bool_val = expr (fst mb) (snd mb) p in
218:     let merge_bb = L.append_block context "merge" fdef in
219:
```

```
220:     let then_bb = L.append_block context "then" fdef in
221:     add_terminal (snd (fstmt (fst mb, L.builder_at_end context then_bb) then_stmt))
222:       (L.build_br merge_bb);
223:
224:     let else_bb = L.append_block context "else" fdef in
225:     add_terminal (snd (fstmt (fst mb, L.builder_at_end context else_bb) else_stmt))
226:       (L.build_br merge_bb);
227:
228:     ignore (L.build_cond_br bool_val then_bb else_bb (snd mb));
229:      (fst mb, L.builder_at_end context merge_bb)
230:      | A.Return e -> ignore(match fdecl.A.typ with
231:                A.Void -> L.build_ret_void (snd mb)
232:              | _ -> L.build_ret (expr (fst mb) (snd mb) e) (snd mb)); mb
233:      | A.VarDecl (t, n) -> let init =
234:            (match t with
235:                A.Int -> L.build_alloca i32_t n (snd mb)
236:              | A.String -> L.build_alloca str_ptr_t n (snd mb)
237:              | A.File -> L.build_alloca str_ptr_t n (snd mb)
238:              | A.Dir -> L.build_alloca str_ptr_t n (snd mb)
239:              | A.Void -> L.build_ret_void (snd mb)
240:            ) in
241:            ((StringMap.add n init (fst mb)), snd mb)
242:      | A.VarDeclAsn (t, n, e) -> let init =
243:            (match t with
244:                A.Int -> L.build_alloca i32_t n (snd mb)
245:              | A.String -> L.build_alloca str_ptr_t n (snd mb)
246:              | A.File -> L.build_alloca str_ptr_t n (snd mb)
247:              | A.Dir -> raise (Failure ("not implemented yet"))
248:              | A.Void -> L.build_ret_void (snd mb)
249:            ) in
250:            let m = (StringMap.add n init (fst mb)) in
251:            let e' = expr (m) (snd mb) e in
252:            ignore(L.build_store e' (lookup n (m)) (snd mb)) ; (m, (snd mb))
253:      | A.Asn (s, e) -> let e' = expr (fst mb) (snd mb) e in
254:        ignore(L.build_store e' (lookup s (fst mb)) (snd mb)) ; mb
255:    in
256:
257:    let builder = (snd (fstmt (local_vars, builder) (A.Block fdecl.A.body)))
258:    in
259:    (* Add a return if the last block falls off the end *)
260:    add_terminal builder (match fdecl.A.typ with
261:        A.Void -> L.build_ret_void
262:      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
```

```
263:  in
264:
265:  (* Declare main function *)
266:  let main_t = L.function_type i32_t [| |] in
267:  let main_func = L.define_function "main" main_t the_module in
268:
269:  (* Variables in main are the empty set initially *)
270:  let main_vars = StringMap.empty in
271:
272:  (* Init builder inside of main *)
273:  let builder = L.builder_at_end context (L.entry_block main_func) in
274:
275:  (* Build statments inside of the main function *)
276:  let build_stmts s =
277:
278:    (* Build the code for the given statement; return the StringMap and builder
279:       for the statement's successor *)
280:    let rec stmt mb = function
281:        A.Block sl -> (List.fold_left stmt mb sl)
282:      | A.Expr e -> ignore (expr (fst mb) (snd mb) e); mb
283:      | A.Nostmt -> mb
284:      | A.For (s1, e, s2, body) ->
285:              (* Construct for basic block *)
286:              let init_bb = L.append_block context "init" main_func in
287:              ignore (L.build_br init_bb (snd mb));
288:
289:              let pred_bb = L.append_block context "for" main_func in
290:              ignore(pred_bb);
291:
292:              let init = stmt (fst mb, L.builder_at_end context init_bb) s1 in
293:              add_terminal (snd init) (L.build_br pred_bb);
294:
295:              (* Construct body basic block, and add s2 at the tail *)
296:              let body_bb = L.append_block context "for_body" main_func in
297:              let b = (stmt (fst init, L.builder_at_end context body_bb) body) in
298:              add_terminal (snd (stmt b s2)) (L.build_br pred_bb);
299:
300:              (* Do initialization before checking the predicate e *)
301:              let pred_builder = L.builder_at_end context pred_bb in
302:              let bool_val = (expr (fst init) pred_builder e) in
303:
304:              (* Construct merge basic block *)
305:              let merge_bb = L.append_block context "merge" main_func in
```

```
306:                ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
307:                (fst mb, L.builder_at_end context merge_bb)
308:       | A.If (p, then_stmt, else_stmt) ->
309:          let bool_val = (expr (fst mb) (snd mb) p) in
310:      let merge_bb = L.append_block context "merge" main_func in
311:
312:      let then_bb = L.append_block context "then" main_func in
313:      add_terminal (snd (stmt (fst mb, L.builder_at_end context then_bb) then_stmt))
314:        (L.build_br merge_bb);
315:
316:      let else_bb = L.append_block context "else" main_func in
317:      add_terminal (snd (stmt (fst mb, L.builder_at_end context else_bb) else_stmt))
318:        (L.build_br merge_bb);
319:
320:      ignore (L.build_cond_br bool_val then_bb else_bb (snd mb));
321:       (fst mb, L.builder_at_end context merge_bb)
322:       | A.Return _ -> raise (Failure ("returns are not allowed in main"))
323:       | A.VarDecl (t, n) -> let init =
324:              (match t with
325:                  A.Int -> L.build_alloca i32_t n (snd mb)
326:                | A.String -> L.build_alloca str_ptr_t n (snd mb)
327:                | A.File -> L.build_alloca str_ptr_t n (snd mb)
328:                | A.Dir -> L.build_alloca str_ptr_t n (snd mb)
329:                | A.Void -> L.build_ret_void (snd mb)
330:              ) in
331:              ((StringMap.add n init (fst mb)), snd mb)
332:       | A.VarDeclAsn (t, n, e) -> let init =
333:              (match t with
334:                  A.Int -> L.build_alloca i32_t n (snd mb)
335:                | A.String -> L.build_alloca str_ptr_t n (snd mb)
336:                | A.File -> L.build_alloca str_ptr_t n (snd mb)
337:                | A.Dir -> L.build_alloca str_ptr_t n (snd mb)
338:                | A.Void -> L.build_ret_void (snd mb)
339:              ) in
340:              let m = (StringMap.add n init (fst mb)) in
341:              let e' = (expr (m) (snd mb) e) in
342:              ignore(L.build_store e' (lookup n (m)) (snd mb)) ; (m, (snd mb))
343:       | A.Asn (s, e) -> let e' = (expr (fst mb) (snd mb) e) in
344:         ignore(L.build_store e' (lookup s (fst mb)) (snd mb)) ; mb
345:     in
346:
347:     (* Build the code for each statement in the function *)
348:     let retval = stmt (main_vars, builder) (A.Block s) in
```

```
349:    ignore(add_terminal (snd retval) (L.build_ret (L.const_int i32_t 0))) ; fst retval
350:   in
351:
352:
353:   ignore(List.iter (build_function_body (build_stmts program.A.stmts)) program.A.funcs);
354:
355:   (* Add terminal for main function *)
356:   the_module
```

9.6 Fli/o

src/flio.ml

```
01: (*
02:  * flio.ml
03:  * Author: Matthew Chan
04:  *)
05: type action = Ast | LLVM_IR | Compile
06:
07: let () =
08:         let action = ref Compile in
09:         let set_action a () = action := a in
10:         let speclist = [
11:                 ("-a", Arg.Unit (set_action Ast), "Print the
SAST");
12:                 ("-l", Arg.Unit (set_action LLVM_IR), "Print the
generated LLVM IR");
13:                 ("-c", Arg.Unit (set_action Compile), "Check and
print the generated LLVM IR (default)");
14:                 ] in
15:         let usage_msg = "usage: ./filo.native [-a|-s|-l|-c]
[file.f]" in
16:         let channel = ref stdin in
17:         Arg.parse speclist
18:                 (fun filename -> channel := open_in filename)
usage_msg;
19:
20:         let lexbuf = Lexing.from_channel !channel in
21:
22:         let ast = Parser.program Scanner.token lexbuf in
23:         ignore(Semant.check ast);
24:
```

```
25:         match !action with
26:              Ast -> print_string (Ast.string_of_program ast)
27:            | LLVM_IR -> print_string (Llvm.string_of_llmodule
(Codegen.translate ast))
28:            | Compile -> let m = Codegen.translate ast in
29:                 Llvm_analysis.assert_valid_module m;
30:                 print_string (Llvm.string_of_llmodule m)
```

9.7 Standard Library

src/stdlib.c

```
001: /*
002:  * stdlib.c
003:  * Author: Matthew Chan
004:  */
005: #include <fcntl.h>
006: #include <stdio.h>
007: #include <sys/wait.h>
008: #include <unistd.h>
009: #include <sys/types.h>
010: #include <errno.h>
011: #include <string.h>
012: #include <stdlib.h>
013:
014: int copy(char *src, char *dest)
015: {
016:  int status;
017:  pid_t pid = fork();
018:  /* Child proc */
019:  if (pid == 0) {
020:       char *const args[] = {"/bin/cp", src, dest, NULL};
021:       /* Syscall interrupt */
022:       execv("/bin/cp", args);
023:
024:       /* Child should not reach this point */
025:       fprintf(stderr, "error: %s\n", strerror(errno));
026:       exit(1);
027:  }
028:  /* Parent proc */
029:  else {
030:       wait(&status);
```

```
031:        return WEXITSTATUS(status);
032:  }
033:  return -1;
034: }
035:
036: int create(char *filename)
037: {
038:  int status;
039:  pid_t pid = fork();
040:  /* Child proc */
041:  if (pid == 0) {
042:        char *const args[] = {"/bin/touch", filename, NULL};
043:        /* Syscall interrupt */
044:        execv("/bin/touch", args);
045:
046:        /* Child should not reach this point */
047:        fprintf(stderr, "error: %s\n", strerror(errno));
048:        exit(1);
049:  }
050:  /* Parent proc */
051:  else {
052:        wait(&status);
053:        return WEXITSTATUS(status);
054:  }
055:  return -1;
056: }
057:
058: int move(char *src, char *dest)
059: {
060:  int status;
061:  pid_t pid = fork();
062:  /* Child proc */
063:  if (pid == 0) {
064:        char *const args[] = {"/bin/cp", src, dest, NULL};
065:        /* Syscall interrupt */
066:        execv("/bin/mv", args);
067:
068:        /* Child should not reach this point */
069:        fprintf(stderr, "error: %s\n", strerror(errno));
070:        exit(1);
071:  }
072:  /* Parent proc */
073:  else {
```

```
074:       wait(&status);
075:       return WEXITSTATUS(status);
076:  }
077:  return -1;
078: }
079:
080: ssize_t bwrite(FILE *f, const char *buf)
081: {
082:  /* int fd = fileno(f); */
083:  /* return write(fd, buf, strlen(buf)); */
084:  return fputs(buf, f);
085: }
086:
087: char *bread(FILE *f, size_t count)
088: {
089:  char *buf = malloc(sizeof(char) * count);
090:  fgets(buf, count + 1, f);
091:  return buf;
092: }
093:
094: char *readLine(FILE *f)
095: {
096:  return bread(f, 1000);
097: }
098:
099: int appendString(const char *f, const char *buf)
100: {
101:
102:  FILE *file = fopen(f, "a");
103:  int fd = fileno(file);
104:  int ret = write(fd, buf, strlen(buf));
105:  fclose(file);
106:  return ret;
107: }
108:
109: char *concat(const char *s1, const char *s2)
110: {
111:  char *c = malloc(strlen(s1) + strlen(s2) + 1);
112:  strcpy(c, s1);
113:  strcat(c, s2);
114:  return c;
115: }
116:
```

```
117: char *intToStr(int value)
118: {
119:  char *buf = malloc(sizeof(char) * 10);
120:  sprintf(buf, "%d", value);
121:  return buf;
122: }
123:
124: #ifdef DEBUG
125: int main(int argc, char **argv)
126: {
127:  /* copy("./test.txt", "./test2.txt"); */
128:  /* move("./test.txt", "./test3.txt"); */
129:  /* FILE *f = fopen("test3.txt", "r+"); */
130:  /* bwrite(fd, "hi there"); */
131:  /* char *s = bread(fd, 2); */
132:  /* char *s; */
133:  /* size_t len; */
134:  /* getline(&s, &len, fd); */
135:
136:  /* printf("%s", readLine(f)); */
137:  return 0;
138: }
139: #endif
```

src/testall.sh

```
001:
002: #!/bin/sh
003: #  testall.sh
004: #  Author: Matthew Chan
005: #  Inspiration: MicroC
006:
007: #  Fli/o regression testing script
008: #  executes then checks the errors of all tests expected to fail
009:
010: # LLVM interpreter path
011: LLI="lli"
012: #LLI="/usr/local/opt/llvm/bin/lli"
013:
014: # Path to the LLVM compiler
015: LLC="llc"
016:
017: # Path to the C compiler
```

```
018: CC="cc"
019:
020: FLIO="./flio.native"
021: #flio="_build/flio.native"
022:
023: # time limit for operations
024: ulimit -t 30
025:
026: globallog=testall.log
027: rm -f $globallog
028: error=0
029: globalerror=0
030:
031: keep=0
032:
033: Usage() {
034:     echo "Usage: testall.sh [options] [.f files]"
035:     echo "-k    Keep intermediate files"
036:     echo "-h    Print this help"
037:     exit 1
038: }
039:
040: SignalError() {
041:     if [ $error -eq 0 ] ; then
042:  echo "FAILED"
043:  error=1
044:     fi
045:     echo "  $1"
046: }
047:
048: # Compare <outfile> <reffile> <difffile>
049: # Compares the outfile with reffile.  Differences, if any,
written to difffile
050: Compare() {
051:     generatedfiles="$generatedfiles $3"
052:     echo diff -b $1 $2 ">" $3 1>&2
053:     diff -b "$1" "$2" > "$3" 2>&1 || {
054:  SignalError "$1 differs from $2"
055:  echo "FAILED $1 differs from $2" 1>&2
056:     }
057: }
058:
059: # Run <args>
```

```
060: # Report the command, run it, and report any errors
061: Run() {
062:     echo $* 1>&2
063:     eval $* || {
064:  SignalError "$1 failed on $*"
065:  return 1
066:     }
067: }
068:
069: # RunFail <args>
070: # Report the command, run it, and expect an error
071: RunFail() {
072:     echo $* 1>&2
073:     eval $* && {
074:  SignalError "failed: $* did not report an error"
075:  return 1
076:     }
077:     return 0
078: }
079:
080: Check() {
081:     error=0
082:     basename=`echo $1 | sed 's/.*\\///
083:                              s/\.f//'`
084:     reffile=`echo $1 | sed 's/\.f$//'`
085:     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
086:
087:     echo -n "$basename..."
088:
089:     echo 1>&2
090:     echo "###### Testing $basename" 1>&2
091:
092:     generatedfiles=""
093:
094:     generatedfiles="$generatedfiles ${basename}.ll
${basename}.out" &&
095:     Run "$FLIO" "<" $1 ">" "${basename}.ll" &&
096:     Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">"
"${basename}.s" &&
097:     Run "$CC" "-o" "${basename}.exe" "${basename}.s"
"../src/stdlib.o" &&
098:     Run "./${basename}.exe" > "${basename}.out" &&
099:     Compare ${basename}.out ${basename}.out ${basename}.diff
```

```
100:
101:     # Report the status and clean up the generated files
102:
103:     if [ $error -eq 0 ] ; then
104:  if [ $keep -eq 0 ] ; then
105:      rm -f $generatedfiles
106:  fi
107:  echo "OK"
108:  echo "###### SUCCESS" 1>&2
109:     else
110:  echo "###### FAILED" 1>&2
111:  globalerror=$error
112:     fi
113: }
114:
115: CheckFail() {
116:     error=0
117:     basename=`echo $1 | sed 's/.*\\///
118:                             s/\.f//'`
119:     reffile=`echo $1 | sed 's/\.f$//'`
120:     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
121:
122:     echo -n "$basename..."
123:
124:     echo 1>&2
125:     echo "###### Testing $basename" 1>&2
126:
127:     generatedfiles=""
128:
129:     generatedfiles="$generatedfiles ${basename}.err
${basename}.diff" &&
130:     RunFail "$FLIO" "<" $1 "2>" "${basename}.err" ">>" $globallog
&&
131:     Compare ${basename}.err ${basename}.err ${basename}.diff
132:
133:     # Report the status and clean up the generated files
134:
135:     if [ $error -eq 0 ] ; then
136:  if [ $keep -eq 0 ] ; then
137:      rm -f $generatedfiles
138:  fi
139:  echo "OK"
140:  echo "###### SUCCESS" 1>&2
```

```
141:     else
142:  echo "###### FAILED" 1>&2
143:  globalerror=$error
144:     fi
145: }
146:
147: while getopts kdpsh c; do
148:     case $c in
149:  k) # Keep intermediate files
150:      keep=1
151:      ;;
152:  h) # Help
153:      Usage
154:      ;;
155:     esac
156: done
157:
158: shift `expr $OPTIND - 1`
159:
160: LLIFail() {
161:   echo "LLVM interpreter not found \"$LLI\"."
162:   echo "Check LLVM installation/modify the LLI variable in
testall.sh"
163:   exit 1
164: }
165:
166: which "$LLI" >> $globallog || LLIFail
167:
168: # if [ ! -f stlib.o ]
169: # then
170: #     echo "Could not find stdlib.o"
171: #     echo "Try \"make stdlib.o\""
172: #     exit 1
173: # fi
174:
175: if [ $# -ge 1 ]
176: then
177:     files=$@
178: else
179:     files="../test/test-*.f ../test/fail-*.f"
180: fi
181:
182: for file in $files
```

```
183: do
184:     case $file in
185:  *test-*)
186:      Check $file 2>> $globallog
187:      ;;
188:  *fail-*)
189:      CheckFail $file 2>> $globallog
190:      ;;
191:  # *)
192:  #     echo "unknown file type $file"
193:  #     globalerror=1
194:  #     ;;
195:     esac
196: done
197:
198: exit $globalerror
```

9.8 Tests

9.8.1 Fail Tests

test/fail-assign.f

```
1: // fail-assign.f
2: // Author: Matthew Chan
3: a = 5;
4: int a;
```

test/fail-fdecl.f

```
1: // fail-fdecl.f
2: // Author: Matthew Chan
3: def voidfunc() void
4: {
5:       prints('Functions should not declare void return type
explicity');
6: }
```

test/fail-fdecl2.f

```
1: // fail-fdecl2.f
2: // Author: Matthew Chan
3: def myfunc() {}
```

```
4: def myfunc() {}
```

test/fail-fdecl3.f

```
1: // fail-fdecl3.f
2: // Author: Matthew Chan
3: def dup_param(int a, int a) {}
```

test/fail-for.f

```
1: // fail-for.f
2: // Author: Matthew Chan
3: for(int i = 0; i < 5; i = i + 1) {
4:       prints('hello world');
5: }
```

test/fail-hello.f

```
1: // fail-hello.f
2: // Author: Matthew Chan
3: prints("hello world!");
```

test/fail-if.f

```
01: // fail-if.f
02: // Author: Matthew Chan
03: int a = 5;
04:
05: else {
06:         prints('a is equal to 5');
07: }
08: if (a != 5) {
09:         prints('a is not equal to 5');
10: }
```

test/fail-return.f

```
1: // fail-return.f
2: // Author: Matthew Chan
3: return 1;
```

test/fail-scope.f

```
01: // fail-scope.f
02: // Author: Matthew Chan
03: int a = 5;
04:
05: def printa()
06: {
```

```
07:         print(a);
08: }
09:
10: printa();
```

test/fail-types.f

```
1: // fail-types.f
2: // Author: Matthew Chan
3: int a;
4: int b;
5: string s;
6: a = b + s;
```

9.8.2 Success Tests

test/test-binop.f

```
01: // test-binop.f
02: // Author: Matthew Chan
03: int a = 10;
04: int b = 50;
05:
06: b - a;
07: b + a;
08: b * a;
09: b / a;
10: b == a;
11: b != a;
12: b > a;
13: b < a;
```

test/test-concat.f

```
1: // test-concat.f
2: // Author: Matthew Chan
3: prints(concat('hello', 'world'));
```

test/test-copyfile.f

```
01: // test-copyfile.f
02: // Author: Matthew Chan
03: string filename = 'myfile.txt';
04: create(filename);
05: string copyname = 'copyfile.txt';
06: file f = fopen(filename);
```

```
07:
08: copy(filename, copyname);
09:
10: delete(filename);
11: delete(copyname);
```

test/test-createfile.f

```
1: // test-createfile.f
2: // Author: Matthew Chan
3: string filename = 'myfile.txt';
4: file f = filename;
5:
6: appendString(filename, 'hola mundo!');
7:
8: delete(filename);
```

test/test-decl2.f

```
01: // test-decl2.f
02: // Author: Matthew Chan
03: string s = 'hello';
04: prints(s);
05:
06: int a;
07: a = 5;
08: print(5 * a + 10);
09: for (int b = 0; b < 4; b = b + 1;) {
10: }
11:
12: def hi() {
13:         prints('hi');
14: }
15:
16: hi();
17:
18: def get5() int {
19:         return 5;
20: }
21:
22: print(get5());
23:
24: //file f = './test.txt';
25:
26:
```

```
27:
28: //copy(f, './test2.txt');
29: //move(f, './test3.txt');
30: //file f2 = './test3.txt');
31: //int fd = fopen(f2);
32: //write(fd, 'hithere');
33:
34: file f = fopen('test3.txt');
35: // write(f, 'hellofriend\n');
36: // prints(readLine(f));
37: // appendString('test3.txt', 'appendingSTRING!');
38:
39: // rmdir('deleteme');
40: // string[5] a;
```

test/test-fcall.f

```
01: // test-fcall.f
02: // Author: Matthew Chan
03: def sum(int a, int b) int
04: {
05:       return a + b;
06: }
07:
08: int a = 10;
09: int b = 5;
10: print(sum(a, b));
```

test/test-fcall2.f

```
01: // test-fcall2.f
02: // Author: Matthew Chan
03: int a = 10;
04: int b = 5;
05: print(sum(a, b));
06:
07: def sum(int a, int b) int
08: {
09:       return a + b;
10: }
```

test/test-fdecl.f

```
1: // test-fdecl.f
2: // Author: Matthew Chan
3: def sum(int a, int b) int
```

```
4: {
5:       return a + b;
6: }
```

test/test-fdecl2.f

```
1: // test-fdecl2.f
2: // Author: Matthew Chan
3: def filefun() file
4: {
5:         file f = fopen('myfile.txt');
6:         return f;
7: }
```

test/test-for.f

```
1: // test-for.f
2: // Author: Matthew Chan
3: int j = 10;
4: for(int i = 0; i < 10; i = i + 1;) {
5:         j = j + i;
6: }
```

test/test-hello.f

```
1: // test-hello.f
2: // Author: Matthew Chan
3: prints('hello world!');
```

test/test-hello2.f

```
1: // test-hello2.f
2: // Author: Matthew Chan
3: print(1);
4: print(10);
5: print(55);
```

test/test-if.f

```
01: // test-if.f
02: // Author: Matthew Chan
03: for (int i = 0; i < 10; i = i + 1;) {
04:       if (i > 5 or i == 5) {
05:         prints('i is less or equal to 5');
06:       }
07:       else {
08:         prints('i is more than 5');
```

```
09:       }
10: }
```

test/test-logic.f

```
01: // test-logic.f
02: // Author: Matthew Chan
03: def go_outside(int dow, string weather) int
04: {
05:       if (dow == 5 or dow == 6 and strcmp(weather, 'horrible') !=
0)
06:         return 1;
07:       return 0;
08: }
09:
10: print(go_outside(5, 'great'));
```

test/test-movefile.f

```
01: // test-movefile.f
02: // Author: Matthew Chan
03: string filename = 'myfile.txt';
04: create(filename);
05: string newname = 'renamedfile.txt';
06:
07: move(filename, newname);
08:
09: delete(newname);
```

test/test-number.f

```
1: // test-number.f
2: // Author: Matthew Chan
3: int a = 5;
4: a = a * 10;
5: int b = a / 5;
```

test/test-readfile.f

```
01: // test-readfile.f
02: // Author: Matthew Chan
03: string filename = 'myfile.txt';
04: create(filename);
05: file f = fopen(filename);
06:
07: appendString(filename, 'hola mundo!');
08: prints(readLine(f));
```

```
09: delete(filename);
```

test/test-rmdir.f

```
1: // test-rmdir.f
2: // Author: Matthew Chan
3: rmdir('removeme');
```

test/test-string.f

```
1: // test-string.f
2: // Author: Matthew Chan
3: 'hi';
4: string s = 'there';
```

test/test-void.f

```
1: // test-void.f
2: // Author: Matthew Chan
3: def voidfunc()
4: {
5:       prints('Functions should not declare void return type
explicity');
6: }
```

9.9 Demo

demo/demo.f

```
01: // demo.f
02: // Author: Matthew Chan
03: // Demo program that shows off Fli/o
04:
05:
06: def addLineNumbers(string filename)
07: {
08:       file f = fopen(filename);
09:
10:       // Create a copy of the current file
11:       string copyName = concat('lined_', filename);
12:       copy(filename, copyName);
13:
14:       file newFile = fopen(copyName);
15:
```

```
16:       string line = readLine(f);
17:
18:       // Keep track of which line we are on
19:       int lineNo = 0;
20:       string prefix;
21:
22:       // Loop through all of the lines in file f
23:       for(; strcmp(line, '') != 0;;) {
24:               prefix = concat('[', intToStr(lineNo));
25:               prefix = concat(prefix, '] ');
26:
27:               // Write the lined version to the new file
28:               write(newFile, concat(prefix, line));
29:               line = readLine(f);
30:               lineNo = lineNo + 1;
31:       }
32:
33: }
34:
35: addLineNumbers('sample.txt');
```

demo/rundemo.sh

```
01: # demo.f
02: # Author: Matthew Chan
03: # Build the project
04: cd ../src
05: make
06:
07: cd -
08:
09: # Compile and run the demo script
10: ../src/flio.native < demo.f > demo.ll
11: llc demo.ll
12: clang demo.s ../src/stdlib.c -o demo
13: ./demo
14:
15: rm demo.ll demo.s demo
```

9.10 Miscellaneous

README.md

```
01: # Fli/o
02: A programming language developed to create a seamless way for
users to interact with files.
03:
04: # Installation
05:
06: ## Prerequisites
07:
08: LLVM, OCaml and cc are required to build the compiler for Fli/o.
09:
10: ## Steps
11:
12: Run `make` in the /src directory. Then, to compile a program named
`myprogram.f`, run `flio.native < myprogram.f > myprogram.s` to build
the LLVM IR.
13:
14: Then run `llc myprogram.ll` to build the Assembly code. Finally,
run `clang myprogram.s stdlib.c -o myprogram` to link the Assembly
code with Fli/o standard library and create an executable named
`myprogram`.
15:
16: Note, you can use other C compilers instead of clang here. (Ex:
gcc, cc)
17:
18: # Testing
19:
20: To run all tests, run `bash testall.sh`.
```

src/Makefile

```
01: # Makefile
02: # Author: Matthew Chan
03: $CC = gcc
04:
05: .PHONY : all
06: all : clean flio.native
07:
08: .PHONY : flio.native
09: flio.native :
10:   ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags
-w,+a-4 \
11:        flio.native
12:   $(CC) -c stdlib.c
```

```
13:
14: .PHONY: clean
15: clean:
16:   ocamlbuild -clean
17:   rm -rf scanner.ml parser.ml parser.mlii flio
18:   rm -rf *.cmx *.cmi *.cmo *.cmx *.o
19:   rm -rf testall.log *.diff *.err *.ll
20:   rm -rf *.exe *s *.out flio.tar.gz
21:
22: TESTS = binop concat copyfile createfile decl2 fcall fcall2 fdecl
fdecl2 \
23:   for hello hello2 if logic movefile number readfile rmdir string
void
24:
25: FAILS = assign fdecl fdecl2 fdecl3 for hello if return scope types
26:
27: TESTFILES = $(TESTS:%=test-%.f) $(FAILS:%=fail-%.f)
28:
29: SOURCEFILES = Makefile scanner.mll parser.mly ast.ml sast.ml
semant.ml \
30:         testall.sh codegen.ml flio.ml
31:
32: DEMOFILES = rundemo.sh sample.txt demo.f
33:
34: TARFILES = ../README.md $(SOURCEFILES) $(TESTFILES:%=../test/%)
$(DEMOFILES:%=../demo/%)
35:
36: OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx
flio.cmx
37:
38: flio : $(OBJS)
39:   ocamlfind ocamlopt -linkpkg -package llvm -package
40:   llvm.analysis $(OBJS) -o flio
41:
42: scanner.ml : scanner.mll
43:   ocamllex scanner.mll
44:
45: parser.ml parser.mli : parser.mly
46:   ocamlyacc parser.mly
47:
48: %.cmo : %.ml
49:   ocamlc -c $<
50:
```

```
51: %.cmi : %.mli
52:   ocamlc -c $<
53: %.cmx: %.ml
54:   ocamlfind ocamlopt -c -package llvm $<
55:
56: flio.tar.gz : $(TARFILES)
57:   cd .. && tar zcf src/flio.tar.gz $(TARFILES:%=src/%)
58:
59: #.PHONY : clean
60: #clean :
61: # rm -f flio parser.ml parser.mli scanner.ml *.cmo *.cmi
62:
63: # Generated by ocamldep *.ml *.mli
64: ast.cmo:
65: ast.cmx:
66: codegen.cmo: ast.cmo
67: codegen.cmx: ast.cmx
68: flio.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
69: flio.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
70: parser.cmo: ast.cmo parser.cmi
71: parser.cmx: ast.cmx parser.cmi
72: scanner.cmo: parser.cmi
73: scanner.cmx: parser.cmx
74: semant.cmo : ast.cmo
75: semant.cmx : ast.cmx
76: parser.cmi: ast.cmo
```

9.11 Git Log

```
commit ed6786a22061339f9cb2068e67faf6f464ad28e8
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 19 17:36:39 2018 -0500

    Add stdlib to Makefile

commit cf81da910ce0d60fe6a940ea0e1f22517f6e49f3
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 19 15:46:45 2018 -0500

    Update testall.sh
```

```
commit 522ed6244b666b00b4119a805dbf67292365bafa
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 19 15:37:49 2018 -0500

    Update Makefile

commit 8ea729dea57b20b6ebf5df6e4d4de64c6dac7167
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 19 15:13:48 2018 -0500

    Fix demo script

commit 0503bc23876768332d73f3c02841d67332d827dd
Merge: fb89205 2b89995
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 19 15:11:23 2018 -0500

    Resolve merge conflict

commit 2b89995247b6602d0d266c07af240d6ae7218801
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 19 15:10:27 2018 -0500

    Update README.md

commit fb892054806adab20a9b7d1fd5d43a0558dde820
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Mon Dec 17 00:12:22 2018 -0500

    Add cleanup commands

commit 8461e267166a378525e6cafcaf3a4f4043304fdd
Merge: 8921276 caf87ef
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Dec 16 23:14:51 2018 -0500

    Merge pull request #8 from matthewachan/development

    Prepare for demo

commit caf87ef5ad51564c4e639c11495be82b39b2dc47
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sun Dec 16 23:14:14 2018 -0500
```

```
    Remove extraneous files

commit 00d4fe4aaa98b5e1082ad1fed993bbc3c7cb3df1
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sun Dec 16 23:11:23 2018 -0500

    Add strcmp

commit f73bcf1673a34e42351a49338eb0948c6316aaec
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sun Dec 16 23:06:43 2018 -0500

    Add demo program

commit 7d348791f054cd544acd86f13943b29e0ad52427
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sun Dec 16 23:06:36 2018 -0500

    Add additional builtin functions and fix tests

commit 8921276ac3a6d49284ccb853cb6d93c2a15b50a4
Merge: e6d7c28 faa91cd
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 19:28:30 2018 -0500

    Merge pull request #7 from matthewachan/development

    Merge stable version of development branch into master

commit faa91cd20db64a53de09bd06d37339e61d44c74b
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 19:25:56 2018 -0500

    Clean up codebase

commit 11f49f09bc3994b649420262a9b26cc9d0449876
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 18:20:34 2018 -0500

    Update test suite

commit d151dc647f9390004405c0a765c64c3dcf73faaf
```

```
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 17:41:02 2018 -0500

    Add tests

commit 755e249755bdb249e3962ad1fa94a59c646c5e48
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 17:40:55 2018 -0500

    Add concat builtin function

commit 3fde8b23cfd78eba6ee01ca944a16632e16f4419
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 17:17:38 2018 -0500

    Remove arrays and proc type

commit 7c05c5bedeafd54150295e9f9cc59c241f9db69f
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 16:40:15 2018 -0500

    Remove pipe operator

commit e22fd25eaf8bb107019639ae613a97f027b1aae4
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 16:36:02 2018 -0500

    Remove foreach

commit 73aea60767bf5ae4c428f0de91eb8e98d12569ab
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 16:35:08 2018 -0500

    Remove dot operator

commit e240250bb3f319f77e4bd006ee7380bcd675be4a
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 16:34:20 2018 -0500

    Remove elif

commit 7b7d16af828bac5e7fc710c533cbeca6e565b9a6
Author: Matt Chan <matthew.a.chan@gmail.com>
```

```
Date:   Sat Dec 15 16:29:32 2018 -0500

    Remove imports

commit 2f53328312529a3169f8aa596e9cb91794fe19b2
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Sat Dec 15 16:26:54 2018 -0500

    Saving changes before cleanup

commit a75123b41ead36bec7e0556f73be9f99c471048f
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Thu Dec 13 16:46:41 2018 -0500

    Revert "Testing..."

    This reverts commit c32b178de000a849daeb16f6720533aea8481b0e.

commit c32b178de000a849daeb16f6720533aea8481b0e
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Thu Dec 13 16:27:44 2018 -0500

    Testing...

commit 38c4ed7df870f35030cee3a5fd8278ebb609d019
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Thu Dec 13 15:57:43 2018 -0500

    Implement proc type

commit fef60237ea1c15e1aa6b7a15eca2a7185467b286
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Thu Dec 13 15:32:55 2018 -0500

    Add array literals

commit b28e821bc9308e65b649dbcd5d252686f2350dcd
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Thu Dec 13 13:41:28 2018 -0500

    Finish with builtin functions

commit 6cefdefd1ce88e24049c9ad4bcc20bf6076566c7
```

```
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 21:20:52 2018 -0500

    Fix file builtins and Makefile

commit 247bdb7f0b3eea279bb1e8a5667f43b28b2f602f
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 20:45:42 2018 -0500

    Implement copy function

commit 562da89fefe45f752c9e70aeeacb749abebc98d1
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 20:12:51 2018 -0500

    Start building the standard library

commit 6369b6ea968d1ae499a61904813d5ffaf177e841
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 19:45:07 2018 -0500

    Add open builtin function for files

commit 676574232ebe79dca37f37a326461f6063d90f3c
Merge: 75ccc63 bb85ca4
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 19:19:25 2018 -0500

    Resolve merge conflicts

commit 75ccc63d704b20c3effd99a8ab0051735adebf4a
Merge: 00dab6f 50247b5
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 19:16:44 2018 -0500

    Merge from mchan

commit 50247b58749b7641e250e9e2bef0bf47db96d556
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 19:06:54 2018 -0500

    Fix for loop initialization
```

```
commit 59b38bd5f94c7009ecbed0bf683d19851a39f0ea
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 18:02:05 2018 -0500

    Added return statements

commit 0cc59dd4349e1ceab642d6d3c5dfc29cc697093d
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 17:55:36 2018 -0500

    Implemented functions

commit 7e46d8f214f2bf34a7ed783ecb1c8e1c69f355ab
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 15:12:17 2018 -0500

    Implement for loop

commit f3064b7652dad6ede9d855d0048bd0f53224729c
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 14:12:45 2018 -0500

    Fix add terminal logic for main

commit b5090535dedf287eaacde3395d75566ff209b9d1
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 13:42:10 2018 -0500

    Implement binary operators

commit 56a8023b78c92f370379ce3d75eea53e7d26ba32
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 13:27:55 2018 -0500

    Fixed main variable declarations

commit 9da82ac53298cd09f2f7bd4af0e222b83f22355f
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 12:26:39 2018 -0500

    Added StringMap for Ids

commit bb85ca47c689d47bc69cfeb3e5f300c2caa4b74e
```

```
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Wed Dec 12 11:52:50 2018 -0500

    Codegen forelif

commit ed306d9e4d3d14087dd2fbdbf3ad0f4c6480fb2a
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 10:39:47 2018 -0500

    Implement string literals

commit fa67886215d7a9c735df80736a7bf46099e8a434
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Dec 12 10:39:14 2018 -0500

    Enable semantic checking (again)

commit 00dab6f7f29199e3e9fd9eab705cafbd1dea808d
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Dec 11 19:14:24 2018 -0500

    Builds all functions and everything in each function

commit 522d1a03ad630241ccb376883a45263b6f999732
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Fri Dec 7 20:00:59 2018 -0500

    Working copy, codegen still needs changes to fully build

commit a2e809713f1b247f82c1657702c71f2d97c15da7
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sat Nov 24 02:04:35 2018 -0500

    Defined variable types, globals, and functions, filled out some
cases for statements and expressions. See commented out code around
66-76 for some problems with filling out functions, and the for case
in the pattern matching

commit e6d7c28fc23d97c71eec1252098cb98cb6c36365
Merge: d88c122 5951499
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Nov 19 22:38:35 2018 -0500
```

```
    Update Makefile

    Merging from development branch

commit 5951499b62711ebd9fe9f6c5ee881d4789f80919
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 16:03:29 2018 -0500

    Update Makefile to add required files to tarball

commit d88c122ac23c1a9d7ca89f7c47945d6e960eccb3
Merge: 247c58a 601db48
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 15:44:51 2018 -0500

    Merge pull request #5 from matthewachan/development

    Fix testall.sh and regression tests

commit 601db48113f8be63dd9932ba7f701bc966a8c00f
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 15:43:45 2018 -0500

    Remove printbig from Makefile

commit fbc0e18d997b67381fe8b7cecfd89f313b7938cd
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 15:40:37 2018 -0500

    Update README test information

commit 77edd244d5fd21133f205055e01dd5b6acb1da56
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 15:38:18 2018 -0500

    Fixed testall script and test suite

commit 247c58adef9d69a99eac2e579ff6b392c852159a
Merge: 39ecfc9 a29d211
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 13:43:48 2018 -0500

    Merge pull request #4 from matthewachan/development
```

```
    Adding in hello world test program

commit a29d2119c7446abfc2c760e856920c2f8a6ae1ca
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 13:42:57 2018 -0500

    Adding in hello world test program

commit 39ecfc94f4931da11c0409fd59f3026cdb577794
Merge: b5163d6 b8aaf5d
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 13:41:31 2018 -0500

    Restructure directory for hello world submission

commit b8aaf5df5ca803e47087b4504e16f8c213fab3f8
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 13:38:18 2018 -0500

    Restructure directory for hello world submission

commit b5163d696ab94f8de502f2743273337a74850118
Merge: 4c548a5 c5c1cab
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 13:32:13 2018 -0500

    Merging working codegen build

    codegen.ml has been fixed and successfully compiles a test
hello_world program.

commit c5c1cab342425c2a51ebfaf11b3012f8d5708ea0
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 13:25:09 2018 -0500

    Fixed codegen and Makefile (working build)

commit a2f54e6b71a9371b427772da62a5037358b5e306
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Wed Nov 14 09:20:56 2018 -0500

    Renamed ast.ml to ast.mli fixed codegen errors
```

```
commit 4c548a5b4a468e5197caefea84eec0c2d7ea9102
Merge: 7cc45df 79fb551
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 02:18:47 2018 -0500

    Merge pull request #1 from matthewachan/development

    Enable pretty printing

commit 79fb55170a263cc4e092e3a37b7e10200b532399
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 02:17:32 2018 -0500

    Enable pretty printing

commit 7cc45df582c44e3b35bf8c25bd6d8b603146bb95
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 02:03:14 2018 -0500

    Add pretty printing to semant exception messages

commit 2dff75f262c671925a9ddd1832769022b5b5d20d
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 01:49:13 2018 -0500

    Pretty printing tested and working

commit dd86fb23adee60473cb32ca0ace3151256d006e7
Merge: 06cfcd0 f85b343
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 01:20:21 2018 -0500

    Working semant build

commit 06cfcd06fa83570f96651862fea520dc1911193d
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Wed Nov 14 01:06:16 2018 -0500

    Revert "parser fix; Pretty Printer almost complete"

    This reverts commit 4b37d148fb4bc98fbeba6975cd3614fbc553d4ee.
```

```
commit f85b343605f9a98c0078e6279a6b345c72b788a3
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 23:52:45 2018 -0500

    Made changes to codegen/top level would work

commit 2074ffb0d810f1679c4b67826abb0b9dd1f72c1f
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 20:43:00 2018 -0500

    Getting codegen to compile

commit ec8bacb07f3caa574dd274626f3b07094e645520
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 19:28:10 2018 -0500

    Removing local variables

commit 826e503761e5eeb1348ad22e5ef6be3e78c30491
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 19:24:35 2018 -0500

    Revert "Pretty Printer Functions/AST mostly done"

    This reverts commit 08f8000fc84274b61cccd3f929ddb9a9b62fc566.

commit 66b96c5de8c165e3dbde6ac87b51e52cfab6f02c
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 19:23:23 2018 -0500

    reverted changes

commit c1dd074ab05b31162f6f50338972e09d0750cbe6
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 19:22:06 2018 -0500

    Revert "parser fix; Pretty Printer almost complete"

    This reverts commit 4b37d148fb4bc98fbeba6975cd3614fbc553d4ee.

commit 0844295d8057bfaa7700c2f81bcebf7cd9cea379
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 19:21:53 2018 -0500
```

```
    Revert "adding back locals"

    This reverts commit 3ba6cbda85143641e080ba28f03ecf84de06a080.

commit 7f766afc2ba2754825b7bf2fae2825565e9c3e8e
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 19:20:48 2018 -0500

    reverting pretty printer changes

commit 3ba6cbda85143641e080ba28f03ecf84de06a080
Author: Talenel <jg3544@columbia.edu>
Date:   Tue Nov 13 07:53:50 2018 -0500

    adding back locals

commit 4b37d148fb4bc98fbeba6975cd3614fbc553d4ee
Author: Talenel <jg3544@columbia.edu>
Date:   Tue Nov 13 07:49:24 2018 -0500

    parser fix; Pretty Printer almost complete

commit b85688f40d0ed1a12c11742dc2076aea02cb85fc
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 03:36:53 2018 -0500

    Updated files to allow for codegen

commit 9b55073e29e80ad7ff885243f3a965a4ada2be38
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Nov 13 00:28:29 2018 -0500

    Fixing codegen syntax errors

commit 284b17b470f88a02c1f3eb5d3c76790282d10711
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Mon Nov 12 23:56:04 2018 -0500

    renamed codegen.ml

commit 0f8fe899d4ea042bc6f1f2b108c2c9745298b9f4
Author: Eyob <eyobtefera26@Yahoo.com>
```

```
Date:   Mon Nov 12 23:50:15 2018 -0500

    fixed Makefile error

commit 08f8000fc84274b61cccd3f929ddb9a9b62fc566
Author: Talenel <jg3544@columbia.edu>
Date:   Mon Nov 12 21:51:58 2018 -0500

    Pretty Printer Functions/AST mostly done

commit 6ffc78898de0318d95b51d5606f3852dac3cf967
Author: Gideon Rono <gideonrono@dyn-160-39-160-232.dyn.columbia.edu>
Date:   Mon Nov 12 15:09:49 2018 -0500

    test file changes

commit cb4649c68621c0849e383b806c36d9a8f4b31aa9
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Mon Nov 12 14:01:18 2018 -0500

    Add checking for global statement list

commit 54417a9cea2febe992e7cb07ec55ad99610175b4
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Mon Nov 12 14:01:01 2018 -0500

    Reverse global statement list order (was reversed)

commit 745a5ad2c728c451d2dd28df36aed9a349092e37
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Mon Nov 12 13:23:14 2018 -0500

    Check functions mostly completed

commit fd64d4448ec16835d9ad87970ee1e4b68c5d85e8
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Mon Nov 12 13:18:29 2018 -0500

    Minor changes to AST and parser structure

commit fc6db9061f5aed109f06a10a20c2e96da274db56
Merge: 444d650 a0945db
Author: Matt Chan <matthew.a.chan@gmail.com>
```

```
Date:   Mon Nov 12 13:17:30 2018 -0500

    Merge branch 'master' of github.com:matthewachan/flio

commit 444d650c37b2f7138d5f7c2ea59ad81f66bf1d8c
Author: Matt Chan <matthew.a.chan@gmail.com>
Date:   Mon Nov 12 13:16:17 2018 -0500

    Semantic checker almost done for functions

commit a0945db4aedb1690bb427e1ba7c8cd92c69b56df
Author: Gideon Rono <gideonrono@Gideons-MacBook-Pro.local>
Date:   Mon Nov 12 01:43:22 2018 -0500

    testall update

commit d1dbf5474b8a44f1e5fa3dc76f10a33f272cc438
Author: Gideon Rono <gideonrono@Gideons-MacBook-Pro.local>
Date:   Mon Nov 12 01:40:35 2018 -0500

    testall

commit 730e764c9b022c824bbe2f96816113a9f2e04cc1
Merge: 32b4368 9f217fe
Author: Gideon Rono <gideonrono@dyn-160-39-160-232.dyn.columbia.edu>
Date:   Mon Nov 12 00:08:10 2018 -0500

    updates
    Merge branch 'master' of https://github.com/matthewachan/flio

commit 32b4368ba6f2ad1e9ab6d72fb0ac49e5b532da89
Author: Gideon Rono <gideonrono@dyn-160-39-160-232.dyn.columbia.edu>
Date:   Mon Nov 12 00:07:40 2018 -0500

    new test files

commit f9794f5d72b1a181d2182c41dc7805e3639a6e9b
Author: Gideon Rono <gideonrono@dyn-160-39-160-232.dyn.columbia.edu>
Date:   Sun Nov 11 23:59:04 2018 -0500

    test_suite changes

commit 9f217fe616493ad86e9921d4c3e60ac4e95bc218
```

```
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sun Nov 11 22:56:04 2018 -0500

    updated makefile to integrate codegen

commit 723a673023cb167c6306505d47ecba95555e5e9e
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sun Nov 11 21:09:28 2018 -0500

    created toplevel for compilation

commit 9521948b49e33671eebf581ab0552ea132967db3
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sun Nov 11 20:49:40 2018 -0500

    Rough outline of codegen, some statements and expression types
still need handling

commit 85838e9dfaee6fb7f8fcc11c76738d95b6cd901e
Merge: 0e6fd02 ffc4061
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sun Nov 11 19:23:32 2018 -0500

    Filled our statements and expression builder
    Merge branch 'master' of https://github.com/matthewachan/flio

commit 0e6fd02e1d7b2583d0c8c44950b6370da3974950
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sun Nov 11 19:23:26 2018 -0500

    cleaned up statement

commit ffc40615b5066f5b3b473dcf895c8424898e9ccd
Author: Gideon Rono <gideonrono@dyn-160-39-160-232.dyn.columbia.edu>
Date:   Sun Nov 11 16:30:15 2018 -0500

    Fli-o

commit 604faaeb01216b371a0f831f148369b481d452bf
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Fri Nov 9 22:08:59 2018 -0500

    added basic expression to the builder
```

```
commit b4a2b7a7d3692aae1cbb3257cae920bca6365ef8
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Fri Nov 9 20:28:43 2018 -0500

    Local/global variable declarations

commit 2686781611d8e00e5d7262ef16f3e4a1c6621a94
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Fri Nov 9 18:59:39 2018 -0500

    Refined the types by introducing the use of structs instead of
pointers

commit 1a87dbdd31410fcc1b84758a3efccd3a1501ecb7
Merge: 60ef2d1 101c075
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Sun Nov 4 19:25:48 2018 -0500

    Merge branch 'master' of https://github.com/matthewachan/flio

commit 101c075e502511fce9e55bf6e2585a71317f517a
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Nov 3 14:31:59 2018 -0400

    Add some pretty printing functions

commit 4e7bd719ecd7960d933f71573945797cd26b4a0b
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Fri Nov 2 10:18:40 2018 -0400

    Add SAST

commit 60ef2d1f106b0e4375c1106498d2f039762bb41a
Author: Eyob <eyobtefera26@Yahoo.com>
Date:   Tue Oct 30 23:50:11 2018 -0400

    beginning of codegen, just included types

commit 3e1d39319eb436518cdda8deebc6aaa773cdb0b8
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Oct 15 22:37:07 2018 -0400
```

```
    Update tests

commit 5e31f66ad017a14f33a2a0d2b6fe4a336ccfcfc1
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Oct 15 22:36:23 2018 -0400

    Add parenthesis expr

commit 40722ac8adff37152d5d7f6392293e6bd97c32df
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Oct 15 21:48:09 2018 -0400

    Make piping a statement rather than an expression

commit 73b97eb4799673997ee02f3f1dc172b865640342
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Oct 15 21:05:05 2018 -0400

    Add imports

commit 520d7776ecbdb5d4e565c4235ae77539607d33d1
Merge: 287b0c4 1b3dba2
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Oct 15 20:15:07 2018 -0400

    Merge branch 'master' of github.com:matthewachan/flio

commit 287b0c47def8eaa8d8828e531dce0fbb5d2a6c7b
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Mon Oct 15 20:14:18 2018 -0400

    Make assignment a statement rather than an expression

commit 1b3dba21c0f8b50e799b6c2fbe89a0ee3094b391
Author: Gideon Rono <gideonrono@dyn-160-39-160-98.dyn.columbia.edu>
Date:   Mon Oct 15 19:27:16 2018 -0400

    second test file, makefile update

commit cc08f8f2586afe351d598c7db3055a81be12bec4
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 02:38:06 2018 -0400
```

```
    Add test files

commit 9201fe4311b19edcec3dca13424b5663e8e8e2ad
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 02:34:11 2018 -0400

    Comment out negative unary operator

commit 9b2e0f6193de479ea9a3e68f6a100f5bb734560f
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 02:28:59 2018 -0400

    Update failure and success messages

commit e89b2aeb8c29e4490913f8d0a636b164edae90e4
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 02:17:59 2018 -0400

    Add comments

commit 2019dccd4825e261532c6c26ccafc1d1e5b2b5b9
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 02:15:11 2018 -0400

    Add elif

commit 838900ed4e07143039f3c0b4b53b21332b331596
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 01:51:02 2018 -0400

    Add pipe operator

commit dddb74f621d082a7f5b465e9b1c72371260e35be
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 01:43:35 2018 -0400

    Add dot operator

commit 13cefc5a32ee0b904540a6ef7088144f8b306696
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 01:25:57 2018 -0400

    Remove old files
```

```
commit 11850aa1ee965b44f6a6b98c386f6fb72b482a16
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 01:23:08 2018 -0400

    Allow void function definitions and add comments

commit 9d693c6d9278aa018191ee1b1630d1234ad05ef5
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 01:22:23 2018 -0400

    Restructure directory

commit d3b05a630df8721f21e83b545c5ccd7c032073dc
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 00:57:49 2018 -0400

    Add array accessing and initialization

commit 1ad05fe9b8dec51956ffdd3658638b46cc6cc4ba
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 00:42:16 2018 -0400

    Add function calls

commit 45dfc5aa7f9adf950d9a88feb016ae49830def1d
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sun Oct 14 00:33:09 2018 -0400

    Add function definitions

commit 0e628d2546ec6ed78bf62ee7760a4ddf8432166c
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 23:16:31 2018 -0400

    Add foreach loop

commit d748fa0cd8c891ec05ca442dca652a9ca6215e30
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 23:11:05 2018 -0400

    Add additional operators
```

```
commit de6c0975b530be54f695328a3028e4bdd733d973
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 22:49:58 2018 -0400

    Add array declaration

commit c6fd042e696575da2409c69a4dfe6f6341d1536f
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 22:43:44 2018 -0400

    Add file and dir types

commit 957c4df62d0964072e1c2a3ccb168a1c13eca9b4
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 22:25:34 2018 -0400

    Add if/else statements

commit 610b86af0abd2e494bbe2a17087ede2870b5f6d8
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 22:07:15 2018 -0400

    Add logical operators, for loops (semi-colon sequencing)

commit 56d337f63354f7d5debd944072f97b2855fabd07
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Sat Oct 13 14:21:09 2018 -0400

    Establish working base

commit 9317ac2c72821b80446aca5b31e0b09348914d83
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Fri Oct 12 15:11:11 2018 -0400

    Update scanner with int, string and var

commit 9b83b6cbb912185717576e9c5403da5f30720b32
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Fri Oct 12 14:58:57 2018 -0400

    Add primitive tokens to scanner

commit b9914d3c3f8c01ed92a7dec5fea312d69f7fcd06
```

```
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Fri Oct 12 14:35:49 2018 -0400

    Init base files

commit b4e4d4630b284b085885bc964ae555ba598b9ba3
Author: Matthew Chan <matthew.a.chan@gmail.com>
Date:   Fri Oct 12 13:55:27 2018 -0400

    Initial commit
```