

# The FFBB Programming Language

## Final Project Report

Bowen Chen, Joseph Yang, Jianan Yao, Xiaosheng Chen

{bc2916, zy2431, jy3022, xc2561}@columbia.edu

April 25, 2021

## Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Overview . . . . .	9
1.2	Features . . . . .	10
<b>2</b>	<b>Language Tutorial</b>	<b>10</b>
2.1	Environment Setup . . . . .	10
2.1.1	Docker . . . . .	10
2.2	Compiling . . . . .	10
2.3	Sample FFBB Program . . . . .	11
<b>3</b>	<b>Language Reference Manual</b>	<b>12</b>
3.1	Lexical conventions . . . . .	12
3.1.1	Comments . . . . .	12
3.1.2	Multi-line Comments . . . . .	12
3.1.3	Identifiers . . . . .	12
3.1.4	Keywords . . . . .	12
3.1.5	Integer constants . . . . .	12
3.1.6	Floating constants . . . . .	12
3.1.7	String constants . . . . .	13

3.2	Syntax notation . . . . .	13
3.3	What's in a Name? . . . . .	13
3.4	Expressions . . . . .	14
3.4.1	Primary expressions . . . . .	14
3.4.2	identifier . . . . .	14
3.4.3	constant . . . . .	14
3.4.4	expression . . . . .	14
3.4.5	expression * expression . . . . .	14
3.4.6	expression / expression . . . . .	14
3.4.7	expression + expression . . . . .	15
3.4.8	expression - expression . . . . .	15
3.5	Relational operators . . . . .	15
3.5.1	expression<expression . . . . .	15
3.5.2	expression>expression . . . . .	15
3.5.3	expression<=expression . . . . .	15
3.5.4	expression>=expression . . . . .	15
3.6	Equality operators . . . . .	15
3.6.1	expression==expression . . . . .	15
3.6.2	expression!=expression . . . . .	15
3.7	Other operators . . . . .	15
3.7.1	expression && expression . . . . .	15
3.7.2	expression    expression . . . . .	16
3.7.3	!expression . . . . .	16
3.7.4	identifier++ . . . . .	16
3.7.5	identifier-- . . . . .	16
3.8	Declarations . . . . .	16
3.9	Type specifiers . . . . .	16
3.9.1	Return specifiers . . . . .	17
3.9.2	Example . . . . .	17
3.10	Statements . . . . .	18
3.10.1	Expression statement . . . . .	18

3.10.2	Compound statement . . . . .	18
3.11	Conditional statement . . . . .	18
3.12	While statement . . . . .	19
3.13	For statement . . . . .	19
3.13.1	For-in statement . . . . .	19
3.13.2	Return statement . . . . .	20
3.14	Functions . . . . .	20
3.14.1	General function form . . . . .	20
3.14.2	Higher order function . . . . .	21
3.14.3	func type declaration . . . . .	21
3.14.4	Definition 1 for higher order function . . . . .	22
3.14.5	Definition 2 for higher order function . . . . .	22
3.15	List . . . . .	22
3.15.1	List Declarations . . . . .	23
3.15.2	Get value at index . . . . .	23
3.15.3	Set value at index . . . . .	23
3.15.4	List Slicing . . . . .	23
3.15.5	Other Built in List functions . . . . .	24
3.16	String . . . . .	24
3.16.1	String operations . . . . .	24
3.17	Dict . . . . .	25
3.17.1	Dict Declaration . . . . .	25
3.17.2	Dict Operations . . . . .	25
3.18	Set . . . . .	27
3.18.1	Set Declaration . . . . .	27
3.18.2	Set Operations . . . . .	27
3.19	Lexical scoperules . . . . .	28
3.20	Constant expressions . . . . .	28
<b>4</b>	<b>Project Plan</b>	<b>28</b>
4.1	Process . . . . .	28
4.2	Style Guide . . . . .	29

4.3	Project Timeline . . . . .	30
4.4	Team Roles . . . . .	30
4.5	Tools . . . . .	30
<b>5</b>	<b>Architectural Design</b>	<b>31</b>
5.1	Block Diagram . . . . .	31
5.2	Scanner . . . . .	31
5.3	Parser, AST . . . . .	31
5.4	Semantic Checking, SAST . . . . .	31
5.5	Code Generation . . . . .	32
5.6	String Library . . . . .	32
5.7	Dictionary Library . . . . .	32
5.8	Set Library . . . . .	32
5.9	List Library . . . . .	32
<b>6</b>	<b>Test Plan</b>	<b>32</b>
6.1	Testing Workflow . . . . .	32
6.2	Sample Tests . . . . .	33
6.2.1	Fibonacci sequence using List and for...in . . . . .	33
6.2.2	Quick sort . . . . .	33
6.2.3	Higher Order Functions . . . . .	34
6.2.4	Bellman–Ford algorithm . . . . .	36
6.3	Responsibilities . . . . .	39
<b>7</b>	<b>Lessons Learned</b>	<b>39</b>
7.1	Bowen Chen . . . . .	39
7.2	Joseph Yang . . . . .	40
7.3	Jianan Yao . . . . .	40
7.4	Xiaosheng Chen . . . . .	40
<b>8</b>	<b>Appendix</b>	<b>42</b>
<b>A</b>	<b>Source code of FFBB</b>	<b>43</b>
A.1	scanner.mll . . . . .	43

A.2	FFBBparser.mly	44
A.3	ast.ml	48
A.4	semant.ml	51
A.5	sast.ml	61
A.6	codegen.ml	63
A.7	FFBB.ml	78
A.8	stringLibrary.c	79
A.9	list.h	80
A.10	list.c	81
A.11	treebasics.h	84
A.12	treebasics.c	85
A.13	treedict.c	86
A.14	treeset.c	91
<b>B</b>	<b>Test Cases</b>	<b>96</b>
B.1	Positive Tests	96
B.1.1	test-add1.mc	96
B.1.2	test-arith1.mc	96
B.1.3	test-arith2.mc	96
B.1.4	test-arith3.mc	97
B.1.5	test-arith4.mc	97
B.1.6	test-demo1.mc	97
B.1.7	test-demo2.mc	99
B.1.8	test-demo3.mc	99
B.1.9	test-demo4.mc	100
B.1.10	test-demo5.mc	103
B.1.11	test-dict1.mc	106
B.1.12	test-dict2.mc	106
B.1.13	test-dict3.mc	107
B.1.14	test-dict4.mc	107
B.1.15	test-dict5.mc	110
B.1.16	test-dict6.mc	110

B.1.17 test-dict7.mc . . . . .	111
B.1.18 test-dict8.mc . . . . .	111
B.1.19 test-fib.mc . . . . .	112
B.1.20 test-float1.mc . . . . .	112
B.1.21 test-float2.mc . . . . .	113
B.1.22 test-float3.mc . . . . .	113
B.1.23 test-float4.mc . . . . .	114
B.1.24 test-float5.mc . . . . .	115
B.1.25 test-float6.mc . . . . .	116
B.1.26 test-for1.mc . . . . .	116
B.1.27 test-for2.mc . . . . .	117
B.1.28 test-forin1.mc . . . . .	117
B.1.29 test-func1.mc . . . . .	118
B.1.30 test-func2.mc . . . . .	119
B.1.31 test-func3.mc . . . . .	119
B.1.32 test-func4.mc . . . . .	120
B.1.33 test-func5.mc . . . . .	120
B.1.34 test-func6.mc . . . . .	120
B.1.35 test-func7.mc . . . . .	121
B.1.36 test-func8.mc . . . . .	121
B.1.37 test-func9.mc . . . . .	121
B.1.38 test-gcd.mc . . . . .	122
B.1.39 test-gcd2.mc . . . . .	122
B.1.40 test-global1.mc . . . . .	123
B.1.41 test-global2.mc . . . . .	124
B.1.42 test-global3.mc . . . . .	124
B.1.43 test-hello-world.mc . . . . .	124
B.1.44 test-hello.mc . . . . .	125
B.1.45 test-if1.mc . . . . .	125
B.1.46 test-if2.mc . . . . .	125
B.1.47 test-if3.mc . . . . .	126

B.1.48	test-if4.mc . . . . .	126
B.1.49	test-if5.mc . . . . .	126
B.1.50	test-if6.mc . . . . .	127
B.1.51	test-lambda.mc . . . . .	127
B.1.52	test-list1.mc . . . . .	129
B.1.53	test-list2.mc . . . . .	129
B.1.54	test-list3.mc . . . . .	130
B.1.55	test-list4.mc . . . . .	131
B.1.56	test-local1.mc . . . . .	131
B.1.57	test-local2.mc . . . . .	132
B.1.58	test-ops1.mc . . . . .	132
B.1.59	test-ops2.mc . . . . .	133
B.1.60	test-ops3.mc . . . . .	134
B.1.61	test-ops4.mc . . . . .	134
B.1.62	test-printbig.mc . . . . .	135
B.1.63	test-set1.mc . . . . .	137
B.1.64	test-set2.mc . . . . .	138
B.1.65	test-set3.mc . . . . .	138
B.1.66	test-set4.mc . . . . .	139
B.1.67	test-set5.mc . . . . .	140
B.1.68	test-string-concat.mc . . . . .	140
B.1.69	test-string-len.mc . . . . .	141
B.1.70	test-string-slice.mc . . . . .	141
B.1.71	test-string1.mc . . . . .	141
B.1.72	test-string2.mc . . . . .	142
B.1.73	test-var1.mc . . . . .	142
B.1.74	test-var2.mc . . . . .	142
B.1.75	test-var3.mc . . . . .	143
B.1.76	test-while1.mc . . . . .	143
B.1.77	test-while2.mc . . . . .	144
B.2	Negative Tests . . . . .	144

B.2.1	fail-assign1.mc . . . . .	144
B.2.2	fail-assign2.mc . . . . .	145
B.2.3	fail-assign3.mc . . . . .	145
B.2.4	fail-dead1.mc . . . . .	145
B.2.5	fail-dead2.mc . . . . .	146
B.2.6	fail-dict1.mc . . . . .	146
B.2.7	fail-dict2.mc . . . . .	146
B.2.8	fail-dict3.mc . . . . .	147
B.2.9	fail-expr1.mc . . . . .	147
B.2.10	fail-expr2.mc . . . . .	148
B.2.11	fail-expr3.mc . . . . .	148
B.2.12	fail-expr4.mc . . . . .	148
B.2.13	fail-expr5.mc . . . . .	149
B.2.14	fail-float1.mc . . . . .	149
B.2.15	fail-float2.mc . . . . .	150
B.2.16	fail-for1.mc . . . . .	150
B.2.17	fail-for2.mc . . . . .	150
B.2.18	fail-for3.mc . . . . .	151
B.2.19	fail-for4.mc . . . . .	151
B.2.20	fail-for5.mc . . . . .	151
B.2.21	fail-func1.mc . . . . .	152
B.2.22	fail-func2.mc . . . . .	152
B.2.23	fail-func3.mc . . . . .	152
B.2.24	fail-func4.mc . . . . .	153
B.2.25	fail-func5.mc . . . . .	153
B.2.26	fail-func6.mc . . . . .	153
B.2.27	fail-func7.mc . . . . .	154
B.2.28	fail-func8.mc . . . . .	154
B.2.29	fail-func9.mc . . . . .	155
B.2.30	fail-global1.mc . . . . .	155
B.2.31	fail-global2.mc . . . . .	155



B.2.32 fail-if1.mc . . . . .	156
B.2.33 fail-if2.mc . . . . .	156
B.2.34 fail-if3.mc . . . . .	156
B.2.35 fail-list1.mc . . . . .	156
B.2.36 fail-list2.mc . . . . .	157
B.2.37 fail-list3.mc . . . . .	157
B.2.38 fail-nomain.mc . . . . .	158
B.2.39 fail-ops1.mc . . . . .	158
B.2.40 fail-ops2.mc . . . . .	158
B.2.41 fail-ops3.mc . . . . .	158
B.2.42 fail-print.mc . . . . .	158
B.2.43 fail-printb.mc . . . . .	159
B.2.44 fail-printbig.mc . . . . .	159
B.2.45 fail-return1.mc . . . . .	159
B.2.46 fail-return2.mc . . . . .	159
B.2.47 fail-set1.mc . . . . .	160
B.2.48 fail-set2.mc . . . . .	160
B.2.49 fail-while1.mc . . . . .	160
B.2.50 fail-while2.mc . . . . .	161

# 1 Introduction

## 1.1 Overview

The FFBB programming language is an imperative language mainly based on the C programming language, with some other features inspired by Java.

It is a general-purpose programming language and even users with non-technical background will be able to study FFBB easily. FFBB will finish syntax-checking during compile time so that programmers won't waste too much time on syntax problem.

The general syntax and language features would be similar to those of the C programming language, with some other operators and features from Java (e.g. type declaration, comment, void keyword).

Also, FFBB programming language accepts some functional programming features like higher-order and lambda functions. We hope that our language could combine the advantages of C and the flexibility of Python to some extent, with certain acceptable trade-off. Our language is written in OCaml/C and then compiled into LLVM code.

## 1.2 Features

- C-style language design with safe explicit type and easy compilation.
- Support of higher-order and lambda functions.
- Built-in data structures of list, dictionary and set like in Python. Implemented using tree to achieve high efficiency.

## 2 Language Tutorial

This tutorial assumes a certain degree of familiarity with programming languages, compilation and the command-line. We recommend brushing up on basic git commands like `git pull` and with `docker` as well. Use `git clone` to download a local copy of our repository, named PLT-FFBB :

```
git clone https://github.com/jyao15/PLT-FFBB.git
```

### 2.1 Environment Setup

It is highly recommended that you use the Docker container utilized by our development team when compiling FFBB programs.

#### 2.1.1 Docker

Run the following command to start the docker

```
docker run --rm -it -v 'pwd':/home/microc -w=/home/microc columbiasedwards/plt
```

### 2.2 Compiling

At the project root directory run

```
make
```

to build the compiler of our language, FFBB.native. Note that make runs an automated test suite, compiling valid and invalid FFBB programs and outputting whether or not they have succeeded or failed. This is a highly useful tool for anyone working on extending or testing the language, but if your goal is only to write FFBB programs, you can build without testing using

```
make all
```

## 2.3 Sample FFBB Program

Consider this simple FFBB program "fib.mc" for computing  $N^{th}$  Fibonacci number:

```
int main () {
    /* Fibonacci number: Compute Nth value */
    int n = 10;
    List<int> f = [0, 1];
    for i in range(n-2) {
        append(f, f[-1] + f[-2]);
    }
    print(f[-1]);
}
```

Using the following commands to compile and execute the above program:

```
./FFBB.native fib.mc > fib.ll
llc -relocation-model=pic fib.ll > fib.s
cc -o fib.exe fib.s printbig.o treebasics.o treeset.o treedict.o list.o
    stringLibrary.o
./fib.exe
```

Running the above will output "34", as expected.

## 3 Language Reference Manual

### 3.1 Lexical conventions

FFBB will include the following types of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. General blanks will be ignored and FFBB needs at least one blank to separate adjacent identifiers, constants, and certain operator-pairs

#### 3.1.1 Comments

#### 3.1.2 Multi-line Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

#### 3.1.3 Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore “`_`” counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

#### 3.1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

*if, else, for, while, return, int, bool, float, int, void, true, false, func, in, List, Set, Dict, lambda*

#### 3.1.5 Integer constants

An integer constant is a sequence of digits. And integer should not have leading zero.

#### 3.1.6 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing.

### 3.1.7 String constants

A string constant is a sequence of characters surrounded by double quotes ”

## 3.2 Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in **gothic**. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript ”opt,” so that

$$\{expression_{opt}\}$$

would indicate an optional expression in braces.

## 3.3 What’s in a Name?

FFBB bases the interpretation of an identifier upon its *type*. The type determines the meaning of the values found in the identifier’s storage.

There are four fundamental types of objects: strings, integers, floating-point numbers, and booleans.

- Strings (declared, and hereinafter called, **string**) is an immutable data structure that contains a variable-length sequence of characters. Each character can be accessed in constant time through its index.
- Integers (**int**) are represented in 16-bit 2’s complement notation.
- Single precision floating point (**float**) quantities have magnitude in the range approximately  $10^{38}$  or 0; their precision is 24 bits or about seven decimal digits.
- Booleans (**bool**) are represented by **true** or **false**.

Besides the four fundamental types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *lists* of objects of a given type;
- *sets* of objects of a given type;
- *dictionaries* of objects of two given types;

FFBB also supports the concept of function pointer(func) which can be used to store the address of a function that can be called later.

## **3.4 Expressions**

The precedence of expression operators follow the conventions of order of operations. Within each subsection, the operators have the same precedence.

### **3.4.1 Primary expressions**

Primary expressions involve only function calls and group left to right.

### **3.4.2 identifier**

An identifier is one of the most primitive expression, and it will be used to identify an unique object or function in FFBB

### **3.4.3 constant**

Constant is one of the most fundamental expression in FFBB. The value of a constant is fixed and remains the same during the entire execution of the program.

### **3.4.4 expression**

Expression in FFBB are usually linked by different operands. An expression can be a variable, constant or some other expressions.

### **3.4.5 expression \* expression**

The binary \* operator indicates multiplication. If both operands are float/int, the result are float/int; If operands are float and int, the type of results will be float.

### **3.4.6 expression / expression**

The binary / operator indicates division. The same type considerations as for multiplication apply.

### **3.4.7 expression + expression**

The binary + operator indicates addition. If both operands are float/int, the result are float/int; If operands are float and int, the type of results will be float. Other type combinations might be discussed later .

### **3.4.8 expression - expression**

The binary - operator indicates subtraction. The same type considerations as for addition apply.

## **3.5 Relational operators**

### **3.5.1 expression < expression**

### **3.5.2 expression > expression**

### **3.5.3 expression <= expression**

### **3.5.4 expression >= expression**

The relational operators group left-to-right. and all of the operators are following the mathematical conventions: < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to). 1 indicates true while 0 will be considered as false.

## **3.6 Equality operators**

### **3.6.1 expression == expression**

### **3.6.2 expression != expression**

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence.

## **3.7 Other operators**

### **3.7.1 expression && expression**

The && operator returns 1 if both its operands are non-zero, 0 otherwise. The second operand is not evaluated if the first operand is 0.

### 3.7.2 `expression || expression`

The `||` operator returns 1 if either of its operands is non-zero, and 0 otherwise. The second operand is not evaluated if the value of the first operand is non-zero.

### 3.7.3 `!expression`

The result of the logical negation operator `!` is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is `int`. This operator is applicable only to `ints` or `booleans`.

### 3.7.4 `identifier++`

Increment the value of `identifier` by 1. This is applicable only to `ints`.

### 3.7.5 `identifier--`

Decrement the value of `identifier` by -1. This is applicable only to `ints`.

## 3.8 Declarations

In function/variable definitions, declarations are used to specify the interpretation which FFBB gives to each identifier. There are two types of declarations in the FFBB language.

## 3.9 Type specifiers

The type-specifiers are

*type-specifier* :

`bool`

`int`

`float`

`string`

`List<type-specifier>`

`Set<type-specifier>`



Dict<type-specifier, type-specifier>

func<type-list>

These specifiers explicitly define the type of

- function return value, if explicitly used in function declarations. In this case, the defined function must return a value with the specified type. Otherwise compile error will be raised.
- variable stored value, if explicitly used in variable declarations. In this case, the defined variable must always store values with the specified type. If ever try to assign a value with other types to the variable, compile error will be raised.

### 3.9.1 Return specifiers

There is only one return-specifier

*type-specifier* :

void

This return-specifier can only be used in function declarations. Also, it cannot be used together with type specifiers. If this return-specifier is used in function declarations, that means the defined function cannot return anything as output. An analogy in Python would be: the defined function works like a procedure, instead of a function.

### 3.9.2 Example

There are two types of variable declarations in FFBB: simple declaration and declaration with assigned value. Here are the examples:

*type-specifier identifier*;

*type-specifier identifier = expression*;

For example,

---

```
// A function pointer points to
// a function taking an integer and returning void
int i;
float j = 0.1;
```

---

## 3.10 Statements

Except as indicated, statements are executed in sequence.

### 3.10.1 Expression statement

Most statements are expression statements, which have the form

*expression*;

Usually expression statements are assignments or function calls.

### 3.10.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

*compound-statement* :

{ *statement-list* }

*statement-statement* :

*statement*

*statement statement-list*

## 3.11 Conditional statement

The two forms of the conditional statement are

`if ( expression ) statement`

`if ( expression ) statement else statement`

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed.

In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered `if`.

## 3.12 While statement

The while statement has the form

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero.

The test takes place before each execution of the statement.

## 3.13 For statement

The for statement has the form

```
for ( expression-1opt; expression-2opt; expression-3opt; ) statement
```

This statement is equivalent to

```
expression-1;  
  
while ( expression-2 ){  
  
    statement;  
  
    expression-3;  
  
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied **while** clause equivalent to "while(1)"; other missing expressions are simply dropped from the expansion above.

### 3.13.1 For-in statement

The for-in statement has the form

```
for identifier-1 in identifier-2 statement
```

This statement is equivalent to

```

int i = 0;

while (i < len(list)){

    identifier-1 = identifier-2[i];

    statement;

    i++;

}

```

Thus the *identifier-1* is the iterator of the list *identifier-2*.

Note the type of *identifier-1* is not needed, it will be inferenced at compile time.

### 3.13.2 Return statement

A function returns to its caller by means of the **return** statement, which has one of the forms

```

return ;

return ( expression );

```

## 3.14 Functions

### 3.14.1 General function form

Function definitions have the form

*function-definition* :

*type-specifier function-declarator function-body*

*function-declarator* :

*declarator ( parameter-list<sub>opt</sub> )*

*parameter-list* :

*type-specifier*

*type-specifier, parameter-list*

The function-body has the form

$$\begin{aligned} & \textit{function-body} : \\ & \qquad \qquad \qquad \textit{function-statement} \end{aligned}$$

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here. The function-statement is just a compound statement which may have declarations at the start.

$$\begin{aligned} & \textit{function-statement} : \\ & \qquad \qquad \qquad \{ \textit{statement-list} \} \end{aligned}$$

### 3.14.2 Higher order function

Besides from the normal format of function, FFBB also supports higher order function. It can take one or more functions as arguments and return a function as its result. And there are two representations available for its definitions.

### 3.14.3 func type declaration

The declaration of func type will be:

$$\text{func} < \textit{type-list} >$$

The type-list is similar with parameter-list. However, the first element in the type-list will be used as higher order function's return type while the following elements will still be the formal argument types of function.

$$\begin{aligned} & \textit{type-list} : \\ & \qquad \qquad \qquad \textit{type-specifier} \\ & \qquad \qquad \qquad \textit{type-specifier}, \textit{type-list} \end{aligned}$$

For example,

---

```
// A function pointer points to
// a function taking an integer and returning void
func<void, int> printbig_ptr;
```

---

### 3.14.4 Definition 1 for higher order function

There are two definitions available for FFBB. The first representation is similar with normal function.

*type-specifier lambda parameter-list<sub>opt</sub> function-body*

Since it is the anonymous function, lambda keyword is mandatory inside the definition and *type-specifier* will be used to indicate the return type of this anonymous function.

The lambda function can be stored in a function pointer with type `func`. For example,

---

```
func<int, int> mul2 = int lambda int x { return x * 2; };  
print(mul2(1)); // 2
```

---

### 3.14.5 Definition 2 for higher order function

The second approach to define higher order function is the shortcut of the previous definition. The format will follow:

*type-specifier lambda parameter-list<sub>opt</sub> → expr*

The evaluated value of *expr* will be the return value of the lambda function.

For example,

---

```
bool lambda int x -> x > 2;  
// is equivalent to  
bool lambda int x {  
    return x > 2;  
};  
// store lambda function in a function pointer  
func<int, int> mul2 = int lambda int x -> x * 2;
```

---

## 3.15 List

A list is an ordered collection of elements of with same type.

### 3.15.1 List Declarations

List can be initialized without value.

$$\text{List} < \textit{type-specifier} > \textit{identifier};$$

List can also be initialized with a list expressions.

$$\text{List} < \textit{type-specifier} > \textit{identifier} = [\textit{expr-list}];$$

For example,

---

```
List<int> list1; // an empty list
List<int> list2 = [1, 2, 3, 4, 5] // a list with 5 integer elements;
```

---

### 3.15.2 Get value at index

The  $i^{\text{th}}$  value of the array can be accessed by

$$\textit{identifier}[i]$$

The  $i^{\text{th}}$  value of the array counting from right to left can be accessed by a negative index

$$\textit{identifier}[-i]$$

### 3.15.3 Set value at index

The  $i^{\text{th}}$  value of the array can be set by

$$\textit{identifier}[i] = \textit{expr};$$

### 3.15.4 List Slicing

A range of elements in a list can be returned by List slicing

$$\textit{identifier}[\textit{start} : \textit{end}]$$

With this operator, one can specify where to start the slicing and where to end. List slicing returns a new list from the existing list. For example

---

```
List<float> sublist = list[2:-2]; // negative index slicing
```

---

### 3.15.5 Other Built in List functions

- `void append(List<type-specifier> list, type-specifier x):`

Add an element to the end of list.

---

```
List<float> list = [0.0];  
append(list, 1.0);
```

---

Internally, each list has a capacity attribute to track how much memory it has in total. If the memory is not enough to hold an additional element, then all original values in the list is copied over to a new memory block with doubled capacity.

- `List<int> range(int n):`

Create and return a List of int with value 0, 1, ..., n-1.

For example

---

```
for i in range(5) {  
    print(i);  
}  
  
// is equivalent to  
for i in [0, 1, 2, 3, 4] {  
    print(i);  
}
```

---

- `int len(List<type-specifier> list):`

Return the length of the list. For example

---

```
for i in range(len(list)) {  
    print(list[i]);  
}
```

---

## 3.16 String

String is declared as an array of char and it uses built-in type of Ocaml.

### 3.16.1 String operations

- `int len(string str):`

Find the length of string



---

```
string str = "Nice to meet u";  
int res = length(str);
```

---

- `string concat(string str1, string str2):`

append `str2` at the end of `str1` and return `str1`;

---

```
string str1 = "Nice to";  
string str2 = "meet u";  
string res = concat(str1, str2);
```

---

- `string slice(string str1, int start, int end):`

find the substring of original string, slice it and return the substring

---

```
string str = "Hello world";  
string substr = slice(str, 0, 4);
```

---

## 3.17 Dict

A *Dict* is a dictionary which maps a key to a value. The dictionary template type is implemented with binary trees in C.

### 3.17.1 Dict Declaration

A Dict variable should be declared with a key type and a value type. That is,

---

```
Dict<key-type, value-type> identifier;
```

---

For example,

---

```
Dict<int, bool> mydict;
```

---

Upon declaration, an empty dictionary is created.

### 3.17.2 Dict Operations

- `void dictAdd(Dict<key-type, value-type> dict, key-type key, value-type val)`

Insert a key-value pair to a dictionary. For example,

---

```
dictAdd(mydict, 10, true);  
dictAdd(mydict, 18, false);
```

---

The compiler will throw an error if either the key type or value type does not match the declared types of the dictionary. Same for the other functions below.

- `int dictSize(Dict<key-type, value-type> dict)`

Return the number of key-value pairs in the dictionary. For example,

---

```
int size = dictSize(mydict);           // size = 2
```

---

- `bool dictHasKey(Dict<key-type, value-type> dict, key-type key)`

Check whether the given key exists in the dictionary. For example,

---

```
bool found = dictHasKey(mydict, 10);   // found = true  
found = dictHasKey(mydict, 30);       // found = false
```

---

- `bool dictGetBool(Dict<key-type, value-type> dict, key-type key)`

Retrieve a bool value from the dictionary. The dictionary must have value type bool, otherwise an error will be thrown. For example,

---

```
bool val = dictGetBool(mydict, 18);    // val = false
```

---

Attempting to retrieve the value of a non-existent key will cause an error. If uncertain, use `dictHasKey` to check its existence first.

- `int dictGetInt(Dict<key-type, value-type> dict, key-type key)`

Retrieve a int value from the dictionary.

- `int dictGetList(Dict<key-type, value-type> dict, key-type key)`

Retrieve a List from the dictionary.

- `float dictGetFloat(Dict<key-type, value-type> dict, key-type key)`

Retrieve a float value from the dictionary.

- `void dictRemove(Dict<key-type, value-type> dict, key-type key)`

Remove a key and its associated value from the dictionary. If the key does not exist, the operation will have no effect. For example,

---

```
dictRemove(mydict, 5);           // nothing happens
dictRemove(mydict, 10);
found = dictHasKey(mydict, 10); // found = false
```

---

## 3.18 Set

### 3.18.1 Set Declaration

A Set variable should be declared with an element type. That is,

---

```
Dict<type-specifier> identifier;
```

---

For example,

---

```
Set<float> myset;
```

---

Upon declaration, an empty set is created.

### 3.18.2 Set Operations

- `void setAdd(Set<type-specifier> set, type-specifier item)`

Insert an element to the set. The element type must match the declared type of the set, otherwise an error will be thrown. Same for the functions below. For example,

---

```
setAdd(myset, 10.5);
setAdd(myset, 7.2);
```

---

- `int setSize(Set<type-specifier> set)`

Return the number of elements in the set. For example,

---

```
int setsize = dictSize(mydict); // setsize = 2
```

---

- `bool setFind(Set<type-specifier> set, type-specifier item)`

Check whether the given element exists in the set. For example,

---

```
bool exists = setFind(myset, 10.5); // exists = true
exists = setFind(myset, 2.7);      // exists = false
```

---

- `void setRemove(Set<type-specifier> set, type-specifier item)`

Remove an element from the set. The operation has no effect if the element does not exist.

For example,

---

```
setRemove(myset, 3.14);           // nothing happens
setRemove(myset, 10.5);
exists = dictHasKey(mydict, 10.5); // exists = false
```

---

### 3.19 Lexical scoperules

The lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; FFBB is not a block-structured language; this may fairly be considered a defect. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function. It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

### 3.20 Constant expressions

In several places FFBB requires expressions that have to evaluate to a constant: in array bounds, the expression can only involve integer constants, possibly connected by binary arithmetic operators; in control flow condition statements, the expression has to evaluate to booleans.

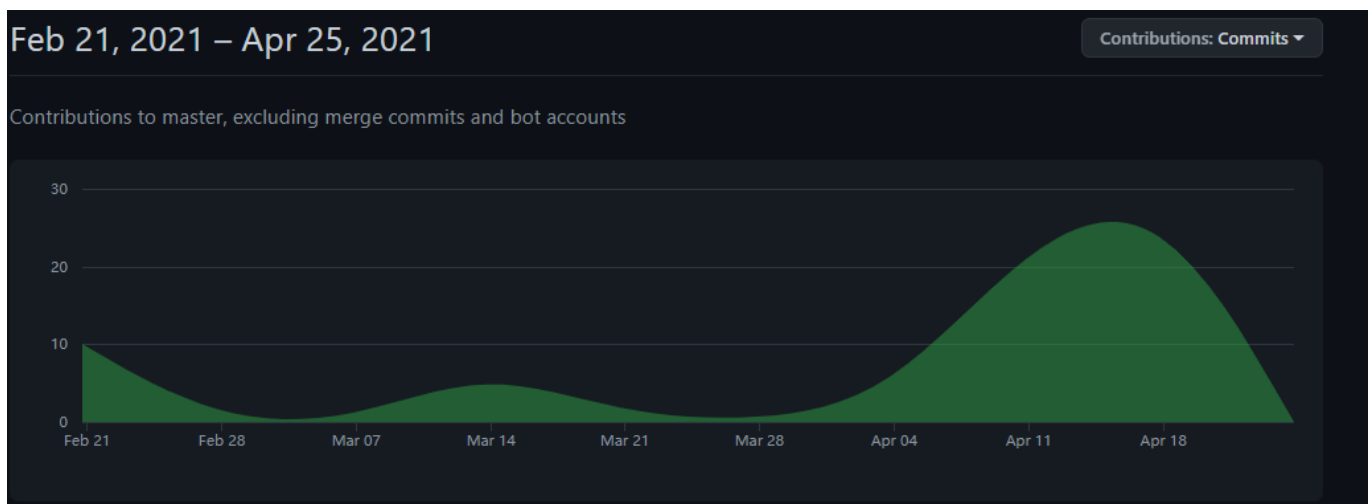
## 4 Project Plan

### 4.1 Process

Throughout the semester we had a wechat group 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. In addition, we have weekly meetings on every Saturday using Zoom. During the beginning of the project we would meet multiple times a week to figure out the design of our language. We also used Google Docs to write todo list and features we want to add in FFBB. This worked exceptionally well because we could start a Zoom meeting, work on the same google document together and then quickly complete the task. After the planning stage, we also discuss difficulties we have met and try to solve them together. We also used GitHub as tool for version control, we had a master branch and a few development branches. Usually, each contributor has his own branch for a feature. During the weekly meeting, we also do code review and merge feature branches to master branch in case there are any conflicts.

Below is a screenshot of our Github Project Insight page for showing our project process throughout the semester:



## 4.2 Style Guide

Our goal was primarily readability, though some of our test suite programs broke these rules to properly vet features. We enforce: 1. Individual lines of code cannot exceed 80 characters. 2. Employ Snakecase when naming variables and functions. 3. Blocks of non-trivial code should have terse but useful comments. 4. Use of 2-space or 4-space for space indentation.

Milestone	Date
Proposal	February 2
Reference Manual	February 22
Parser Draft	February 24
Hello World	March 24
Essential Functions	April 16
Additional Functions	April 24
Project Presentation	April 25
Final Report	April 26

Role	Team Member	UNI
System Architect	Bowen Chen	bc2916
Manager	Jianan Yao	jy3022
Language Guru	Joseph Yang	zy2431
Tester	Xiaosheng Chen	xc2561

### 4.3 Project Timeline

### 4.4 Team Roles

### 4.5 Tools

**Languages:** OCaml, C, Bash, zsh

**Version Control:** Git

**Repository Management:** Github

**Testing:** Bash, FFBB

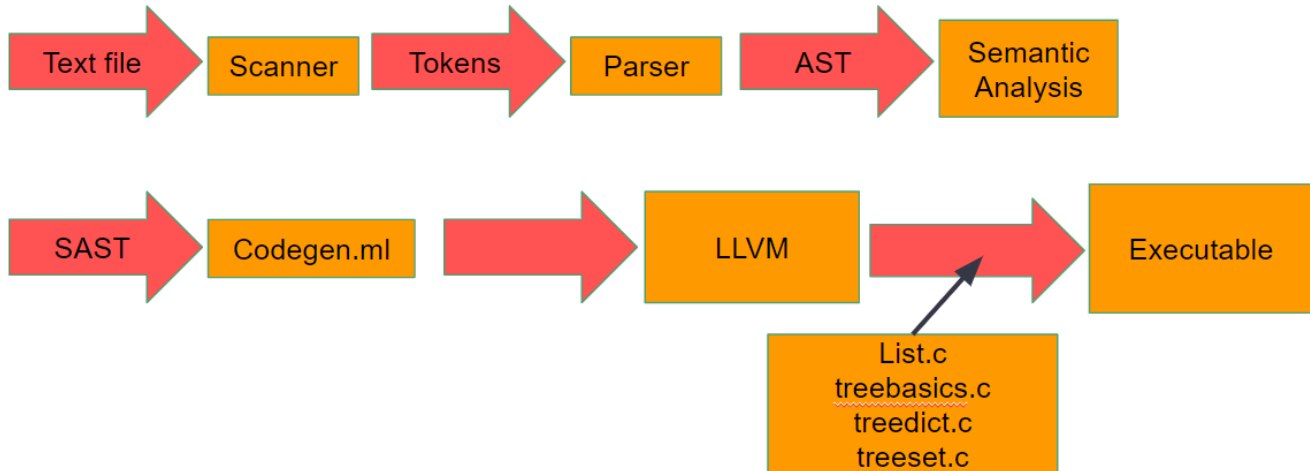
**Editors:** Vim, VS Code, sublime

**Platforms:** Window10, Ubuntu (via Docker), macOS

**Communication:** Wechat, Zoom, Slack

## 5 Architectural Design

### 5.1 Block Diagram



### 5.2 Scanner

Scanner will take in raw source codes, cluster sequences of characters into different groups and recognize them as different tokens. These tokens will be fed into parser. Comments are removed during this stage.

Authors: Bowen Chen, Jianan Yao

### 5.3 Parser, AST

Parser will use tokens generated from scanner, analyze their positions and compare with our grammar from abstract syntax tree. If source codes are not written in a correct manner, parser will be able to recognize the faults and prevent the source code from moving on to the next stages.

Authors: Bowen Chen, Jianan Yao, Xiaosheng Chen, Joseph Yang

### 5.4 Semantic Checking, SAST

SAST is the semantically checked abstract syntax tree representation of our FFBB source code.

Authors: Bowen Chen, Jianan Yao, Joseph Yang

## 5.5 Code Generation

This takes the SAST as input and generates the LLVM code.

Authors: Bowen Chen, Jianan Yao, Xiaosheng Chen, Joseph Yang

## 5.6 String Library

Authors: Xiaosheng Chen

## 5.7 Dictionary Library

Authors: Jianan Yao

## 5.8 Set Library

Authors: Jianan Yao

## 5.9 List Library

Authors: Bowen Chen

# 6 Test Plan

## 6.1 Testing Workflow

The structure of a FFBB test is to generate a test case for each feature store the expected output of that program in a file titled \*.out or \*.err , and then run a script that compiles the program and compares the generated output to the expected output. The script utilizes UNIX's diff tools to perform this comparison - if a difference is detected, the test fails, but if not, it passes. All of the testcases are stored inside tests folder. The source code of success testcases should start at test-\*.mc while Erroneous testcases should start at fail-\*.c. Our Workflow was inspired by MicroC's test suite.

Erroneous testcases also play important roles during our testing workflow. Users' behaviors are usually unexpected and it is important to for a program to hold error checking mechanisms and protect program from being crashed by codes with wrong syntax or logic. This is also essential for



testing our semantic checker.

You can run the following command

```
make
```

to run the tests.

## 6.2 Sample Tests

Our test suite has over 125 tests covering list, set, dictionary, lambda functions, function pointers, semantic checking. Here we show some interesting one. All test cases can be found in appendix.

### 6.2.1 Fibonacci sequence using List and for...in

Fibonacci sequence will always be a classic example for novice to grasp the basic concepts of programming. To complete such task, list will a good helper to store values calculated previously and for in loop are very handy to iterate the whole list.

```
1 /* Fibonacci number: Compute Nth value */
2 int main () {
3     int n = 10;
4     List<int> f = [0, 1];
5     for i in range(n-2) {
6         append(f, f[-1] + f[-2]);
7     }
8     print(f[-1]);
9 }
```

### 6.2.2 Quick sort

Recursion will always be engineers' supportive friends. Instead of trying to solve a complicated problem directly, programmers will usually break the whole problem into small pieces. And FFBB will protect recursive functions from running smoothly.

```
1 void swap(List<int> A, int i, int j) {
2     int t = A[i]; A[i] = A[j]; A[j] = t;
3 }
4
```

```

5 int partition(List<int> A, int p, int r) {
6     int x = A[r];
7     int i = p - 1;
8     for j in range(r - p + 1) {
9         if (A[j+p] <= x) {
10             i++;
11             swap(A, i, j+p);
12         }
13     }
14     swap(A, i+1, r);
15     return i;
16 }
17
18 /* Recursive function to sort list A using quick-sort */
19 void quicksort(List<int> A, int p, int r) {
20     if (p < r) {
21         int q = partition(A, p, r);
22         quicksort(A, p, q-1);
23         quicksort(A, q+1, r);
24     }
25 }
26
27 int main () {
28     /* Using quicksort */
29     List<int> A = [4, 2, 7, 3, 1, 9, 6, 10, 5, 8];
30     quicksort(A, 0, len(A) - 1);
31     for a in A {
32         print(a);
33     }
34 }

```

### 6.2.3 Higher Order Functions

To write concise and neat code, Higher order functions are commonly used for experienced programmers. And FFBB will provide programmers with a simple way to utilize higher order functions freely.

```

1 List<int> map(func<int, int> f, List<int> list) {

```

```

2     List<int> out;
3
4     for x in list {
5         append(out, f(x));
6     }
7     return out;
8 }
9
10 List<int> filter(func<bool, int> f, List<int> list) {
11     List<int> out;
12
13     for x in list {
14         if (f(x)) {
15             append(out, x);
16         }
17     }
18     return out;
19 }
20
21 func<int, int> sum2() {
22     return int lambda int x -> x+2;
23 }
24
25 void print_list(List<int> list) {
26     for x in list {
27         print(x);
28     }
29 }
30
31 int main()
32 {
33     /* [0, 1, 2, 3, 4] */
34     List<int> my_list = range(5);
35
36     List<int> out = map(int lambda int x -> x * 2, my_list);
37     print_list(out); /* 0, 2, 4, 6, 8 */
38
39     out = map(sum2(), my_list);

```

```

40     print_list(out); /* 2, 3, 4, 5, 6 */
41
42     out = filter(bool lambda int x -> x > 2, my_list);
43     print_list(out); /* 3, 4 */
44     return 0;
45 }

```

## 6.2.4 Bellman–Ford algorithm

FFBB will also support some implementations of graph algorithm. For dictionary data structure, it can store nodes in the graph as keys and dict's values will be lists which consists of each nodes' neighbors and weights.

```

1  int INF;
2  void graphInit(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
3      for i in range(n) {
4          List<int> l;
5          List<int> w;
6          dictAdd(E, i, l);
7          dictAdd(W, i, w);
8      }
9  }
10
11 void addEdge(Dict<int, List<int>> E, Dict<int, List<int>> W, int u, int v, int
    w) {
12     List<int> le = dictGetList(E, u);
13     List<int> lw = dictGetList(W, u);
14     append(le, v);
15     append(lw, w);
16 }
17
18 void printGraph(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
19     for i in range(n) {
20         prints("-----");
21         print(i);
22         List<int> le = dictGetList(E, i);
23         List<int> lw = dictGetList(W, i);
24

```

```

25     if (len(le) > 0) {
26         prints("neighbors");
27         for v in le {
28             print(v);
29         }
30         prints("weights");
31         for w in lw {
32             print(w);
33         }
34     }
35 }
36 }
37
38
39
40 bool checkNegativeCycle(Dict<int, List<int>> E, Dict<int, List<int>> W, int src
, int n, List<int> dist) {
41     /* for every edge */
42     for u in range(n) {
43         List<int> le = dictGetList(E, u);
44         List<int> lw = dictGetList(W, u);
45         int m = len(le);
46         if (m > 0) {
47             for k in range(m) {
48                 int v = le[k];
49                 int w = lw[k];
50
51                 /* we have u, v, w */
52                 if (dist[u] != INF && dist[u] + w < dist[v]) {
53                     prints("Graph contains negative weight cycle");
54                     return false;
55                 }
56             }
57         }
58     }
59     return true;
60 }
61

```

```

62 List<int> BellmanFord(Dict<int, List<int>> E, Dict<int, List<int>> W, int src,
    int n) {
63
64     List<int> dist;
65     for z in range(n) {
66         append(dist, INF);
67     }
68     dist[src] = 0;
69
70     /* Loop |V| times */
71     for i in range(n - 1) {
72         /* for every edge */
73         for u in range(n) {
74             List<int> le = dictGetList(E, u);
75             List<int> lw = dictGetList(W, u);
76             int m = len(le);
77             if (m > 0) {
78                 for k in range(m) {
79                     int v = le[k];
80                     int w = lw[k];
81
82                     /* we have u, v, w */
83                     if (dist[u] != INF && dist[u] + w < dist[v]) {
84                         dist[v] = dist[u] + w;
85                     }
86                 }
87             }
88         }
89     }
90     checkNegativeCycle(E, W, src, n, dist);
91     return dist;
92 }
93
94 int main() {
95     int n = 5;
96     INF = 100000;
97     Dict<int, List<int>> E;
98     Dict<int, List<int>> W;

```

```

99 graphInit(E, W, n);
100 addEdge(E, W, 0, 1, -1);
101 addEdge(E, W, 0, 2, 4);
102 addEdge(E, W, 1, 2, 3);
103 addEdge(E, W, 1, 3, 2);
104 addEdge(E, W, 1, 4, 2);
105 addEdge(E, W, 3, 2, 5);
106 addEdge(E, W, 3, 1, 1);
107 addEdge(E, W, 4, 3, -3);
108 printGraph(E, W, n);
109 prints("Vertex Distance from Source");
110 List<int> dist = BellmanFord(E, W, 0, n);
111 for d in dist {
112     print(d);
113 }
114
115 return 0;
116 }

```

## 6.3 Responsibilities

The majority of the test cases were written by Bowen Chen and Jianan Yao. Additional test cases were provided by Xiaosheng Chen and Joseph Yang. The rest of the test cases were inherited from MicroC.

# 7 Lessons Learned

## 7.1 Bowen Chen

The most important experience I had during this course was the demystification of compilers, interpreters, and translators. For example, when implementing the higher order functions and function pointers, I need to first go through how the original MicroC functions are parsed, stored and used. Then, after carefully thinking, I came up an approach to not only support function pointer, but also lambda function and even nested local function. Another example is when implementing mutable list which allows user to push element in the end, I have to look into the

generated LLVM code for debugging. I benefit a lot from these debugging processes and understand more deeply how most languages works I used everyday.

For future teams, I think starting early and demystification of compilers are the most important things to do well in the project.

## 7.2 Joseph Yang

When we were writing our language proposal at the beginning of this project, I completely have no idea what to do, as I had no knowledge about compiler structure and writing one seemed to be an impossible task. However, as we proceeded in this course, I not only came to understand the basic structures of a compiler, but walked through a clear example which is MicroC. Besides course contents, I also learned on how to do team projects. It's very important to meet with teammates regularly and update one's own progress. Also, a clear and fair task allocation also helps to improve the progress of a team project.

## 7.3 Jianan Yao

Originally I took this compiler course because I need a systems course and one of two TA units to graduate, and previous students and TAs recommended it. The project turns out more than helpful. I came to understand the workflow of a compiler, especially the semantic checking and code generation part. I understand how compilers take the syntax tree and build the basic blocks steps-by-step and how C-level control flow is translated into assembly-level branch instructions. I also learned a lot about OCaml and LLVM. These knowledge and experience have already been proved useful for my ongoing research on relaxed memory models, where normal compiler optimizations can be troublesome on relaxed memory hardware.

For future teams that plan to support `List<int>` style types, we find it difficult to support templates in external C. We eventually use less elegant workarounds (`void*` pointer and union data type). It can be worthwhile to try external C++ and its template class and functions instead.

## 7.4 Xiaosheng Chen

After spending a couple of months with FFBB, I have a better understanding about Ocaml, LLVM. I also know more about the mechanism of compilers, both in theory and in practice, and understand



why they could provide programmers with chances to write codes and build products. Moreover, it might not be easy to solve a big problem directly at the beginning. But it will be simpler if we can make careful analytic, break them down into smaller pieces, and conquer all of subproblems steps by steps.

## 8 Appendix

# A Source code of FFBB

## A.1 scanner.mll

```
1 (* Ocamllex scanner for FFBB *)
2
3 { open FFBBparser }
4
5 let digit = ['0' - '9']
6 let digits = digit+
7
8 rule token = parse
9   [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
10  | "/*"      { comment lexbuf }        (* Comments *)
11  | '('       { LPAREN }
12  | ')'       { RPAREN }
13  | '{'       { LBRACE }
14  | '}'       { RBRACE }
15  | '['       { LSQUARE }
16  | ']'       { RSQUARE }
17  | ';'       { SEMI }
18  | ','       { COMMA }
19  | '+'       { PLUS }
20  | "++"      { DOUBLEPLUS }
21  | '-'       { MINUS }
22  | "--"      { DOUBLEMINUS }
23  | '*'       { TIMES }
24  | '/'       { DIVIDE }
25  | '='       { ASSIGN }
26  | "=="      { EQ }
27  | "!="      { NEQ }
28  | '<'       { LT }
29  | "<="      { LEQ }
30  | ">"       { GT }
31  | ">="      { GEQ }
32  | "&&"      { AND }
33  | "||"      { OR }
34  | "!"       { NOT }
35  | "if"      { IF }
36  | "else"    { ELSE }
37  | "for"     { FOR }
38  | "while"   { WHILE }
39  | "return"  { RETURN }
40  | "int"     { INT }
41  | "bool"    { BOOL }
42  | "float"   { FLOAT }
43  | "List"    { LISTT }
44  | "string"  { STRING }
45  | "Set"     { SETT }
```

```

46 | "Dict"    { DICTT }
47 | "void"   { VOID }
48 | "in"     { IN }
49 | "lambda" { LAMBDA }
50 | "func"   { FUNC }
51 | ":"      { COLON }
52 | "->"     { ARROW }
53 | "true"   { BLIT(true) }
54 | "false"  { BLIT(false) }
55 | digits as lxm { LITERAL(int_of_string lxm) }
56 | digits '.' digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
57 | ['a'-'z' 'A'-'Z' '@']['a'-'z' 'A'-'Z' '0'-'9' '_' ]* as lxm { ID(lxm) }
58 | '"'      { read_string (Buffer.create 10) lexbuf }
59 | eof { EOF }
60 | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
61
62 and comment = parse
63   "*/" { token lexbuf }
64 | _    { comment lexbuf }
65
66 and read_string buf =
67   parse
68   | '"'      { STRLIT (Buffer.contents buf) }
69   | [^ '"' '\\\']+
70     { Buffer.add_string buf (Lexing.lexeme lexbuf);
71       read_string buf lexbuf
72     }
73 | _ { raise (Failure ("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
74 | eof { raise (Failure ("String is not terminated")) }

```

## A.2 FFBBparser.mly

```

1 /* Ocaml yacc parser for FFBB */
2
3 %{
4   open Ast
5   let lambda_num = ref 0
6
7   %}
8
9 %token SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA PLUS MINUS TIMES DIVIDE
10  ASSIGN COLON IN
11 %token NOT EQ NEQ LT LEQ GT GEQ AND OR DOUBLEPLUS DOUBLEMINUS LAMBDA FUNC ARROW
12 %token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID STRING LISTT DICTT SETT
13 %token <int> LITERAL
14 %token <bool> BLIT
15 %token <string> ID FLIT STRLIT
16 %token EOF

```

```

16
17 %start program
18 %type <Ast.program> program
19
20 %nonassoc NOELSE
21 %nonassoc ELSE
22 %right ASSIGN COLON
23 %left OR
24 %left AND
25 %left DOUBLEPLUS DOUBLEMINUS
26 %left EQ NEQ
27 %left LT GT LEQ GEQ
28 %left PLUS MINUS
29 %left TIMES DIVIDE
30 %right NOT
31 %left LSQUARE
32
33 %%
34
35 program:
36   decls EOF { $1 }
37
38 decls:
39   /* nothing */ { ([], []) }
40 | decls vdecl { (($2 :: fst $1), snd $1) }
41 | decls fdecl { (fst $1, ($2 :: snd $1)) }
42
43 formals_opt:
44   /* nothing */ { [] }
45 | formal_list { $1 }
46
47 formal_list:
48   typ ID { [($1,$2)] }
49 | formal_list COMMA typ ID { ($3,$4) :: $1 }
50
51 formals_opt_lambda:
52   /* nothing */ { ([], []) }
53 | formal_list_lambda { (List.rev (fst $1), List.rev (snd $1)) }
54
55 formal_list_lambda:
56   typ ID { ([$1],[2]) }
57 | formal_list_lambda COMMA typ ID { ($3 :: fst $1), ($4 :: snd $1) }
58
59 fdecl:
60   typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
61   {
62     { typ = $1;
63       fname = $2;
64       formals = List.rev $4;
65       body = List.rev $7 }

```

```

66     }
67
68
69 typ_list:
70     typ                {[$1]}
71     | typ COMMA typ_list { $1 :: $3 }
72
73 high_typ:
74     LISTT { LIST }
75     | SETT { SET }
76     | DICTT { DICT }
77
78 typ:
79     INT      { Int   }
80     | BOOL   { Bool  }
81     | FLOAT  { Float }
82     | STRING { String}
83     | VOID   { Void  }
84     | FUNC LT typ_list GT { FunctionType($3) }
85     | high_typ LT typ_list GT { CompositeType($1, $3) }
86
87 vdecl_list:
88     /* nothing */ { [] }
89     | vdecl_list vdecl { $2 :: $1 }
90
91 vdecl:
92     typ ID SEMI { ($1, $2) }
93
94 stmt_list:
95     /* nothing */ { [] }
96     | stmt_list stmt { $2 :: $1 }
97
98 stmt:
99     expr SEMI                                { Expr $1                }
100    | RETURN expr_opt SEMI                    { Return $2              }
101    | LBRACE stmt_list RBRACE                  { Block(List.rev $2)     }
102    | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([]))  }
103    | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
104    | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
105                                             { For($3, $5, $7, $9)   }
106    | FOR ID IN expr stmt                      { Forin($2, $4, $5)     }
107    | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }
108    | typ ID SEMI                              { VarDecl($1, $2)      }
109    | typ ID ASSIGN expr SEMI                  { VarDeclAssign(($1, $2), $4) }
110    | ID DOUBLEPLUS SEMI                       { Expr(Unop(Inc, Id($1))) }
111    | ID DOUBLEMINUS SEMI                      { Expr(Unop(Dec, Id($1))) }
112    | ID LSQUARE expr RSQUARE DOUBLEPLUS SEMI { Expr(IndexUnop($1, $3, Inc)) }
113    | ID LSQUARE expr RSQUARE DOUBLEMINUS SEMI { Expr(IndexUnop($1, $3, Dec)) }
114
115 expr_opt:

```

```

116     /* nothing */ { Noexpr }
117 | expr          { $1 }
118
119 expr:
120     LITERAL      { Literal($1)          }
121 | FLIT          { Fliteral($1)         }
122 | BLIT          { BoolLit($1)          }
123 | STRLIT        { StrLit($1)           }
124 | function_literal { $1 }
125 | ID            { Id($1)               }
126 | expr PLUS     expr { Binop($1, Add,   $3) }
127 | expr MINUS    expr { Binop($1, Sub,   $3) }
128 | expr TIMES    expr { Binop($1, Mult,  $3) }
129 | expr DIVIDE   expr { Binop($1, Div,   $3) }
130 | expr EQ       expr { Binop($1, Equal, $3) }
131 | expr NEQ      expr { Binop($1, Neq,   $3) }
132 | expr LT       expr { Binop($1, Less,  $3) }
133 | expr LEQ      expr { Binop($1, Leq,   $3) }
134 | expr GT       expr { Binop($1, Greater, $3) }
135 | expr GEQ      expr { Binop($1, Geq,   $3) }
136 | expr AND      expr { Binop($1, And,   $3) }
137 | expr OR       expr { Binop($1, Or,    $3) }
138 | list_expr     { $1 }
139 | MINUS expr %prec NOT { Unop(Neg, $2)    }
140 | NOT expr      { Unop(Not, $2)          }
141 | ID LSQUARE expr RSQUARE ASSIGN expr { IndexAssign($1, $3, $6) }
142 | ID ASSIGN expr { Assign($1, $3)        }
143 | ID LPAREN args_opt RPAREN { Call($1, $3) }
144 | ID LSQUARE expr COLON expr RSQUARE { GetSlice($1, $3, $5) }
145 | ID LSQUARE expr RSQUARE { GetIndex($1, $3) }
146 | LPAREN expr RPAREN { $2 }
147
148 function_literal:
149     typ LAMBDA formals_opt LBRACE stmt_list RBRACE{
150         lambda_num := !lambda_num + 1;
151         FunctionLit({
152             typ = $1;
153             fname = "lambda_" ^ string_of_int !lambda_num;
154             formals = List.rev $3;
155             body = List.rev $5
156         })
157     }
158 | typ LAMBDA formals_opt ARROW expr {
159     lambda_num := !lambda_num + 1;
160     FunctionLit({
161         typ = $1;
162         fname = "lambda_" ^ string_of_int !lambda_num;
163         formals = List.rev $3;
164         body = [Return $5]
165     })

```

```

166     }
167
168 list_expr:
169     LSQUARE RSQUARE          { ListExpr([]) }
170     | LSQUARE list_expr_core RSQUARE { ListExpr($2)}
171
172 list_expr_core:
173     expr                      { [$1] }
174     | expr COMMA list_expr_core { $1 :: $3 }
175
176 args_opt:
177     /* nothing */ { [] }
178     | args_list { List.rev $1 }
179
180 args_list:
181     expr                      { [$1] }
182     | args_list COMMA expr { $3 :: $1 }

```

### A.3 ast.ml

```

1 (* Abstract Syntax Tree and functions for printing it *)
2
3 type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
4         And | Or
5
6 type uop = Neg | Not | Inc | Dec
7
8 (* type prm_typ = Int | Bool | Float | Void | String *)
9 type high_typ = LIST | DICT | SET
10
11 type typ =
12     Int | Bool | Float | Void | String | Func
13     | CompositeType of high_typ * typ list
14     | FunctionType of typ list
15
16
17 and func_decl = {
18     typ : typ;
19     fname : string;
20     formals : bind list;
21     body : stmt list;
22 }
23
24 and bind = typ * string
25
26 and expr =
27     Literal of int
28     | Fliteral of string

```



```

29 | BoolLit of bool
30 | StrLit of string
31 | FunctionLit of func_decl
32 | GetSlice of string * expr * expr
33 | GetIndex of string * expr
34 | Id of string
35 | Binop of expr * op * expr
36 | Unop of uop * expr
37 | IndexAssign of string * expr * expr
38 | Assign of string * expr
39 | ListExpr of expr list
40 | Call of string * expr list
41 | IndexUnop of string * expr * uop
42 | Noexpr
43
44 and stmt =
45   Block of stmt list
46 | Expr of expr
47 | Return of expr
48 | If of expr * stmt * stmt
49 | For of expr * expr * expr * stmt
50 | Forin of string * expr * stmt
51 | While of expr * stmt
52 | VarDecl of bind
53 | VarDeclAssign of bind * expr
54
55 type program = bind list * func_decl list
56
57
58
59 (* Pretty-printing functions *)
60
61 let string_of_op = function
62   Add -> "+"
63 | Sub -> "-"
64 | Mult -> "*"
65 | Div -> "/"
66 | Equal -> "=="
67 | Neq -> "!="
68 | Less -> "<"
69 | Leq -> "<="
70 | Greater -> ">"
71 | Geq -> ">="
72 | And -> "&&"
73 | Or -> "||"
74
75 let string_of_uop = function
76   Neg -> "-"
77 | Not -> "!"
78 | Inc -> "++"

```

```

79 | Dec -> "--"
80
81 let string_of_high_typ = function
82   LIST -> "List"
83   | DICT -> "Dict"
84   | SET -> "Set"
85
86 let rec string_of_typ = function
87   Int -> "int"
88   | Bool -> "bool"
89   | Float -> "float"
90   | Void -> "void"
91   | String -> "string"
92   | Func -> "func"
93   | CompositeType(ht, tl) ->
94     string_of_high_typ ht ^ "<" ^ String.concat ", " (List.map string_of_typ tl) ^ ">"
95   | FunctionType(rt::rest) ->
96     string_of_typ rt ^ ":" ^ String.concat ", " (List.map string_of_typ rest) ^ ""
97
98 let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
99
100 let string_of_formal (t, id) = string_of_typ t ^ " " ^ id
101
102 let rec string_of_expr = function
103   Literal(l) -> string_of_int l
104   | Fliteral(l) -> l
105   | BoolLit(true) -> "true"
106   | BoolLit(false) -> "false"
107   | StrLit(s) -> "\"" ^ s ^ "\""
108   | FunctionLit(l) -> string_of_typ l.typ ^ " " ^
109     l.fname ^ "(" ^ String.concat ", " (List.map string_of_formal l.formals) ^
110     ")\n{}\n"
111   | Id(s) -> s
112   | IndexAssign(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ "]" = " ^ string_of_expr e2
113   | GetIndex(v, e2) -> v ^ "[" ^ string_of_expr e2 ^ "]"
114   | GetSlice(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ ":" ^ string_of_expr e2 ^ "]"
115   | Binop(e1, o, e2) ->
116     string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
117   | Unop(o, e) ->
118     (match o with
119     | Neg | Not -> string_of_uop o ^ string_of_expr e
120     | Inc | Dec -> string_of_expr e ^ string_of_uop o
121     )
122   | ListExpr(el) -> "[" ^ String.concat ", " (List.map string_of_expr el) ^ "]"
123   | Assign(v, e) -> v ^ " = " ^ string_of_expr e
124   | Call(f, el) ->
125     f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
126   | IndexUnop(v, e, o) ->
127     v ^ "[" ^ string_of_expr e ^ "]" ^ string_of_uop o
128   | Noexpr -> ""

```

```

129
130 let rec string_of_stmt = function
131   Block(stmts) ->
132     "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
133 | Expr(expr) -> string_of_expr expr ^ ";\n"
134 | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
135 | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
136 | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
137   string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
138 | For(e1, e2, e3, s) ->
139   "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
140   string_of_expr e3 ^ ") " ^ string_of_stmt s
141 | Forin(id1, id2, s) ->
142   "for " ^ id1 ^ " in " ^ string_of_expr id2 ^ " " ^ string_of_stmt s
143 | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
144 | VarDecl(t, id) -> string_of_typ t ^ " " ^ id ^ ";\n"
145 | VarDeclAssign((t, id), e) -> string_of_typ t ^ " " ^ id ^ " = " ^ string_of_expr e ^
146   ";\n"
147
148
149 let string_of_fdecl fdecl =
150   string_of_typ fdecl.typ ^ " " ^
151   fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_formal fdecl.formals) ^
152   ")\n{\n" ^
153   String.concat "" (List.map string_of_stmt fdecl.body) ^
154   "}\n"
155
156 let string_of_program (vars, funcs) =
157   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
158   String.concat "\n" (List.map string_of_fdecl funcs)

```

## A.4 semant.ml

```

1 (* Semantic checking for the FFBB compiler *)
2
3 open Ast
4 open Sast
5 module StringMap = Map.Make (String)
6
7 (* Semantic checking of the AST. Returns an SAST if successful,
8   throws an exception if something is wrong.
9
10   Check each global variable, then check each function *)
11 let local_sfunc = ref []
12
13 let function_decls = ref StringMap.empty
14

```

```

15 let check (globals, functions) =
16   (* Verify a list of bindings has no void types or duplicate names *)
17   let check_binds (kind : string) (binds : bind list) =
18     List.iter
19       (function
20         | Void, b -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
21         | _ -> ())
22     binds;
23   let rec dups = function
24     | [] -> ()
25     | (_, n1) :: (_, n2) :: _ when n1 = n2 ->
26       raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
27     | _ :: t -> dups t
28   in
29   dups (List.sort (fun (_, a) (_, b) -> compare a b) binds)
30 in
31
32 (* Verify the new binding to be added is not a duplicate *)
33 let check_dup (binding : bind) (bindings : bindtbl) =
34   match binding with
35   | Void, s -> raise (Failure ("illegal void local " ^ s))
36   | _, s ->
37     if Hashtbl.mem bindings s then raise (Failure ("duplicate local " ^ s))
38     else ()
39 in
40
41 (**** Check global variables ****)
42 check_binds "global" globals;
43
44 (**** Check functions ****)
45 let lambda_num = ref 0 in
46
47 (* Collect function declarations for built-in functions: no bodies *)
48 let get_atypes fdecl = List.map (fun x -> fst x) fdecl.formals in
49 let get_ftype fdecl = FunctionType (fdecl.typ :: get_atypes fdecl) in
50 let built_in_decls =
51   let add_bind map (name, return_ty, tys) =
52     StringMap.add name
53       {
54         typ = return_ty;
55         fname = name;
56         formals =
57           (match List.length tys with
58            | 0 -> []
59            | 1 -> [ (List.nth tys 0, "arg1") ]
60            | 2 -> [ (List.nth tys 0, "arg1"); (List.nth tys 1, "arg2") ]
61            | 3 ->
62              [
63                (List.nth tys 0, "arg1");
64                (List.nth tys 1, "arg2");

```

```

65         (List.nth tys 2, "arg3");
66     ]
67     | _ -> raise (Failure "Built-in function must take 0-3 arguments");
68     body = [];
69 }
70 map
71 in
72 List.fold_left add_bind StringMap.empty
73 [
74     ("print", Void, [ Int ]);
75     ("printb", Void, [ Bool ]);
76     ("prints", Void, [ String ]);
77     ("printf", Void, [ Float ]);
78     ("printbig", Void, [ Int ]);
79     ("setAdd", Void, [ CompositeType (SET, [ Int ]); Int ]);
80     ("setFind", Bool, [ CompositeType (SET, [ Int ]); Int ]);
81     ("setRemove", Void, [ CompositeType (SET, [ Int ]); Int ]);
82     ("setSize", Int, [ CompositeType (SET, [ Int ]) ]);
83     ("dictAdd", Void, [ CompositeType (DICT, [ Int; Int ]); Int; Int ]);
84     ("dictSize", Int, [ CompositeType (DICT, [ Int; Int ]) ]);
85     ("dictHasKey", Bool, [ CompositeType (DICT, [ Int; Int ]); Int ]);
86     ("dictGetInt", Int, [ CompositeType (DICT, [ Int; Int ]); Int ]);
87     ("dictGetBool", Bool, [ CompositeType (DICT, [ Int; Int ]); Int ]);
88     ("dictGetFloat", Float, [ CompositeType (DICT, [ Int; Int ]); Int ]);
89     ("dictGetList", CompositeType (LIST, [Int]), [ CompositeType (DICT, [ Int; Int
90 ]; Int ));
91     ("dictRemove", Int, [ CompositeType (DICT, [ Int; Int ]); Int ]);
92     ("len", Int, [ CompositeType (LIST, [ Void ]) ]);
93     ("range", CompositeType (LIST, [ Int ]), [ Int ]);
94     ("append", Void, [ CompositeType (LIST, [ Void ]); Void ]);
95     ("slice", String, [ String; Int; Int ]);
96     ("length", Int, [ String ]);
97     ("concat", String, [ String; String ]);
98 ]
99 in
100 (* Add function name to symbol table *)
101 let add_func map fd =
102     let built_in_err = "function " ^ fd.fname ^ " may not be defined"
103     and dup_err = "duplicate function " ^ fd.fname
104     and make_err er = raise (Failure er)
105     and n = fd.fname (* Name of the function *) in
106     match fd with
107     (* No duplicate functions or redefinitions of built-ins *)
108     | _ when StringMap.mem n built_in_decls -> make_err built_in_err
109     | _ when StringMap.mem n map -> make_err dup_err
110     | _ -> StringMap.add n fd map
111 in
112
113 (* Collect all function names into one symbol table *)

```

```

114 let _ =
115     ignore (function_decls := List.fold_left add_func built_in_decls functions)
116 in
117
118 (* Return a function from our symbol table *)
119 let find_func s =
120     try StringMap.find s !function_decls
121     with Not_found -> raise (Failure ("unrecognized function " ^ s))
122 in
123
124 let _ = find_func "main" in
125
126 (* Ensure "main" is defined *)
127 let rec check_function func =
128     let bindings = Hashtbl.create 10 in
129     (* Make sure no formals are void or duplicates *)
130     check_binds "formal" func.formals;
131
132     let add_formal_to_bindings formal =
133         Hashtbl.add bindings (snd formal) (fst formal)
134     in
135     List.iter add_formal_to_bindings func.formals;
136
137     (* Raise an exception if the given rvalue type cannot be assigned to
138        the given lvalue type *)
139     let check_assign fname lvaluet rvaluet err =
140         match fname with
141         | "append" | "len" | "setAdd" | "setFind" | "setRemove" | "setSize"
142         | "dictAdd" | "dictHasKey" | "dictGetInt" | "dictGetBool" | "dictGetFloat" | "
143 dictGetList"
144         | "dictRemove" | "dictSize" ->
145             lvaluet
146         | _ -> if lvaluet = rvaluet then lvaluet else raise (Failure err)
147     in
148
149     (* Raise an exception if the given rvalue type cannot be assigned to
150        the given lvalue type *)
151     let get_list_prim_typ t =
152         match t with CompositeType (LIST, [ typ ]) -> typ
153     in
154     let check_get_index lvaluet rvaluet err = get_list_prim_typ lvaluet in
155
156     (* Build local symbol table of variables for this function *)
157     let symbols =
158         List.fold_left
159             (fun m (ty, name) -> StringMap.add name ty m)
160             StringMap.empty globals
161     in
162
163     (* Return a variable from our local symbol table *)

```

```

163 let type_of_identifier s =
164     try Hashtbl.find bindings s
165     with Not_found -> (
166         try StringMap.find s symbols
167         with Not_found -> raise (Failure ("undeclared identifier " ^ s)))
168     in
169
170 (* Return a semantically-checked expression, i.e., with a type *)
171 let rec expr = function
172 | Literal l -> (Int, SLiteral l)
173 | Fliteral l -> (Float, SFliteral l)
174 | BoolLit l -> (Bool, SBoolLit l)
175 | FunctionLit l ->
176     function_decls := List.fold_left add_func !function_decls [ l ];
177     let local_sf = check_function l in
178     let _ = local_sfunc := local_sf :: !local_sfunc in
179     (get_ftype l, SFunctionLit local_sf)
180 | StrLit l -> (String, SStrLit l)
181 | Noexpr -> (Void, SNoexpr)
182 | Id s -> (type_of_identifier s, SId s)
183 | IndexAssign (var, e1, e2) as ex ->
184     let lt = type_of_identifier var
185     and rt1, e1' = expr e1
186     and rt2, e2' = expr e2 in
187     let err = "illegal index assignment" in
188     (lt, SIndexAssign (var, (rt1, e1'), (rt2, e2')))
189 | GetIndex (var, e2) as ex ->
190     let lt = type_of_identifier var and rt, e' = expr e2 in
191     let err =
192         "illegal index access " ^ string_of_typ lt ^ "[" ^ string_of_expr ex
193         ^ "]"
194     in
195     (check_get_index lt rt err, SGetIndex (var, (rt, e')))
196 | GetSlice (var, e1, e2) as ex ->
197     let lt = type_of_identifier var
198     and rt1, e1' = expr e1
199     and rt2, e2' = expr e2 in
200     let err =
201         "illegal slice access " ^ string_of_typ lt ^ "[" ^ string_of_expr ex
202         ^ "]"
203     in
204     (lt, SGetSlice (var, (rt1, e1'), (rt2, e2')))
205 | Assign (var, e) as ex ->
206     let lt = type_of_identifier var and rt, e' = expr e in
207     let err =
208         "illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ rt
209         ^ " in " ^ string_of_expr ex
210     in
211     (check_assign var lt rt err, SAssign (var, (rt, e')))
212 | ListExpr el as ex ->

```

```

213     let check_list e =
214         let et, e' = expr e in
215         (et, e')
216     in
217     let el' = List.map check_list el in
218     let head = List.hd el' in
219     let get_rt (a, _) = [ a ] in
220     (CompositeType (LIST, get_rt head), SListExpr el')
221 | Unop (op, e) as ex -> (
222     let t, e' = expr e in
223     let ty =
224         match op with
225         | Neg when t = Int || t = Float -> t
226         | Not when t = Bool -> Bool
227         | (Inc | Dec) when t = Int -> t
228         | Inc | Dec ->
229             raise
230             (Failure
231              ("illegal unary operator " ^ string_of_typ t
232               ^ string_of_uop op ^ " in " ^ string_of_expr ex))
233         | _ ->
234             raise
235             (Failure
236              ("illegal unary operator " ^ string_of_uop op
237               ^ string_of_typ t ^ " in " ^ string_of_expr ex))
238     in
239     match ex with
240     | Unop (Neg, _) | Unop (Not, _) -> (ty, SUnop (op, (t, e')))
241     | Unop (Inc, Id s) ->
242         (ty, SAssign (s, (t, SBinop ((t, SId s), Add, (t, SLiteral 1)))))
243     | Unop (Dec, Id s) ->
244         (ty, SAssign (s, (t, SBinop ((t, SId s), Sub, (t, SLiteral 1)))))
245 | IndexUnop (v, e, o) as ex -> (
246     let list_t = type_of_identifier v and index_t, e' = expr e in
247     let err =
248         "illegal index access " ^ string_of_typ list_t ^ "["
249         ^ string_of_expr ex ^ "]"
250     in
251     let index_element =
252         (check_get_index list_t index_t err, SGetIndex (v, (index_t, e')))
253     in
254     match o with
255     | Inc ->
256         ( list_t,
257           SIndexAssign
258           ( v,
259             (index_t, e'),
260             (Int, SBinop (index_element, Add, (Int, SLiteral 1))) ) )
261     | Dec ->
262         ( list_t,

```



```

263         SIndexAssign
264         ( v,
265           (index_t, e'),
266           (Int, SBinop (index_element, Sub, (Int, SLiteral 1))) ) )
267 | _ -> raise (Failure "Case should never be reached")
268 | Binop (e1, op, e2) as e ->
269   let t1, e1' = expr e1 and t2, e2' = expr e2 in
270   (* All binary operators require operands of the same type *)
271   let same = t1 = t2 in
272   (* Determine expression type based on operator and operand types *)
273   let ty =
274     match op with
275     | (Add | Sub | Mult | Div) when same && t1 = Int -> Int
276     | (Add | Sub | Mult | Div) when same && t1 = Float -> Float
277     | (Equal | Neq) when same -> Bool
278     | (Less | Leq | Greater | Geq) when same && (t1 = Int || t1 = Float)
279     ->
280       Bool
281     | (And | Or) when same && t1 = Bool -> Bool
282     | _ ->
283       raise
284         (Failure
285          ("illegal binary operator " ^ string_of_typ t1 ^ " "
286           ^ string_of_op op ^ " " ^ string_of_typ t2 ^ " in "
287            ^ string_of_expr e))
288   in
289   (ty, SBinop ((t1, e1'), op, (t2, e2')))
290 | Call (fname, args) as call ->
291   let fd =
292     if StringMap.mem fname !function_decls then find_func fname
293     else if Hashtbl.mem bindings fname then
294       match type_of_identifier fname with
295       | FunctionType (rt :: rest) ->
296         {
297           typ = rt;
298           fname;
299           formals = List.mapi (fun i x -> (x, string_of_int i)) rest;
300           body = [];
301         }
302       | _ -> raise (Failure ("unrecognized function " ^ fname))
303     else raise (Failure ("unrecognized function " ^ fname))
304   in
305   let param_length = List.length fd.formals in
306   if List.length args != param_length then
307     raise
308       (Failure
309        ("expecting " ^ string_of_int param_length ^ " arguments in "
310         ^ string_of_expr call))
311   else
312     (* check if formal_type == expr_type *)

```

```

313 let _check_call fname ft e =
314     let et, e' = expr e in
315     let err =
316         "illegal argument found " ^ string_of_ttyp et ^ " expected "
317         ^ string_of_ttyp ft ^ " in " ^ string_of_expr e
318     in
319     (check_assign fname ft et err, e')
320 in
321 let get_et e =
322     let et, e' = expr e in
323     et
324 in
325 let err_msg = fname ^ " type incorrect" in
326 let err_msg_2 =
327     "first argument of " ^ fname ^ " should have composite type"
328 in
329 (* set arg_idx to -1 when internal type check is no needed *)
330 let check_custom_fun_type exp_high_ttyp len arg_idxs =
331     match List.map get_et args with
332     | CompositeType (high_ttyp, [ prim_ttyp ]) :: rest ->
333         if
334             List.length rest = len
335             && high_ttyp = exp_high_ttyp
336             &&
337             match arg_idxs with
338             | [ arg_idx ] -> prim_ttyp = List.nth rest arg_idx
339             | [] -> true
340             | _ -> raise (Failure "Internal error at arg_idxs")
341         then fname
342         else raise (Failure err_msg)
343     | CompositeType (high_ttyp, [ key_ttyp; value_ttyp ]) :: rest ->
344         if
345             List.length rest = len
346             && high_ttyp = exp_high_ttyp
347             &&
348             match arg_idxs with
349             | [ arg_idx1; arg_idx2 ] ->
350                 key_ttyp = List.nth rest arg_idx1
351                 && value_ttyp = List.nth rest arg_idx2
352             | [ arg_idx ] -> key_ttyp = List.nth rest arg_idx
353             | [] -> true
354             | _ -> raise (Failure "Internal error at arg_idxs")
355         then fname
356         else raise (Failure err_msg)
357     | _ -> raise (Failure err_msg_2)
358 in
359 let _ =
360     match fname with
361     | "append" -> check_custom_fun_type LIST 1 [ 0 ]
362     | "len" -> check_custom_fun_type LIST 0 []

```

```

363     | "setAdd" -> check_custom_fun_type SET 1 [ 0 ]
364     | "setFind" -> check_custom_fun_type SET 1 [ 0 ]
365     | "setRemove" -> check_custom_fun_type SET 1 [ 0 ]
366     | "setSize" -> check_custom_fun_type SET 0 []
367     | "dictAdd" -> check_custom_fun_type DICT 2 [ 0; 1 ]
368     | "dictHasKey" -> check_custom_fun_type DICT 1 [ 0 ]
369     | "dictGetInt" -> check_custom_fun_type DICT 1 [ 0 ]
370     | "dictGetBool" -> check_custom_fun_type DICT 1 [ 0 ]
371     | "dictGetFloat" -> check_custom_fun_type DICT 1 [ 0 ]
372     | "dictGetList" -> check_custom_fun_type DICT 1 [ 0 ]
373     | "dictRemove" -> check_custom_fun_type DICT 1 [ 0 ]
374     | "dictSize" -> check_custom_fun_type DICT 0 []
375     | _ -> fname
376   in
377   let check_call (ft, _) e = _check_call fname ft e in
378   let args' = List.map2 check_call fd.formals args in
379   (fd.typ, SCall (fname, args'))
380 in
381
382 let check_bool_expr e =
383   let t', e' = expr e
384   and err = "expected Boolean expression in " ^ string_of_expr e in
385   if t' != Bool then raise (Failure err) else (t', e')
386 in
387
388 (* Return a semantically-checked statement i.e. containing sexprs *)
389 let rec check_stmt = function
390 | Expr e -> SExpr (expr e)
391 | If (p, b1, b2) -> SIf (check_bool_expr p, check_stmt b1, check_stmt b2)
392 | For (e1, e2, e3, st) ->
393   SFor (expr e1, check_bool_expr e2, expr e3, check_stmt st)
394 | Forin (e1, e2, st) ->
395   lambda_num := !lambda_num + 1;
396   let rt, e2' = expr e2 in
397   let lt = get_list_prim_typ rt in
398   let iter_n = "iter" ^ string_of_int !lambda_num in
399   let clist_n = "clist" ^ string_of_int !lambda_num in
400   let iter = check_stmt (VarDeclAssign ((Int, iter_n), Literal 0)) in
401   let clist = check_stmt (VarDeclAssign ((rt, clist_n), e2)) in
402   let pred =
403     expr (Binop (Id iter_n, Less, Call ("len", [ Id clist_n ])))
404   in
405   let inc = expr (Unop (Inc, Id iter_n)) in
406   let x =
407     check_stmt (VarDeclAssign ((lt, e1), GetIndex (clist_n, Id iter_n)))
408   in
409   let next = expr (Assign (e1, GetIndex (clist_n, Id iter_n))) in
410   SForin (iter, clist, x, pred, inc, next, check_stmt st)
411 | While (p, s) -> SWhile (check_bool_expr p, check_stmt s)
412 | Return e ->

```

```

413     let t, e' = expr e in
414     if t = func.typ then SReturn (t, e')
415     else
416         raise
417             (Failure
418                 ("return gives " ^ string_of_typ t ^ " expected "
419                     ^ string_of_typ func.typ ^ " in " ^ string_of_expr e))
420 (* A block is correct if each statement is correct and nothing
421     follows any Return statement. Nested blocks are flattened. *)
422 | Block s1 ->
423     let rec check_stmt_list = function
424     | [ (Return _ as s) ] -> [ check_stmt s ]
425     | Return _ :: _ -> raise (Failure "nothing may follow a return")
426     | Block s1 :: ss -> check_stmt_list (s1 @ ss) (* Flatten blocks *)
427     | s :: ss ->
428         let tmp = check_stmt s in
429         tmp :: check_stmt_list ss
430     | [] -> []
431     in
432     SBlock (check_stmt_list s1)
433 | VarDecl (t, s) ->
434     check_dup (t, s) bindings;
435     Hashtbl.add bindings s t;
436     SVarDecl (t, s)
437 | VarDeclAssign ((t, s), e) as stmt ->
438     check_dup (t, s) bindings;
439     Hashtbl.add bindings s t;
440     let lt = t and rt, e' = expr e in
441     let err =
442         "illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ rt
443         ^ " in " ^ string_of_stmt stmt
444     in
445     let tmp =
446         (check_assign s lt rt err, SVarDeclAssign ((t, s), (rt, e')))
447     in
448     snd tmp
449 in
450
451 (* body of check_function *)
452 {
453     styp = func.typ;
454     sfname = func.fname;
455     sformals = func.formals;
456     sftype = get_fstype func;
457     slocals = bindings;
458     sbody =
459         (match check_stmt (Block func.body) with
460         | SBlock s1 -> s1
461         | _ -> raise (Failure "internal error: block didn't become a block?"));
462 }

```

```

463   in
464   let sfunc_decls = List.map check_function functions in
465   ((globals, List.rev !local_sfunc @ sfunc_decls), sfunc_decls)

```

## A.5 sast.ml

```

1  (* Semantically-checked Abstract Syntax Tree and functions for printing it *)
2
3  open Ast
4
5  type sexpr = typ * sx
6
7  and sfunc_decl = {
8      styp : typ;
9      sfname : string;
10     sftype : typ;
11     sformals : bind list;
12     slocals : bindtbl;
13     sbody : sstmt list;
14 }
15
16 and sx =
17   | SLiteral of int
18   | SFliteral of string
19   | SBoolLit of bool
20   | SStrLit of string
21   | SFunctionLit of sfunc_decl
22   | SIndexAssign of string * sexpr * sexpr
23   | SGetSlice of string * sexpr * sexpr
24   | SGetIndex of string * sexpr
25   | SId of string
26   | SBinop of sexpr * op * sexpr
27   | SUnop of uop * sexpr
28   | SAssign of string * sexpr
29   | SListExpr of sexpr list
30   | SCall of string * sexpr list
31   | SNoexpr
32
33 and sstmt =
34   | SBlock of sstmt list
35   | SExpr of sexpr
36   | SReturn of sexpr
37   | SIf of sexpr * sstmt * sstmt
38   | SFor of sexpr * sexpr * sexpr * sstmt
39   | SForin of sstmt * sstmt * sstmt * sexpr * sexpr * sexpr * sstmt
40   | SWhile of sexpr * sstmt
41   | SVarDecl of bind
42   | SVarDeclAssign of bind * sexpr

```

```

43
44 and bindtbl = (string, typ) Hashtbl.t
45
46 and sprogram = bind list * sfunc_decl list
47
48 (* Pretty-printing functions *)
49
50 let rec string_of_sexpr (t, e) =
51   "(" ^ string_of_typ t ^ " : "
52   ^ (match e with
53     | SLiteral l -> string_of_int l
54     | SBoolLit true -> "true"
55     | SBoolLit false -> "false"
56     | SFliteral l -> l
57     | SFunctionLit(l) -> string_of_typ l.styp ^ " " ^
58       l.sfname ^ "(" ^ String.concat ", " (List.map string_of_formal l.sformals) ^
59       ")\n{}\n"
60     | SStrLit(l) -> "\"" ^ l ^ "\""
61     | SId s -> s
62     | SIndexAssign (v, e1, e2) ->
63       v ^ "[" ^ string_of_sexpr e1 ^ "]" = " ^ string_of_sexpr e2
64     | SGetIndex (e1, e2) -> e1 ^ "[" ^ string_of_sexpr e2 ^ "]"
65     | SGetSlice(v, e1, e2) -> v ^ "[" ^ string_of_sexpr e1 ^ ":" ^ string_of_sexpr e2 ^
66       "]"
67     | SBinop (e1, o, e2) ->
68       string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
69     | SUnop (o, e) -> string_of_uop o ^ string_of_sexpr e
70     | SListExpr e1 ->
71       "[" ^ String.concat ", " (List.map string_of_sexpr e1) ^ "]"
72     | SAssign (v, e) -> v ^ " = " ^ string_of_sexpr e
73     | SCall (f, e1) ->
74       f ^ "(" ^ String.concat ", " (List.map string_of_sexpr e1) ^ ")"
75     | SNoexpr -> ""
76   ^ ")")
77
78 let rec string_of_sstmt = function
79   | SBlock stmts ->
80     "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
81   | SExpr expr -> string_of_sexpr expr ^ ";\n"
82   (* | SVarDecl(expr) -> "decldre " ^ string_of_sexpr expr ^ ";\n"; *)
83   | SReturn expr -> "return " ^ string_of_sexpr expr ^ ";\n"
84   | SIf (e, s, SBlock []) ->
85     "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
86   | SIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
87     string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
88   | SFor(e1, e2, e3, s) ->
89     "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
90     string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
91   | SForin(s1, s2, s3, e1, e2, e3, s) ->
92     string_of_sstmt s1 ^ string_of_sstmt s2 ^ string_of_sstmt s3 ^

```

```

92     "while(" ^ string_of_sexpr e1 ^ ") {\n" ^ string_of_sstmt s
93     ^ ", " ^ string_of_sexpr e2 ^ "}" ^ string_of_sexpr e3 ^ "}\n"
94 | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s
95 | SVarDecl(t, id) -> string_of_typ t ^ " " ^ id ^ ";\n"
96 | SVarDeclAssign((t, id), e) -> string_of_typ t ^ " " ^ id ^ " = " ^ string_of_sexpr e
    ^ ";\n"
97
98 let string_of_sfdecl fdecl =
99     string_of_typ fdecl.styp ^ " " ^
100     fdecl.sfname ^ "(" ^ String.concat ", " (List.map string_of_typ (List.map fst fdecl.
    sformals)) ^
101     ")\n{\n" ^
102     String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
103     "}\n"
104
105 let string_of_sprogram (vars, funcs) =
106     String.concat "" (List.map string_of_vdecl vars)
107     ^ "\n"
108     ^ String.concat "\n" (List.map string_of_sfdecl funcs)

```

## A.6 codegen.ml

```

1 (* Code generation: translate takes a semantically checked AST and
2 produces LLVM IR
3
4 LLVM tutorial: Make sure to read the OCaml version of the tutorial
5
6 http://llvm.org/docs/tutorial/index.html
7
8 Detailed documentation on the OCaml LLVM library:
9
10 http://llvm.moe/
11 http://llvm.moe/ocaml/
12
13 *)
14
15 module L = Llvmlib
16 module A = Ast
17 open Sast
18 module StringMap = Map.Make (String)
19
20 (* translate : Sast.program -> Llvmlib.module *)
21 let translate ((globals, functions), top_funcs) =
22     let context = L.global_context () in
23
24     (* Create the LLVM compilation module into which
25        we will generate code *)
26     let the_module = L.create_module context "FFBB" in

```

```

27
28 (* Get types from the context *)
29 let i32_t = L.i32_type context
30 and i8_t = L.i8_type context
31 and i1_t = L.i1_type context
32 and float_t = L.double_type context
33 and str_t = L.pointer_type (L.i8_type context)
34 and void_t = L.void_type context in
35 let list_t typ ltyp =
36   let typ_str = A.string_of_typ (A.CompositeType (A.LIST, [ typ ])) in
37   let t = L.named_struct_type context typ_str in
38   let list_body =
39     [| i32_t; i32_t; i32_t; L.pointer_type ltyp |]
40     (* length, type, capacity, pointer *)
41   in
42   ignore (L.struct_set_body t list_body false);
43   t
44 in
45 let list_int_t = list_t A.Int i32_t
46 and list_bool_t = list_t A.Bool i1_t
47 and list_float_t = list_t A.Float float_t in
48 let set_t =
49   let t = L.named_struct_type context "TreeSet" in
50   let set_body = [| i32_t; i32_t; L.pointer_type i8_t |] in
51   ignore (L.struct_set_body t set_body false);
52   t
53 in
54 let dict_t =
55   let t = L.named_struct_type context "TreeDict" in
56   let set_body = [| i32_t; i32_t; L.pointer_type i8_t |] in
57   ignore (L.struct_set_body t set_body false);
58   t
59 in
60 (* Return the LLVM type for a FFBB type *)
61 let rec ltype_of_typ = function
62 | A.Int -> i32_t
63 | A.Bool -> i1_t
64 | A.Float -> float_t
65 | A.Void -> void_t
66 | A.String -> str_t
67 | A.FunctionType (rt :: rest) ->
68   let llargs =
69     List.fold_left (fun l arg -> ltype_of_typ arg :: l) [] (List.rev rest)
70   in
71   L.pointer_type
72     (L.function_type (ltype_of_typ rt) (Array.of_list llargs))
73 | A.CompositeType (A.LIST, [ A.Int ]) -> L.pointer_type list_int_t
74 | A.CompositeType (A.LIST, [ A.Bool ]) -> L.pointer_type list_bool_t
75 | A.CompositeType (A.LIST, [ A.Float ]) -> L.pointer_type list_float_t
76 | A.CompositeType (A.SET, [ _ ]) -> L.pointer_type set_t

```



```

77 | A.CompositeType (A.DICT, [ _; _ ]) -> L.pointer_type dict_t
78 in
79
80 (* Create a map of global variables after creating each *)
81 let global_vars : L.llvalue StringMap.t =
82   let global_var m (t, n) =
83     let init =
84       match t with
85       | A.Float -> L.const_float (ltype_of_typ t) 0.0
86       | _ -> L.const_int (ltype_of_typ t) 0
87     in
88     StringMap.add n (L.define_global n init the_module) m
89   in
90   List.fold_left global_var StringMap.empty globals
91 in
92
93 let printf_t : L.lltype =
94   L.var_arg_function_type i32_t [| L.pointer_type i8_t |]
95 in
96 let printf_func : L.llvalue =
97   L.declare_function "printf" printf_t the_module
98 in
99
100 let printbig_t : L.lltype = L.function_type i32_t [| i32_t |] in
101 let printbig_func : L.llvalue =
102   L.declare_function "printbig" printbig_t the_module
103 in
104
105 let slice_t : L.lltype =
106   L.function_type str_t [| L.pointer_type i8_t; i32_t; i32_t |]
107 in
108 let slice_func : L.llvalue = L.declare_function "slice" slice_t the_module in
109
110 let length_t : L.lltype = L.function_type i32_t [| L.pointer_type i8_t |] in
111 let length_func : L.llvalue =
112   L.declare_function "length" length_t the_module
113 in
114
115 let concat_t : L.lltype =
116   L.function_type str_t [| L.pointer_type i8_t; L.pointer_type i8_t |]
117 in
118 let concat_func : L.llvalue =
119   L.declare_function "concat" concat_t the_module
120 in
121
122 let range_t : L.lltype =
123   L.function_type (L.pointer_type list_int_t) [| i32_t |]
124 in
125 let range_func : L.llvalue = L.declare_function "range" range_t the_module in
126

```

```

127 let create_empty_list_int_t : L.lltype =
128     L.function_type (L.pointer_type list_int_t) [| i32_t |]
129 in
130 let create_empty_list_int_func : L.llvalue = L.declare_function "create_empty_list_int
    " create_empty_list_int_t the_module in
131
132 let create_empty_list_bool_t : L.lltype =
133     L.function_type (L.pointer_type list_bool_t) [| i32_t |]
134 in
135 let create_empty_list_bool_func : L.llvalue = L.declare_function "
    create_empty_list_bool" create_empty_list_bool_t the_module in
136
137 let create_empty_list_float_t : L.lltype =
138     L.function_type (L.pointer_type list_float_t) [| i32_t |]
139 in
140 let create_empty_list_float_func : L.llvalue = L.declare_function "
    create_empty_list_float" create_empty_list_float_t the_module in
141
142 let len_t : L.lltype = L.function_type i32_t [| L.pointer_type i8_t |] in
143 let len_func : L.llvalue = L.declare_function "len" len_t the_module in
144
145 let append_t : L.lltype =
146     L.function_type i32_t [| L.pointer_type i8_t; L.pointer_type i8_t |]
147 in
148 let append_func : L.llvalue =
149     L.declare_function "append" append_t the_module
150 in
151
152 let idxtrans_t : L.lltype =
153     L.function_type i32_t [| L.pointer_type i8_t; i32_t |]
154 in
155 let idxtrans_func : L.llvalue =
156     L.declare_function "index_transform" idxtrans_t the_module
157 in
158
159 let list_slice_t : L.lltype =
160     L.function_type
161         (L.pointer_type list_int_t)
162         [| L.pointer_type i8_t; i32_t; i32_t |]
163 in
164 let list_slice_func : L.llvalue =
165     L.declare_function "list_slice" list_slice_t the_module
166 in
167
168 let create_empty_set_t : L.lltype =
169     L.function_type (L.pointer_type set_t) [| i32_t |]
170 in
171 let create_empty_set_func : L.llvalue =
172     L.declare_function "create_empty_set" create_empty_set_t the_module
173 in

```

```

174
175 let add_elem_t : L.lltype =
176     L.function_type i32_t [| L.pointer_type set_t; L.pointer_type i8_t |]
177 in
178 let add_elem_func : L.llvalue =
179     L.declare_function "add_elem" add_elem_t the_module
180 in
181
182 let search_elem_t : L.lltype =
183     L.function_type i1_t [| L.pointer_type set_t; L.pointer_type i8_t |]
184 in
185 let search_elem_func : L.llvalue =
186     L.declare_function "search_elem" search_elem_t the_module
187 in
188
189 let delete_elem_t : L.lltype =
190     L.function_type i32_t [| L.pointer_type set_t; L.pointer_type i8_t |]
191 in
192 let delete_elem_func : L.llvalue =
193     L.declare_function "delete_elem" delete_elem_t the_module
194 in
195
196 let get_set_size_t : L.lltype =
197     L.function_type i32_t [| L.pointer_type set_t |]
198 in
199 let get_set_size_func : L.llvalue =
200     L.declare_function "get_set_size" get_set_size_t the_module
201 in
202
203 let create_empty_dict_t : L.lltype =
204     L.function_type (L.pointer_type dict_t) [| i32_t; i32_t |]
205 in
206 let create_empty_dict_func : L.llvalue =
207     L.declare_function "create_empty_dict" create_empty_dict_t the_module
208 in
209
210 let add_key_value_t : L.lltype =
211     L.function_type i32_t
212     [| L.pointer_type dict_t; L.pointer_type i8_t; L.pointer_type i8_t |]
213 in
214 let add_key_value_func : L.llvalue =
215     L.declare_function "add_key_value" add_key_value_t the_module
216 in
217
218 let get_dict_size_t : L.lltype =
219     L.function_type i32_t [| L.pointer_type dict_t |]
220 in
221 let get_dict_size_func : L.llvalue =
222     L.declare_function "get_dict_size" get_dict_size_t the_module
223 in

```

```

224
225 let key_exists_t : L.lltype =
226     L.function_type i1_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
227 in
228 let key_exists_func : L.llvalue =
229     L.declare_function "key_exists" key_exists_t the_module
230 in
231
232 let retrieve_value_int_t : L.lltype =
233     L.function_type i32_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
234 in
235 let retrieve_value_int_func : L.llvalue =
236     L.declare_function "retrieve_value_int" retrieve_value_int_t the_module
237 in
238
239 let retrieve_value_float_t : L.lltype =
240     L.function_type float_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
241 in
242 let retrieve_value_float_func : L.llvalue =
243     L.declare_function "retrieve_value_float" retrieve_value_float_t the_module
244 in
245
246 let retrieve_value_bool_t : L.lltype =
247     L.function_type i1_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
248 in
249 let retrieve_value_bool_func : L.llvalue =
250     L.declare_function "retrieve_value_bool" retrieve_value_bool_t the_module
251 in
252
253 let retrieve_value_list_t : L.lltype =
254     L.function_type (L.pointer_type list_int_t) [| L.pointer_type dict_t; L.pointer_type
255         i8_t |]
256 in
257 let retrieve_value_list_func : L.llvalue =
258     L.declare_function "retrieve_value_list" retrieve_value_list_t the_module
259 in
260
261 let delete_key_t : L.lltype =
262     L.function_type i32_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
263 in
264 let delete_key_func : L.llvalue =
265     L.declare_function "delete_key" delete_key_t the_module
266 in
267
268 (* Define each function (arguments and return type) so we can
269    call it even before we've created its body *)
270 let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
271     let function_decl m fdecl =
272         let name = fdecl.sfname

```

```

273     Array.of_list (List.map (fun (t, _) -> ltype_of_typ t) fdecl.sformals)
274   in
275     let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
276     StringMap.add name (L.define_function name ftype the_module, fdecl) m
277   in
278     List.fold_left function_decl StringMap.empty functions
279   in
280
281   (* Fill in the body of the given function *)
282   let rec build_function_body fdecl =
283     let the_function, _ = StringMap.find fdecl.sfname function_decls in
284     let builder = L.builder_at_end context (L.entry_block the_function) in
285
286     let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
287     and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
288     and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
289
290     (* Construct the function's "locals": formal arguments and locally
291        declared variables.  Allocate each on the stack, initialize their
292        value, if appropriate, and remember their values in the "locals" map *)
293     let local_vars =
294       let local_vars_tbl = Hashtbl.create 10 in
295       let add_formal (t, n) p =
296         L.set_value_name n p;
297         let local = L.build_alloc (ltype_of_typ t) n builder in
298         ignore (L.build_store p local builder);
299         Hashtbl.add local_vars_tbl n local
300       in
301
302       List.iter2 add_formal fdecl.sformals
303         (Array.to_list (L.params the_function));
304       local_vars_tbl
305     in
306
307     (* Return the value for a variable or formal argument.
308        Check local names first, then global names *)
309     let lookup n =
310       try Hashtbl.find local_vars n
311       with Not_found -> StringMap.find n global_vars
312     in
313
314     let list_type_to_idx typ =
315       match typ with
316       | A.CompositeType (A.LIST, [ A.Int ]) -> 0
317       | A.CompositeType (A.LIST, [ A.Bool ]) -> 1
318       | A.CompositeType (A.LIST, [ A.Float ]) -> 2
319       (* | A.CompositeType (A.LIST, [ A.String ]) -> 3 *)
320       | _ -> raise (Failure "Invalid list type")
321     in
322     let list_prim_type typ =

```

```

323     match typ with
324     | A.CompositeType (A.LIST, [ t ]) -> t
325     | _ -> raise (Failure "Invalid list type")
326 in
327
328 let type_to_idx_map typ =
329     match typ with
330     | A.Int -> 0
331     | A.Bool -> 1
332     | A.Float -> 2
333     | A.CompositeType(LIST, _) -> 4
334     | _ ->
335         raise (Failure "Internal error. Prime type has to be int/bool/float")
336 in
337
338 (* Construct code for an expression; return its value *)
339 let rec expr builder ((et, e) : sexpr) =
340     let alloc_var_and_return_voidptr_from_expr e =
341         let e' = expr builder e in
342         let p_e' = L.build_alloca (L.type_of e') "ptmp" builder in
343         ignore (L.build_store e' p_e' builder);
344         L.build_bitcast p_e' (L.pointer_type i8_t) "tmpvoidptr" builder
345     in
346
347     match e with
348     | SLiteral i -> L.const_int i32_t i
349     | SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
350     | SFliteral l -> L.const_float_of_string float_t l
351     | SStrLit l -> L.build_global_stringptr l "string" builder
352     (*@ *)
353     (* | SFunctionLit s -> L.build_load (lookup s) s builder *)
354     | SNoexpr -> L.const_int i32_t 0
355     | SId s -> L.build_load (lookup s) s builder
356     | SFunctionLit sfunc ->
357         build_function_body sfunc;
358         let fdef, fdecl = StringMap.find sfunc.sfname function_decls in
359         fdef
360     | SListExpr el ->
361         let vl = List.map (expr builder) el in
362         let e_type = L.pointer_type (L.type_of (List.hd vl)) in
363         let size = L.const_int i32_t (List.length vl) in
364         let prim_type = list_prim_type et in
365         let list_typ =
366             match prim_type with
367             | A.Int -> list_int_t
368             | A.Bool -> list_bool_t
369             | A.Float -> list_float_t
370         in
371         let typ_idx = list_type_to_idx et in
372         (* Declare LIST *)

```

```

373 let list = L.build_array_malloc e_typ size "list" builder in
374 let list = L.build_pointercast list e_typ "list" builder in
375 let list_struct = L.build_malloc list_typ "list_struct" builder in
376 (* variable storing current size of list *)
377 let list_size =
378     L.build_struct_gep list_struct 0 "list_size" builder
379 in
380 ignore
381     (L.build_store
382         (L.const_int i32_t (List.length vl))
383         list_size builder);
384 (* variable storing type of list *)
385 let list_type =
386     L.build_struct_gep list_struct 1 "list_type" builder
387 in
388 ignore (L.build_store (L.const_int i32_t typ_idx) list_type builder);
389 (* variable storing capacity of list, used later by append() *)
390 let list_capacity =
391     L.build_struct_gep list_struct 2 "list_capacity" builder
392 in
393 ignore
394     (L.build_store
395         (L.const_int i32_t (List.length vl))
396         list_capacity builder);
397 (* assign actual list with values *)
398 let assign i v =
399     let vp =
400         L.build_gep list [| L.const_int i32_t i |] "list_v" builder
401     in
402     ignore (L.build_store v vp builder)
403 in
404 List.iteri assign vl;
405 (* pointer to the actual list *)
406 let list_ptr =
407     L.build_struct_gep list_struct 3 "list_struct" builder
408 in
409 ignore (L.build_store list list_ptr builder);
410 list_struct
411 | SIndexAssign (s, e1, e2) ->
412     let idx = expr builder e1
413     and value = expr builder e2
414     and list = L.build_load (lookup s) "getIndex" builder in
415     (* Cast list to void * to send to len function *)
416     let p_list = L.build_alloca (L.type_of list) "pcst" builder in
417     ignore (L.build_store list p_list builder);
418     let p_list =
419         L.build_bitcast p_list (L.pointer_type i8_t) "cs" builder
420     in
421     let idx =
422         L.build_call idxtrans_func [| p_list; idx |] "index_transform_call"

```

```

423         builder
424     in
425     (* Get list pointer from struct *)
426     let list = L.build_struct_gep list 3 "list_v" builder in
427     let list = L.build_load list "list_v" builder in
428     (* list[idx] *)
429     let list_idx = L.build_gep list [| idx |] "getIndex_e" builder in
430     ignore (L.build_store value list_idx builder);
431     value
432 | SAssign (s, e) ->
433     let e' = expr builder e in
434     ignore (L.build_store e' (lookup s) builder);
435     e'
436 | SGetIndex (s, e) ->
437     let idx = expr builder e
438     and list = L.build_load (lookup s) "getIndex" builder in
439     let p_list = L.build_alloca (L.type_of list) "pcst" builder in
440     ignore (L.build_store list p_list builder);
441     let p_list =
442         L.build_bitcast p_list (L.pointer_type i8_t) "cs" builder
443     in
444     let idx =
445         L.build_call idxtrans_func [| p_list; idx |] "index_transform_call"
446         builder
447     in
448     (* Get list pointer from struct *)
449     let list = L.build_struct_gep list 3 "list_v" builder in
450     let list = L.build_load list "list_v" builder in
451     let sp = L.build_gep list [| idx |] "getIndex_e" builder in
452     L.build_load sp "getIndex" builder
453 | SGetSlice (s, e1, e2) ->
454     let l = expr builder e1
455     and r = expr builder e2
456     and list = L.build_load (lookup s) "getList" builder in
457     (* Cast list to void * to send to len function *)
458     let p_list = L.build_alloca (L.type_of list) "pcst" builder in
459     ignore (L.build_store list p_list builder);
460     let p_list =
461         L.build_bitcast p_list (L.pointer_type i8_t) "cs" builder
462     in
463     (* Get list pointer from struct *)
464     L.build_call list_slice_func [| p_list; l; r |] "list_slice_call"
465     builder
466 | SBinop (((A.Float, _) as e1), op, e2) ->
467     let e1' = expr builder e1 and e2' = expr builder e2 in
468     (match op with
469     | A.Add -> L.build_fadd
470     | A.Sub -> L.build_fsub
471     | A.Mult -> L.build_fmuls
472     | A.Div -> L.build_fdiv

```



```

473 | A.Equal -> L.build_fcmp L.Fcmp.Oeq
474 | A.Neq -> L.build_fcmp L.Fcmp.One
475 | A.Less -> L.build_fcmp L.Fcmp.Olt
476 | A.Leq -> L.build_fcmp L.Fcmp.Ole
477 | A.Greater -> L.build_fcmp L.Fcmp.Ogt
478 | A.Geq -> L.build_fcmp L.Fcmp.Oge
479 | A.And | A.Or ->
480 |     raise
481 |     (Failure
482 |      "internal error: semant should have rejected and/or on float"))
483 |     e1' e2' "tmp" builder
484 | SBinop (e1, op, e2) ->
485 |     let e1' = expr builder e1 and e2' = expr builder e2 in
486 |     (match op with
487 |     | A.Add -> L.build_add
488 |     | A.Sub -> L.build_sub
489 |     | A.Mult -> L.build_mul
490 |     | A.Div -> L.build_sdiv
491 |     | A.And -> L.build_and
492 |     | A.Or -> L.build_or
493 |     | A.Equal -> L.build_icmp L.Icmp.Eq
494 |     | A.Neq -> L.build_icmp L.Icmp.Ne
495 |     | A.Less -> L.build_icmp L.Icmp.Slt
496 |     | A.Leq -> L.build_icmp L.Icmp.Sle
497 |     | A.Greater -> L.build_icmp L.Icmp.Sgt
498 |     | A.Geq -> L.build_icmp L.Icmp.Sge)
499 |     e1' e2' "tmp" builder
500 | SUnop (op, ((t, _) as e)) ->
501 |     let e' = expr builder e in
502 |     (match op with
503 |     | A.Neg when t = A.Float -> L.build_fneg
504 |     | A.Neg -> L.build_neg
505 |     | A.Not -> L.build_not)
506 |     e' "tmp" builder
507 | SCall ("print", [ e ]) | SCall ("printb", [ e ]) ->
508 |     L.build_call printf_func
509 |     [| int_format_str; expr builder e |]
510 |     "printf" builder
511 | SCall ("printbig", [ e ]) ->
512 |     L.build_call printbig_func [| expr builder e |] "printbig" builder
513 | SCall ("printf", [ e ]) ->
514 |     L.build_call printf_func
515 |     [| float_format_str; expr builder e |]
516 |     "printf" builder
517 | SCall ("prints", [ e ]) ->
518 |     L.build_call printf_func
519 |     [| string_format_str; expr builder e |]
520 |     "printf" builder
521 | SCall ("slice", [ e; e1; e2 ]) ->
522 |     L.build_call slice_func

```

```

523     [| expr builder e; expr builder e1; expr builder e2 |]
524     "slice" builder
525 | SCall ("length", [ e ]) ->
526     L.build_call length_func [| expr builder e |] "length" builder
527 | SCall ("concat", [ e; e1 ]) ->
528     L.build_call concat_func
529     [| expr builder e; expr builder e1 |]
530     "concat" builder
531 | SCall ("range", [ e ]) ->
532     L.build_call range_func [| expr builder e |] "range_call" builder
533 | SCall ("len", [ e ]) ->
534     let e' = expr builder e in
535     (* Cast list to void * to send to len function *)
536     let p_e' = L.build_alloca (L.type_of e') "pcst" builder in
537     ignore (L.build_store e' p_e' builder);
538     let e' = L.build_bitcast p_e' (L.pointer_type i8_t) "cs" builder in
539     L.build_call len_func [| e' |] "list_len_call" builder
540 | SCall ("append", [ e1; e2 ]) ->
541     let e1' = expr builder e1 in
542     let e2' = expr builder e2 in
543     (* Cast list to void * to send to append function *)
544     let p_e1' = L.build_alloca (L.type_of e1') "pcst" builder in
545     ignore (L.build_store e1' p_e1' builder);
546     let p_e2' = L.build_alloca (L.type_of e2') "pcst" builder in
547     ignore (L.build_store e2' p_e2' builder);
548     let e1' = L.build_bitcast p_e1' (L.pointer_type i8_t) "cs" builder in
549     let e2' = L.build_bitcast p_e2' (L.pointer_type i8_t) "cs" builder in
550     L.build_call append_func [| e1'; e2' |] "list_append_call" builder
551 | SCall ("setAdd", [ se; e ]) ->
552     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
553     L.build_call add_elem_func
554     [| expr builder se; e_voidptr |]
555     "add_elem" builder
556 | SCall ("setFind", [ se; e ]) ->
557     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
558     L.build_call search_elem_func
559     [| expr builder se; e_voidptr |]
560     "search_elem" builder
561 | SCall ("setRemove", [ se; e ]) ->
562     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
563     L.build_call delete_elem_func
564     [| expr builder se; e_voidptr |]
565     "delete_elem" builder
566 | SCall ("setSize", [ se ]) ->
567     L.build_call get_set_size_func
568     [| expr builder se |]
569     "get_set_size" builder
570 | SCall ("dictAdd", [ de; e1; e2 ]) ->
571     let e1_voidptr = alloc_var_and_return_voidptr_from_expr e1 in
572     let e2_voidptr = alloc_var_and_return_voidptr_from_expr e2 in

```

```

573     L.build_call add_key_value_func
574     [| expr builder de; e1_voidptr; e2_voidptr |]
575     "add_key_value" builder
576 | SCall ("dictSize", [ de ]) ->
577     L.build_call get_dict_size_func
578     [| expr builder de |]
579     "get_dict_size" builder
580 | SCall ("dictHasKey", [ de; e ]) ->
581     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
582     L.build_call key_exists_func
583     [| expr builder de; e_voidptr |]
584     "key_exists" builder
585 | SCall ("dictGetInt", [ de; e ]) ->
586     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
587     L.build_call retrieve_value_int_func
588     [| expr builder de; e_voidptr |]
589     "retrieve_value_int" builder
590 | SCall ("dictGetBool", [ de; e ]) ->
591     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
592     L.build_call retrieve_value_bool_func
593     [| expr builder de; e_voidptr |]
594     "retrieve_value_bool" builder
595 | SCall ("dictGetFloat", [ de; e ]) ->
596     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
597     L.build_call retrieve_value_float_func
598     [| expr builder de; e_voidptr |]
599     "retrieve_value_float" builder
600 | SCall ("dictGetList", [ de; e ]) ->
601     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
602     L.build_call retrieve_value_list_func
603     [| expr builder de; e_voidptr |]
604     "retrieve_value_list" builder
605 | SCall ("dictRemove", [ de; e ]) ->
606     let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
607     L.build_call delete_key_func
608     [| expr builder de; e_voidptr |]
609     "delete_key" builder
610 | SCall (f, args) ->
611     let result = match et with A.Void -> "" | _ -> f ^ "_result" in
612     let llargs = List.rev (List.map (expr builder) (List.rev args)) in
613     let fdef =
614         if StringMap.mem f function_decls then
615             fst (StringMap.find f function_decls)
616         else L.build_load (lookup f) f builder
617     in
618     L.build_call fdef (Array.of_list llargs) result builder
619 in
620
621 (* LLVM insists each basic block end with exactly one "terminator"
622    instruction that transfers control. This function runs "instr builder"

```

```

623     if the current block does not already have a terminator. Used,
624     e.g., to handle the "fall off the end of the function" case. *)
625 let add_terminal builder instr =
626     match L.block_terminator (L.insertion_block builder) with
627     | Some _ -> ()
628     | None -> ignore (instr builder)
629 in
630
631 (* Build the code for the given statement; return the builder for
632    the statement's successor (i.e., the next instruction will be built
633    after the one generated by this call) *)
634 let rec stmt builder = function
635 | SBlock sl -> List.fold_left stmt builder sl
636 | SExpr e ->
637     ignore (expr builder e);
638     builder
639 | SReturn e ->
640     ignore
641     (match fdecl.styp with
642     (* Special "return nothing" instr *)
643     | A.Void -> L.build_ret_void builder (* Build return statement *)
644     | _ -> L.build_ret (expr builder e) builder);
645     builder
646 | SIf (predicate, then_stmt, else_stmt) ->
647     let bool_val = expr builder predicate in
648     let merge_bb = L.append_block context "merge" the_function in
649     let build_br_merge = L.build_br merge_bb in
650
651     (* partial function *)
652     let then_bb = L.append_block context "then" the_function in
653     add_terminal
654     (stmt (L.builder_at_end context then_bb) then_stmt)
655     build_br_merge;
656
657     let else_bb = L.append_block context "else" the_function in
658     add_terminal
659     (stmt (L.builder_at_end context else_bb) else_stmt)
660     build_br_merge;
661
662     ignore (L.build_cond_br bool_val then_bb else_bb builder);
663     L.builder_at_end context merge_bb
664 | SWhile (predicate, body) ->
665     let pred_bb = L.append_block context "while" the_function in
666     ignore (L.build_br pred_bb builder);
667
668     let body_bb = L.append_block context "while_body" the_function in
669     add_terminal
670     (stmt (L.builder_at_end context body_bb) body)
671     (L.build_br pred_bb);
672

```

```

673     let pred_builder = L.builder_at_end context pred_bb in
674     let bool_val = expr pred_builder predicate in
675
676     let merge_bb = L.append_block context "merge" the_function in
677     ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
678     L.builder_at_end context merge_bb
679 (* Implement for loops as while loops *)
680 | SFor (e1, e2, e3, body) ->
681     stmt builder
682     (SBlock [ SExpr e1; SWhile (e2, SBlock [ body; SExpr e3 ]) ])
683 | SForin (iter, clist, x, pred, inc, next, body) ->
684     stmt builder
685     (SBlock
686     [
687         iter;
688         clist;
689         x;
690         SWhile (pred, SBlock [ body; SExpr inc; SExpr next ]);
691     ])
692 | SVarDecl (t, id) ->
693     let local_var = L.build_alloca (ltype_of_typ t) id builder in
694     Hashtbl.add local_vars id local_var;
695     ignore
696     (match t with
697     | CompositeType (LIST, [ Int ]) ->
698         ignore
699         (L.build_store
700         (L.build_call create_empty_list_int_func
701         [ L.const_int i32_t (type_to_idx_map Int) ]
702         "create_empty_list_int" builder)
703         local_var builder)
704     | CompositeType (LIST, [ Bool ]) ->
705         ignore
706         (L.build_store
707         (L.build_call create_empty_list_bool_func
708         [ L.const_int i32_t (type_to_idx_map Bool) ]
709         "create_empty_list_bool" builder)
710         local_var builder)
711     | CompositeType (LIST, [ Float ]) ->
712         ignore
713         (L.build_store
714         (L.build_call create_empty_list_float_func
715         [ L.const_int i32_t (type_to_idx_map Float) ]
716         "create_empty_list_float" builder)
717         local_var builder)
718     | CompositeType (SET, [ prim_typ ]) ->
719         ignore
720         (L.build_store
721         (L.build_call create_empty_set_func
722         [ L.const_int i32_t (type_to_idx_map prim_typ) ]

```

```

723         "create_empty_set" builder)
724         local_var builder)
725     | CompositeType (DICT, [ key_typ; value_typ ]) ->
726         ignore
727             (L.build_store
728                 (L.build_call create_empty_dict_func
729                     [|
730                         L.const_int i32_t (type_to_idx_map key_typ);
731                         L.const_int i32_t (type_to_idx_map value_typ);
732                     |]
733                     "create_empty_dict" builder)
734                 local_var builder)
735     | _ -> ());
736     builder
737 | SVarDeclAssign ((t, id), e) ->
738     let local_var = L.build_alloca (ltype_of_typ t) id builder in
739     Hashtbl.add local_vars id local_var;
740     let e' = expr builder e in
741     ignore (L.build_store e' (lookup id) builder);
742     builder
743 in
744
745 (* Build the code for each statement in the function *)
746 let builder = stmt builder (SBlock fdecl.sbody) in
747
748 (* Add a return if the last block falls off the end *)
749 add_terminal builder
750 (match fdecl.styp with
751 | A.Void -> L.build_ret_void
752 | A.Float -> L.build_ret (L.const_float float_t 0.0)
753 | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
754 in
755
756 List.iter build_function_body top_funcs;
757 the_module

```

## A.7 FFBB.ml

```

1 (* Top-level of the FFBB compiler: scan & parse the input,
2    check the resulting AST and generate an SAST from it, generate LLVM IR,
3    and dump the module *)
4
5 type action = Ast | Sast | LLVM_IR | Compile
6
7 let () =
8     let action = ref Compile in
9     let set_action a () = action := a in
10    let speclist = [

```

```

11  ("a", Arg.Unit (set_action Ast), "Print the AST");
12  ("-s", Arg.Unit (set_action Sast), "Print the SAST");
13  ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
14  ("-c", Arg.Unit (set_action Compile),
15     "Check and print the generated LLVM IR (default)");
16  ] in
17  let usage_msg = "usage: ./FFBB.native [-a|-s|-l|-c] [file.mc]" in
18  let channel = ref stdin in
19  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
20
21  let lexbuf = Lexing.from_channel !channel in
22  let ast = FFBBparser.program Scanner.token lexbuf in
23  match !action with
24  | Ast -> print_string (Ast.string_of_program ast)
25  | _ -> let sast = Semant.check ast in
26     match !action with
27     | Ast -> ()
28     | Sast -> print_string (Sast.string_of_sprogram (fst sast))
29     | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
30     | Compile -> let m = Codegen.translate sast in
31  Llvm_analysis.assert_valid_module m;
32  print_string (Llvm.string_of_llmodule m)

```

## A.8 stringLibrary.c

```

1  #include <string.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5  #include <errno.h>
6
7
8  char *slice(char *src, int begin, int end){
9
10     int i;
11     char* res = malloc((end - begin+1)*sizeof(char));
12
13     if (!res)
14         goto exit;
15
16     if (begin>end){
17         printf("Runtime Error: Slice begin integer %d is greater than end integer %d,",
18             begin, end);
19         goto exit;
20     }
21
22     int idx = 0;
23     for (i=begin;i<end&&src[i]!='\0' ; i++){

```

```

23     res[idx++] = src[i];
24 }
25
26 res[idx] = '\0';
27 return res;
28
29
30 exit:
31     free(res);
32     return NULL;
33 }
34
35
36
37 int length(char *src){
38     return strlen(src);
39 }
40
41
42 char *concat(char *a, char *b){
43     int len1 = strlen(a);
44     int len2 = strlen(b);
45     char *res = malloc(len1 + len2 + 1);
46     if (!res)
47         goto exit;
48
49     memcpy(res, a, len1);
50     memcpy(res + len1, b, len2 + 1);
51     return res;
52
53
54
55 exit:
56     free(res);
57     return NULL;
58 }

```

## A.9 list.h

```

1 #ifndef LIST_H
2 #define LIST_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7
8
9 // List Struct

```



```

10 typedef struct LIST {
11     int size;
12     int type;
13     int capacity;
14     void *ptr;
15 } List;
16
17 int len(void *_list);
18
19 void append(void *_list, void *value);
20
21 struct LIST * range(int n);
22
23 struct LIST * create_empty_list(int type);
24
25 #endif

```

## A.10 list.c

```

1 #include "list.h"
2
3
4 int len(void *_list) {
5     struct LIST *list = *(struct LIST **) _list;
6     // printf("here in len\n");
7     // printf("len=%d\n", list->size);
8     // printf("type=%d\n", list->type);
9     // printf("capacity=%d\n", list->capacity);
10    return list->size;
11 }
12
13 struct LIST * range(int n) {
14     struct LIST *list = malloc(sizeof(struct LIST));
15     int *arr = (int *) malloc(n * sizeof(int));
16     for (int i = 0; i < n; i++)
17         arr[i] = i;
18     list->size = n;
19     list->type = 0;
20     list->capacity = n;
21     list->ptr = (void *) arr;
22     return list;
23 }
24
25
26
27 struct LIST * create_empty_list(int type) {
28     struct LIST *list = malloc(sizeof(struct LIST));
29     // int *arr = (int *) malloc(n * sizeof(int));

```

```

30     list->size = 0;
31     list->type = type;
32     list->capacity = 0;
33     // list->ptr = (void *) arr;
34     return list;
35 }
36
37 struct LIST * create_empty_list_int(int type) {
38     return create_empty_list(type);
39 }
40
41 struct LIST * create_empty_list_bool(int type) {
42     return create_empty_list(type);
43 }
44
45 struct LIST * create_empty_list_float(int type) {
46     return create_empty_list(type);
47 }
48
49
50 int index_transform(void *_list, int i) {
51     if (i < 0) {
52         i += len(_list);
53     }
54     return i;
55 }
56
57
58 // l, r inclusive
59 struct LIST * list_slice(void *_list, int l, int r) {
60     struct LIST *list = *(struct LIST **) _list;
61     l = index_transform(_list, l);
62     r = index_transform(_list, r);
63     int n = r - l + 1;
64     struct LIST *new_list = malloc(sizeof(struct LIST));
65     void *new_arr;
66     void *old_list_ptr = list->ptr;
67     // extend capacity
68     if (list->type == 0) {
69         new_arr = (void *) malloc(n * sizeof(int));
70         // copy over old values
71         for (int i = 0; i < n; i++)
72             ((int *)new_arr)[i] = ((int *)old_list_ptr)[l+i];
73     }
74     else if (list->type == 1) {
75         new_arr = (void *) malloc(n * sizeof(bool));
76         // copy over old values
77         for (int i = 0; i < n; i++)
78             ((bool *)new_arr)[i] = ((bool *)old_list_ptr)[l+i];
79     }

```

```

80     else if (list->type == 2) {
81         new_arr = (void *) malloc(n * sizeof(double));
82         // copy over old values
83         for (int i = 0; i < n; i++)
84             ((double *)new_arr)[i] = ((double *)old_list_ptr)[i+i];
85     }
86     new_list->size = n;
87     new_list->type = list->type;
88     new_list->capacity = n;
89     new_list->ptr = (void *) new_arr;
90     return new_list;
91 }
92
93 void append(void *_list, void *value) {
94     struct LIST *list = *(struct LIST **) _list;
95
96     // capacity not enough
97     if (list->size >= list->capacity) {
98         void *old_list_ptr = list->ptr;
99         void *new_list_ptr;
100        // extend capacity
101        list->capacity *= 2;
102        if (list->type == 0) {
103            new_list_ptr = (void *) malloc(list->capacity * sizeof(int));
104            // copy over old values
105            for (int i = 0; i < list->size; i++)
106                ((int *)new_list_ptr)[i] = ((int *)old_list_ptr)[i];
107        }
108        else if (list->type == 1) {
109            new_list_ptr = (void *) malloc(list->capacity * sizeof(bool));
110            // copy over old values
111            for (int i = 0; i < list->size; i++)
112                ((bool *)new_list_ptr)[i] = ((bool *)old_list_ptr)[i];
113        }
114        else if (list->type == 2) {
115            new_list_ptr = (void *) malloc(list->capacity * sizeof(double));
116            // copy over old values
117            for (int i = 0; i < list->size; i++)
118                ((double *)new_list_ptr)[i] = ((double *)old_list_ptr)[i];
119        }
120
121        list->ptr = new_list_ptr;
122        free(old_list_ptr);
123    }
124    if (list->type == 0)
125        ((int *)list->ptr)[list->size++] = *(int *) value;
126    else if (list->type == 1)
127        ((bool *)list->ptr)[list->size++] = *(bool *) value;
128    else if (list->type == 2)
129        ((double *)list->ptr)[list->size++] = *(double *) value;

```

```

130
131 // printf("here in append\n");
132 // printf("len=%d\n", list->size);
133 // printf("type=%d\n", list->type);
134 // printf("capacity=%d\n", list->capacity);
135 }
136
137
138 #ifdef BUILD_TEST
139 int main()
140 {
141     printf("here\n");
142     struct LIST *list = malloc(sizeof(struct LIST));
143     int *arr = (int *) malloc(5 * sizeof(int));
144     arr[0] = 6;
145     arr[1] = 3;
146     arr[2] = 9;
147     arr[3] = 8;
148     list->size = 4;
149     list->type = 0;
150     list->capacity = 4;
151     list->ptr = (void *) arr;
152     printf("Len(list) = %d\n", len(list));
153     for (int i = 0; i < len(list); i++)
154         printf("list->ptr[i]=%d\n", ((int *) (list->ptr))[i]);
155     // printf("Len(list) = %d\n", list->size);
156 }
157 #endif

```

## A.11 treebasics.h

```

1 #ifndef DICTSET_H
2 #define DICTSET_H
3
4 #include <stdlib.h>
5 #include <stdio.h>
6 #include <stdbool.h>
7 #include "list.h"
8
9 #define is_left_child(x) (x->parent->lc == x)
10
11 union Value
12 {
13     int i;
14     float f;
15     bool b;
16     struct LIST* l;
17 };

```

```

18
19 struct TreeNode
20 {
21     union Value key;
22     union Value value;
23     struct TreeNode* parent;
24     struct TreeNode* lc;
25     struct TreeNode* rc;
26 };
27
28 union Value parse_value(void* value_ptr, int type_id);
29
30 bool uvalue_equal(union Value uvalue1, union Value uvalue2, int type_id);
31
32 bool uvalue_lt(union Value uvalue1, union Value uvalue2, int type_id);
33
34 #endif

```

## A.12 treebasics.c

```

1 #include "treebasics.h"
2
3 union Value parse_value(void* value_ptr, int type_id)
4 {
5     // printf("Enter parse_value");
6     union Value uvalue;
7     switch (type_id)
8     {
9         case 0 :
10            uvalue.i = *(int*)value_ptr;
11            break;
12        case 1 :
13            uvalue.b = *(bool*)value_ptr;
14            break;
15        case 2 :
16            uvalue.f = *(double*)value_ptr;
17            break;
18        case 4 :
19            uvalue.l = *(struct LIST**)value_ptr;
20            break;
21    }
22    // printf("Leave parse_value");
23    return uvalue;
24 }
25
26 bool uvalue_equal(union Value uvalue1, union Value uvalue2, int type_id)
27 {
28     switch (type_id)

```

```

29     {
30         case 0 :
31             return uvalue1.i == uvalue2.i;
32         case 1 :
33             return uvalue1.b == uvalue2.b;
34         case 2 :
35             return uvalue1.f == uvalue2.f;
36     }
37 }
38
39 bool uvalue_lt(union Value uvalue1, union Value uvalue2, int type_id)
40 {
41     switch (type_id)
42     {
43         case 0 :
44             return uvalue1.i < uvalue2.i;
45         case 1 :
46             return (int)uvalue1.b < (int)uvalue2.b;
47         case 2 :
48             return uvalue1.f < uvalue2.f;
49     }
50 }

```

## A.13 treedict.c

```

1 #include "treebasics.h"
2
3 struct TreeDict
4 {
5     int size;
6     int key_type; // 0: int, 1: bool, 2: double, 3: string
7     int value_type; // 0: int, 1: bool, 2: double, 3: string, 4: list
8     struct TreeNode* root;
9 };
10
11 struct TreeDict* create_empty_dict(int key_type_id, int value_type_id)
12 {
13     struct TreeDict* treedict = (struct TreeDict*)malloc(sizeof(struct TreeDict));
14     treedict->size = 0;
15     treedict->key_type = key_type_id;
16     treedict->value_type = value_type_id;
17     treedict->root = NULL;
18     return treedict;
19 }
20
21 struct TreeNode* create_dict_node(struct TreeNode* parent, union Value key, union Value
    value)
22 {

```

```

23     struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
24     node->key = key;
25     node->value = value;
26     node->parent = parent;
27     node->lc = NULL;
28     node->rc = NULL;
29     return node;
30 }
31
32 void add_key_value(struct TreeDict* treedict, void* key_ptr, void* value_ptr)
33 {
34     union Value ukey = parse_value(key_ptr, treedict->key_type);
35     union Value uvalue = parse_value(value_ptr, treedict->value_type);
36     if (treedict->root == NULL)
37     {
38         treedict->root = create_dict_node(NULL, ukey, uvalue);
39         treedict->size = 1;
40         return;
41     }
42     struct TreeNode* curr_node = treedict->root;
43     while (true)
44     {
45         if (uvalue_equal(ukey, curr_node->key, treedict->key_type))
46         {
47             curr_node->value = uvalue;
48             return;
49         }
50         if (uvalue_lt(ukey, curr_node->key, treedict->key_type))
51         {
52             if (curr_node->lc == NULL)
53             {
54                 curr_node->lc = create_dict_node(curr_node, ukey, uvalue);
55                 treedict->size++;
56                 return;
57             }
58             else
59             {
60                 curr_node = curr_node->lc;
61             }
62         }
63         else
64         {
65             if (curr_node->rc == NULL)
66             {
67                 curr_node->rc = create_dict_node(curr_node, ukey, uvalue);
68                 treedict->size++;
69                 return;
70             }
71             else
72             {

```

```

73         curr_node = curr_node->rc;
74     }
75 }
76 }
77 }
78
79 struct TreeNode* search_key_internal(struct TreeDict* treedict, union Value ukey)
80 {
81     if (treedict->root == NULL) return NULL;
82     struct TreeNode* curr_node = treedict->root;
83     while (true)
84     {
85         if (uvalue_equal(ukey, curr_node->key, treedict->key_type)) return curr_node;
86         if (uvalue_lt(ukey, curr_node->key, treedict->key_type))
87         {
88             if (curr_node->lc == NULL) return NULL;
89             else
90             {
91                 curr_node = curr_node->lc;
92             }
93         }
94         else
95         {
96             if (curr_node->rc == NULL) return NULL;
97             else
98             {
99                 curr_node = curr_node->rc;
100            }
101        }
102    }
103 }
104
105 bool key_exists(struct TreeDict* treedict, void* key_ptr)
106 {
107     union Value ukey = parse_value(key_ptr, treedict->key_type);
108     struct TreeNode* curr_node = search_key_internal(treedict, ukey);
109     return (curr_node != NULL);
110 }
111
112 union Value retrieve_value_union(struct TreeDict* treedict, void* key_ptr)
113 {
114     union Value ukey = parse_value(key_ptr, treedict->key_type);
115     struct TreeNode* curr_node = search_key_internal(treedict, ukey);
116     if (curr_node == NULL)
117     {
118         printf("Error! Key %d does not exists\n.", ukey.i);
119         exit(-1);
120     }
121     return curr_node->value;
122 }

```



```

123
124 int retrieve_value_int(struct TreeDict* treedict, void* key_ptr)
125 {
126     return retrieve_value_union(treedict, key_ptr).i;
127 }
128
129 double retrieve_value_float(struct TreeDict* treedict, void* key_ptr)
130 {
131     double result = retrieve_value_union(treedict, key_ptr).f;
132     return result;
133 }
134
135 bool retrieve_value_bool(struct TreeDict* treedict, void* key_ptr)
136 {
137     return retrieve_value_union(treedict, key_ptr).b;
138 }
139
140 struct LIST* retrieve_value_list(struct TreeDict* treedict, void* key_ptr)
141 {
142     return retrieve_value_union(treedict, key_ptr).l;
143 }
144
145 void delete_key(struct TreeDict* treedict, void* key_ptr)
146 {
147     union Value ukey = parse_value(key_ptr, treedict->key_type);
148     // reference: https://blog.csdn.net/zxnsirius/article/details/52131433
149     struct TreeNode* curr_node = search_key_internal(treedict, ukey);
150     if (curr_node == NULL) return;
151     treedict->size--;
152     if (curr_node->lc == NULL)
153     {
154         if (curr_node == treedict->root) // root node
155         {
156             treedict->root = curr_node->rc;
157             if (treedict->root != NULL) treedict->root->parent = NULL;
158         }
159         else if (curr_node->rc == NULL) // leaf node
160         {
161             if (is_left_child(curr_node)) curr_node->parent->lc = NULL;
162             else curr_node->parent->rc = NULL;
163         }
164         else // middle node
165         {
166             if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->rc;
167             else curr_node->parent->rc = curr_node->rc;
168             curr_node->rc->parent = curr_node->parent;
169         }
170     }
171     else if (curr_node->rc == NULL)
172     {

```

```

173     if (curr_node == treedict->root)
174     {
175         treedict->root = curr_node->lc;
176         if (treedict->root != NULL) treedict->root->parent = NULL;
177     }
178     else
179     {
180         if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->lc;
181         else curr_node->parent->rc = curr_node->lc;
182         curr_node->lc->parent = curr_node->parent;
183     }
184 }
185 else // both left and right child exist, find the largest element that is smaller
186     than the deleted element
187     {
188         struct TreeNode* left_neighbor = curr_node->lc;
189         while (true)
190         {
191             if (left_neighbor->rc != NULL) left_neighbor = left_neighbor->rc;
192             else break;
193         }
194         if (left_neighbor->lc == NULL) // leaf node
195         {
196             if (is_left_child(left_neighbor)) left_neighbor->parent->lc = NULL;
197             else left_neighbor->parent->rc = NULL;
198         }
199         else
200         {
201             if (is_left_child(left_neighbor)) left_neighbor->parent->lc = left_neighbor
202             ->lc;
203             else left_neighbor->parent->rc = left_neighbor->lc;
204             left_neighbor->lc->parent = left_neighbor->parent;
205         }
206         curr_node->key = left_neighbor->key;
207         curr_node->value = left_neighbor->value;
208     }
209 }
210 int get_dict_size(struct TreeDict* treedict)
211 {
212     return treedict->size;
213 }
214
215 // int main(void)
216 // {
217 //     struct TreeDict* dict1 = create_empty_dict(0, 4);
218 //     int i3 = 3;
219 //     int i5 = 5;
220 //     struct LIST* list1 = create_empty_list(0);

```

```

221 //     struct LIST* list2 = range(5);
222 //     struct LIST* list3 = range(3);
223 //     add_key_value(dict1, (void*)&i3, (void*)&list1));
224 //     add_key_value(dict1, (void*)&i5, (void*)&list2));
225 //     add_key_value(dict1, (void*)&i3, (void*)&list3));
226 //     int dict1_size = get_dict_size(dict1);
227 //     printf("dict1 size: %d\n", dict1_size);
228 //     struct LIST* value1 = retrieve_value_list(dict1, (void*)&i3));
229 //     printf("retrieved list size: %d\n", len((void*)&value1));
230 //     append((void*)&value1, (void*)&i5));
231 //     struct LIST* value2 = retrieve_value_list(dict1, (void*)&i3));
232 //     printf("retrieved list size: %d\n", len((void*)&value2));
233 //     return 0;
234 // }

```

## A.14 treeset.c

```

1 #include "treebasics.h"
2
3 struct TreeSet
4 {
5     int size;
6     int value_type; // 0: int, 1: bool, 2: float, 3: string
7     struct TreeNode* root;
8 };
9
10 struct TreeSet* create_empty_set(int type_id)
11 {
12     struct TreeSet* treeset = (struct TreeSet*)malloc(sizeof(struct TreeSet));
13     treeset->size = 0;
14     treeset->value_type = type_id;
15     treeset->root = NULL;
16     return treeset;
17 }
18
19 struct TreeNode* create_node(struct TreeNode* parent, union Value value)
20 {
21     struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
22     node->value = value;
23     node->parent = parent;
24     node->lc = NULL;
25     node->rc = NULL;
26     return node;
27 }
28
29 void add_elem(struct TreeSet* treeset, void* value_ptr)
30 {
31     union Value uvalue = parse_value(value_ptr, treeset->value_type);

```

```

32     if (treeset->root == NULL)
33     {
34         treeset->root = create_node(NULL, uvalue);
35         treeset->size = 1;
36         return;
37     }
38     struct TreeNode* curr_node = treeset->root;
39     while (true)
40     {
41         if (uvalue_equal(uvalue, curr_node->value, treeset->value_type)) return;
42         if (uvalue_lt(uvalue, curr_node->value, treeset->value_type))
43         {
44             if (curr_node->lc == NULL)
45             {
46                 curr_node->lc = create_node(curr_node, uvalue);
47                 treeset->size++;
48                 return;
49             }
50             else
51             {
52                 curr_node = curr_node->lc;
53             }
54         }
55         else
56         {
57             if (curr_node->rc == NULL)
58             {
59                 curr_node->rc = create_node(curr_node, uvalue);
60                 treeset->size++;
61                 return;
62             }
63             else
64             {
65                 curr_node = curr_node->rc;
66             }
67         }
68     }
69 }
70
71 struct TreeNode* search_elem_internal(struct TreeSet* treeset, union Value uvalue)
72 {
73     if (treeset->root == NULL) return NULL;
74     struct TreeNode* curr_node = treeset->root;
75     while (true)
76     {
77         if (uvalue_equal(uvalue, curr_node->value, treeset->value_type)) return
curr_node;
78         if (uvalue_lt(uvalue, curr_node->value, treeset->value_type))
79         {
80             if (curr_node->lc == NULL) return NULL;

```

```

81     else
82     {
83         curr_node = curr_node->lc;
84     }
85 }
86 else
87 {
88     if (curr_node->rc == NULL) return NULL;
89     else
90     {
91         curr_node = curr_node->rc;
92     }
93 }
94 }
95 }
96
97 bool search_elem(struct TreeSet* treeset, void* value_ptr)
98 {
99     union Value uvalue = parse_value(value_ptr, treeset->value_type);
100    struct TreeNode* curr_node = search_elem_internal(treeset, uvalue);
101    return (curr_node != NULL);
102 }
103
104 void delete_elem(struct TreeSet* treeset, void* value_ptr)
105 {
106     union Value uvalue = parse_value(value_ptr, treeset->value_type);
107     // reference: https://blog.csdn.net/zxnsirius/article/details/52131433
108     struct TreeNode* curr_node = search_elem_internal(treeset, uvalue);
109     if (curr_node == NULL) return;
110     treeset->size--;
111     if (curr_node->lc == NULL)
112     {
113         if (curr_node == treeset->root) // root node
114         {
115             treeset->root = curr_node->rc;
116             if (treeset->root != NULL) treeset->root->parent = NULL;
117         }
118         else if (curr_node->rc == NULL) // leaf node
119         {
120             if (is_left_child(curr_node)) curr_node->parent->lc = NULL;
121             else curr_node->parent->rc = NULL;
122         }
123         else // middle node
124         {
125             if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->rc;
126             else curr_node->parent->rc = curr_node->rc;
127             curr_node->rc->parent = curr_node->parent;
128         }
129     }
130     else if (curr_node->rc == NULL)

```

```

131 {
132     if (curr_node == treeset->root)
133     {
134         treeset->root = curr_node->lc;
135         if (treeset->root != NULL) treeset->root->parent = NULL;
136     }
137     else
138     {
139         if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->lc;
140         else curr_node->parent->rc = curr_node->lc;
141         curr_node->lc->parent = curr_node->parent;
142     }
143 }
144 else // both left and right child exist, find the largest element that is smaller
than the deleted element
145 {
146     struct TreeNode* left_neighbor = curr_node->lc;
147     while (true)
148     {
149         if (left_neighbor->rc != NULL) left_neighbor = left_neighbor->rc;
150         else break;
151     }
152     if (left_neighbor->lc == NULL) // leaf node
153     {
154         if (is_left_child(left_neighbor)) left_neighbor->parent->lc = NULL;
155         else left_neighbor->parent->rc = NULL;
156     }
157     else
158     {
159         if (is_left_child(left_neighbor)) left_neighbor->parent->lc = left_neighbor
->lc;
160         else left_neighbor->parent->rc = left_neighbor->lc;
161         left_neighbor->lc->parent = left_neighbor->parent;
162     }
163     curr_node->value = left_neighbor->value;
164 }
165 }
166
167 int get_set_size(struct TreeSet* treeset)
168 {
169     return treeset->size;
170 }
171
172
173 // int main(void)
174 // {
175 //     struct TreeSet* set1 = create_empty_set(2);
176 //     float a = 3;
177 //     add_elem(set1, (void*)&a);
178 //     a = 5;

```

```
179 //     add_elem(set1, (void*)&a);
180 //     a = 3;
181 //     add_elem(set1, (void*)&a);
182 //     int set1_size = get_set_size(set1);
183 //     printf("set1 size: %d\n", set1_size);
184 //     delete_elem(set1, (void*)&a);
185 //     set1_size = get_set_size(set1);
186 //     printf("set1 size: %d\n", set1_size);
187 //     return 0;
188 // }
```

## B Test Cases

### B.1 Positive Tests

#### B.1.1 test-add1.mc

```
1 int add(int x, int y)
2 {
3     return x + y;
4 }
5
6 int main()
7 {
8     print( add(17, 25) );
9     return 0;
10 }
```

Expected result:

```
1 42
```

#### B.1.2 test-arith1.mc

```
1 int main()
2 {
3     print(39 + 3);
4     return 0;
5 }
```

Expected result:

```
1 42
```

#### B.1.3 test-arith2.mc

```
1 int main()
2 {
3     print(1 + 2 * 3 + 4);
4     return 0;
5 }
```

Expected result:

```
1 11
```



### B.1.4 test-arith3.mc

```
1 int foo(int a)
2 {
3     return a;
4 }
5
6 int main()
7 {
8     int a;
9     a = 42;
10    a = a + 5;
11    print(a);
12    return 0;
13 }
```

Expected result:

```
1 47
```

### B.1.5 test-arith4.mc

```
1 int foo(int a)
2 {
3     return a;
4 }
5
6 int main()
7 {
8     int a = 42;
9     a = a + 5;
10    print(a);
11    return 0;
12 }
```

Expected result:

```
1 47
```

### B.1.6 test-demo1.mc

```
1 List<int> map(func<int, int> f, List<int> list) {
2     List<int> out;
3
4     for x in list {
5         append(out, f(x));
6     }
```

```

7     return out;
8 }
9
10 List<int> filter(func<bool, int> f, List<int> list) {
11     List<int> out;
12
13     for x in list {
14         if (f(x)) {
15             append(out, x);
16         }
17     }
18     return out;
19 }
20
21 func<int, int> sum2() {
22     return int lambda int x -> x+2;
23 }
24
25 void print_list(List<int> list) {
26     for x in list {
27         print(x);
28     }
29 }
30
31 int main()
32 {
33     /* [0, 1, 2, 3, 4] */
34     List<int> my_list = range(5);
35
36     List<bool> out = map(int lambda int x -> x * 2, my_list);
37     print_list(out);
38
39     out = map(sum2(), my_list);
40     print_list(out);
41
42     out = filter(bool lambda int x -> x > 2, my_list);
43     print_list(out);
44     return 0;
45 }

```

Expected result:

```

1 0
2 2
3 4
4 6
5 8
6 2
7 3
8 4
9 5

```

```
10 6
11 3
12 4
```

### B.1.7 test-demo2.mc

```
1
2 int main () {
3     /* Fibonacci number: Compute Nth value */
4     int n = 10;
5     List<int> f = [0, 1];
6     for i in range(n-2) {
7         append(f, f[-1] + f[-2]);
8     }
9     print(f[-1]);
10 }
```

Expected result:

```
1 34
```

### B.1.8 test-demo3.mc

```
1 void swap(List<int> A, int i, int j) {
2     int t = A[i]; A[i] = A[j]; A[j] = t;
3 }
4
5 int partition(List<int> A, int p, int r) {
6     int x = A[r];
7     int i = p - 1;
8     for j in range(r - p + 1) {
9         if (A[j+p] <= x) {
10            i++;
11            swap(A, i, j+p);
12        }
13    }
14    swap(A, i+1, r);
15    return i;
16 }
17
18 /* Recursive function to sort list A using quick-sort */
19 void quicksort(List<int> A, int p, int r) {
20     if (p < r) {
21         int q = partition(A, p, r);
22         quicksort(A, p, q-1);
23         quicksort(A, q+1, r);
24     }
```

```

25 }
26
27 int main () {
28     /* Using quicksort */
29     List<int> A = [4, 2, 7, 3, 1, 9, 6, 10, 5, 8];
30     quicksort(A, 0, len(A) - 1);
31     for a in A {
32         print(a);
33     }
34 }

```

Expected result:

```

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10

```

### B.1.9 test-demo4.mc

```

1
2 int INF;
3 void graphInit(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
4     for i in range(n) {
5         List<int> l;
6         List<int> w;
7         dictAdd(E, i, l);
8         dictAdd(W, i, w);
9     }
10 }
11
12 void addEdge(Dict<int, List<int>> E, Dict<int, List<int>> W, int u, int v, int w) {
13     List<int> le = dictGetList(E, u);
14     List<int> lw = dictGetList(W, u);
15     append(le, v);
16     append(lw, w);
17 }
18
19 void printGraph(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
20     for i in range(n) {
21         prints("-----");
22         print(i);
23         List<int> le = dictGetList(E, i);

```

```

24     List<int> lw = dictGetList(W, i);
25
26     if (len(le) > 0) {
27         prints("neighbors");
28         for v in le {
29             print(v);
30         }
31         prints("weights");
32         for w in lw {
33             print(w);
34         }
35     }
36 }
37 }
38
39
40
41 bool checkNegativeCycle(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n,
42 List<int> dist) {
43     /* for every edge */
44     for u in range(n) {
45         List<int> le = dictGetList(E, u);
46         List<int> lw = dictGetList(W, u);
47         int m = len(le);
48         if (m > 0) {
49             for k in range(m) {
50                 int v = le[k];
51                 int w = lw[k];
52
53                 /* we have u, v, w */
54                 if (dist[u] != INF && dist[u] + w < dist[v]) {
55                     prints("Graph contains negative weight cycle");
56                     return false;
57                 }
58             }
59         }
60     }
61     return true;
62 }
63
64 List<int> BellmanFord(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n) {
65     List<int> dist;
66     for z in range(n) {
67         append(dist, INF);
68     }
69     dist[src] = 0;
70
71     /* Loop |V| times */
72     for i in range(n - 1) {

```

```

73     /* for every edge */
74     for u in range(n) {
75         List<int> le = dictGetList(E, u);
76         List<int> lw = dictGetList(W, u);
77         int m = len(le);
78         if (m > 0) {
79             for k in range(m) {
80                 int v = le[k];
81                 int w = lw[k];
82
83                 /* we have u, v, w */
84                 if (dist[u] != INF && dist[u] + w < dist[v]) {
85                     dist[v] = dist[u] + w;
86                 }
87             }
88         }
89     }
90 }
91 checkNegativeCycle(E, W, src, n, dist);
92 return dist;
93 }
94
95 int main() {
96     int n = 5;
97     INF = 100000;
98     Dict<int, List<int>> E;
99     Dict<int, List<int>> W;
100    graphInit(E, W, n);
101    addEdge(E, W, 0, 1, -1);
102    addEdge(E, W, 0, 2, 4);
103    addEdge(E, W, 1, 2, 3);
104    addEdge(E, W, 1, 3, 2);
105    addEdge(E, W, 1, 4, 2);
106    addEdge(E, W, 3, 2, 5);
107    addEdge(E, W, 3, 1, 1);
108    addEdge(E, W, 4, 3, -3);
109    printGraph(E, W, n);
110    prints("Vertex Distance from Source");
111    List<int> dist = BellmanFord(E, W, 0, n);
112    for d in dist {
113        print(d);
114    }
115
116    return 0;
117 }

```

Expected result:

```

1 -----
2 0
3 neighbors

```

```

4 1
5 2
6 weights
7 -1
8 4
9 -----
10 1
11 neighbors
12 2
13 3
14 4
15 weights
16 3
17 2
18 2
19 -----
20 2
21 -----
22 3
23 neighbors
24 2
25 1
26 weights
27 5
28 1
29 -----
30 4
31 neighbors
32 3
33 weights
34 -3
35 Vertex Distance from Source
36 0
37 -1
38 2
39 -2
40 1

```

### B.1.10 test-demo5.mc

```

1
2 int INF;
3 void graphInit(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
4     for i in range(n) {
5         List<int> l;
6         List<int> w;
7         dictAdd(E, i, l);
8         dictAdd(W, i, w);
9     }

```

```

10 }
11
12 void addEdge(Dict<int, List<int>> E, Dict<int, List<int>> W, int u, int v, int w) {
13     List<int> le = dictGetList(E, u);
14     List<int> lw = dictGetList(W, u);
15     append(le, v);
16     append(lw, w);
17 }
18
19 void printGraph(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
20     for i in range(n) {
21         prints("-----");
22         print(i);
23         List<int> le = dictGetList(E, i);
24         List<int> lw = dictGetList(W, i);
25
26         if (len(le) > 0) {
27             prints("neighbors");
28             for v in le {
29                 print(v);
30             }
31             prints("weights");
32             for w in lw {
33                 print(w);
34             }
35         }
36     }
37 }
38
39
40
41 bool checkNegativeCycle(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n,
42     List<int> dist) {
43     /* for every edge */
44     for u in range(n) {
45         List<int> le = dictGetList(E, u);
46         List<int> lw = dictGetList(W, u);
47         int m = len(le);
48         if (m > 0) {
49             for k in range(m) {
50                 int v = le[k];
51                 int w = lw[k];
52
53                 /* we have u, v, w */
54                 if (dist[u] != INF && dist[u] + w < dist[v]) {
55                     prints("Graph contains negative weight cycle");
56                     return false;
57                 }
58             }
59         }
60     }

```



```

59     }
60     return true;
61 }
62
63 bool BellmanFord(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n, List<
64     int> dist) {
65     for z in range(n) {
66         append(dist, INF);
67     }
68     dist[src] = 0;
69
70     /* Loop |V| times */
71     for i in range(n - 1) {
72         /* for every edge */
73         for u in range(n) {
74             List<int> le = dictGetList(E, u);
75             List<int> lw = dictGetList(W, u);
76             int m = len(le);
77             if (m > 0) {
78                 for k in range(m) {
79                     int v = le[k];
80                     int w = lw[k];
81
82                     /* we have u, v, w */
83                     if (dist[u] != INF && dist[u] + w < dist[v]) {
84                         dist[v] = dist[u] + w;
85                     }
86                 }
87             }
88         }
89     }
90     return checkNegativeCycle(E, W, src, n, dist);
91 }
92
93 int main() {
94     int n = 5;
95     INF = 100000;
96     Dict<int, List<int>> E;
97     Dict<int, List<int>> W;
98     graphInit(E, W, n);
99     addEdge(E, W, 0, 1, -1);
100    addEdge(E, W, 0, 2, 4);
101    addEdge(E, W, 1, 2, 3);
102    addEdge(E, W, 1, 3, 2);
103    addEdge(E, W, 1, 4, 2);
104    addEdge(E, W, 2, 0, -3);
105    addEdge(E, W, 3, 2, 5);
106    addEdge(E, W, 3, 1, 1);
107    addEdge(E, W, 4, 3, -3);

```

```

108  /* printGraph(E, W, n); */
109  List<int> dist;
110  if (BellmanFord(E, W, 0, n, dist)) {
111      prints("Vertex Distance from Source");
112      for d in dist {
113          print(d);
114      }
115  }
116  return 0;
117 }

```

Expected result:

```

1 Graph contains negative weight cycle

```

### B.1.11 test-dict1.mc

```

1 int main()
2 {
3     Dict<int, int> idict;
4     int i;
5     int size;
6     dictAdd(idict, 5, 10);
7     size = dictSize(idict);
8     print(size);
9     i = 10;
10    dictAdd(idict, i, 12);
11    size = dictSize(idict);
12    print(size);
13    dictAdd(idict, 5, 15);
14    size = dictSize(idict);
15    print(size);
16    return 0;
17 }

```

Expected result:

```

1 1
2 2
3 2

```

### B.1.12 test-dict2.mc

```

1 int main()
2 {
3     Dict<int, int> idict;
4     bool found;
5     dictAdd(idict, 7, 10);

```

```

6 dictAdd(idict, 9, 14);
7 found = dictHasKey(idict, 7);
8 printb(found);
9 found = dictHasKey(idict, 10);
10 printb(found);
11 return 0;
12 }

```

Expected result:

```

1 1
2 0

```

### B.1.13 test-dict3.mc

```

1 int main()
2 {
3     Dict<int, int> idict;
4     int val;
5     dictAdd(idict, 2, 9);
6     dictAdd(idict, 8, 3);
7     dictAdd(idict, 10, 6);
8     dictAdd(idict, 7, 11);
9     val = dictGetInt(idict, 8);
10    print(val);
11    val = dictGetInt(idict, 10);
12    print(val);
13    return 0;
14 }

```

Expected result:

```

1 3
2 6

```

### B.1.14 test-dict4.mc

```

1 int main()
2 {
3     Dict<int, int> idict;
4     int val;
5     int i;
6     for (i = 0; i < 100; i = i + 1)
7     {
8         dictAdd(idict, i, i);
9     }
10    for (i = 0; i < 100; i = i + 1)
11    {

```

```
12     val = dictGetInt(idict, i);
13     print(val);
14 }
15 return 0;
16 }
```

Expected result:

```
1 0
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9
11 10
12 11
13 12
14 13
15 14
16 15
17 16
18 17
19 18
20 19
21 20
22 21
23 22
24 23
25 24
26 25
27 26
28 27
29 28
30 29
31 30
32 31
33 32
34 33
35 34
36 35
37 36
38 37
39 38
40 39
41 40
42 41
43 42
```

44 43  
45 44  
46 45  
47 46  
48 47  
49 48  
50 49  
51 50  
52 51  
53 52  
54 53  
55 54  
56 55  
57 56  
58 57  
59 58  
60 59  
61 60  
62 61  
63 62  
64 63  
65 64  
66 65  
67 66  
68 67  
69 68  
70 69  
71 70  
72 71  
73 72  
74 73  
75 74  
76 75  
77 76  
78 77  
79 78  
80 79  
81 80  
82 81  
83 82  
84 83  
85 84  
86 85  
87 86  
88 87  
89 88  
90 89  
91 90  
92 91  
93 92

```
94 93
95 94
96 95
97 96
98 97
99 98
100 99
```

### B.1.15 test-dict5.mc

```
1 int main()
2 {
3     Dict<int, int> idict;
4     int val;
5     bool flag;
6     dictAdd(idict, 5, 7);
7     dictAdd(idict, 9, 3);
8     dictAdd(idict, 1, 15);
9     dictAdd(idict, 2, 8);
10    flag = dictHasKey(idict, 9);
11    printb(flag);
12    dictRemove(idict, 9);
13    dictRemove(idict, 2);
14    dictRemove(idict, 10);
15    flag = dictHasKey(idict, 9);
16    printb(flag);
17    return 0;
18 }
```

Expected result:

```
1 1
2 0
```

### B.1.16 test-dict6.mc

```
1 int main()
2 {
3     Dict<int, float> idict;
4     int i;
5     int size;
6     dictAdd(idict, 5, 10.2);
7     size = dictSize(idict);
8     print(size);
9     i = 10;
10    dictAdd(idict, i, 12.0);
11    size = dictSize(idict);
```

```
12 print(size);
13 dictAdd(idict, 5, 15.0);
14 size = dictSize(idict);
15 print(size);
16 return 0;
17 }
```

Expected result:

```
1 1
2 2
3 2
```

### B.1.17 test-dict7.mc

```
1 int main()
2 {
3     Dict<int, bool> bdict;
4     bool result;
5     dictAdd(bdict, 2, true);
6     result = dictGetBool(bdict, 2);
7     printb(result);
8     Dict<float, float> fdict;
9     dictAdd(fdict, 2.4, 4.8);
10    dictAdd(fdict, 5.6, 11.2);
11    float sim = dictGetFloat(fdict, 2.4);
12    printf(sim);
13    return 0;
14 }
```

Expected result:

```
1 1
2 4.8
```

### B.1.18 test-dict8.mc

```
1 int main()
2 {
3     Dict<int, int> idict;
4     int result;
5     dictAdd(idict, 2, 5);
6     result = dictGetInt(idict, 2);
7     print(result);
8     return 0;
9 }
```

Expected result:

```
1 5
```

### B.1.19 test-fib.mc

```
1 int fib(int x)
2 {
3     if (x < 2) return 1;
4     return fib(x-1) + fib(x-2);
5 }
6
7 int main()
8 {
9     print(fib(0));
10    print(fib(1));
11    print(fib(2));
12    print(fib(3));
13    print(fib(4));
14    print(fib(5));
15    return 0;
16 }
```

Expected result:

```
1 1
2 1
3 2
4 3
5 5
6 8
```

### B.1.20 test-float1.mc

```
1 int main()
2 {
3     float a;
4     a = 3.14159267;
5     printf(a);
6     return 0;
7 }
```

Expected result:

```
1 3.14159
```



### B.1.21 test-float2.mc

```
1 int main()
2 {
3     float a;
4     float b;
5     float c;
6     a = 3.14159267;
7     b = -2.71828;
8     c = a + b;
9     printf(c);
10    return 0;
11 }
```

Expected result:

```
1 0.423313
```

### B.1.22 test-float3.mc

```
1 void testfloat(float a, float b)
2 {
3     printf(a + b);
4     printf(a - b);
5     printf(a * b);
6     printf(a / b);
7     printb(a == b);
8     printb(a == a);
9     printb(a != b);
10    printb(a != a);
11    printb(a > b);
12    printb(a >= b);
13    printb(a < b);
14    printb(a <= b);
15 }
16
17 int main()
18 {
19     float c;
20     float d;
21
22     c = 42.0;
23     d = 3.14159;
24
25     testfloat(c, d);
26
27     testfloat(d, d);
28
29     return 0;
```

```
30 }
```

Expected result:

```
1 45.1416
2 38.8584
3 131.947
4 13.369
5 0
6 1
7 1
8 0
9 1
10 1
11 0
12 0
13 6.28318
14 0
15 9.86959
16 1
17 1
18 1
19 0
20 0
21 0
22 1
23 0
24 1
```

### B.1.23 test-float4.mc

```
1 void testfloat(float a, float b)
2 {
3     printf(a + b);
4     printf(a - b);
5     printf(a * b);
6     printf(a / b);
7     printb(a == b);
8     printb(a == a);
9     printb(a != b);
10    printb(a != a);
11    printb(a > b);
12    printb(a >= b);
13    printb(a < b);
14    printb(a <= b);
15 }
16
17 int main()
18 {
```

```

19 float c = 42.0;
20 float d = 3.14159;
21
22 testfloat(c, d);
23
24 testfloat(d, d);
25
26 return 0;
27 }

```

Expected result:

```

1 45.1416
2 38.8584
3 131.947
4 13.369
5 0
6 1
7 1
8 0
9 1
10 1
11 0
12 0
13 6.28318
14 0
15 9.86959
16 1
17 1
18 1
19 0
20 0
21 0
22 1
23 0
24 1

```

### B.1.24 test-float5.mc

```

1 int main()
2 {
3     float a = 3.14159267;
4     float b = -2.71828;
5     float c;
6     a = 3.14159267;
7     b = -2.71828;
8     c = a + b + 2.1;
9     printf(c);
10    b = 2.83;

```

```
11 c = a + b + 1.0;
12 printf(c);
13 return 0;
14 }
```

Expected result:

```
1 2.52331
2 6.97159
```

### B.1.25 test-float6.mc

```
1 int main()
2 {
3     float a = 1.1;
4     float b = -1.1;
5     printf(a*b);
6     printf(a/b);
7     printf(a+b-1.0);
8 }
```

Expected result:

```
1 -1.21
2 -1
3 -1
```

### B.1.26 test-for1.mc

```
1 int main()
2 {
3     int i;
4     for (i = 0 ; i < 5 ; i = i + 1) {
5         print(i);
6     }
7     print(42);
8     return 0;
9 }
```

Expected result:

```
1 0
2 1
3 2
4 3
5 4
6 42
```

### B.1.27 test-for2.mc

```
1 int main()
2 {
3     int i;
4     i = 0;
5     for ( ; i < 5; ) {
6         print(i);
7         i = i + 1;
8     }
9     print(42);
10    return 0;
11 }
```

Expected result:

```
1 0
2 1
3 2
4 3
5 4
6 42
```

### B.1.28 test-forin1.mc

```
1 int main()
2 {
3     int n = 5;
4     for i in range(n) {
5         print(i);
6         for j in range(n) {
7             print(j);
8         }
9     }
10    for k in [3, 6, 9, 10] {
11        print(k);
12    }
13    return 0;
14 }
```

Expected result:

```
1 0
2 0
3 1
4 2
5 3
6 4
7 1
```

```
8 0
9 1
10 2
11 3
12 4
13 2
14 0
15 1
16 2
17 3
18 4
19 3
20 0
21 1
22 2
23 3
24 4
25 4
26 0
27 1
28 2
29 3
30 4
31 3
32 6
33 9
34 10
```

### B.1.29 test-func1.mc

```
1 int add(int a, int b)
2 {
3     return a + b;
4 }
5
6 int main()
7 {
8     int a;
9     a = add(39, 3);
10    print(a);
11    return 0;
12 }
```

Expected result:

```
1 42
```

### B.1.30 test-func2.mc

```
1 /* Bug noticed by Pin-Chin Huang */
2
3 int fun(int x, int y)
4 {
5     return 0;
6 }
7
8 int main()
9 {
10    int i;
11    i = 1;
12
13    fun(i = 2, i = i+1);
14
15    print(i);
16    return 0;
17 }
```

Expected result:

```
1 2
```

### B.1.31 test-func3.mc

```
1 void printem(int a, int b, int c, int d)
2 {
3     print(a);
4     print(b);
5     print(c);
6     print(d);
7 }
8
9 int main()
10 {
11    printem(42,17,192,8);
12    return 0;
13 }
```

Expected result:

```
1 42
2 17
3 192
4 8
```

### B.1.32 test-func4.mc

```
1 int add(int a, int b)
2 {
3     int c;
4     c = a + b;
5     return c;
6 }
7
8 int main()
9 {
10    int d;
11    d = add(52, 10);
12    print(d);
13    return 0;
14 }
```

Expected result:

```
1 62
```

### B.1.33 test-func5.mc

```
1 int foo(int a)
2 {
3     return a;
4 }
5
6 int main()
7 {
8     return 0;
9 }
```

Expected result:

### B.1.34 test-func6.mc

```
1 void foo() {}
2
3 int bar(int a, bool b, int c) { return a + c; }
4
5 int main()
6 {
7     print(bar(17, false, 25));
8     return 0;
9 }
```



Expected result:

```
1 42
```

### B.1.35 test-func7.mc

```
1 int a;  
2  
3 void foo(int c)  
4 {  
5     a = c + 42;  
6 }  
7  
8 int main()  
9 {  
10    foo(73);  
11    print(a);  
12    return 0;  
13 }
```

Expected result:

```
1 115
```

### B.1.36 test-func8.mc

```
1 void foo(int a)  
2 {  
3     print(a + 3);  
4 }  
5  
6 int main()  
7 {  
8     foo(40);  
9     return 0;  
10 }
```

Expected result:

```
1 43
```

### B.1.37 test-func9.mc

```
1 void foo(int a)  
2 {  
3     print(a + 3);
```

```
4   return;
5 }
6
7 int main()
8 {
9   foo(40);
10  return 0;
11 }
```

Expected result:

```
1 43
```

### B.1.38 test-gcd.mc

```
1 int gcd(int a, int b) {
2   while (a != b) {
3     if (a > b) a = a - b;
4     else b = b - a;
5   }
6   return a;
7 }
8
9 int main()
10 {
11  print(gcd(2,14));
12  print(gcd(3,15));
13  print(gcd(99,121));
14  return 0;
15 }
```

Expected result:

```
1 2
2 3
3 11
```

### B.1.39 test-gcd2.mc

```
1 int gcd(int a, int b) {
2   while (a != b)
3     if (a > b) a = a - b;
4     else b = b - a;
5   return a;
6 }
7
8 int main()
9 {
```

```
10 print(gcd(14,21));
11 print(gcd(8,36));
12 print(gcd(99,121));
13 return 0;
14 }
```

Expected result:

```
1 7
2 4
3 11
```

### B.1.40 test-global1.mc

```
1 int a;
2 int b;
3
4 void printa()
5 {
6     print(a);
7 }
8
9 void printbb()
10 {
11     print(b);
12 }
13
14 void incab()
15 {
16     a = a + 1;
17     b = b + 1;
18 }
19
20 int main()
21 {
22     a = 42;
23     b = 21;
24     printa();
25     printbb();
26     incab();
27     printa();
28     printbb();
29     return 0;
30 }
```

Expected result:

```
1 42
2 21
```

```
3 43
4 22
```

### B.1.41 test-global2.mc

```
1 bool i;
2
3 int main()
4 {
5     int i; /* Should hide the global i */
6
7     i = 42;
8     print(i + i);
9     return 0;
10 }
```

Expected result:

```
1 84
```

### B.1.42 test-global3.mc

```
1 int i;
2 bool b;
3 int j;
4
5 int main()
6 {
7     i = 42;
8     j = 10;
9     print(i + j);
10    return 0;
11 }
```

Expected result:

```
1 52
```

### B.1.43 test-hello-world.mc

```
1 int main()
2 {
3     prints("Hello, World!");
4 }
```

Expected result:

```
1 Hello, World!
```

#### B.1.44 test-hello.mc

```
1 int main()  
2 {  
3     print(42);  
4     print(71);  
5     print(1);  
6     return 0;  
7 }
```

Expected result:

```
1 42  
2 71  
3 1
```

#### B.1.45 test-if1.mc

```
1 int main()  
2 {  
3     if (true) print(42);  
4     print(17);  
5     return 0;  
6 }
```

Expected result:

```
1 42  
2 17
```

#### B.1.46 test-if2.mc

```
1 int main()  
2 {  
3     if (true) print(42); else print(8);  
4     print(17);  
5     return 0;  
6 }
```

Expected result:

```
1 42  
2 17
```

### B.1.47 test-if3.mc

```
1 int main()
2 {
3     if (false) print(42);
4     print(17);
5     return 0;
6 }
```

Expected result:

```
1 17
```

### B.1.48 test-if4.mc

```
1 int main()
2 {
3     if (false) print(42); else print(8);
4     print(17);
5     return 0;
6 }
```

Expected result:

```
1 8
2 17
```

### B.1.49 test-if5.mc

```
1 int cond(bool b)
2 {
3     int x;
4     if (b)
5         x = 42;
6     else
7         x = 17;
8     return x;
9 }
10
11 int main()
12 {
13     print(cond(true));
14     print(cond(false));
15     return 0;
16 }
```

Expected result:

```
1 42
2 17
```

### B.1.50 test-if6.mc

```
1 int cond(bool b)
2 {
3     int x;
4     x = 10;
5     if (b)
6         if (x == 10)
7             x = 42;
8     else
9         x = 17;
10    return x;
11 }
12
13 int main()
14 {
15     print(cond(true));
16     print(cond(false));
17     return 0;
18 }
```

Expected result:

```
1 42
2 10
```

### B.1.51 test-lambda.mc

```
1 int g;
2 List<int> map(func<int, int> f, List<int> list) {
3     List<int> out = [0];
4
5     for x in list {
6         append(out, f(x));
7     }
8     return out[1:-1];
9 }
10
11 List<int> filter(func<bool, int> f, List<int> list) {
12     List<int> out = [0];
13
14     for x in list {
15         if (f(x)) {
16             append(out, x);
```

```

17     }
18 }
19     return out[1:-1];
20 }
21
22 func<int, int> sum2() {
23     return int lambda int x -> x+2;
24 }
25
26
27 int main()
28 {
29     List<int> my_list = range(5);
30     func<void, List<int>> print_list = void lambda List<int> list {
31         for x in list {
32             print(x);
33         }
34     };
35
36     g = 2;
37     List<int> out = map(int lambda int x { return x*g; }, my_list);
38     print_list(out);
39     out = map(int lambda int x -> x*2, my_list);
40     print_list(out);
41     out = map(sum2(), my_list);
42     print_list(out);
43     out = filter(bool lambda int x -> x > 2, my_list);
44     print_list(out);
45     return 0;
46 }

```

Expected result:

```

1 0
2 2
3 4
4 6
5 8
6 0
7 2
8 4
9 6
10 8
11 2
12 3
13 4
14 5
15 6
16 3
17 4

```



### B.1.52 test-list1.mc

```
1 int main()
2 {
3     List<int> arr;
4     int i;
5     arr = [5, 4, 3, 2, 1];
6     for (i = 0 ; i < 5 ; i = i + 1) {
7         print(arr[i]);
8     }
9     arr[1] = 100;
10    for (i = 0 ; i < 5 ; i = i + 1) {
11        print(arr[i]);
12    }
13    return 0;
14 }
```

Expected result:

```
1 5
2 4
3 3
4 2
5 1
6 5
7 100
8 3
9 2
10 1
```

### B.1.53 test-list2.mc

```
1 int main()
2 {
3     List<int> int_arr;
4     List<float> float_arr;
5     int i;
6     int_arr = [100];
7     append(int_arr, 666);
8     append(int_arr, 999);
9     print(len(int_arr));
10    append(int_arr, 1111);
11    append(int_arr, 2222);
12    for (i = 0; i < len(int_arr); i=i+1) {
13        print(int_arr[i]);
14    }
15    float_arr = [3.1, 5.2];
16    append(float_arr, 6.3);
17    append(float_arr, 9.4);
```

```

18 print(len(float_arr));
19 append(float_arr, 11.5);
20 append(float_arr, 2.6);
21 float_arr[0] = 0.1;
22 for (i = 0; i < len(float_arr); i=i+1) {
23     printf(float_arr[i]);
24 }
25 return 0;
26 }

```

Expected result:

```

1 3
2 100
3 666
4 999
5 1111
6 2222
7 4
8 0.1
9 5.2
10 6.3
11 9.4
12 11.5
13 2.6

```

### B.1.54 test-list3.mc

```

1 int main()
2 {
3     List<int> int_arr;
4     int i;
5     int n;
6     n = 5;
7     int_arr = range(n);
8     for (i = 0; i < n; i=i+1) {
9         print(int_arr[i]);
10    }
11    return 0;
12 }

```

Expected result:

```

1 0
2 1
3 2
4 3
5 4

```

### B.1.55 test-list4.mc

```
1 int main()
2 {
3     List<int> int_arr;
4     List<int> slice_arr;
5     List<float> float_arr;
6     int i;
7     int n;
8
9     n = 5;
10    List<int> range_list = range(n);
11    range_list[-1] = 10;
12    for (i = -1; i >= -n; i=i-1) {
13        print(range_list[i]);
14    }
15    int_arr = [100, 666, 999, 1111, 2222, 3333];
16    slice_arr = int_arr[-3:-1];
17    for (i = 0; i < len(slice_arr); i=i+1) {
18        print(slice_arr[i]);
19    }
20    return 0;
21 }
```

Expected result:

```
1 10
2 3
3 2
4 1
5 0
6 1111
7 2222
8 3333
```

### B.1.56 test-local1.mc

```
1 void foo(bool j)
2 {
3     int i;
4
5     i = 42;
6     print(i + i);
7 }
8
9 int main()
10 {
11     foo(true);
12     return 0;
```

```
13 }
```

Expected result:

```
1 84
```

### B.1.57 test-local2.mc

```
1 int foo(int a, bool b)
2 {
3     int c;
4     bool d;
5
6     c = a;
7
8     return c + 10;
9 }
10
11 int main() {
12     print(foo(37, false));
13     return 0;
14 }
```

Expected result:

```
1 47
```

### B.1.58 test-ops1.mc

```
1 int main()
2 {
3     print(1 + 2);
4     print(1 - 2);
5     print(1 * 2);
6     print(100 / 2);
7     print(99);
8     printb(1 == 2);
9     printb(1 == 1);
10    print(99);
11    printb(1 != 2);
12    printb(1 != 1);
13    print(99);
14    printb(1 < 2);
15    printb(2 < 1);
16    print(99);
17    printb(1 <= 2);
18    printb(1 <= 1);
19    printb(2 <= 1);
```

```

20 print(99);
21 printb(1 > 2);
22 printb(2 > 1);
23 print(99);
24 printb(1 >= 2);
25 printb(1 >= 1);
26 printb(2 >= 1);
27 return 0;
28 }

```

Expected result:

```

1 3
2 -1
3 2
4 50
5 99
6 0
7 1
8 99
9 1
10 0
11 99
12 1
13 0
14 99
15 1
16 1
17 0
18 99
19 0
20 1
21 99
22 0
23 1
24 1

```

### B.1.59 test-ops2.mc

```

1 int main()
2 {
3     printb(true);
4     printb(false);
5     printb(true && true);
6     printb(true && false);
7     printb(false && true);
8     printb(false && false);
9     printb(true || true);
10    printb(true || false);

```

```
11 printb(false || true);
12 printb(false || false);
13 printb(!false);
14 printb(!true);
15 print(-10);
16 }
```

Expected result:

```
1 1
2 0
3 1
4 0
5 0
6 0
7 1
8 1
9 1
10 0
11 1
12 0
13 -10
```

### B.1.60 test-ops3.mc

```
1 int main()
2 {
3     int x = 0;
4     x ++;
5     print(x);
6     x --;
7     print(x);
8     x--;
9     print(x);
10    x++;
11    print(x);
12 }
```

Expected result:

```
1 1
2 0
3 -1
4 0
```

### B.1.61 test-ops4.mc

```

1 int main()
2 {
3     List<int> arr = [5, 4, 3, 2, 1];
4     print(arr[0]);
5     arr[0] ++;
6     print(arr[0]);
7     arr[1]--;
8     print(arr[1]);
9     return 0;
10 }

```

Expected result:

```

1 5
2 6
3 3

```

### B.1.62 test-printbig.mc

```

1 /*
2  * Test for linking external C functions to LLVM-generated code
3  *
4  * printbig is defined as an external function, much like printf
5  * The C compiler generates printbig.o
6  * The LLVM compiler, llc, translates the .ll to an assembly .s file
7  * The C compiler assembles the .s file and links the .o file to generate
8  * an executable
9  */
10
11 int main()
12 {
13     printbig(72); /* H */
14     printbig(69); /* E */
15     printbig(76); /* L */
16     printbig(76); /* L */
17     printbig(79); /* O */
18     printbig(32); /*  */
19     printbig(87); /* W */
20     printbig(79); /* O */
21     printbig(82); /* R */
22     printbig(76); /* L */
23     printbig(68); /* D */
24     return 0;
25 }

```

Expected result:

```

1 XXXXXXXXXXXXXXXX

```

2 XXXXXXXXXXXXXXXX  
3 XX  
4 XX  
5 XX  
6 XXXXXXXXXXXXXXXX  
7 XXXXXXXXXXXXXXXX  
8  
9  
10 XXXXXXXXXXXXXXXX  
11 XXXXXXXXXXXXXXXX  
12 XX XX XX  
13 XX XX XX  
14 XX XX XX  
15 XX XX  
16  
17  
18 XXXXXXXXXXXXXXXX  
19 XXXXXXXXXXXXXXXX  
20 XX  
21 XX  
22 XX  
23 XX  
24  
25  
26 XXXXXXXXXXXXXXXX  
27 XXXXXXXXXXXXXXXX  
28 XX  
29 XX  
30 XX  
31 XX  
32  
33 XXXXXXXXXX  
34 XXXXXXXXXXXXXXXX  
35 XX XX  
36 XX XX  
37 XX XX  
38 XXXXXXXXXXXXXXXX  
39 XXXXXXXXXX  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49 XXXXXXXXXX  
50 XXXXXXXXXXXXXXXX  
51 XXXXXX



```

52     XXXXXX
53     XXXXXX
54     XXXXXXXXXXXXXXXX
55     XXXXXXXXXXXXXXXX
56
57     XXXXXXXXXXXXXXXX
58     XXXXXXXXXXXXXXXX
59     XX             XX
60     XX             XX
61     XX             XX
62     XXXXXXXXXXXXXXXX
63     XXXXXXXXXXXXXXXX
64
65     XXXXXXXXXXXXXXXX
66     XXXXXXXXXXXXXXXX
67     XX             XX
68     XXXX          XX
69     XXXXXXXX      XX
70     XXXX  XXXXXXXX
71     XX   XXXXXXXX
72
73
74     XXXXXXXXXXXXXXXX
75     XXXXXXXXXXXXXXXX
76     XX
77     XX
78     XX
79     XX
80
81     XXXXXXXXXXXXXXXX
82     XXXXXXXXXXXXXXXX
83     XX             XX
84     XX             XX
85     XXXX          XXXX
86     XXXXXXXXXXXXXXXX
87     XXXXXXXX

```

### B.1.63 test-set1.mc

```

1  int main()
2  {
3      Set<int> iset;
4      int i;
5      int size;
6      size = setSize(iset);
7      print(size);
8      setAdd(iset, 5);
9      size = setSize(iset);
10     print(size);

```

```
11 i = 10;
12 setAdd(iset, i);
13 size = setSize(iset);
14 print(size);
15 setAdd(iset, 5);
16 size = setSize(iset);
17 print(size);
18 return 0;
19 }
```

Expected result:

```
1 0
2 1
3 2
4 2
```

### B.1.64 test-set2.mc

```
1 int main()
2 {
3     Set<int> iset;
4     bool found;
5     setAdd(iset, 7);
6     setAdd(iset, 9);
7     found = setFind(iset, 7);
8     printb(found);
9     found = setFind(iset, 10);
10    printb(found);
11    return 0;
12 }
```

Expected result:

```
1 1
2 0
```

### B.1.65 test-set3.mc

```
1 int main()
2 {
3     Set<int> iset;
4     int size;
5     bool found;
6     setAdd(iset, 4);
7     setAdd(iset, 8);
8     setAdd(iset, 16);
9     setAdd(iset, 7);
```

```

10 found = setFind(iset, 8);
11 printb(found);
12 setRemove(iset, 5);
13 setRemove(iset, 8);
14 found = setFind(iset, 8);
15 printb(found);
16 setRemove(iset, 16);
17 size = setSize(iset);
18 print(size);
19 return 0;
20 }

```

Expected result:

```

1 1
2 0
3 2

```

### B.1.66 test-set4.mc

```

1 int main()
2 {
3   Set<float> fset;
4   float i;
5   int size;
6   size = setSize(fset);
7   print(size);
8   setAdd(fset, 5.2);
9   size = setSize(fset);
10  print(size);
11  i = 10.6;
12  setAdd(fset, i);
13  size = setSize(fset);
14  print(size);
15  setAdd(fset, 5.2);
16  size = setSize(fset);
17  print(size);
18  return 0;
19 }

```

Expected result:

```

1 0
2 1
3 2
4 2

```

### B.1.67 test-set5.mc

```
1 int main()
2 {
3     Set<bool> bset;
4     bool found;
5     bool b;
6     int size;
7     setAdd(bset, true);
8     found = setFind(bset, true);
9     printb(found);
10    found = setFind(bset, false);
11    printb(found);
12    b = false;
13    setAdd(bset, b);
14    found = setFind(bset, false);
15    printb(found);
16    setRemove(bset, true);
17    size = setSize(bset);
18    print(size);
19    return 0;
20 }
```

Expected result:

```
1 1
2 0
3 1
4 1
```

### B.1.68 test-string-concat.mc

```
1 int main(){
2     string s1;
3     string s2;
4     string s3;
5     s1= "Good ";
6     s2= "morning";
7
8     s3= concat(s1,s2);
9     prints(s3);
10
11    return 0;
12 }
```

Expected result:

```
1 Good morning
```

### B.1.69 test-string-len.mc

```
1 int main(){
2     string s;
3     int len;
4
5     s= "foobar";
6     len= length(s);
7     print(len);
8     return 0;
9 }
```

Expected result:

```
1 6
```

### B.1.70 test-string-slice.mc

```
1 int main(){
2     string s;
3     string sliced;
4
5     s= "foobar";
6     sliced = slice(s, 2, 5);
7     prints(sliced);
8     return 0;
9 }
```

Expected result:

```
1 oba
```

### B.1.71 test-string1.mc

```
1 int main()
2 {
3     string str;
4     str = "nice to meet u";
5     prints(str);
6     return 0;
7 }
```

Expected result:

```
1 nice to meet u
```

### B.1.72 test-string2.mc

```
1 int main()
2 {
3     string str = "nice to meet u";
4     prints(str);
5     return 0;
6 }
```

Expected result:

```
1 nice to meet u
```

### B.1.73 test-var1.mc

```
1 int main()
2 {
3     int a;
4     a = 42;
5     print(a);
6     return 0;
7 }
```

Expected result:

```
1 42
```

### B.1.74 test-var2.mc

```
1 int a;
2
3 void foo(int c)
4 {
5     a = c + 42;
6 }
7
8 int main()
9 {
10    foo(73);
11    print(a);
12    return 0;
13 }
```

Expected result:

```
1 115
```

### B.1.75 test-var3.mc

```
1 int main()
2 {
3     int pos_int = 42;
4     print(pos_int);
5     int neg_int = -42;
6     print(neg_int);
7     bool true_bool;
8     true_bool = true;
9     printb(true_bool);
10    bool false_bool = false;
11    printb(false_bool);
12    float pos_float = 0.1;
13    printf(pos_float);
14    float neg_float = -0.1;
15    printf(neg_float);
16    string empty_string = "";
17    prints(empty_string);
18    string short_string = "abcd";
19    prints(short_string);
20    string long_string = "abcdefghijklmnopqrstuvwxy";
21    prints(long_string);
22    return 0;
23 }
```

Expected result:

```
1 42
2 -42
3 1
4 0
5 0.1
6 -0.1
7
8 abcd
9 abcdefghijklmnopqrstuvwxy
```

### B.1.76 test-while1.mc

```
1 int main()
2 {
3     int i;
4     i = 5;
5     while (i > 0) {
6         print(i);
7         i = i - 1;
8     }
9     print(42);
```

```
10 | return 0;
11 | }
```

Expected result:

```
1 | 5
2 | 4
3 | 3
4 | 2
5 | 1
6 | 42
```

### B.1.77 test-while2.mc

```
1 | int foo(int a)
2 | {
3 |     int j;
4 |     j = 0;
5 |     while (a > 0) {
6 |         j = j + 2;
7 |         a = a - 1;
8 |     }
9 |     return j;
10 | }
11 |
12 | int main()
13 | {
14 |     print(foo(7));
15 |     return 0;
16 | }
```

Expected result:

```
1 | 14
```

## B.2 Negative Tests

### B.2.1 fail-assign1.mc

```
1 | int main()
2 | {
3 |     int i;
4 |     bool b;
5 |
6 |     i = 42;
7 |     i = 10;
8 |     b = true;
```



```
9  b = false;
10 i = false; /* Fail: assigning a bool to an integer */
11 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal assignment int = bool in i = false")
```

## B.2.2 fail-assign2.mc

```
1 int main()
2 {
3   int i;
4   bool b;
5
6   b = 48; /* Fail: assigning an integer to a bool */
7 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal assignment bool = int in b = 48")
```

## B.2.3 fail-assign3.mc

```
1 void myvoid()
2 {
3   return;
4 }
5
6 int main()
7 {
8   int i;
9
10  i = myvoid(); /* Fail: assigning a void to an integer */
11 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal assignment int = void in i = myvoid()")
```

## B.2.4 fail-dead1.mc

```
1 int main()
2 {
3   int i;
4 }
```

```
5  i = 15;
6  return i;
7  i = 32; /* Error: code after a return */
8  }
```

Expected error:

```
1 Fatal error: exception Failure("nothing may follow a return")
```

### B.2.5 fail-dead2.mc

```
1  int main()
2  {
3    int i;
4
5    {
6      i = 15;
7      return i;
8    }
9    i = 32; /* Error: code after a return */
10 }
```

Expected error:

```
1 Fatal error: exception Failure("nothing may follow a return")
```

### B.2.6 fail-dict1.mc

```
1  int main()
2  {
3    Dict<int, float> ifdict;
4    dictAdd(ifdict, 3.2, 5.0);
5    return 0;
6  }
```

Expected error:

```
1 Fatal error: exception Failure("dictAdd type incorrect")
```

### B.2.7 fail-dict2.mc

```
1  int main()
2  {
3    Dict<float, bool> fbdict;
4    dictAdd(fbdict, 3.2, true);
5    dictGetBool(fbdict, false);
```

```
6   return 0;
7 }
```

Expected error:

```
1 Fatal error: exception Failure("dictGetBool type incorrect")
```

### B.2.8 fail-dict3.mc

```
1 int main()
2 {
3   Dict<int, float> ifdict;
4   dictAdd(ifdict, 3, 5);
5   return 0;
6 }
```

Expected error:

```
1 Fatal error: exception Failure("dictAdd type incorrect")
```

### B.2.9 fail-expr1.mc

```
1 int a;
2 bool b;
3
4 void foo(int c, bool d)
5 {
6   int dd;
7   bool e;
8   a + c;
9   c - a;
10  a * 3;
11  c / 2;
12  d + a; /* Error: bool + int */
13 }
14
15 int main()
16 {
17   return 0;
18 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal binary operator bool + int in d + a")
```

### B.2.10 fail-expr2.mc

```
1 int a;
2 bool b;
3
4 void foo(int c, bool d)
5 {
6     int d; /* Error: redeclaring d */
7     bool e;
8     b + a;
9 }
10
11 int main()
12 {
13     return 0;
14 }
```

Expected error:

```
1 Fatal error: exception Failure("duplicate local d")
```

### B.2.11 fail-expr3.mc

```
1 int a;
2 float b;
3
4 void foo(int c, float d)
5 {
6     int d; /* Error: redeclaring d */
7     float e;
8     b + a;
9 }
10
11 int main()
12 {
13     return 0;
14 }
```

Expected error:

```
1 Fatal error: exception Failure("duplicate local d")
```

### B.2.12 fail-expr4.mc

```
1 int a;
2 bool b;
3
```

```

4 void foo(int c, bool d)
5 {
6     int f;
7     bool e;
8     b + a; /* Error: bool + int */
9 }
10
11 int main()
12 {
13     return 0;
14 }

```

Expected error:

```

1 Fatal error: exception Failure("illegal binary operator bool + int in b + a")

```

### B.2.13 fail-expr5.mc

```

1 int a;
2 float b;
3
4 void foo(int c, float d)
5 {
6     int f;
7     bool e;
8     b + e; /* Error: float + bool */
9 }
10
11 int main()
12 {
13     return 0;
14 }

```

Expected error:

```

1 Fatal error: exception Failure("illegal binary operator float + bool in b + e")

```

### B.2.14 fail-float1.mc

```

1 int main()
2 {
3     -3.5 && 1; /* Float with AND? */
4     return 0;
5 }

```

Expected error:

```

1 Fatal error: exception Failure("illegal binary operator float && int in -3.5 && 1")

```

### B.2.15 fail-float2.mc

```
1 int main()
2 {
3   -3.5 && 2.5; /* Float with AND? */
4   return 0;
5 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal binary operator float && float in -3.5 && 2.5")
```

### B.2.16 fail-for1.mc

```
1 int main()
2 {
3   int i;
4   for ( ; true ; ) {} /* OK: Forever */
5
6   for (i = 0 ; i < 10 ; i = i + 1) {
7     if (i == 3) return 42;
8   }
9
10  for (j = 0; i < 10 ; i = i + 1) {} /* j undefined */
11
12  return 0;
13 }
```

Expected error:

```
1 Fatal error: exception Failure("undeclared identifier j")
```

### B.2.17 fail-for2.mc

```
1 int main()
2 {
3   int i;
4
5   for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */
6
7   return 0;
8 }
```

Expected error:

```
1 Fatal error: exception Failure("undeclared identifier j")
```

### B.2.18 fail-for3.mc

```
1 int main()
2 {
3     int i;
4
5     for (i = 0; i ; i = i + 1) {} /* i is an integer, not Boolean */
6
7     return 0;
8 }
```

Expected error:

```
1 Fatal error: exception Failure("expected Boolean expression in i")
```

### B.2.19 fail-for4.mc

```
1 int main()
2 {
3     int i;
4
5     for (i = 0; i < 10 ; i = j + 1) {} /* j undefined */
6
7     return 0;
8 }
```

Expected error:

```
1 Fatal error: exception Failure("undeclared identifier j")
```

### B.2.20 fail-for5.mc

```
1 int main()
2 {
3     int i;
4
5     for (i = 0; i < 10 ; i = i + 1) {
6         foo(); /* Error: no function foo */
7     }
8
9     return 0;
10 }
```

Expected error:

```
1 Fatal error: exception Failure("unrecognized function foo")
```

### B.2.21 fail-func1.mc

```
1 int foo() {}
2
3 int bar() {}
4
5 int baz() {}
6
7 void bar() {} /* Error: duplicate function bar */
8
9 int main()
10 {
11     return 0;
12 }
```

Expected error:

```
1 Fatal error: exception Failure("duplicate function bar")
```

### B.2.22 fail-func2.mc

```
1 int foo(int a, bool b, int c) { }
2
3 void bar(int a, bool b, int a) {} /* Error: duplicate formal a in bar */
4
5 int main()
6 {
7     return 0;
8 }
```

Expected error:

```
1 Fatal error: exception Failure("duplicate formal a")
```

### B.2.23 fail-func3.mc

```
1 int foo(int a, bool b, int c) { }
2
3 void bar(int a, void b, int c) {} /* Error: illegal void formal b */
4
5 int main()
6 {
7     return 0;
8 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal void formal b")
```



### B.2.24 fail-func4.mc

```
1 int foo() {}
2
3 void bar() {}
4
5 int print() {} /* Should not be able to define print */
6
7 void baz() {}
8
9 int main()
10 {
11     return 0;
12 }
```

Expected error:

```
1 Fatal error: exception Failure("function print may not be defined")
```

### B.2.25 fail-func5.mc

```
1 int foo() {}
2
3 int bar() {
4     int a;
5     void b; /* Error: illegal void local b */
6     bool c;
7
8     return 0;
9 }
10
11 int main()
12 {
13     return 0;
14 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal void local b")
```

### B.2.26 fail-func6.mc

```
1 void foo(int a, bool b)
2 {
3 }
4
5 int main()
```

```
6 {
7   foo(42, true);
8   foo(42); /* Wrong number of arguments */
9 }
```

Expected error:

```
1 Fatal error: exception Failure("expecting 2 arguments in foo(42)")
```

### B.2.27 fail-func7.mc

```
1 void foo(int a, bool b)
2 {
3 }
4
5 int main()
6 {
7   foo(42, true);
8   foo(42, true, false); /* Wrong number of arguments */
9 }
```

Expected error:

```
1 Fatal error: exception Failure("expecting 2 arguments in foo(42, true, false)")
```

### B.2.28 fail-func8.mc

```
1 void foo(int a, bool b)
2 {
3 }
4
5 void bar()
6 {
7 }
8
9 int main()
10 {
11   foo(42, true);
12   foo(42, bar()); /* int and void, not int and bool */
13 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal argument found void expected bool in bar()")
```

### B.2.29 fail-func9.mc

```
1 void foo(int a, bool b)
2 {
3 }
4
5 int main()
6 {
7     foo(42, true);
8     foo(42, 42); /* Fail: int, not bool */
9 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal argument found int expected bool in 42")
```

### B.2.30 fail-global1.mc

```
1 int c;
2 bool b;
3 void a; /* global variables should not be void */
4
5
6 int main()
7 {
8     return 0;
9 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal void global a")
```

### B.2.31 fail-global2.mc

```
1 int b;
2 bool c;
3 int a;
4 int b; /* Duplicate global variable */
5
6 int main()
7 {
8     return 0;
9 }
```

Expected error:

```
1 Fatal error: exception Failure("duplicate global b")
```

### B.2.32 fail-if1.mc

```
1 int main()
2 {
3     if (true) {}
4     if (false) {} else {}
5     if (42) {} /* Error: non-bool predicate */
6 }
```

Expected error:

```
1 Fatal error: exception Failure("expected Boolean expression in 42")
```

### B.2.33 fail-if2.mc

```
1 int main()
2 {
3     if (true) {
4         foo; /* Error: undeclared variable */
5     }
6 }
```

Expected error:

```
1 Fatal error: exception Failure("undeclared identifier foo")
```

### B.2.34 fail-if3.mc

```
1 int main()
2 {
3     if (true) {
4         42;
5     } else {
6         bar; /* Error: undeclared variable */
7     }
8 }
```

Expected error:

```
1 Fatal error: exception Failure("undeclared identifier bar")
```

### B.2.35 fail-list1.mc

```
1 int main()
2 {
3     List<int> range_list;
```

```
4  int i;
5  int n;
6
7  n = 5;
8  range_list = range(n);
9  append(range_list, 999.0);
10 return 0;
11 }
```

Expected error:

```
1 Fatal error: exception Failure("append type incorrect")
```

### B.2.36 fail-list2.mc

```
1 int main()
2 {
3   int i;
4   int n;
5
6   n = 5;
7   i = len(n);
8   return 0;
9 }
```

Expected error:

```
1 Fatal error: exception Failure("first argument of len should have composite type")
```

### B.2.37 fail-list3.mc

```
1 int main()
2 {
3   List<float> float_list;
4   int n;
5   n = 5;
6   float_list = [1.2, 2.4];
7   append(float_list, n);
8   return 0;
9 }
```

Expected error:

```
1 Fatal error: exception Failure("append type incorrect")
```

### B.2.38 fail-nomain.mc

Expected error:

```
1 Fatal error: exception Failure("unrecognized function main")
```

### B.2.39 fail-ops1.mc

```
1 void main() {  
2   float x = 0.0;  
3   x ++; /* Error: illegal unary operator float++ in x++ */  
4 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal unary operator float++ in x++")
```

### B.2.40 fail-ops2.mc

```
1 void main() {  
2   bool x = false;  
3   x ++; /* Error: illegal unary operator bool++ in x++ */  
4 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal unary operator bool++ in x++")
```

### B.2.41 fail-ops3.mc

```
1 void main() {  
2   string x = "123";  
3   x ++; /* Error: illegal unary operator string++ in x++ */  
4 }
```

Expected error:

```
1 Fatal error: exception Failure("illegal unary operator string++ in x++")
```

### B.2.42 fail-print.mc

```
1 /* Should be illegal to redefine */  
2 void print() {}
```

Expected error:

```
1 Fatal error: exception Failure("function print may not be defined")
```

### B.2.43 fail-printb.mc

```
1 /* Should be illegal to redefine */  
2 void printb() {}
```

Expected error:

```
1 Fatal error: exception Failure("function printb may not be defined")
```

### B.2.44 fail-printbig.mc

```
1 /* Should be illegal to redefine */  
2 void printbig() {}
```

Expected error:

```
1 Fatal error: exception Failure("function printbig may not be defined")
```

### B.2.45 fail-return1.mc

```
1 int main()  
2 {  
3     return true; /* Should return int */  
4 }
```

Expected error:

```
1 Fatal error: exception Failure("return gives bool expected int in true")
```

### B.2.46 fail-return2.mc

```
1 void foo()  
2 {  
3     if (true) return 42; /* Should return void */  
4     else return;  
5 }  
6  
7 int main()  
8 {  
9     return 42;  
10 }
```

Expected error:

```
1 Fatal error: exception Failure("return gives int expected void in 42")
```

### B.2.47 fail-set1.mc

```
1 int main()
2 {
3   Set<float> fset;
4   float f = 10.0;
5   int a = 10;
6   setAdd(fset, f);
7   setRemove(fset, 10);
8   return 0;
9 }
```

Expected error:

```
1 Fatal error: exception Failure("setRemove type incorrect")
```

### B.2.48 fail-set2.mc

```
1 int main()
2 {
3   Set<int> iset;
4   setAdd(iset, true);
5   return 0;
6 }
```

Expected error:

```
1 Fatal error: exception Failure("setAdd type incorrect")
```

### B.2.49 fail-while1.mc

```
1 int main()
2 {
3   int i;
4
5   while (true) {
6     i = i + 1;
7   }
8
9   while (42) { /* Should be boolean */
10    i = i + 1;
11 }
```



```
12 |
13 | }
```

Expected error:

```
1 Fatal error: exception Failure("expected Boolean expression in 42")
```

### B.2.50 fail-while2.mc

```
1 int main()
2 {
3   int i;
4
5   while (true) {
6     i = i + 1;
7   }
8
9   while (true) {
10    foo(); /* foo undefined */
11  }
12
13 }
```

Expected error:

```
1 Fatal error: exception Failure("unrecognized function foo")
```