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

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 elseless `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-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$;) *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$_{opt}$)

parameter-list :

type-specifier

type-specifier, parameter-list

The function-body has the form

function-body :

function-statement

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.

function-statement :

{ *statement-list*}

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 are 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:

`func` < *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.

type-list :

type-specifier

type-specifier, type-list

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` $\textit{parameter-list}_{opt}$ *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` $\textit{parameter-list}_{opt} \rightarrow \textit{expr}$

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.

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

List can also be initialized with a list expressions.

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

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^{th} value of the array can be accessed by

$$identifier[i]$$

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

$$identifier[-i]$$

3.15.3 Set value at index

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

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

3.15.4 List Slicing

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

$$identifier[start : end]$$

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

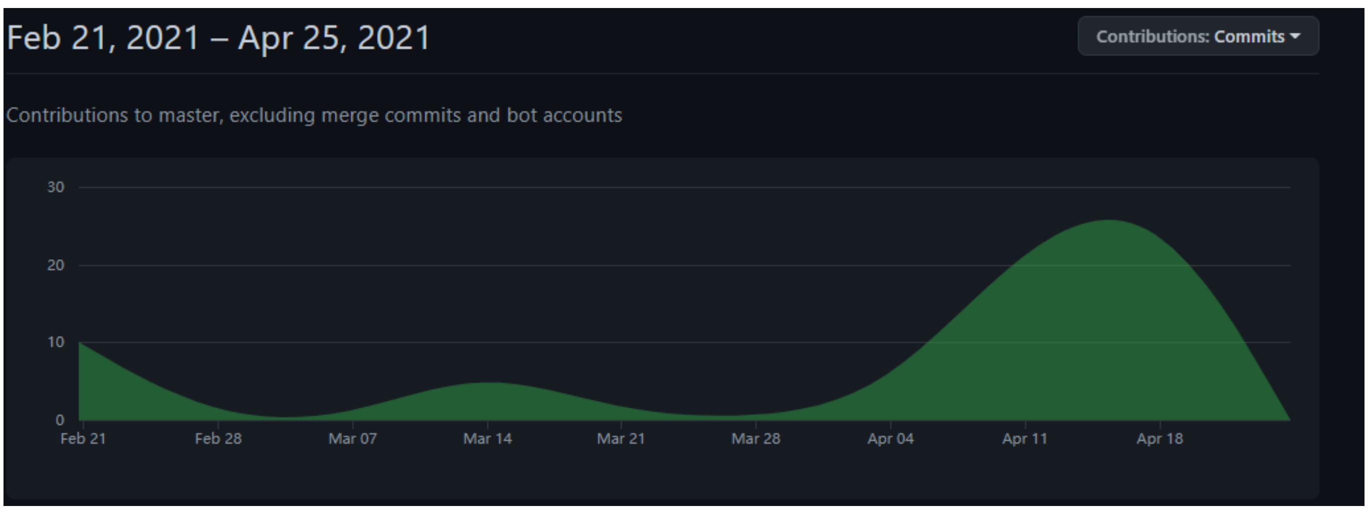
In several places FFBB requires expressions that have to evaluate to a constant: in array bounds, the expression can only involve integer constants, possibly connected by binary arithmetic operators; in control flow condition statements, the expression has to evaluate to booleans.

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 the 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

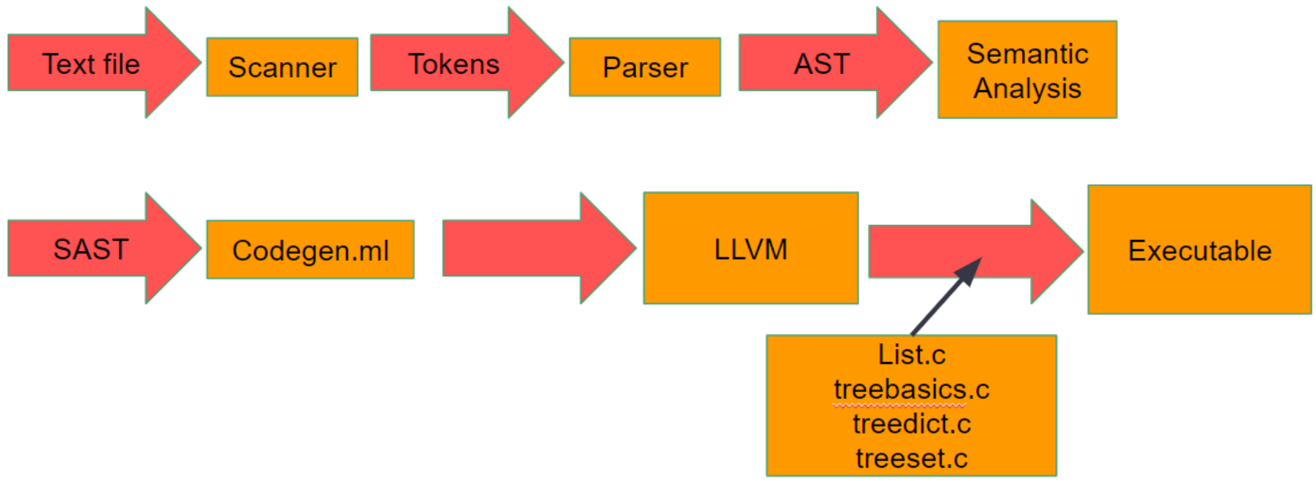
Editors: Vim, VS Code, sublime

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.

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

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.

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

```

```
int partition(List<int> A, int p, int r) {
  int x = A[r];
  int i = p - 1;
  for j in range(r - p + 1) {
      if (A[j+p] <= x) {
          i++;
          swap(A, i, j+p);
      }
  }
  swap(A, i+1, r);
  return i;
}

/* Recursive function to sort list A using quick-sort */
void quicksort(List<int> A, int p, int r) {
  if (p < r) {
      int q = partition(A, p, r);
      quicksort(A, p, q-1);
      quicksort(A, q+1, r);
  }
}

int main () {
    /* Using quicksort */
    List<int> A = [4, 2, 7, 3, 1, 9, 6, 10, 5, 8];
    quicksort(A, 0, len(A) - 1);
    for a in A {
        print(a);
    }
}
```

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.

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

```
    List<int> out;

    for x in list {
        append(out, f(x));
    }
    return out;
}

List<int> filter(func<bool, int> f, List<int> list) {
    List<int> out;

    for x in list {
        if (f(x)) {
            append(out, x);
        }
    }
    return out;
}

func<int, int> sum2() {
    return int lambda int x -> x+2;
}

void print_list(List<int> list) {
    for x in list {
        print(x);
    }
}

int main()
{
    /* [0, 1, 2, 3, 4] */
    List<int> my_list = range(5);

    List<int> out = map(int lambda int x -> x * 2, my_list);
    print_list(out); /* 0, 2, 4, 6, 8 */

    out = map(sum2(), my_list);
```

```
    print_list(out); /* 2, 3, 4, 5, 6 */

    out = filter(bool lambda int x -> x > 2, my_list);
    print_list(out); /* 3, 4 */
    return 0;
}
```

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.

```
int INF;
void graphInit(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
    for i in range(n) {
        List<int> l;
        List<int> w;
        dictAdd(E, i, l);
        dictAdd(W, i, w);
    }
}

void addEdge(Dict<int, List<int>> E, Dict<int, List<int>> W, int u, int v, int
   w) {
    List<int> le = dictGetList(E, u);
    List<int> lw = dictGetList(W, u);
    append(le, v);
    append(lw, w);
}

void printGraph(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
    for i in range(n) {
        prints("------------------------");
        print(i);
        List<int> le = dictGetList(E, i);
        List<int> lw = dictGetList(W, i);

```

```
        if (len(le) > 0) {
            prints("neighbors");
            for v in le {
                print(v);
            }
            prints("weights");
            for w in lw {
                print(w);
            }
        }
    }
}



bool checkNegativeCycle(Dict<int, List<int>> E, Dict<int, List<int>> W, int src
    , int n, List<int> dist) {
    /* for every edge */
    for u in range(n) {
        List<int> le = dictGetList(E, u);
        List<int> lw = dictGetList(W, u);
        int m = len(le);
        if (m > 0) {
            for k in range(m) {
                int v = le[k];
                int w = lw[k];

                /* we have u, v, w */
                if (dist[u] != INF && dist[u] + w < dist[v]) {
                    prints("Graph contains negative weight cycle");
                    return false;
                }
            }
        }
    }
    return true;
}

```

```
List<int> BellmanFord(Dict<int, List<int>> E, Dict<int, List<int>> W, int src,
    int n) {

    List<int> dist;
    for z in range(n) {
        append(dist, INF);
    }
    dist[src] = 0;

    /* Loop |V| times */
    for i in range(n - 1) {
        /* for every edge */
        for u in range(n) {
            List<int> le = dictGetList(E, u);
            List<int> lw = dictGetList(W, u);
            int m = len(le);
            if (m > 0) {
                for k in range(m) {
                    int v = le[k];
                    int w = lw[k];

                    /* we have u, v, w */
                    if (dist[u] != INF && dist[u] + w < dist[v]) {
                        dist[v] = dist[u] + w;
                    }
                }
            }
        }
    }
    checkNegativeCycle(E, W, src, n, dist);
    return dist;
}

int main() {
    int n = 5;
    INF = 100000;
    Dict<int, List<int>> E;
    Dict<int, List<int>> W;
```

```
    graphInit(E, W, n);
    addEdge(E, W,  0, 1, -1);
    addEdge(E, W,  0, 2, 4);
    addEdge(E, W,  1, 2, 3);
    addEdge(E, W,  1, 3, 2);
    addEdge(E, W,  1, 4, 2);
    addEdge(E, W,  3, 2, 5);
    addEdge(E, W,  3, 1, 1);
    addEdge(E, W,  4, 3, -3);
    printGraph(E, W, n);
    prints("Vertex Distance from Source");
    List<int> dist = BellmanFord(E, W, 0, n);
    for d in dist {
        print(d);
    }

    return 0;
}
```

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

```
(* Ocamllex scanner for FFBB *)

{ open FFBBparser }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LSQUARE }
| ']'      { RSQUARE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| "++"     { DOUBLEPLUS }
| '-'      { MINUS }
| "--"     { DOUBLEMINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "List"   { LISTT }
| "string" { STRING }
| "Set"    { SETT  }
```

```
| "Dict"   { DICTT }
| "void"   { VOID }
| "in"     { IN }
| "lambda" { LAMBDA }
| "func"   { FUNC }
| ":"      { COLON }
| "->"      { ARROW }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z' '@']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| '"'         { read_string (Buffer.create 10) lexbuf }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

and read_string buf =
  parse
  | '"'       { STRLIT (Buffer.contents buf) }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      read_string buf lexbuf
    }
  | _ { raise (Failure ("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
  | eof { raise (Failure ("String is not terminated")) }
```

A.2 FFBBparser.mly

```
/* Ocamlyacc parser for FFBB */

%{
open Ast
let lambda_num = ref 0

%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA PLUS MINUS TIMES DIVIDE
    ASSIGN COLON IN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR DOUBLEPLUS DOUBLEMINUS LAMBDA FUNC ARROW
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID STRING LISTT DICTT SETT
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT STRLIT
%token EOF
```

```

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN COLON
%left OR
%left AND
%left DOUBLEPLUS DOUBLEMINUS
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT
%left LSQUARE

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { ([], [])               }
 | decls vdecl { (($2 :: fst $1), snd $1) }
 | decls fdecl { (fst $1, ($2 :: snd $1)) }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }

formal_list:
    typ ID                     { [($1,$2)]     }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

formals_opt_lambda:
    /* nothing */  { ([], []) }
  | formal_list_lambda   { (List.rev (fst $1), List.rev (snd $1)) }

formal_list_lambda:
    typ ID                     { ([$1],[$2])     }
  | formal_list_lambda COMMA typ ID { ($3 :: fst $1), ($4 :: snd $1) }

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    {
    { typ = $1;
   fname = $2;
   formals = List.rev $4;
   body = List.rev $7 }
```

```
  }


typ_list:
    typ                {[$1]}
  | typ COMMA typ_list { $1 :: $3 }

high_typ:
    LISTT { LIST }
  | SETT  { SET }
  | DICTT { DICT }

typ:
    INT      { Int   }
  | BOOL     { Bool  }
  | FLOAT    { Float }
  | STRING   { String }
  | VOID     { Void  }
  | FUNC LT typ_list GT { FunctionType($3) }
  | high_typ LT typ_list GT { CompositeType($1, $3) }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
    typ ID SEMI { ($1, $2) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI                               { Expr $1                     }
  | RETURN expr_opt SEMI                    { Return $2                   }
  | LBRACE stmt_list RBRACE                 { Block(List.rev $2)          }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([]))       }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)              }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                            { For($3, $5, $7, $9)         }
  | FOR ID IN expr stmt                       { Forin($2, $4, $5)           }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)               }
  | typ ID SEMI                             { VarDecl($1, $2)             }
  | typ ID ASSIGN expr SEMI                 { VarDeclAssign(($1, $2), $4) }
  | ID DOUBLEPLUS SEMI                      { Expr(Unop(Inc, Id($1)))     }
  | ID DOUBLEMINUS SEMI                     { Expr(Unop(Dec, Id($1)))     }
  | ID LSQUARE expr RSQUARE DOUBLEPLUS SEMI  { Expr(IndexUnop($1, $3, Inc)) }
  | ID LSQUARE expr RSQUARE DOUBLEMINUS SEMI { Expr(IndexUnop($1, $3, Dec)) }

expr_opt:
```

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

expr:
    LITERAL          { Literal($1)            }
  | FLIT           { Fliteral($1)           }
  | BLIT             { BoolLit($1)            }
  | STRLIT           { StrLit($1)          }
  | function_literal { $1 }
  | ID               { Id($1)                 }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq,   $3)   }
  | expr LT     expr { Binop($1, Less,  $3)   }
  | expr LEQ    expr { Binop($1, Leq,   $3)   }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | list_expr        { $1 }
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
  | NOT expr         { Unop(Not, $2)          }
  | ID LSQUARE expr RSQUARE ASSIGN expr { IndexAssign($1, $3, $6) }
  | ID ASSIGN expr   { Assign($1, $3)         }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | ID LSQUARE expr COLON expr RSQUARE { GetSlice($1, $3, $5)  }
  | ID LSQUARE expr RSQUARE { GetIndex($1, $3)  }
  | LPAREN expr RPAREN { $2                   }

function_literal:
   typ LAMBDA formals_opt LBRACE stmt_list RBRACE{
        lambda_num := !lambda_num + 1;
        FunctionLit({
            typ = $1;
            fname = "lambda_" ^ string_of_int !lambda_num;
            formals = List.rev $3;
            body = List.rev $5
        })
    }
  | typ LAMBDA formals_opt ARROW expr {
        lambda_num := !lambda_num + 1;
        FunctionLit({
            typ = $1;
            fname = "lambda_" ^ string_of_int !lambda_num;
            formals = List.rev $3;
            body = [Return $5]
        })
```

```
    }

list_expr:
    LSQUARE RSQUARE                 { ListExpr([]) }
  | LSQUARE list_expr_core RSQUARE { ListExpr($2)}

list_expr_core:
    expr                       { [$1] }
  | expr COMMA list_expr_core { $1 :: $3 }

args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

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

A.3 ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

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

type uop = Neg | Not | Inc | Dec

(* type prm_typ = Int | Bool | Float | Void | String *)
type high_typ = LIST | DICT | SET

type typ =
    Int | Bool | Float | Void | String | Func
  | CompositeType of high_typ * typ list
  | FunctionType of typ list


and func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    body : stmt list;
  }

and bind = typ * string

and expr =
    Literal of int
  | Fliteral of string
```

```
  | BoolLit of bool
  | StrLit of string
  | FunctionLit of func_decl
  | GetSlice of string * expr * expr
  | GetIndex of string * expr
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | IndexAssign of string * expr * expr
  | Assign of string * expr
  | ListExpr of expr list
  | Call of string * expr list
  | IndexUnop of string * expr * uop
  | Noexpr

and stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | Forin of string * expr * stmt
  | While of expr * stmt
  | VarDecl of bind
  | VarDeclAssign of bind * expr

type program = bind list * func_decl list



(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"
  | Inc -> "++"
```

```
  | Dec -> "--"

let string_of_high_typ = function
    LIST -> "List"
  | DICT -> "Dict"
  | SET ->  "Set"

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | String -> "string"
  | Func -> "func"
  | CompositeType(ht, tl) ->
      string_of_high_typ ht ^ "<" ^ String.concat ", " (List.map string_of_typ tl) ^ ">"
  | FunctionType(rt::rest) ->
      string_of_typ rt ^ ":" ^ String.concat ", " (List.map string_of_typ rest) ^ ""

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_formal (t, id) = string_of_typ t ^ " " ^ id

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | StrLit(s) -> "\"" ^ s ^ "\""
  | FunctionLit(l) -> string_of_typ l.typ ^ " " ^
    l.fname ^ "(" ^ String.concat ", " (List.map string_of_formal l.formals) ^
    ")\n{}\n"
  | Id(s) -> s
  | IndexAssign(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ "] = " ^ string_of_expr e2
  | GetIndex(v, e2) -> v ^ "[" ^ string_of_expr e2 ^ "]"
  | GetSlice(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ ":" ^ string_of_expr e2 ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) ->
    (match o with
    | Neg | Not -> string_of_uop o ^ string_of_expr e
    | Inc | Dec -> string_of_expr e ^ string_of_uop o
    )
  | ListExpr(el) -> "[ " ^ String.concat ", " (List.map string_of_expr el) ^ " ]"
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | IndexUnop(v, e, o) ->
      v ^ "[" ^ string_of_expr e ^ "]" ^ string_of_uop o
  | Noexpr -> ""
```

```
let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | Forin(id1, id2, s) ->
      "for " ^ id1 ^ " in " ^ string_of_expr id2 ^ " " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | VarDecl(t, id) -> string_of_typ t ^ " " ^ id ^ ";\n"
  | VarDeclAssign((t, id), e) -> string_of_typ t ^ " " ^ id ^ " = " ^ string_of_expr e ^
     ";\n"



let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_formal fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

A.4 semant.ml

```
(* Semantic checking for the FFBB compiler *)

open Ast
open Sast
module StringMap = Map.Make (String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)
let local_sfunc = ref []

let function_decls = ref StringMap.empty

```

```
let check (globals, functions) =
  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter
      (function
        | Void, b -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
        | _ -> ())
      binds;
    let rec dups = function
      | [] -> ()
      | (_, n1) :: (_, n2) :: _ when n1 = n2 ->
          raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in
    dups (List.sort (fun (_, a) (_, b) -> compare a b) binds)
  in

  (* Verify the new binding to be added is not a duplicate *)
  let check_dup (binding : bind) (bindings : bindtbl) =
    match binding with
    | Void, s -> raise (Failure ("illegal void local " ^ s))
    | _, s ->
        if Hashtbl.mem bindings s then raise (Failure ("duplicate local " ^ s))
        else ()
  in

  (**** Check global variables ****)
  check_binds "global" globals;

  (**** Check functions ****)
  let lambda_num = ref 0 in

  (* Collect function declarations for built-in functions: no bodies *)
  let get_atypes fdecl = List.map (fun x -> fst x) fdecl.formals in
  let get_ftype fdecl = FunctionType (fdecl.typ :: get_atypes fdecl) in
  let built_in_decls =
    let add_bind map (name, return_ty, tys) =
      StringMap.add name
        {
          typ = return_ty;
          fname = name;
          formals =
            (match List.length tys with
            | 0 -> []
            | 1 -> [ (List.nth tys 0, "arg1") ]
            | 2 -> [ (List.nth tys 0, "arg1"); (List.nth tys 1, "arg2") ]
            | 3 ->
                [
                  (List.nth tys 0, "arg1");
                  (List.nth tys 1, "arg2");
```

```
                  (List.nth tys 2, "arg3");
                ]
            | _ -> raise (Failure "Built-in function must take 0-3 arguments"));
          body = [];
        }
        map
    in
    List.fold_left add_bind StringMap.empty
      [
        ("print", Void, [ Int ]);
        ("printb", Void, [ Bool ]);
        ("prints", Void, [ String ]);
        ("printf", Void, [ Float ]);
        ("printbig", Void, [ Int ]);
        ("setAdd", Void, [ CompositeType (SET, [ Int ]); Int ]);
        ("setFind", Bool, [ CompositeType (SET, [ Int ]); Int ]);
        ("setRemove", Void, [ CompositeType (SET, [ Int ]); Int ]);
        ("setSize", Int, [ CompositeType (SET, [ Int ]) ]);
        ("dictAdd", Void, [ CompositeType (DICT, [ Int; Int ]); Int; Int ]);
        ("dictSize", Int, [ CompositeType (DICT, [ Int; Int ]) ]);
        ("dictHasKey", Bool, [ CompositeType (DICT, [ Int; Int ]); Int ]);
        ("dictGetInt", Int, [ CompositeType (DICT, [ Int; Int ]); Int ]);
        ("dictGetBool", Bool, [ CompositeType (DICT, [ Int; Int ]); Int ]);
        ("dictGetFloat", Float, [ CompositeType (DICT, [ Int; Int ]); Int ]);
        ("dictGetList", CompositeType (LIST, [Int]), [ CompositeType (DICT, [ Int; Int
 ]); Int ]);
        ("dictRemove", Int, [ CompositeType (DICT, [ Int; Int ]); Int ]);
        ("len", Int, [ CompositeType (LIST, [ Void ]) ]);
        ("range", CompositeType (LIST, [ Int ]), [ Int ]);
        ("append", Void, [ CompositeType (LIST, [ Void ]); Void ]);
        ("slice", String, [ String; Int; Int ]);
        ("length", Int, [ String ]);
        ("concat", String, [ String; String ]);
      ]
  in

  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *) in
    match fd with
    (* No duplicate functions or redefinitions of built-ins *)
    | _ when StringMap.mem n built_in_decls -> make_err built_in_err
    | _ when StringMap.mem n map -> make_err dup_err
    | _ -> StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
```

```
  let _ =
    ignore (function_decls := List.fold_left add_func built_in_decls functions)
  in

  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s !function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let _ = find_func "main" in

  (* Ensure "main" is defined *)
  let rec check_function func =
    let bindings = Hashtbl.create 10 in
    (* Make sure no formals are void or duplicates *)
    check_binds "formal" func.formals;

    let add_formal_to_bindings formal =
      Hashtbl.add bindings (snd formal) (fst formal)
    in
    List.iter add_formal_to_bindings func.formals;

    (* Raise an exception if the given rvalue type cannot be assigned to
       the given lvalue type *)
    let check_assign fname lvaluet rvaluet err =
      match fname with
      | "append" | "len" | "setAdd" | "setFind" | "setRemove" | "setSize"
      | "dictAdd" | "dictHasKey" | "dictGetInt" | "dictGetBool" | "dictGetFloat" | "dictGetList"
      | "dictRemove" | "dictSize" ->
          lvaluet
      | _ -> if lvaluet = rvaluet then lvaluet else raise (Failure err)
    in

    (* Raise an exception if the given rvalue type cannot be assigned to
       the given lvalue type *)
    let get_list_prim_typ t =
      match t with CompositeType (LIST, [ typ ]) -> typ
    in
    let check_get_index lvaluet rvaluet err = get_list_prim_typ lvaluet in

    (* Build local symbol table of variables for this function *)
    let symbols =
      List.fold_left
        (fun m (ty, name) -> StringMap.add name ty m)
        StringMap.empty globals
    in

    (* Return a variable from our local symbol table *)
```

```
    let type_of_identifier s =
      try Hashtbl.find bindings s
      with Not_found -> (
        try StringMap.find s symbols
        with Not_found -> raise (Failure ("undeclared identifier " ^ s)))
    in

    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr = function
      | Literal l -> (Int, SLiteral l)
      | Fliteral l -> (Float, SFliteral l)
      | BoolLit l -> (Bool, SBoolLit l)
      | FunctionLit l ->
          function_decls := List.fold_left add_func !function_decls [ l ];
          let local_sf = check_function l in
          let _ = local_sfunc := local_sf :: !local_sfunc in
          (get_ftype l, SFunctionLit local_sf)
      | StrLit l -> (String, SStrLit l)
      | Noexpr -> (Void, SNoexpr)
      | Id s -> (type_of_identifier s, SId s)
      | IndexAssign (var, e1, e2) as ex ->
          let lt = type_of_identifier var
          and rt1, e1' = expr e1
          and rt2, e2' = expr e2 in
          let err = "illegal index assignment" in
          (lt, SIndexAssign (var, (rt1, e1'), (rt2, e2')))
      | GetIndex (var, e2) as ex ->
          let lt = type_of_identifier var and rt, e' = expr e2 in
          let err =
            "illegal index access " ^ string_of_typ lt ^ "[" ^ string_of_expr ex
            ^ "]"
          in
          (check_get_index lt rt err, SGetIndex (var, (rt, e')))
      | GetSlice (var, e1, e2) as ex ->
          let lt = type_of_identifier var
          and rt1, e1' = expr e1
          and rt2, e2' = expr e2 in
          let err =
            "illegal slice access " ^ string_of_typ lt ^ "[" ^ string_of_expr ex
            ^ "]"
          in
          (lt, SGetSlice (var, (rt1, e1'), (rt2, e2')))
      | Assign (var, e) as ex ->
          let lt = type_of_identifier var and rt, e' = expr e in
          let err =
            "illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ rt
            ^ " in " ^ string_of_expr ex
          in
          (check_assign var lt rt err, SAssign (var, (rt, e')))
      | ListExpr el as ex ->
```

```
          let check_list e =
            let et, e' = expr e in
            (et, e')
          in
          let el' = List.map check_list el in
          let head = List.hd el' in
          let get_rt (a, _) = [ a ] in
          (CompositeType (LIST, get_rt head), SListExpr el')
      | Unop (op, e) as ex -> (
          let t, e' = expr e in
          let ty =
            match op with
            | Neg when t = Int || t = Float -> t
            | Not when t = Bool -> Bool
            | (Inc | Dec) when t = Int -> t
            | Inc | Dec ->
                raise
                  (Failure
                     ("illegal unary operator " ^ string_of_typ t
                    ^ string_of_uop op ^ " in " ^ string_of_expr ex))
            | _ ->
                raise
                  (Failure
                     ("illegal unary operator " ^ string_of_uop op
                    ^ string_of_typ t ^ " in " ^ string_of_expr ex))
          in
          match ex with
          | Unop (Neg, _) | Unop (Not, _) -> (ty, SUnop (op, (t, e')))
          | Unop (Inc, Id s) ->
              (ty, SAssign (s, (t, SBinop ((t, SId s), Add, (t, SLiteral 1)))))
          | Unop (Dec, Id s) ->
              (ty, SAssign (s, (t, SBinop ((t, SId s), Sub, (t, SLiteral 1))))))
      | IndexUnop (v, e, o) as ex -> (
          let list_t = type_of_identifier v and index_t, e' = expr e in
          let err =
            "illegal index access " ^ string_of_typ list_t ^ "["
            ^ string_of_expr ex ^ "]"
          in
          let index_element =
            (check_get_index list_t index_t err, SGetIndex (v, (index_t, e')))
          in
          match o with
          | Inc ->
              ( list_t,
                SIndexAssign
                  ( v,
                    (index_t, e'),
                    (Int, SBinop (index_element, Add, (Int, SLiteral 1))) ) )
          | Dec ->
              ( list_t,
```

```
                SIndexAssign
                  ( v,
                    (index_t, e'),
                    (Int, SBinop (index_element, Sub, (Int, SLiteral 1))) ) )
          | _ -> raise (Failure "Case should never be reached"))
      | Binop (e1, op, e2) as e ->
          let t1, e1' = expr e1 and t2, e2' = expr e2 in
          (* All binary operators require operands of the same type *)
          let same = t1 = t2 in
          (* Determine expression type based on operator and operand types *)
          let ty =
            match op with
            | (Add | Sub | Mult | Div) when same && t1 = Int -> Int
            | (Add | Sub | Mult | Div) when same && t1 = Float -> Float
            | (Equal | Neq) when same -> Bool
            | (Less | Leq | Greater | Geq) when same && (t1 = Int || t1 = Float)
              ->
                Bool
            | (And | Or) when same && t1 = Bool -> Bool
            | _ ->
                raise
                  (Failure
                     ("illegal binary operator " ^ string_of_typ t1 ^ " "
                    ^ string_of_op op ^ " " ^ string_of_typ t2 ^ " in "
                    ^ string_of_expr e))
          in
          (ty, SBinop ((t1, e1'), op, (t2, e2')))
      | Call (fname, args) as call ->
          let fd =
            if StringMap.mem fname !function_decls then find_func fname
            else if Hashtbl.mem bindings fname then
              match type_of_identifier fname with
              | FunctionType (rt :: rest) ->
                  {
                    typ = rt;
                    fname;
                    formals = List.mapi (fun i x -> (x, string_of_int i)) rest;
                    body = [];
                  }
              | _ -> raise (Failure ("unrecognized function " ^ fname))
            else raise (Failure ("unrecognized function " ^ fname))
          in
          let param_length = List.length fd.formals in
          if List.length args != param_length then
            raise
              (Failure
                 ("expecting " ^ string_of_int param_length ^ " arguments in "
                ^ string_of_expr call))
          else
            (* check if formal_type == expr_type *)
```

```
          let _check_call fname ft e =
            let et, e' = expr e in
            let err =
              "illegal argument found " ^ string_of_typ et ^ " expected "
              ^ string_of_typ ft ^ " in " ^ string_of_expr e
            in
            (check_assign fname ft et err, e')
          in
          let get_et e =
            let et, e' = expr e in
            et
          in
          let err_msg = fname ^ " type incorrect" in
          let err_msg_2 =
            "first argument of " ^ fname ^ " should have composite type"
          in
          (* set arg_idx to -1 when internal type check is no needed *)
          let check_custom_fun_type exp_high_typ len arg_idxs =
            match List.map get_et args with
            | CompositeType (high_typ, [ prim_typ ]) :: rest ->
                if
                  List.length rest = len
                  && high_typ = exp_high_typ
                  &&
                  match arg_idxs with
                  | [ arg_idx ] -> prim_typ = List.nth rest arg_idx
                  | [] -> true
                  | _ -> raise (Failure "Internal error at arg_idxs")
                then fname
                else raise (Failure err_msg)
            | CompositeType (high_typ, [ key_typ; value_typ ]) :: rest ->
                if
                  List.length rest = len
                  && high_typ = exp_high_typ
                  &&
                  match arg_idxs with
                  | [ arg_idx1; arg_idx2 ] ->
                      key_typ = List.nth rest arg_idx1
                      && value_typ = List.nth rest arg_idx2
                  | [ arg_idx ] -> key_typ = List.nth rest arg_idx
                  | [] -> true
                  | _ -> raise (Failure "Internal error at arg_idxs")
                then fname
                else raise (Failure err_msg)
            | _ -> raise (Failure err_msg_2)
          in
          let _ =
            match fname with
            | "append" -> check_custom_fun_type LIST 1 [ 0 ]
            | "len" -> check_custom_fun_type LIST 0 []
```

```
              | "setAdd" -> check_custom_fun_type SET 1 [ 0 ]
              | "setFind" -> check_custom_fun_type SET 1 [ 0 ]
              | "setRemove" -> check_custom_fun_type SET 1 [ 0 ]
              | "setSize" -> check_custom_fun_type SET 0 []
              | "dictAdd" -> check_custom_fun_type DICT 2 [ 0; 1 ]
              | "dictHasKey" -> check_custom_fun_type DICT 1 [ 0 ]
              | "dictGetInt" -> check_custom_fun_type DICT 1 [ 0 ]
              | "dictGetBool" -> check_custom_fun_type DICT 1 [ 0 ]
              | "dictGetFloat" -> check_custom_fun_type DICT 1 [ 0 ]
              | "dictGetList" -> check_custom_fun_type DICT 1 [ 0 ]
              | "dictRemove" -> check_custom_fun_type DICT 1 [ 0 ]
              | "dictSize" -> check_custom_fun_type DICT 0 []
              | _ -> fname
            in
            let check_call (ft, _) e = _check_call fname ft e in
            let args' = List.map2 check_call fd.formals args in
            (fd.typ, SCall (fname, args'))
    in

    let check_bool_expr e =
      let t', e' = expr e
      and err = "expected Boolean expression in " ^ string_of_expr e in
      if t' != Bool then raise (Failure err) else (t', e')
    in

    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt = function
      | Expr e -> SExpr (expr e)
      | If (p, b1, b2) -> SIf (check_bool_expr p, check_stmt b1, check_stmt b2)
      | For (e1, e2, e3, st) ->
          SFor (expr e1, check_bool_expr e2, expr e3, check_stmt st)
      | Forin (e1, e2, st) ->
          lambda_num := !lambda_num + 1;
          let rt, e2' = expr e2 in
          let lt = get_list_prim_typ rt in
          let iter_n = "iter" ^ string_of_int !lambda_num in
          let clist_n = "clist" ^ string_of_int !lambda_num in
          let iter = check_stmt (VarDeclAssign ((Int, iter_n), Literal 0)) in
          let clist = check_stmt (VarDeclAssign ((rt, clist_n), e2)) in
          let pred =
            expr (Binop (Id iter_n, Less, Call ("len", [ Id clist_n ])))
          in
          let inc = expr (Unop (Inc, Id iter_n)) in
          let x =
            check_stmt (VarDeclAssign ((lt, e1), GetIndex (clist_n, Id iter_n)))
          in
          let next = expr (Assign (e1, GetIndex (clist_n, Id iter_n))) in
          SForin (iter, clist, x, pred, inc, next, check_stmt st)
      | While (p, s) -> SWhile (check_bool_expr p, check_stmt s)
      | Return e ->
```

```
          let t, e' = expr e in
          if t = func.typ then SReturn (t, e')
          else
            raise
              (Failure
                 ("return gives " ^ string_of_typ t ^ " expected "
                ^ string_of_typ func.typ ^ " in " ^ string_of_expr e))
      (* A block is correct if each statement is correct and nothing
         follows any Return statement.  Nested blocks are flattened. *)
      | Block sl ->
          let rec check_stmt_list = function
            | [ (Return _ as s) ] -> [ check_stmt s ]
            | Return _ :: _ -> raise (Failure "nothing may follow a return")
            | Block sl :: ss -> check_stmt_list (sl @ ss) (* Flatten blocks *)
            | s :: ss ->
                let tmp = check_stmt s in
                tmp :: check_stmt_list ss
            | [] -> []
          in
          SBlock (check_stmt_list sl)
      | VarDecl (t, s) ->
          check_dup (t, s) bindings;
          Hashtbl.add bindings s t;
          SVarDecl (t, s)
      | VarDeclAssign ((t, s), e) as stmt ->
          check_dup (t, s) bindings;
          Hashtbl.add bindings s t;
          let lt = t and rt, e' = expr e in
          let err =
            "illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ rt
            ^ " in " ^ string_of_stmt stmt
          in
          let tmp =
            (check_assign s lt rt err, SVarDeclAssign ((t, s), (rt, e')))
          in
          snd tmp
    in

    (* body of check_function *)
    {
      styp = func.typ;
      sfname = func.fname;
      sformals = func.formals;
      sftype = get_ftype func;
      slocals = bindings;
      sbody =
        (match check_stmt (Block func.body) with
        | SBlock sl -> sl
        | _ -> raise (Failure "internal error: block didn't become a block?"));
    }
```

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

A.5 sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast

type sexpr = typ * sx

and sfunc_decl = {
    styp : typ;
    sfname : string;
    sftype : typ;
    sformals : bind list;
    slocals : bindtbl;
    sbody : sstmt list;
  }

and sx =
  | SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SStrLit of string
  | SFunctionLit of sfunc_decl
  | SIndexAssign of string * sexpr * sexpr
  | SGetSlice of string * sexpr * sexpr
  | SGetIndex of string * sexpr
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SListExpr of sexpr list
  | SCall of string * sexpr list
  | SNoexpr

and sstmt =
  | SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SForin of sstmt * sstmt * sstmt * sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SVarDecl of bind
  | SVarDeclAssign of bind * sexpr
```

```
and bindtbl = (string, typ) Hashtbl.t

and sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : "
  ^ (match e with
    | SLiteral l -> string_of_int l
    | SBoolLit true -> "true"
    | SBoolLit false -> "false"
    | SFliteral l -> l
    | SFunctionLit(l) -> string_of_typ l.styp ^ " " ^
      l.sfname ^ "(" ^ String.concat ", " (List.map string_of_formal l.sformals) ^
      ")\n{}\n"
    | SStrLit(l) -> "\"" ^ l ^ "\""
    | SId s -> s
    | SIndexAssign (v, e1, e2) ->
        v ^ "[" ^ string_of_sexpr e1 ^ "] = " ^ string_of_sexpr e2
    | SGetIndex (e1, e2) -> e1 ^ "[" ^ string_of_sexpr e2 ^ "]"
    | SGetSlice(v, e1, e2) -> v ^ "[" ^ string_of_sexpr e1 ^ ":" ^ string_of_sexpr e2 ^
   "]"
    | SBinop (e1, o, e2) ->
        string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
    | SUnop (o, e) -> string_of_uop o ^ string_of_sexpr e
    | SListExpr el ->
        "[ " ^ String.concat ", " (List.map string_of_sexpr el) ^ " ]"
    | SAssign (v, e) -> v ^ " = " ^ string_of_sexpr e
    | SCall (f, el) ->
        f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
    | SNoexpr -> "")
  ^ ")"

let rec string_of_sstmt = function
  | SBlock stmts ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr expr -> string_of_sexpr expr ^ ";\n"
  (* | SVarDecl(expr) -> "decalre " ^ string_of_sexpr expr ^ ";\n"; *)
  | SReturn expr -> "return " ^ string_of_sexpr expr ^ ";\n"
  | SIf (e, s, SBlock []) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SForin(s1, s2, s3, e1, e2, e3, s) ->
      string_of_sstmt s1 ^ string_of_sstmt s2 ^ string_of_sstmt s3 ^
```

```
      "while(" ^ string_of_sexpr e1 ^ ") {\n" ^ string_of_sstmt s
      ^ ", " ^ string_of_sexpr e2 ^ ")" ^ string_of_sexpr e3 ^ "}\n"
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s
  | SVarDecl(t, id) -> string_of_typ t ^ " " ^ id ^ ";\n"
  | SVarDeclAssign((t, id), e) -> string_of_typ t ^ " " ^ id ^ " = " ^ string_of_sexpr e
    ^ ";\n"

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map string_of_typ (List.map fst fdecl.
    sformals)) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars)
  ^ "\n"
  ^ String.concat "\n" (List.map string_of_sfdecl funcs)
```

A.6 codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

module L = Llvm
module A = Ast
open Sast
module StringMap = Map.Make (String)

(* translate : Sast.program -> Llvm.module *)
let translate ((globals, functions), top_funcs) =
  let context = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "FFBB" in
```

```
(* Get types from the context *)
let i32_t = L.i32_type context
and i8_t = L.i8_type context
and i1_t = L.i1_type context
and float_t = L.double_type context
and str_t = L.pointer_type (L.i8_type context)
and void_t = L.void_type context in
let list_t typ ltyp =
  let typ_str = A.string_of_typ (A.CompositeType (A.LIST, [ typ ])) in
  let t = L.named_struct_type context typ_str in
  let list_body =
    [| i32_t; i32_t; i32_t; L.pointer_type ltyp |]
    (* length, type, capacity, pointer *)
  in
  ignore (L.struct_set_body t list_body false);
  t
in
let list_int_t = list_t A.Int i32_t
and list_bool_t = list_t A.Bool i1_t
and list_float_t = list_t A.Float float_t in
let set_t =
  let t = L.named_struct_type context "TreeSet" in
  let set_body = [| i32_t; i32_t; L.pointer_type i8_t |] in
  ignore (L.struct_set_body t set_body false);
  t
in
let dict_t =
  let t = L.named_struct_type context "TreeDict" in
  let set_body = [| i32_t; i32_t; L.pointer_type i8_t |] in
  ignore (L.struct_set_body t set_body false);
  t
in
(* Return the LLVM type for a FFBB type *)
let rec ltype_of_typ = function
  | A.Int -> i32_t
  | A.Bool -> i1_t
  | A.Float -> float_t
  | A.Void -> void_t
  | A.String -> str_t
  | A.FunctionType (rt :: rest) ->
      let llargs =
        List.fold_left (fun l arg -> ltype_of_typ arg :: l) [] (List.rev rest)
      in
      L.pointer_type
        (L.function_type (ltype_of_typ rt) (Array.of_list llargs))
  | A.CompositeType (A.LIST, [ A.Int ]) -> L.pointer_type list_int_t
  | A.CompositeType (A.LIST, [ A.Bool ]) -> L.pointer_type list_bool_t
  | A.CompositeType (A.LIST, [ A.Float ]) -> L.pointer_type list_float_t
  | A.CompositeType (A.SET, [ _ ]) -> L.pointer_type set_t
```

```
    | A.CompositeType (A.DICT, [ _; _ ]) -> L.pointer_type dict_t
  in

  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
    let global_var m (t, n) =
      let init =
        match t with
        | A.Float -> L.const_float (ltype_of_typ t) 0.0
        | _ -> L.const_int (ltype_of_typ t) 0
      in
      StringMap.add n (L.define_global n init the_module) m
    in
    List.fold_left global_var StringMap.empty globals
  in

  let printf_t : L.lltype =
    L.var_arg_function_type i32_t [| L.pointer_type i8_t |]
  in
  let printf_func : L.llvalue =
    L.declare_function "printf" printf_t the_module
  in

  let printbig_t : L.lltype = L.function_type i32_t [| i32_t |] in
  let printbig_func : L.llvalue =
    L.declare_function "printbig" printbig_t the_module
  in

  let slice_t : L.lltype =
    L.function_type str_t [| L.pointer_type i8_t; i32_t; i32_t |]
  in
  let slice_func : L.llvalue = L.declare_function "slice" slice_t the_module in

  let length_t : L.lltype = L.function_type i32_t [| L.pointer_type i8_t |] in
  let length_func : L.llvalue =
    L.declare_function "length" length_t the_module
  in

  let concat_t : L.lltype =
    L.function_type str_t [| L.pointer_type i8_t; L.pointer_type i8_t |]
  in
  let concat_func : L.llvalue =
    L.declare_function "concat" concat_t the_module
  in

  let range_t : L.lltype =
    L.function_type (L.pointer_type list_int_t) [| i32_t |]
  in
  let range_func : L.llvalue = L.declare_function "range" range_t the_module in

```

```
let create_empty_list_int_t : L.lltype =
  L.function_type (L.pointer_type list_int_t) [| i32_t |]
in
let create_empty_list_int_func : L.llvalue = L.declare_function "create_empty_list_int" create_empty_list_int_t the_module in

let create_empty_list_bool_t : L.lltype =
  L.function_type (L.pointer_type list_bool_t) [| i32_t |]
in
let create_empty_list_bool_func : L.llvalue = L.declare_function "create_empty_list_bool" create_empty_list_bool_t the_module in

let create_empty_list_float_t : L.lltype =
  L.function_type (L.pointer_type list_float_t) [| i32_t |]
in
let create_empty_list_float_func : L.llvalue = L.declare_function "create_empty_list_float" create_empty_list_float_t the_module in

let len_t : L.lltype = L.function_type i32_t [| L.pointer_type i8_t |] in
let len_func : L.llvalue = L.declare_function "len" len_t the_module in

let append_t : L.lltype =
  L.function_type i32_t [| L.pointer_type i8_t; L.pointer_type i8_t |]
in
let append_func : L.llvalue =
  L.declare_function "append" append_t the_module
in

let idxtrans_t : L.lltype =
  L.function_type i32_t [| L.pointer_type i8_t; i32_t |]
in
let idxtrans_func : L.llvalue =
  L.declare_function "index_transform" idxtrans_t the_module
in

let list_slice_t : L.lltype =
  L.function_type
    (L.pointer_type list_int_t)
    [| L.pointer_type i8_t; i32_t; i32_t |]
in
let list_slice_func : L.llvalue =
  L.declare_function "list_slice" list_slice_t the_module
in

let create_empty_set_t : L.lltype =
  L.function_type (L.pointer_type set_t) [| i32_t |]
in
let create_empty_set_func : L.llvalue =
  L.declare_function "create_empty_set" create_empty_set_t the_module
in
```

```
let add_elem_t : L.lltype =
  L.function_type i32_t [| L.pointer_type set_t; L.pointer_type i8_t |]
in
let add_elem_func : L.llvalue =
  L.declare_function "add_elem" add_elem_t the_module
in

let search_elem_t : L.lltype =
  L.function_type i1_t [| L.pointer_type set_t; L.pointer_type i8_t |]
in
let search_elem_func : L.llvalue =
  L.declare_function "search_elem" search_elem_t the_module
in

let delete_elem_t : L.lltype =
  L.function_type i32_t [| L.pointer_type set_t; L.pointer_type i8_t |]
in
let delete_elem_func : L.llvalue =
  L.declare_function "delete_elem" delete_elem_t the_module
in

let get_set_size_t : L.lltype =
  L.function_type i32_t [| L.pointer_type set_t |]
in
let get_set_size_func : L.llvalue =
  L.declare_function "get_set_size" get_set_size_t the_module
in

let create_empty_dict_t : L.lltype =
  L.function_type (L.pointer_type dict_t) [| i32_t; i32_t |]
in
let create_empty_dict_func : L.llvalue =
  L.declare_function "create_empty_dict" create_empty_dict_t the_module
in

let add_key_value_t : L.lltype =
  L.function_type i32_t
    [| L.pointer_type dict_t; L.pointer_type i8_t; L.pointer_type i8_t |]
in
let add_key_value_func : L.llvalue =
  L.declare_function "add_key_value" add_key_value_t the_module
in

let get_dict_size_t : L.lltype =
  L.function_type i32_t [| L.pointer_type dict_t |]
in
let get_dict_size_func : L.llvalue =
  L.declare_function "get_dict_size" get_dict_size_t the_module
in
```

```

  let key_exists_t : L.lltype =
    L.function_type i1_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
  in
  let key_exists_func : L.llvalue =
    L.declare_function "key_exists" key_exists_t the_module
  in

  let retrieve_value_int_t : L.lltype =
    L.function_type i32_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
  in
  let retrieve_value_int_func : L.llvalue =
    L.declare_function "retrieve_value_int" retrieve_value_int_t the_module
  in

  let retrieve_value_float_t : L.lltype =
    L.function_type float_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
  in
  let retrieve_value_float_func : L.llvalue =
    L.declare_function "retrieve_value_float" retrieve_value_float_t the_module
  in

  let retrieve_value_bool_t : L.lltype =
    L.function_type i1_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
  in
  let retrieve_value_bool_func : L.llvalue =
    L.declare_function "retrieve_value_bool" retrieve_value_bool_t the_module
  in

  let retrieve_value_list_t : L.lltype =
    L.function_type (L.pointer_type list_int_t) [| L.pointer_type dict_t; L.pointer_type
     i8_t |]
  in
  let retrieve_value_list_func : L.llvalue =
    L.declare_function "retrieve_value_list" retrieve_value_list_t the_module
  in

  let delete_key_t : L.lltype =
    L.function_type i32_t [| L.pointer_type dict_t; L.pointer_type i8_t |]
  in
  let delete_key_func : L.llvalue =
    L.declare_function "delete_key" delete_key_t the_module
  in

  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.sfname
      and formal_types =
```

```
        Array.of_list (List.map (fun (t, _) -> ltype_of_typ t) fdecl.sformals)
      in
      let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m
    in
    List.fold_left function_decl StringMap.empty functions
  in

  (* Fill in the body of the given function *)
  let rec build_function_body fdecl =
    let the_function, _ = StringMap.find fdecl.sfname function_decls in
    let builder = L.builder_at_end context (L.entry_block the_function) in

    let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
    and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
    and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder in

    (* Construct the function's "locals": formal arguments and locally
       declared variables.  Allocate each on the stack, initialize their
       value, if appropriate, and remember their values in the "locals" map *)
    let local_vars =
      let local_vars_tbl = Hashtbl.create 10 in
      let add_formal (t, n) p =
        L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n builder in
        ignore (L.build_store p local builder);
        Hashtbl.add local_vars_tbl n local
      in

      List.iter2 add_formal fdecl.sformals
        (Array.to_list (L.params the_function));
      local_vars_tbl
    in

    (* Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let lookup n =
      try Hashtbl.find local_vars n
      with Not_found -> StringMap.find n global_vars
    in

    let list_type_to_idx typ =
      match typ with
      | A.CompositeType (A.LIST, [ A.Int ]) -> 0
      | A.CompositeType (A.LIST, [ A.Bool ]) -> 1
      | A.CompositeType (A.LIST, [ A.Float ]) -> 2
      (* | A.CompositeType(A.LIST, [A.String]) -> 3 *)
      | _ -> raise (Failure "Invalid list type")
    in
    let list_prim_type typ =
```

```
    match typ with
    | A.CompositeType (A.LIST, [ t ]) -> t
    | _ -> raise (Failure "Invalid list type")
  in

  let type_to_idx_map typ =
    match typ with
    | A.Int -> 0
    | A.Bool -> 1
    | A.Float -> 2
    | A.CompositeType(LIST, _) -> 4
    | _ ->
        raise (Failure "Internal error. Prime type has to be int/bool/float")
  in

  (* Construct code for an expression; return its value *)
  let rec expr builder ((et, e) : sexpr) =
    let alloc_var_and_return_voidptr_from_expr e =
      let e' = expr builder e in
      let p_e' = L.build_alloca (L.type_of e') "ptmp" builder in
      ignore (L.build_store e' p_e' builder);
      L.build_bitcast p_e' (L.pointer_type i8_t) "tmpvoidptr" builder
    in

    match e with
    | SLiteral i -> L.const_int i32_t i
    | SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
    | SFliteral l -> L.const_float_of_string float_t l
    | SStrLit l -> L.build_global_stringptr l "string" builder
    (*@ *)
    (* | SFunctionLit s -> L.build_load (lookup s) s builder *)
    | SNoexpr -> L.const_int i32_t 0
    | SId s -> L.build_load (lookup s) s builder
    | SFunctionLit sfunc ->
        build_function_body sfunc;
        let fdef, fdecl = StringMap.find sfunc.sfname function_decls in
        fdef
    | SListExpr el ->
        let vl = List.map (expr builder) el in
        let e_typ = L.pointer_type (L.type_of (List.hd vl)) in
        let size = L.const_int i32_t (List.length vl) in
        let prim_type = list_prim_type et in
        let list_typ =
          match prim_type with
          | A.Int -> list_int_t
          | A.Bool -> list_bool_t
          | A.Float -> list_float_t
        in
        let typ_idx = list_type_to_idx et in
        (* Declare LIST *)
```

```
        let list = L.build_array_malloc e_typ size "list" builder in
        let list = L.build_pointercast list e_typ "list" builder in
        let list_struct = L.build_malloc list_typ "list_struct" builder in
        (* variable storing current size of list *)
        let list_size =
          L.build_struct_gep list_struct 0 "list_size" builder
        in
        ignore
          (L.build_store
             (L.const_int i32_t (List.length vl))
             list_size builder);
        (* variable storing type of list *)
        let list_type =
          L.build_struct_gep list_struct 1 "list_type" builder
        in
        ignore (L.build_store (L.const_int i32_t typ_idx) list_type builder);
        (* variable storing capacity of list, used later by append() *)
        let list_capacity =
          L.build_struct_gep list_struct 2 "list_capacity" builder
        in
        ignore
          (L.build_store
             (L.const_int i32_t (List.length vl))
             list_capacity builder);
        (* assign actual list with values *)
        let assign i v =
          let vp =
            L.build_gep list [| L.const_int i32_t i |] "list_v" builder
          in
          ignore (L.build_store v vp builder)
        in
        List.iteri assign vl;
        (* pointer to the actual list *)
        let list_ptr =
          L.build_struct_gep list_struct 3 "list_struct" builder
        in
        ignore (L.build_store list list_ptr builder);
        list_struct
    | SIndexAssign (s, e1, e2) ->
        let idx = expr builder e1
        and value = expr builder e2
        and list = L.build_load (lookup s) "getIndex" builder in
        (* Cast list to void * to send to len function *)
        let p_list = L.build_alloca (L.type_of list) "pcst" builder in
        ignore (L.build_store list p_list builder);
        let p_list =
          L.build_bitcast p_list (L.pointer_type i8_t) "cs" builder
        in
        let idx =
          L.build_call idxtrans_func [| p_list; idx |] "index_transform_call"
```

```
              builder
          in
          (* Get list pointer from struct *)
          let list = L.build_struct_gep list 3 "list_v" builder in
          let list = L.build_load list "list_v" builder in
          (* list[idx]*)
          let list_idx = L.build_gep list [| idx |] "getIndex_e" builder in
          ignore (L.build_store value list_idx builder);
          value
      | SAssign (s, e) ->
          let e' = expr builder e in
          ignore (L.build_store e' (lookup s) builder);
          e'
      | SGetIndex (s, e) ->
          let idx = expr builder e
          and list = L.build_load (lookup s) "getIndex" builder in
          let p_list = L.build_alloca (L.type_of list) "pcst" builder in
          ignore (L.build_store list p_list builder);
          let p_list =
            L.build_bitcast p_list (L.pointer_type i8_t) "cs" builder
          in
          let idx =
            L.build_call idxtrans_func [| p_list; idx |] "index_transform_call"
              builder
          in
          (* Get list pointer from struct *)
          let list = L.build_struct_gep list 3 "list_v" builder in
          let list = L.build_load list "list_v" builder in
          let sp = L.build_gep list [| idx |] "getIndex_e" builder in
          L.build_load sp "getIndex" builder
      | SGetSlice (s, e1, e2) ->
          let l = expr builder e1
          and r = expr builder e2
          and list = L.build_load (lookup s) "getlist" builder in
          (* Cast list to void * to send to len function *)
          let p_list = L.build_alloca (L.type_of list) "pcst" builder in
          ignore (L.build_store list p_list builder);
          let p_list =
            L.build_bitcast p_list (L.pointer_type i8_t) "cs" builder
          in
          (* Get list pointer from struct *)
          L.build_call list_slice_func [| p_list; l; r |] "list_slice_call"
            builder
      | SBinop (((A.Float, _) as e1), op, e2) ->
          let e1' = expr builder e1 and e2' = expr builder e2 in
          (match op with
          | A.Add -> L.build_fadd
          | A.Sub -> L.build_fsub
          | A.Mult -> L.build_fmul
          | A.Div -> L.build_fdiv
```

```
          | A.Equal -> L.build_fcmp L.Fcmp.Oeq
          | A.Neq -> L.build_fcmp L.Fcmp.One
          | A.Less -> L.build_fcmp L.Fcmp.Olt
          | A.Leq -> L.build_fcmp L.Fcmp.Ole
          | A.Greater -> L.build_fcmp L.Fcmp.Ogt
          | A.Geq -> L.build_fcmp L.Fcmp.Oge
          | A.And | A.Or ->
            raise
              (Failure
                 "internal error: semant should have rejected and/or on float"))
            e1' e2' "tmp" builder
      | SBinop (e1, op, e2) ->
          let e1' = expr builder e1 and e2' = expr builder e2 in
          (match op with
          | A.Add -> L.build_add
          | A.Sub -> L.build_sub
          | A.Mult -> L.build_mul
          | A.Div -> L.build_sdiv
          | A.And -> L.build_and
          | A.Or -> L.build_or
          | A.Equal -> L.build_icmp L.Icmp.Eq
          | A.Neq -> L.build_icmp L.Icmp.Ne
          | A.Less -> L.build_icmp L.Icmp.Slt
          | A.Leq -> L.build_icmp L.Icmp.Sle
          | A.Greater -> L.build_icmp L.Icmp.Sgt
          | A.Geq -> L.build_icmp L.Icmp.Sge)
            e1' e2' "tmp" builder
      | SUnop (op, ((t, _) as e)) ->
          let e' = expr builder e in
          (match op with
          | A.Neg when t = A.Float -> L.build_fneg
          | A.Neg -> L.build_neg
          | A.Not -> L.build_not)
            e' "tmp" builder
      | SCall ("print", [ e ]) | SCall ("printb", [ e ]) ->
          L.build_call printf_func
            [| int_format_str; expr builder e |]
            "printf" builder
      | SCall ("printbig", [ e ]) ->
          L.build_call printbig_func [| expr builder e |] "printbig" builder
      | SCall ("printf", [ e ]) ->
          L.build_call printf_func
            [| float_format_str; expr builder e |]
            "printf" builder
      | SCall ("prints", [ e ]) ->
          L.build_call printf_func
            [| string_format_str; expr builder e |]
            "printf" builder
      | SCall ("slice", [ e; e1; e2 ]) ->
          L.build_call slice_func
```

```
            [| expr builder e; expr builder e1; expr builder e2 |]
            "slice" builder
      | SCall ("length", [ e ]) ->
          L.build_call length_func [| expr builder e |] "length" builder
      | SCall ("concat", [ e; e1 ]) ->
          L.build_call concat_func
            [| expr builder e; expr builder e1 |]
            "concat" builder
      | SCall ("range", [ e ]) ->
          L.build_call range_func [| expr builder e |] "range_call" builder
      | SCall ("len", [ e ]) ->
          let e' = expr builder e in
          (* Cast list to void * to send to len function *)
          let p_e' = L.build_alloca (L.type_of e') "pcst" builder in
          ignore (L.build_store e' p_e' builder);
          let e' = L.build_bitcast p_e' (L.pointer_type i8_t) "cs" builder in
          L.build_call len_func [| e' |] "list_len_call" builder
      | SCall ("append", [ e1; e2 ]) ->
          let e1' = expr builder e1 in
          let e2' = expr builder e2 in
          (* Cast list to void * to send to append function *)
          let p_e1' = L.build_alloca (L.type_of e1') "pcst" builder in
          ignore (L.build_store e1' p_e1' builder);
          let p_e2' = L.build_alloca (L.type_of e2') "pcst" builder in
          ignore (L.build_store e2' p_e2' builder);
          let e1' = L.build_bitcast p_e1' (L.pointer_type i8_t) "cs" builder in
          let e2' = L.build_bitcast p_e2' (L.pointer_type i8_t) "cs" builder in
          L.build_call append_func [| e1'; e2' |] "list_append_call" builder
      | SCall ("setAdd", [ se; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call add_elem_func
            [| expr builder se; e_voidptr |]
            "add_elem" builder
      | SCall ("setFind", [ se; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call search_elem_func
            [| expr builder se; e_voidptr |]
            "search_elem" builder
      | SCall ("setRemove", [ se; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call delete_elem_func
            [| expr builder se; e_voidptr |]
            "delete_elem" builder
      | SCall ("setSize", [ se ]) ->
          L.build_call get_set_size_func
            [| expr builder se |]
            "get_set_size" builder
      | SCall ("dictAdd", [ de; e1; e2 ]) ->
          let e1_voidptr = alloc_var_and_return_voidptr_from_expr e1 in
          let e2_voidptr = alloc_var_and_return_voidptr_from_expr e2 in
```

```
          L.build_call add_key_value_func
            [| expr builder de; e1_voidptr; e2_voidptr |]
            "add_key_value" builder
      | SCall ("dictSize", [ de ]) ->
          L.build_call get_dict_size_func
            [| expr builder de |]
            "get_dict_size" builder
      | SCall ("dictHasKey", [ de; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call key_exists_func
            [| expr builder de; e_voidptr |]
            "key_exists" builder
      | SCall ("dictGetInt", [ de; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call retrieve_value_int_func
            [| expr builder de; e_voidptr |]
            "retrieve_value_int" builder
      | SCall ("dictGetBool", [ de; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call retrieve_value_bool_func
            [| expr builder de; e_voidptr |]
            "retrieve_value_bool" builder
      | SCall ("dictGetFloat", [ de; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call retrieve_value_float_func
            [| expr builder de; e_voidptr |]
            "retrieve_value_float" builder
      | SCall ("dictGetList", [ de; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call retrieve_value_list_func
            [| expr builder de; e_voidptr |]
            "retrieve_value_list" builder
      | SCall ("dictRemove", [ de; e ]) ->
          let e_voidptr = alloc_var_and_return_voidptr_from_expr e in
          L.build_call delete_key_func
            [| expr builder de; e_voidptr |]
            "delete_key" builder
      | SCall (f, args) ->
          let result = match et with A.Void -> "" | _ -> f ^ "_result" in
          let llargs = List.rev (List.map (expr builder) (List.rev args)) in
          let fdef =
            if StringMap.mem f function_decls then
              fst (StringMap.find f function_decls)
            else L.build_load (lookup f) f builder
          in
          L.build_call fdef (Array.of_list llargs) result builder
    in

    (* LLVM insists each basic block end with exactly one "terminator"
       instruction that transfers control.  This function runs "instr builder"
```

```
       if the current block does not already have a terminator.  Used,
       e.g., to handle the "fall off the end of the function" case. *)
    let add_terminal builder instr =
      match L.block_terminator (L.insertion_block builder) with
      | Some _ -> ()
      | None -> ignore (instr builder)
    in

    (* Build the code for the given statement; return the builder for
       the statement's successor (i.e., the next instruction will be built
       after the one generated by this call) *)
    let rec stmt builder = function
      | SBlock sl -> List.fold_left stmt builder sl
      | SExpr e ->
          ignore (expr builder e);
          builder
      | SReturn e ->
          ignore
            (match fdecl.styp with
            (* Special "return nothing" instr *)
            | A.Void -> L.build_ret_void builder (* Build return statement *)
            | _ -> L.build_ret (expr builder e) builder);
          builder
      | SIf (predicate, then_stmt, else_stmt) ->
          let bool_val = expr builder predicate in
          let merge_bb = L.append_block context "merge" the_function in
          let build_br_merge = L.build_br merge_bb in

          (* partial function *)
          let then_bb = L.append_block context "then" the_function in
          add_terminal
            (stmt (L.builder_at_end context then_bb) then_stmt)
            build_br_merge;

          let else_bb = L.append_block context "else" the_function in
          add_terminal
            (stmt (L.builder_at_end context else_bb) else_stmt)
            build_br_merge;

          ignore (L.build_cond_br bool_val then_bb else_bb builder);
          L.builder_at_end context merge_bb
      | SWhile (predicate, body) ->
          let pred_bb = L.append_block context "while" the_function in
          ignore (L.build_br pred_bb builder);

          let body_bb = L.append_block context "while_body" the_function in
          add_terminal
            (stmt (L.builder_at_end context body_bb) body)
            (L.build_br pred_bb);

```

```
          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = expr pred_builder predicate in

          let merge_bb = L.append_block context "merge" the_function in
          ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
          L.builder_at_end context merge_bb
      (* Implement for loops as while loops *)
      | SFor (e1, e2, e3, body) ->
          stmt builder
            (SBlock [ SExpr e1; SWhile (e2, SBlock [ body; SExpr e3 ]) ])
      | SForin (iter, clist, x, pred, inc, next, body) ->
          stmt builder
            (SBlock
               [
                 iter;
                 clist;
                 x;
                 SWhile (pred, SBlock [ body; SExpr inc; SExpr next ]);
               ])
      | SVarDecl (t, id) ->
          let local_var = L.build_alloca (ltype_of_typ t) id builder in
          Hashtbl.add local_vars id local_var;
          ignore
            (match t with
            | CompositeType (LIST, [ Int ]) ->
                ignore
                  (L.build_store
                     (L.build_call create_empty_list_int_func
                        [| L.const_int i32_t (type_to_idx_map Int) |]
                        "create_empty_list_int" builder)
                     local_var builder)
            | CompositeType (LIST, [ Bool ]) ->
                ignore
                  (L.build_store
                     (L.build_call create_empty_list_bool_func
                        [| L.const_int i32_t (type_to_idx_map Bool) |]
                        "create_empty_list_bool" builder)
                     local_var builder)
            | CompositeType (LIST, [ Float ]) ->
                ignore
                  (L.build_store
                     (L.build_call create_empty_list_float_func
                        [| L.const_int i32_t (type_to_idx_map Float) |]
                        "create_empty_list_float" builder)
                     local_var builder)
            | CompositeType (SET, [ prim_typ ]) ->
                ignore
                  (L.build_store
                     (L.build_call create_empty_set_func
                        [| L.const_int i32_t (type_to_idx_map prim_typ) |]
```

```
                            "create_empty_set" builder)
                         local_var builder)
              | CompositeType (DICT, [ key_typ; value_typ ]) ->
                  ignore
                    (L.build_store
                       (L.build_call create_empty_dict_func
                          [|
                            L.const_int i32_t (type_to_idx_map key_typ);
                            L.const_int i32_t (type_to_idx_map value_typ);
                          |]
                          "create_empty_dict" builder)
                       local_var builder)
              | _ -> ());
            builder
        | SVarDeclAssign ((t, id), e) ->
            let local_var = L.build_alloca (ltype_of_typ t) id builder in
            Hashtbl.add local_vars id local_var;
            let e' = expr builder e in
            ignore (L.build_store e' (lookup id) builder);
            builder
    in

    (* Build the code for each statement in the function *)
    let builder = stmt builder (SBlock fdecl.sbody) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder
      (match fdecl.styp with
      | A.Void -> L.build_ret_void
      | A.Float -> L.build_ret (L.const_float float_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body top_funcs;
  the_module
```

A.7 FFBB.ml

```
(* Top-level of the FFBB compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
```

```
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./FFBB.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = FFBBparser.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram (fst sast))
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

A.8 stringLibrary.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <errno.h>


char *slice(char *src, int begin, int end){

  int i;
  char* res = malloc((end - begin+1)*sizeof(char));

  if (!res)
    goto exit;

  if (begin>end){
        printf("Runtime Error: Slice begin integer %d is greater than end integer %d,", begin, end);
      goto exit;
  }

  int idx = 0;
  for (i=begin;i<end&&src[i]!='\0' ; i++){
```

```c
    res[idx++] = src[i];
  }

  res[idx] = '\0';
  return res;


exit:
  free(res);
  return NULL;
}



int  length(char *src){
  return strlen(src);
}


char *concat(char *a, char *b){
  int len1 = strlen(a);
  int len2 = strlen(b);
  char *res = malloc(len1 + len2 + 1);
  if (!res)
    goto exit;

  memcpy(res, a, len1);
  memcpy(res + len1, b, len2 + 1);
  return res;



exit:
  free(res);
  return NULL;
}
```

A.9 list.h

```c
#ifndef LIST_H
#define LIST_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>


// List Struct
```

```
typedef struct LIST {
  int size;
  int type;
  int capacity;
  void *ptr;
} List;

int len(void *_list);

void append(void *_list, void *value);

struct LIST * range(int n);

struct LIST * create_empty_list(int type);

#endif
```

A.10 list.c

```
#include "list.h"


int len(void *_list) {
    struct LIST *list = *(struct LIST **) _list;
    // printf("here in len\n");
    // printf("len=%d\n", list->size);
    // printf("type=%d\n", list->type);
    // printf("capacity=%d\n", list->capacity);
    return list->size;
}

struct LIST * range(int n) {
    struct LIST *list = malloc(sizeof(struct LIST));
    int *arr = (int *) malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        arr[i] = i;
    list->size = n;
    list->type = 0;
    list->capacity = n;
    list->ptr = (void *) arr;
    return list;
}



struct LIST * create_empty_list(int type) {
    struct LIST *list = malloc(sizeof(struct LIST));
    // int *arr = (int *) malloc(n * sizeof(int));
```

```c
    list->size = 0;
    list->type = type;
    list->capacity = 0;
    // list->ptr = (void *) arr;
    return list;
}

struct LIST * create_empty_list_int(int type) {
    return create_empty_list(type);
}

struct LIST * create_empty_list_bool(int type) {
    return create_empty_list(type);
}

struct LIST * create_empty_list_float(int type) {
    return create_empty_list(type);
}


int index_transform(void *_list, int i) {
    if (i < 0) {
        i += len(_list);
    }
    return i;
}


// l, r inclusive
struct LIST * list_slice(void *_list, int l, int r) {
    struct LIST *list = *(struct LIST **) _list;
    l = index_transform(_list, l);
    r = index_transform(_list, r);
    int n = r - l + 1;
    struct LIST *new_list = malloc(sizeof(struct LIST));
    void *new_arr;
    void *old_list_ptr = list->ptr;
    // extend capacity
    if (list->type == 0) {
        new_arr = (void *) malloc(n * sizeof(int));
        // copy over old values
        for (int i = 0; i < n; i++)
            ((int *)new_arr)[i] = ((int *)old_list_ptr)[l+i];
    }
    else if (list->type == 1) {
        new_arr = (void *) malloc(n * sizeof(bool));
        // copy over old values
        for (int i = 0; i < n; i++)
            ((bool *)new_arr)[i] = ((bool *)old_list_ptr)[l+i];
    }
```

```
    else if (list->type == 2) {
        new_arr = (void *) malloc(n * sizeof(double));
        // copy over old values
        for (int i = 0; i < n; i++)
            ((double *)new_arr)[i] = ((double *)old_list_ptr)[l+i];
    }
    new_list->size = n;
    new_list->type = list->type;
    new_list->capacity = n;
    new_list->ptr = (void *) new_arr;
    return new_list;
}

void append(void *_list, void *value) {
    struct LIST *list = *(struct LIST **) _list;

    // capacity not enough
    if (list->size >= list->capacity) {
        void *old_list_ptr = list->ptr;
        void *new_list_ptr;
        // extend capacity
        list->capacity *= 2;
        if (list->type == 0) {
            new_list_ptr = (void *) malloc(list->capacity * sizeof(int));
            // copy over old values
            for (int i = 0; i < list->size; i++)
                ((int *)new_list_ptr)[i] = ((int *)old_list_ptr)[i];
        }
        else if (list->type == 1) {
            new_list_ptr = (void *) malloc(list->capacity * sizeof(bool));
            // copy over old values
            for (int i = 0; i < list->size; i++)
                ((bool *)new_list_ptr)[i] = ((bool *)old_list_ptr)[i];
        }
        else if (list->type == 2) {
            new_list_ptr = (void *) malloc(list->capacity * sizeof(double));
            // copy over old values
            for (int i = 0; i < list->size; i++)
                ((double *)new_list_ptr)[i] = ((double *)old_list_ptr)[i];
        }

        list->ptr = new_list_ptr;
        free(old_list_ptr);
    }
    if (list->type == 0)
        ((int *)list->ptr)[list->size++] = *(int *) value;
    else if (list->type == 1)
        ((bool *)list->ptr)[list->size++] = *(bool *) value;
    else if (list->type == 2)
        ((double *)list->ptr)[list->size++] = *(double *) value;
```

```c

    // printf("here in append\n");
    // printf("len=%d\n", list->size);
    // printf("type=%d\n", list->type);
    // printf("capacity=%d\n", list->capacity);
}


#ifdef BUILD_TEST
int main()
{
    printf("here\n");
    struct LIST *list = malloc(sizeof(struct LIST));
    int *arr = (int *) malloc(5 * sizeof(int));
    arr[0] = 6;
    arr[1] = 3;
    arr[2] = 9;
    arr[3] = 8;
    list->size = 4;
    list->type = 0;
    list->capacity = 4;
    list->ptr = (void *) arr;
    printf("Len(list) = %d\n", len(list));
    for (int i = 0; i < len(list); i++)
        printf("list->ptr[i]=%d\n", ((int *)(list->ptr))[i]);
    // printf("Len(list) = %d\n", list->size);
}
#endif
```

A.11 treebasics.h

```c
#ifndef DICTSET_H
#define DICTSET_H

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include "list.h"

#define is_left_child(x) (x->parent->lc == x)

union Value
{
    int i;
    float f;
    bool b;
    struct LIST* l;
};
```

```
struct TreeNode
{
    union Value key;
    union Value value;
    struct TreeNode* parent;
    struct TreeNode* lc;
    struct TreeNode* rc;
};

union Value parse_value(void* value_ptr, int type_id);

bool uvalue_equal(union Value uvalue1, union Value uvalue2, int type_id);

bool uvalue_lt(union Value uvalue1, union Value uvalue2, int type_id);

#endif
```

A.12 treebasics.c

```
#include "treebasics.h"

union Value parse_value(void* value_ptr, int type_id)
{
    // printf("Enter parse_value");
    union Value uvalue;
    switch (type_id)
    {
        case 0 :
            uvalue.i = *(int*)value_ptr;
            break;
        case 1 :
            uvalue.b = *(bool*)value_ptr;
            break;
        case 2 :
            uvalue.f = *(double*)value_ptr;
            break;
        case 4 :
            uvalue.l = *(struct LIST**)value_ptr;
            break;
    }
    // printf("Leave parse_value");
    return uvalue;
}

bool uvalue_equal(union Value uvalue1, union Value uvalue2, int type_id)
{
    switch (type_id)
```

```
    {
        case 0 :
            return uvalue1.i == uvalue2.i;
        case 1 :
            return uvalue1.b == uvalue2.b;
        case 2 :
            return uvalue1.f == uvalue2.f;
    }
}

bool uvalue_lt(union Value uvalue1, union Value uvalue2, int type_id)
{
    switch (type_id)
    {
        case 0 :
            return uvalue1.i < uvalue2.i;
        case 1 :
            return (int)uvalue1.b < (int)uvalue2.b;
        case 2 :
            return uvalue1.f < uvalue2.f;
    }
}
```

A.13 treedict.c

```
#include "treebasics.h"

struct TreeDict
{
    int size;
    int key_type;  // 0: int, 1: bool, 2: double, 3: string
    int value_type;  // 0: int, 1: bool, 2: double, 3: string, 4: list
    struct TreeNode* root;
};

struct TreeDict* create_empty_dict(int key_type_id, int value_type_id)
{
    struct TreeDict* treedict = (struct TreeDict*)malloc(sizeof(struct TreeDict));
    treedict->size = 0;
    treedict->key_type = key_type_id;
    treedict->value_type = value_type_id;
    treedict->root = NULL;
    return treedict;
}

struct TreeNode* create_dict_node(struct TreeNode* parent, union Value key, union Value
    value)
{
```

```
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->key = key;
    node->value = value;
    node->parent = parent;
    node->lc = NULL;
    node->rc = NULL;
    return node;
}

void add_key_value(struct TreeDict* treedict, void* key_ptr, void* value_ptr)
{
    union Value ukey = parse_value(key_ptr, treedict->key_type);
    union Value uvalue = parse_value(value_ptr, treedict->value_type);
    if (treedict->root == NULL)
    {
        treedict->root = create_dict_node(NULL, ukey, uvalue);
        treedict->size = 1;
        return;
    }
    struct TreeNode* curr_node = treedict->root;
    while (true)
    {
        if (uvalue_equal(ukey, curr_node->key, treedict->key_type))
        {
            curr_node->value = uvalue;
            return;
        }
        if (uvalue_lt(ukey, curr_node->key, treedict->key_type))
        {
            if (curr_node->lc == NULL)
            {
                curr_node->lc = create_dict_node(curr_node, ukey, uvalue);
                treedict->size++;
                return;
            }
            else
            {
                curr_node = curr_node->lc;
            }
        }
        else
        {
            if (curr_node->rc == NULL)
            {
                curr_node->rc = create_dict_node(curr_node, ukey, uvalue);
                treedict->size++;
                return;
            }
            else
            {
```

```c
                curr_node = curr_node->rc;
            }
        }
    }
}

struct TreeNode* search_key_internal(struct TreeDict* treedict, union Value ukey)
{
    if (treedict->root == NULL) return NULL;
    struct TreeNode* curr_node = treedict->root;
    while (true)
    {
        if (uvalue_equal(ukey, curr_node->key, treedict->key_type)) return curr_node;
        if (uvalue_lt(ukey, curr_node->key, treedict->key_type))
        {
            if (curr_node->lc == NULL) return NULL;
            else
            {
                curr_node = curr_node->lc;
            }
        }
        else
        {
            if (curr_node->rc == NULL) return NULL;
            else
            {
                curr_node = curr_node->rc;
            }
        }
    }
}

bool key_exists(struct TreeDict* treedict, void* key_ptr)
{
    union Value ukey = parse_value(key_ptr, treedict->key_type);
    struct TreeNode* curr_node = search_key_internal(treedict, ukey);
    return (curr_node != NULL);
}

union Value retrieve_value_union(struct TreeDict* treedict, void* key_ptr)
{
    union Value ukey = parse_value(key_ptr, treedict->key_type);
    struct TreeNode* curr_node = search_key_internal(treedict, ukey);
    if (curr_node == NULL)
    {
        printf("Error! Key %d does not exists\n.", ukey.i);
        exit(-1);
    }
    return curr_node->value;
}
```

```c

int retrieve_value_int(struct TreeDict* treedict, void* key_ptr)
{
    return retrieve_value_union(treedict, key_ptr).i;
}

double retrieve_value_float(struct TreeDict* treedict, void* key_ptr)
{
    double result = retrieve_value_union(treedict, key_ptr).f;
    return result;
}

bool retrieve_value_bool(struct TreeDict* treedict, void* key_ptr)
{
    return retrieve_value_union(treedict, key_ptr).b;
}

struct LIST* retrieve_value_list(struct TreeDict* treedict, void* key_ptr)
{
    return retrieve_value_union(treedict, key_ptr).l;
}

void delete_key(struct TreeDict* treedict, void* key_ptr)
{
    union Value ukey = parse_value(key_ptr, treedict->key_type);
    // reference: https://blog.csdn.net/zxnsirius/article/details/52131433
    struct TreeNode* curr_node = search_key_internal(treedict, ukey);
    if (curr_node == NULL) return;
    treedict->size--;
    if (curr_node->lc == NULL)
    {
        if (curr_node == treedict->root)  // root node
        {
            treedict->root = curr_node->rc;
            if (treedict->root != NULL) treedict->root->parent = NULL;
        }
        else if (curr_node->rc == NULL)  // leaf node
        {
            if (is_left_child(curr_node)) curr_node->parent->lc = NULL;
            else curr_node->parent->rc = NULL;
        }
        else                             // middle node
        {
            if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->rc;
            else curr_node->parent->rc = curr_node->rc;
            curr_node->rc->parent = curr_node->parent;
        }
    }
    else if (curr_node->rc == NULL)
    {
```

```
        if (curr_node == treedict->root)
        {
            treedict->root = curr_node->lc;
            if (treedict->root != NULL) treedict->root->parent = NULL;
        }
        else
        {
            if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->lc;
            else curr_node->parent->rc = curr_node->lc;
            curr_node->lc->parent = curr_node->parent;
        }
    }
    else  // both left and right child exist, find the largest element that is smaller than the deleted element
    {
        struct TreeNode* left_neighbor = curr_node->lc;
        while (true)
        {
            if (left_neighbor->rc != NULL) left_neighbor = left_neighbor->rc;
            else break;
        }
        if (left_neighbor->lc == NULL)  // leaf node
        {
            if (is_left_child(left_neighbor)) left_neighbor->parent->lc = NULL;
            else left_neighbor->parent->rc = NULL;
        }
        else
        {
            if (is_left_child(left_neighbor)) left_neighbor->parent->lc = left_neighbor->lc;
            else left_neighbor->parent->rc = left_neighbor->lc;
            left_neighbor->lc->parent = left_neighbor->parent;
        }
        curr_node->key = left_neighbor->key;
        curr_node->value = left_neighbor->value;
    }
}

int get_dict_size(struct TreeDict* treedict)
{
    return treedict->size;
}


// int main(void)
// {
//     struct TreeDict* dict1 = create_empty_dict(0, 4);
//     int i3 = 3;
//     int i5 = 5;
//     struct LIST* list1 = create_empty_list(0);
```

```
//      struct LIST* list2 = range(5);
//      struct LIST* list3 = range(3);
//      add_key_value(dict1, (void*)(&i3), (void*)(&list1));
//      add_key_value(dict1, (void*)(&i5), (void*)(&list2));
//      add_key_value(dict1, (void*)(&i3), (void*)(&list3));
//      int dict1_size = get_dict_size(dict1);
//      printf("dict1 size: %d\n", dict1_size);
//      struct LIST* value1 = retrieve_value_list(dict1, (void*)(&i3));
//      printf("retrieved list size: %d\n", len((void*)(&value1)));
//      append((void*)(&value1), (void*)(&i5));
//      struct LIST* value2 = retrieve_value_list(dict1, (void*)(&i3));
//      printf("retrieved list size: %d\n", len((void*)(&value2)));
//      return 0;
// }
```

A.14 treeset.c

```
#include "treebasics.h"

struct TreeSet
{
    int size;
    int value_type;  // 0: int, 1: bool, 2: float, 3: string
    struct TreeNode* root;
};

struct TreeSet* create_empty_set(int type_id)
{
    struct TreeSet* treeset = (struct TreeSet*)malloc(sizeof(struct TreeSet));
    treeset->size = 0;
    treeset->value_type = type_id;
    treeset->root = NULL;
    return treeset;
}

struct TreeNode* create_node(struct TreeNode* parent, union Value value)
{
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->value = value;
    node->parent = parent;
    node->lc = NULL;
    node->rc = NULL;
    return node;
}

void add_elem(struct TreeSet* treeset, void* value_ptr)
{
    union Value uvalue = parse_value(value_ptr, treeset->value_type);
```

```
    if (treeset->root == NULL)
    {
        treeset->root = create_node(NULL, uvalue);
        treeset->size = 1;
        return;
    }
    struct TreeNode* curr_node = treeset->root;
    while (true)
    {
        if (uvalue_equal(uvalue, curr_node->value, treeset->value_type)) return;
        if (uvalue_lt(uvalue, curr_node->value, treeset->value_type))
        {
            if (curr_node->lc == NULL)
            {
                curr_node->lc = create_node(curr_node, uvalue);
                treeset->size++;
                return;
            }
            else
            {
                curr_node = curr_node->lc;
            }
        }
        else
        {
            if (curr_node->rc == NULL)
            {
                curr_node->rc = create_node(curr_node, uvalue);
                treeset->size++;
                return;
            }
            else
            {
                curr_node = curr_node->rc;
            }
        }
    }
}

struct TreeNode* search_elem_internal(struct TreeSet* treeset, union Value uvalue)
{
    if (treeset->root == NULL) return NULL;
    struct TreeNode* curr_node = treeset->root;
    while (true)
    {
        if (uvalue_equal(uvalue, curr_node->value, treeset->value_type)) return curr_node;
        if (uvalue_lt(uvalue, curr_node->value, treeset->value_type))
        {
            if (curr_node->lc == NULL) return NULL;
```

```
            else
            {
                curr_node = curr_node->lc;
            }
        }
        else
        {
            if (curr_node->rc == NULL) return NULL;
            else
            {
                curr_node = curr_node->rc;
            }
        }
    }
}

bool search_elem(struct TreeSet* treeset, void* value_ptr)
{
    union Value uvalue = parse_value(value_ptr, treeset->value_type);
    struct TreeNode* curr_node = search_elem_internal(treeset, uvalue);
    return (curr_node != NULL);
}

void delete_elem(struct TreeSet* treeset, void* value_ptr)
{
    union Value uvalue = parse_value(value_ptr, treeset->value_type);
    // reference: https://blog.csdn.net/zxnsirius/article/details/52131433
    struct TreeNode* curr_node = search_elem_internal(treeset, uvalue);
    if (curr_node == NULL) return;
    treeset->size--;
    if (curr_node->lc == NULL)
    {
        if (curr_node == treeset->root)  // root node
        {
            treeset->root = curr_node->rc;
            if (treeset->root != NULL) treeset->root->parent = NULL;
        }
        else if (curr_node->rc == NULL)  // leaf node
        {
            if (is_left_child(curr_node)) curr_node->parent->lc = NULL;
            else curr_node->parent->rc = NULL;
        }
        else                             // middle node
        {
            if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->rc;
            else curr_node->parent->rc = curr_node->rc;
            curr_node->rc->parent = curr_node->parent;
        }
    }
    else if (curr_node->rc == NULL)
```

```
    {
        if (curr_node == treeset->root)
        {
            treeset->root = curr_node->lc;
            if (treeset->root != NULL) treeset->root->parent = NULL;
        }
        else
        {
            if (is_left_child(curr_node)) curr_node->parent->lc = curr_node->lc;
            else curr_node->parent->rc = curr_node->lc;
            curr_node->lc->parent = curr_node->parent;
        }
    }
    else  // both left and right child exist, find the largest element that is smaller than the deleted element
    {
        struct TreeNode* left_neighbor = curr_node->lc;
        while (true)
        {
            if (left_neighbor->rc != NULL) left_neighbor = left_neighbor->rc;
            else break;
        }
        if (left_neighbor->lc == NULL)  // leaf node
        {
            if (is_left_child(left_neighbor)) left_neighbor->parent->lc = NULL;
            else left_neighbor->parent->rc = NULL;
        }
        else
        {
            if (is_left_child(left_neighbor)) left_neighbor->parent->lc = left_neighbor->lc;
            else left_neighbor->parent->rc = left_neighbor->lc;
            left_neighbor->lc->parent = left_neighbor->parent;
        }
        curr_node->value = left_neighbor->value;
    }
}

int get_set_size(struct TreeSet* treeset)
{
    return treeset->size;
}


// int main(void)
// {
//     struct TreeSet* set1 = create_empty_set(2);
//     float a = 3;
//     add_elem(set1, (void*)(&a));
//     a = 5;
```

```
//      add_elem(set1, (void*)(&a));
//      a = 3;
//      add_elem(set1, (void*)(&a));
//      int set1_size = get_set_size(set1);
//      printf("set1 size: %d\n", set1_size);
//      delete_elem(set1, (void*)(&a));
//      set1_size = get_set_size(set1);
//      printf("set1 size: %d\n", set1_size);
//      return 0;
// }
```

B Test Cases

B.1 Positive Tests

B.1.1 test-add1.mc

```
int add(int x, int y)
{
  return x + y;
}

int main()
{
  print( add(17, 25) );
  return 0;
}
```

Expected result:

```
42
```

B.1.2 test-arith1.mc

```
int main()
{
  print(39 + 3);
  return 0;
}
```

Expected result:

```
42
```

B.1.3 test-arith2.mc

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

Expected result:

```
11
```

B.1.4 test-arith3.mc

```
int foo(int a)
{
  return a;
}

int main()
{
  int a;
  a = 42;
  a = a + 5;
  print(a);
  return 0;
}
```

Expected result:

```
47
```

B.1.5 test-arith4.mc

```
int foo(int a)
{
  return a;
}

int main()
{
  int a = 42;
  a = a + 5;
  print(a);
  return 0;
}
```

Expected result:

```
47
```

B.1.6 test-demo1.mc

```
List<int> map(func<int, int> f, List<int> list) {
    List<int> out;

    for x in list {
        append(out, f(x));
    }
```

```
    return out;
}

List<int> filter(func<bool, int> f, List<int> list) {
    List<int> out;

    for x in list {
        if (f(x)) {
            append(out, x);
        }
    }
    return out;
}

func<int, int> sum2() {
    return int lambda int x -> x+2;
}

void print_list(List<int> list) {
    for x in list {
        print(x);
    }
}

int main()
{
    /* [0, 1, 2, 3, 4] */
    List<int> my_list = range(5);

    List<bool> out = map(int lambda int x -> x * 2, my_list);
    print_list(out);

    out = map(sum2(), my_list);
    print_list(out);

    out = filter(bool lambda int x -> x > 2, my_list);
    print_list(out);
    return 0;
}
```

Expected result:

```
0
2
4
6
8
2
3
4
5
```

```
6
3
4
```

B.1.7 test-demo2.mc

```

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]);
}
```

Expected result:

```
34
```

B.1.8 test-demo3.mc

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

int partition(List<int> A, int p, int r) {
  int x = A[r];
  int i = p - 1;
  for j in range(r - p + 1) {
      if (A[j+p] <= x) {
          i++;
          swap(A, i, j+p);
      }
  }
  swap(A, i+1, r);
  return i;
}

/* Recursive function to sort list A using quick-sort */
void quicksort(List<int> A, int p, int r) {
  if (p < r) {
      int q = partition(A, p, r);
      quicksort(A, p, q-1);
      quicksort(A, q+1, r);
  }
```

```
}

int main () {
    /* Using quicksort */
    List<int> A = [4, 2, 7, 3, 1, 9, 6, 10, 5, 8];
    quicksort(A, 0, len(A) - 1);
    for a in A {
        print(a);
    }
}
```

Expected result:

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

B.1.9 test-demo4.mc

```

int INF;
void graphInit(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
    for i in range(n) {
        List<int> l;
        List<int> w;
        dictAdd(E, i, l);
        dictAdd(W, i, w);
    }
}

void addEdge(Dict<int, List<int>> E, Dict<int, List<int>> W, int u, int v, int w) {
    List<int> le = dictGetList(E, u);
    List<int> lw = dictGetList(W, u);
    append(le, v);
    append(lw, w);
}

void printGraph(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
    for i in range(n) {
        prints("--------------------------");
        print(i);
        List<int> le = dictGetList(E, i);
```

```
        List<int> lw = dictGetList(W, i);

        if (len(le) > 0) {
            prints("neighbors");
            for v in le {
                print(v);
            }
            prints("weights");
            for w in lw {
                print(w);
            }
        }
    }
}



bool checkNegativeCycle(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n,
    List<int> dist) {
    /* for every edge */
    for u in range(n) {
        List<int> le = dictGetList(E, u);
        List<int> lw = dictGetList(W, u);
        int m = len(le);
        if (m > 0) {
            for k in range(m) {
                int v = le[k];
                int w = lw[k];

                /* we have u, v, w */
                if (dist[u] != INF && dist[u] + w < dist[v]) {
                    prints("Graph contains negative weight cycle");
                    return false;
                }
            }
        }
    }
    return true;
}

List<int> BellmanFord(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n) {

    List<int> dist;
    for z in range(n) {
        append(dist, INF);
    }
    dist[src] = 0;

    /* Loop |V| times */
    for i in range(n - 1) {
```

```
        /* for every edge */
        for u in range(n) {
            List<int> le = dictGetList(E, u);
            List<int> lw = dictGetList(W, u);
            int m = len(le);
            if (m > 0) {
                for k in range(m) {
                    int v = le[k];
                    int w = lw[k];

                    /* we have u, v, w */
                    if (dist[u] != INF && dist[u] + w < dist[v]) {
                        dist[v] = dist[u] + w;
                    }
                }
            }
        }
    }
    checkNegativeCycle(E, W, src, n, dist);
    return dist;
}

int main() {
    int n = 5;
    INF = 100000;
    Dict<int, List<int>> E;
    Dict<int, List<int>> W;
    graphInit(E, W, n);
    addEdge(E, W,  0, 1, -1);
    addEdge(E, W,  0, 2, 4);
    addEdge(E, W,  1, 2, 3);
    addEdge(E, W,  1, 3, 2);
    addEdge(E, W,  1, 4, 2);
    addEdge(E, W,  3, 2, 5);
    addEdge(E, W,  3, 1, 1);
    addEdge(E, W,  4, 3, -3);
    printGraph(E, W, n);
    prints("Vertex Distance from Source");
    List<int> dist = BellmanFord(E, W, 0, n);
    for d in dist {
        print(d);
    }

    return 0;
}
```

Expected result:

```
--------------------------
0
neighbors
```

```
1
2
weights
-1
4
--------------------------
1
neighbors
2
3
4
weights
3
2
2
--------------------------
2
--------------------------
3
neighbors
2
1
weights
5
1
--------------------------
4
neighbors
3
weights
-3
Vertex Distance from Source
0
-1
2
-2
1
```

B.1.10 test-demo5.mc

```

int INF;
void graphInit(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
    for i in range(n) {
        List<int> l;
        List<int> w;
        dictAdd(E, i, l);
        dictAdd(W, i, w);
    }
```

```
}

void addEdge(Dict<int, List<int>> E, Dict<int, List<int>> W, int u, int v, int w) {
    List<int> le = dictGetList(E, u);
    List<int> lw = dictGetList(W, u);
    append(le, v);
    append(lw, w);
}

void printGraph(Dict<int, List<int>> E, Dict<int, List<int>> W, int n) {
    for i in range(n) {
        prints("--------------------------");
        print(i);
        List<int> le = dictGetList(E, i);
        List<int> lw = dictGetList(W, i);

        if (len(le) > 0) {
            prints("neighbors");
            for v in le {
                print(v);
            }
            prints("weights");
            for w in lw {
                print(w);
            }
        }
    }
}



bool checkNegativeCycle(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n,
    List<int> dist) {
    /* for every edge */
    for u in range(n) {
        List<int> le = dictGetList(E, u);
        List<int> lw = dictGetList(W, u);
        int m = len(le);
        if (m > 0) {
            for k in range(m) {
                int v = le[k];
                int w = lw[k];

                /* we have u, v, w */
                if (dist[u] != INF && dist[u] + w < dist[v]) {
                    prints("Graph contains negative weight cycle");
                    return false;
                }
            }
        }
```

```
    }
    return true;
}

bool BellmanFord(Dict<int, List<int>> E, Dict<int, List<int>> W, int src, int n, List<
    int> dist) {

    for z in range(n) {
        append(dist, INF);
    }
    dist[src] = 0;

    /* Loop |V| times */
    for i in range(n - 1) {
        /* for every edge */
        for u in range(n) {
            List<int> le = dictGetList(E, u);
            List<int> lw = dictGetList(W, u);
            int m = len(le);
            if (m > 0) {
                for k in range(m) {
                    int v = le[k];
                    int w = lw[k];

                    /* we have u, v, w */
                    if (dist[u] != INF && dist[u] + w < dist[v]) {
                        dist[v] = dist[u] + w;
                    }
                }
            }
        }
    }
    return checkNegativeCycle(E, W, src, n, dist);
}

int main() {
    int n = 5;
    INF = 100000;
    Dict<int, List<int>> E;
    Dict<int, List<int>> W;
    graphInit(E, W, n);
    addEdge(E, W,  0, 1, -1);
    addEdge(E, W,  0, 2, 4);
    addEdge(E, W,  1, 2, 3);
    addEdge(E, W,  1, 3, 2);
    addEdge(E, W,  1, 4, 2);
    addEdge(E, W,  2, 0, -3);
    addEdge(E, W,  3, 2, 5);
    addEdge(E, W,  3, 1, 1);
    addEdge(E, W,  4, 3, -3);
```

```
    /* printGraph(E, W, n); */
    List<int> dist;
    if (BellmanFord(E, W, 0, n, dist)) {
        prints("Vertex Distance from Source");
        for d in dist {
            print(d);
        }
    }
    return 0;
}
```

Expected result:

```
Graph contains negative weight cycle
```

B.1.11 test-dict1.mc

```
int main()
{
  Dict<int, int> idict;
  int i;
  int size;
  dictAdd(idict, 5, 10);
  size = dictSize(idict);
  print(size);
  i = 10;
  dictAdd(idict, i, 12);
  size = dictSize(idict);
  print(size);
  dictAdd(idict, 5, 15);
  size = dictSize(idict);
  print(size);
  return 0;
}
```

Expected result:

```
1
2
2
```

B.1.12 test-dict2.mc

```
int main()
{
  Dict<int, int> idict;
  bool found;
  dictAdd(idict, 7, 10);
```

```
  dictAdd(idict, 9, 14);
  found = dictHasKey(idict, 7);
  printb(found);
  found = dictHasKey(idict, 10);
  printb(found);
  return 0;
}
```

Expected result:

```
1
0
```

B.1.13 test-dict3.mc

```
int main()
{
  Dict<int, int> idict;
  int val;
  dictAdd(idict, 2, 9);
  dictAdd(idict, 8, 3);
  dictAdd(idict, 10, 6);
  dictAdd(idict, 7, 11);
  val = dictGetInt(idict, 8);
  print(val);
  val = dictGetInt(idict, 10);
  print(val);
  return 0;
}
```

Expected result:

```
3
6
```

B.1.14 test-dict4.mc

```
int main()
{
  Dict<int, int> idict;
  int val;
  int i;
  for (i = 0; i < 100; i = i + 1)
  {
    dictAdd(idict, i, i);
  }
  for (i = 0; i < 100; i = i + 1)
  {
```

```
    val = dictGetInt(idict, i);
    print(val);
  }
  return 0;
}
```

Expected result:

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

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

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

B.1.15 test-dict5.mc

```
int main()
{
  Dict<int, int> idict;
  int val;
  bool flag;
  dictAdd(idict, 5, 7);
  dictAdd(idict, 9, 3);
  dictAdd(idict, 1, 15);
  dictAdd(idict, 2, 8);
  flag = dictHasKey(idict, 9);
  printb(flag);
  dictRemove(idict, 9);
  dictRemove(idict, 2);
  dictRemove(idict, 10);
  flag = dictHasKey(idict, 9);
  printb(flag);
  return 0;
}
```

Expected result:

```
1
0
```

B.1.16 test-dict6.mc

```
int main()
{
  Dict<int, float> idict;
  int i;
  int size;
  dictAdd(idict, 5, 10.2);
  size = dictSize(idict);
  print(size);
  i = 10;
  dictAdd(idict, i, 12.0);
  size = dictSize(idict);
```

```
  print(size);
  dictAdd(idict, 5, 15.0);
  size = dictSize(idict);
  print(size);
  return 0;
}
```

Expected result:

```
1
2
2
```

B.1.17 test-dict7.mc

```
int main()
{
  Dict<int, bool> bdict;
  bool result;
  dictAdd(bdict, 2, true);
  result = dictGetBool(bdict, 2);
  printb(result);
  Dict<float, float> fdict;
  dictAdd(fdict, 2.4, 4.8);
  dictAdd(fdict, 5.6, 11.2);
  float sim = dictGetFloat(fdict, 2.4);
  printf(sim);
  return 0;
}
```

Expected result:

```
1
4.8
```

B.1.18 test-dict8.mc

```
int main()
{
  Dict<int, int> idict;
  int result;
  dictAdd(idict, 2, 5);
  result = dictGetInt(idict, 2);
  print(result);
  return 0;
}
```

Expected result:

```
5
```

B.1.19 test-fib.mc

```
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

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

Expected result:

```
1
1
2
3
5
8
```

B.1.20 test-float1.mc

```
int main()
{
  float a;
  a = 3.14159267;
  printf(a);
  return 0;
}
```

Expected result:

```
3.14159
```

B.1.21 test-float2.mc

```
int main()
{
  float a;
  float b;
  float c;
  a = 3.14159267;
  b = -2.71828;
  c = a + b;
  printf(c);
  return 0;
}
```

Expected result:

```
0.423313
```

B.1.22 test-float3.mc

```
void testfloat(float a, float b)
{
  printf(a + b);
  printf(a - b);
  printf(a * b);
  printf(a / b);
  printb(a == b);
  printb(a == a);
  printb(a != b);
  printb(a != a);
  printb(a > b);
  printb(a >= b);
  printb(a < b);
  printb(a <= b);
}

int main()
{
  float c;
  float d;

  c = 42.0;
  d = 3.14159;

  testfloat(c, d);

  testfloat(d, d);

  return 0;
```

```
}
```

Expected result:

```
45.1416
38.8584
131.947
13.369
0
1
1
0
1
1
0
0
6.28318
0
9.86959
1
1
1
0
0
0
1
0
1
```

B.1.23 test-float4.mc

```
void testfloat(float a, float b)
{
  printf(a + b);
  printf(a - b);
  printf(a * b);
  printf(a / b);
  printb(a == b);
  printb(a == a);
  printb(a != b);
  printb(a != a);
  printb(a > b);
  printb(a >= b);
  printb(a < b);
  printb(a <= b);
}

int main()
{
```

```
  float c = 42.0;
  float d = 3.14159;

  testfloat(c, d);

  testfloat(d, d);

  return 0;
}
```

Expected result:

```
45.1416
38.8584
131.947
13.369
0
1
1
0
1
1
0
0
6.28318
0
9.86959
1
1
1
0
0
0
1
0
1
```

B.1.24 test-float5.mc

```
int main()
{
  float a = 3.14159267;
  float b = -2.71828;
  float c;
  a = 3.14159267;
  b = -2.71828;
  c = a + b + 2.1;
  printf(c);
  b = 2.83;
```

```
  c = a + b + 1.0;
  printf(c);
  return 0;
}
```

Expected result:

```
2.52331
6.97159
```

B.1.25 test-float6.mc

```
int main()
{
  float a = 1.1;
  float b = -1.1;
  printf(a*b);
  printf(a/b);
  printf(a+b-1.0);
}
```

Expected result:

```
-1.21
-1
-1
```

B.1.26 test-for1.mc

```
int main()
{
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(i);
  }
  print(42);
  return 0;
}
```

Expected result:

```
0
1
2
3
4
42
```

B.1.27 test-for2.mc

```
int main()
{
  int i;
  i = 0;
  for ( ; i < 5; ) {
    print(i);
    i = i + 1;
  }
  print(42);
  return 0;
}
```

Expected result:

```
0
1
2
3
4
42
```

B.1.28 test-forin1.mc

```
int main()
{
  int n = 5;
  for i in range(n) {
    print(i);
    for j in range(n) {
        print(j);
    }
  }
  for k in [3, 6, 9, 10] {
    print(k);
  }
  return 0;
}
```

Expected result:

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

```
0
1
2
3
4
2
0
1
2
3
4
3
0
1
2
3
4
4
0
1
2
3
4
3
6
9
10
```

B.1.29 test-func1.mc

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int a;
  a = add(39, 3);
  print(a);
  return 0;
}
```

Expected result:

```
42
```

B.1.30 test-func2.mc

```
/* Bug noticed by Pin-Chin Huang */

int fun(int x, int y)
{
  return 0;
}

int main()
{
  int i;
  i = 1;

  fun(i = 2, i = i+1);

  print(i);
  return 0;
}
```

Expected result:

```
2
```

B.1.31 test-func3.mc

```
void printem(int a, int b, int c, int d)
{
  print(a);
  print(b);
  print(c);
  print(d);
}

int main()
{
  printem(42,17,192,8);
  return 0;
}
```

Expected result:

```
42
17
192
8
```

B.1.32 test-func4.mc

```
int add(int a, int b)
{
  int c;
  c = a + b;
  return c;
}

int main()
{
  int d;
  d = add(52, 10);
  print(d);
  return 0;
}
```

Expected result:

```
62
```

B.1.33 test-func5.mc

```
int foo(int a)
{
  return a;
}

int main()
{
  return 0;
}
```

Expected result:

```
```

B.1.34 test-func6.mc

```
void foo() {}

int bar(int a, bool b, int c) { return a + c; }

int main()
{
  print(bar(17, false, 25));
  return 0;
}
```

Expected result:

```
42
```

B.1.35 test-func7.mc

```
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  print(a);
  return 0;
}
```

Expected result:

```
115
```

B.1.36 test-func8.mc

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

int main()
{
  foo(40);
  return 0;
}
```

Expected result:

```
43
```

B.1.37 test-func9.mc

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

```
  return;
}

int main()
{
  foo(40);
  return 0;
}
```

Expected result:

```
43
```

B.1.38 test-gcd.mc

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

int main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
  return 0;
}
```

Expected result:

```
2
3
11
```

B.1.39 test-gcd2.mc

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}

int main()
{
```

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

Expected result:

```
7
4
11
```

B.1.40 test-global1.mc

```
int a;
int b;

void printa()
{
  print(a);
}

void printbb()
{
  print(b);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
{
  a = 42;
  b = 21;
  printa();
  printbb();
  incab();
  printa();
  printbb();
  return 0;
}
```

Expected result:

```
42
21
```

```
43
22
```

B.1.41 test-global2.mc

```
bool i;

int main()
{
  int i; /* Should hide the global i */

  i = 42;
  print(i + i);
  return 0;
}
```

Expected result:

```
84
```

B.1.42 test-global3.mc

```
int i;
bool b;
int j;

int main()
{
  i = 42;
  j = 10;
  print(i + j);
  return 0;
}
```

Expected result:

```
52
```

B.1.43 test-hello-world.mc

```
int main()
{
  prints("Hello, World!");
}
```

Expected result:

```
Hello, World!
```

B.1.44 test-hello.mc

```
int main()
{
  print(42);
  print(71);
  print(1);
  return 0;
}
```

Expected result:

```
42
71
1
```

B.1.45 test-if1.mc

```
int main()
{
  if (true) print(42);
  print(17);
  return 0;
}
```

Expected result:

```
42
17
```

B.1.46 test-if2.mc

```
int main()
{
  if (true) print(42); else print(8);
  print(17);
  return 0;
}
```

Expected result:

```
42
17
```

B.1.47 test-if3.mc

```
int main()
{
  if (false) print(42);
  print(17);
  return 0;
}
```

Expected result:

```
17
```

B.1.48 test-if4.mc

```
int main()
{
  if (false) print(42); else print(8);
  print(17);
  return 0;
}
```

Expected result:

```
8
17
```

B.1.49 test-if5.mc

```
int cond(bool b)
{
  int x;
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 print(cond(true));
 print(cond(false));
 return 0;
}
```

Expected result:

```
42
17
```

B.1.50 test-if6.mc

```
int cond(bool b)
{
  int x;
  x = 10;
  if (b)
    if (x == 10)
      x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 print(cond(true));
 print(cond(false));
 return 0;
}
```

Expected result:

```
42
10
```

B.1.51 test-lambda.mc

```
int g;
List<int> map(func<int, int> f, List<int> list) {
    List<int> out = [0];

    for x in list {
        append(out, f(x));
    }
    return out[1:-1];
}

List<int> filter(func<bool, int> f, List<int> list) {
    List<int> out = [0];

    for x in list {
        if (f(x)) {
            append(out, x);
```

```
        }
    }
    return out[1:-1];
}

func<int, int> sum2() {
    return int lambda int x -> x+2;
}


int main()
{
    List<int> my_list = range(5);
    func<void, List<int>> print_list = void lambda List<int> list {
        for x in list {
            print(x);
        }
    };

    g = 2;
    List<int> out = map(int lambda int x { return x*g; }, my_list);
    print_list(out);
    out = map(int lambda int x -> x*2, my_list);
    print_list(out);
    out = map(sum2(), my_list);
    print_list(out);
    out = filter(bool lambda int x -> x > 2, my_list);
    print_list(out);
    return 0;
}
```

Expected result:

```
0
2
4
6
8
0
2
4
6
8
2
3
4
5
6
3
4
```

B.1.52 test-list1.mc

```
int main()
{
  List<int> arr;
  int i;
  arr = [5, 4, 3, 2, 1];
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(arr[i]);
  }
  arr[1] = 100;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(arr[i]);
  }
  return 0;
}
```

Expected result:

```
5
4
3
2
1
5
100
3
2
1
```

B.1.53 test-list2.mc

```
int main()
{
  List<int> int_arr;
  List<float> float_arr;
  int i;
  int_arr = [100];
  append(int_arr, 666);
  append(int_arr, 999);
  print(len(int_arr));
  append(int_arr, 1111);
  append(int_arr, 2222);
  for (i = 0; i < len(int_arr); i=i+1) {
      print(int_arr[i]);
  }
  float_arr = [3.1, 5.2];
  append(float_arr, 6.3);
  append(float_arr, 9.4);
```

```
    print(len(float_arr));
    append(float_arr, 11.5);
    append(float_arr, 2.6);
    float_arr[0] = 0.1;
    for (i = 0; i < len(float_arr); i=i+1) {
        printf(float_arr[i]);
    }
    return 0;
}
```

Expected result:

```
3
100
666
999
1111
2222
4
0.1
5.2
6.3
9.4
11.5
2.6
```

B.1.54 test-list3.mc

```
int main()
{
    List<int> int_arr;
    int i;
    int n;
    n = 5;
    int_arr = range(n);
    for (i = 0; i < n; i=i+1) {
        print(int_arr[i]);
    }
    return 0;
}
```

Expected result:

```
0
1
2
3
4
```

B.1.55 test-list4.mc

```
int main()
{
  List<int> int_arr;
  List<int> slice_arr;
  List<float> float_arr;
  int i;
  int n;

  n = 5;
  List<int> range_list = range(n);
  range_list[-1] = 10;
  for (i = -1; i >= -n; i=i-1) {
      print(range_list[i]);
  }
  int_arr = [100, 666, 999, 1111, 2222, 3333];
  slice_arr = int_arr[-3:-1];
  for (i = 0; i < len(slice_arr); i=i+1) {
      print(slice_arr[i]);
  }
  return 0;
}
```

Expected result:

```
10
3
2
1
0
1111
2222
3333
```

B.1.56 test-local1.mc

```
void foo(bool j)
{
  int i;

  i = 42;
  print(i + i);
}

int main()
{
  foo(true);
  return 0;
```

```
}
```

Expected result:

```
84
```

B.1.57 test-local2.mc

```
int foo(int a, bool b)
{
  int c;
  bool d;

  c = a;

  return c + 10;
}

int main() {
 print(foo(37, false));
 return 0;
}
```

Expected result:

```
47
```

B.1.58 test-ops1.mc

```
int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  printb(1 == 2);
  printb(1 == 1);
  print(99);
  printb(1 != 2);
  printb(1 != 1);
  print(99);
  printb(1 < 2);
  printb(2 < 1);
  print(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
```

```
  print(99);
  printb(1 > 2);
  printb(2 > 1);
  print(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  return 0;
}
```

Expected result:

```
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
1
1
```

B.1.59 test-ops2.mc

```
int main()
{
  printb(true);
  printb(false);
  printb(true && true);
  printb(true && false);
  printb(false && true);
  printb(false && false);
  printb(true || true);
  printb(true || false);
```

```
  printb(false || true);
  printb(false || false);
  printb(!false);
  printb(!true);
  print(-10);
}
```

Expected result:

```
1
0
1
0
0
0
1
1
1
0
1
0
-10
```

B.1.60 test-ops3.mc

```
int main()
{
  int x = 0;
  x ++;
  print(x);
  x --;
  print(x);
  x--;
  print(x);
  x++;
  print(x);
}
```

Expected result:

```
1
0
-1
0
```

B.1.61 test-ops4.mc

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

Expected result:

```
5
6
3
```

B.1.62 test-printbig.mc

```
/*
 * Test for linking external C functions to LLVM-generated code
 *
 * printbig is defined as an external function, much like printf
 * The C compiler generates printbig.o
 * The LLVM compiler, llc, translates the .ll to an assembly .s file
 * The C compiler assembles the .s file and links the .o file to generate
 * an executable
 */

int main()
{
  printbig(72); /* H */
  printbig(69); /* E */
  printbig(76); /* L */
  printbig(76); /* L */
  printbig(79); /* O */
  printbig(32); /*   */
  printbig(87); /* W */
  printbig(79); /* O */
  printbig(82); /* R */
  printbig(76); /* L */
  printbig(68); /* D */
  return 0;
}
```

Expected result:

```
  XXXXXXXXXXXXXX
```

```
XXXXXXXXXXXXXX
      XX
      XX
      XX
XXXXXXXXXXXXXX
XXXXXXXXXXXXXX


XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
XX    XX    XX
XX    XX    XX
XX    XX    XX
XX          XX


XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
XX
XX
XX
XX


XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
XX
XX
XX
XX

  XXXXXXXXXX
XXXXXXXXXXXXXX
XX          XX
XX          XX
XX          XX
XXXXXXXXXXXXXX
  XXXXXXXXXX









    XXXXXXXXXX
XXXXXXXXXXXXXX
  XXXXXX
```

```
      XXXXXX
    XXXXXX
  XXXXXXXXXXXXXX
      XXXXXXXXXX

    XXXXXXXXXX
  XXXXXXXXXXXXXX
  XX          XX
  XX          XX
  XX          XX
  XXXXXXXXXXXXXX
    XXXXXXXXXX

  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX
      XX      XX
    XXXX      XX
  XXXXXXXX    XX
  XXXX  XXXXXXXX
  XX    XXXXXX


  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX
  XX
  XX
  XX
  XX

  XXXXXXXXXXXXXX
  XXXXXXXXXXXXXX
  XX          XX
  XX          XX
  XXXX      XXXX
    XXXXXXXXXX
      XXXXXX
```

B.1.63 test-set1.mc

```
int main()
{
  Set<int> iset;
  int i;
  int size;
  size = setSize(iset);
  print(size);
  setAdd(iset, 5);
  size = setSize(iset);
  print(size);
```

```
  i = 10;
  setAdd(iset, i);
  size = setSize(iset);
  print(size);
  setAdd(iset, 5);
  size = setSize(iset);
  print(size);
  return 0;
}
```

Expected result:

```
0
1
2
2
```

B.1.64 test-set2.mc

```
int main()
{
  Set<int> iset;
  bool found;
  setAdd(iset, 7);
  setAdd(iset, 9);
  found = setFind(iset, 7);
  printb(found);
  found = setFind(iset, 10);
  printb(found);
  return 0;
}
```

Expected result:

```
1
0
```

B.1.65 test-set3.mc

```
int main()
{
  Set<int> iset;
  int size;
  bool found;
  setAdd(iset, 4);
  setAdd(iset, 8);
  setAdd(iset, 16);
  setAdd(iset, 7);
```

```
  found = setFind(iset, 8);
  printb(found);
  setRemove(iset, 5);
  setRemove(iset, 8);
  found = setFind(iset, 8);
  printb(found);
  setRemove(iset, 16);
  size = setSize(iset);
  print(size);
  return 0;
}
```

Expected result:

```
1
0
2
```

B.1.66 test-set4.mc

```
int main()
{
  Set<float> fset;
  float i;
  int size;
  size = setSize(fset);
  print(size);
  setAdd(fset, 5.2);
  size = setSize(fset);
  print(size);
  i = 10.6;
  setAdd(fset, i);
  size = setSize(fset);
  print(size);
  setAdd(fset, 5.2);
  size = setSize(fset);
  print(size);
  return 0;
}
```

Expected result:

```
0
1
2
2
```

B.1.67 test-set5.mc

```
int main()
{
  Set<bool> bset;
  bool found;
  bool b;
  int size;
  setAdd(bset, true);
  found = setFind(bset, true);
  printb(found);
  found = setFind(bset, false);
  printb(found);
  b = false;
  setAdd(bset, b);
  found = setFind(bset, false);
  printb(found);
  setRemove(bset, true);
  size = setSize(bset);
  print(size);
  return 0;
}
```

Expected result:

```
1
0
1
1
```

B.1.68 test-string-concat.mc

```
int main(){
  string s1;
    string s2;
    string s3;
    s1= "Good ";
    s2= "morning";

    s3= concat(s1,s2);
    prints(s3);

    return 0;
}
```

Expected result:

```
Good morning
```

B.1.69 test-string-len.mc

```
int main(){
  string s;
    int len;

    s= "foobar";
    len= length(s);
    print(len);
    return 0;
}
```

Expected result:

```
6
```

B.1.70 test-string-slice.mc

```
int main(){
  string s;
    string sliced;

    s= "foobar";
    sliced = slice(s, 2, 5);
  prints(sliced);
    return 0;
}
```

Expected result:

```
oba
```

B.1.71 test-string1.mc

```
int main()
{
  string str;
  str = "nice to meet u";
  prints(str);
  return 0;
}
```

Expected result:

```
nice to meet u
```

B.1.72 test-string2.mc

```
int main()
{
  string str = "nice to meet u";
  prints(str);
  return 0;
}
```

Expected result:

```
nice to meet u
```

B.1.73 test-var1.mc

```
int main()
{
  int a;
  a = 42;
  print(a);
  return 0;
}
```

Expected result:

```
42
```

B.1.74 test-var2.mc

```
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  print(a);
  return 0;
}
```

Expected result:

```
115
```

B.1.75 test-var3.mc

```
int main()
{
  int pos_int = 42;
  print(pos_int);
  int neg_int = -42;
  print(neg_int);
  bool true_bool;
  true_bool = true;
  printb(true_bool);
  bool false_bool = false;
  printb(false_bool);
  float pos_float = 0.1;
  printf(pos_float);
  float neg_float = -0.1;
  printf(neg_float);
  string empty_string = "";
  prints(empty_string);
  string short_string = "abcd";
  prints(short_string);
  string long_string = "abcdefghijklmnopqrstuvwxyz";
  prints(long_string);
  return 0;
}
```

Expected result:

```
42
-42
1
0
0.1
-0.1

abcd
abcdefghijklmnopqrstuvwxyz
```

B.1.76 test-while1.mc

```
int main()
{
  int i;
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
```

```
  return 0;
}
```

Expected result:

```
5
4
3
2
1
42
```

B.1.77 test-while2.mc

```
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

int main()
{
  print(foo(7));
  return 0;
}
```

Expected result:

```
14
```

B.2 Negative Tests

B.2.1 fail-assign1.mc

```
int main()
{
  int i;
  bool b;

  i = 42;
  i = 10;
  b = true;
```

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

Expected error:

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

B.2.2 fail-assign2.mc

```
int main()
{
  int i;
  bool b;

  b = 48; /* Fail: assigning an integer to a bool */
}
```

Expected error:

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

B.2.3 fail-assign3.mc

```
void myvoid()
{
  return;
}

int main()
{
  int i;

  i = myvoid(); /* Fail: assigning a void to an integer */
}
```

Expected error:

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

B.2.4 fail-dead1.mc

```
int main()
{
  int i;

```

```
  i = 15;
  return i;
  i = 32; /* Error: code after a return */
}
```

Expected error:

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

B.2.5 fail-dead2.mc

```
int main()
{
  int i;

  {
    i = 15;
    return i;
  }
  i = 32; /* Error: code after a return */
}
```

Expected error:

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

B.2.6 fail-dict1.mc

```
int main()
{
  Dict<int, float> ifdict;
  dictAdd(ifdict, 3.2, 5.0);
  return 0;
}
```

Expected error:

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

B.2.7 fail-dict2.mc

```
int main()
{
  Dict<float, bool> fbdict;
  dictAdd(fbdict, 3.2, true);
  dictGetBool(fbdict, false);
```

```
  return 0;
}
```

Expected error:

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

B.2.8 fail-dict3.mc

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

Expected error:

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

B.2.9 fail-expr1.mc

```
int a;
bool b;

void foo(int c, bool d)
{
  int dd;
  bool e;
  a + c;
  c - a;
  a * 3;
  c / 2;
  d + a; /* Error: bool + int */
}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.10 fail-expr2.mc

```
int a;
bool b;

void foo(int c, bool d)
{
  int d; /* Error: redeclaring d */
  bool e;
  b + a;
}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.11 fail-expr3.mc

```
int a;
float b;

void foo(int c, float d)
{
  int d; /* Error: redeclaring d */
  float e;
  b + a;
}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.12 fail-expr4.mc

```
int a;
bool b;

```

```
void foo(int c, bool d)
{
  int f;
  bool e;
  b + a; /* Error: bool + int */
}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.13 fail-expr5.mc

```
int a;
float b;

void foo(int c, float d)
{
  int f;
  bool e;
  b + e; /* Error: float + bool */
}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.14 fail-float1.mc

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

Expected error:

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

B.2.15 fail-float2.mc

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

Expected error:

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

B.2.16 fail-for1.mc

```
int main()
{
  int i;
  for ( ; true ; ) {} /* OK: Forever */

  for (i = 0 ; i < 10 ; i = i + 1) {
    if (i == 3) return 42;
  }

  for (j = 0; i < 10 ; i = i + 1) {} /* j undefined */

  return 0;
}
```

Expected error:

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

B.2.17 fail-for2.mc

```
int main()
{
  int i;

  for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */

  return 0;
}
```

Expected error:

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

B.2.18 fail-for3.mc

```
int main()
{
  int i;

  for (i = 0; i ; i = i + 1) {} /* i is an integer, not Boolean */

  return 0;
}
```

Expected error:

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

B.2.19 fail-for4.mc

```
int main()
{
  int i;

  for (i = 0; i < 10 ; i = j + 1) {} /* j undefined */

  return 0;
}
```

Expected error:

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

B.2.20 fail-for5.mc

```
int main()
{
  int i;

  for (i = 0; i < 10 ; i = i + 1) {
    foo(); /* Error: no function foo */
  }

  return 0;
}
```

Expected error:

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

B.2.21 fail-func1.mc

```
int foo() {}

int bar() {}

int baz() {}

void bar() {} /* Error: duplicate function bar */

int main()
{
  return 0;
}
```

Expected error:

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

B.2.22 fail-func2.mc

```
int foo(int a, bool b, int c) { }

void bar(int a, bool b, int a) {} /* Error: duplicate formal a in bar */

int main()
{
  return 0;
}
```

Expected error:

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

B.2.23 fail-func3.mc

```
int foo(int a, bool b, int c) { }

void bar(int a, void b, int c) {} /* Error: illegal void formal b */

int main()
{
  return 0;
}
```

Expected error:

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

B.2.24 fail-func4.mc

```
int foo() {}

void bar() {}

int print() {} /* Should not be able to define print */

void baz() {}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.25 fail-func5.mc

```
int foo() {}

int bar() {
  int a;
  void b; /* Error: illegal void local b */
  bool c;

  return 0;
}

int main()
{
  return 0;
}
```

Expected error:

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

B.2.26 fail-func6.mc

```
void foo(int a, bool b)
{
}

int main()
```

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

Expected error:

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

B.2.27 fail-func7.mc

```
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42, true, false); /* Wrong number of arguments */
}
```

Expected error:

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

B.2.28 fail-func8.mc

```
void foo(int a, bool b)
{
}

void bar()
{
}

int main()
{
  foo(42, true);
  foo(42, bar()); /* int and void, not int and bool */
}
```

Expected error:

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

B.2.29 fail-func9.mc

```
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42, 42); /* Fail: int, not bool */
}
```

Expected error:

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

B.2.30 fail-global1.mc

```
int c;
bool b;
void a; /* global variables should not be void */


int main()
{
  return 0;
}
```

Expected error:

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

B.2.31 fail-global2.mc

```
int b;
bool c;
int a;
int b; /* Duplicate global variable */

int main()
{
  return 0;
}
```

Expected error:

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

B.2.32 fail-if1.mc

```
int main()
{
  if (true) {}
  if (false) {} else {}
  if (42) {} /* Error: non-bool predicate */
}
```

Expected error:

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

B.2.33 fail-if2.mc

```
int main()
{
  if (true) {
    foo; /* Error: undeclared variable */
  }
}
```

Expected error:

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

B.2.34 fail-if3.mc

```
int main()
{
  if (true) {
    42;
  } else {
    bar; /* Error: undeclared variable */
  }
}
```

Expected error:

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

B.2.35 fail-list1.mc

```
int main()
{
  List<int> range_list;
```

```
  int i;
  int n;

  n = 5;
  range_list = range(n);
  append(range_list, 999.0);
  return 0;
}
```

Expected error:

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

B.2.36 fail-list2.mc

```
int main()
{
  int i;
  int n;

  n = 5;
  i = len(n);
  return 0;
}
```

Expected error:

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

B.2.37 fail-list3.mc

```
int main()
{
  List<float> float_list;
  int n;
  n = 5;
  float_list = [1.2, 2.4];
  append(float_list, n);
  return 0;
}
```

Expected error:

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

B.2.38 fail-nomain.mc

```
```

Expected error:

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

B.2.39 fail-ops1.mc

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

Expected error:

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

B.2.40 fail-ops2.mc

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

Expected error:

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

B.2.41 fail-ops3.mc

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

Expected error:

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

B.2.42 fail-print.mc

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

Expected error:

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

B.2.43 fail-printb.mc

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

Expected error:

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

B.2.44 fail-printbig.mc

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

Expected error:

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

B.2.45 fail-return1.mc

```
int main()
{
  return true; /* Should return int */
}
```

Expected error:

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

B.2.46 fail-return2.mc

```
void foo()
{
  if (true) return 42; /* Should return void */
  else return;
}

int main()
{
  return 42;
}
```

Expected error:

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

B.2.47 fail-set1.mc

```
int main()
{
  Set<float> fset;
  float f = 10.0;
  int a = 10;
  setAdd(fset, f);
  setRemove(fset, 10);
  return 0;
}
```

Expected error:

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

B.2.48 fail-set2.mc

```
int main()
{
  Set<int> iset;
  setAdd(iset, true);
  return 0;
}
```

Expected error:

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

B.2.49 fail-while1.mc

```
int main()
{
  int i;

  while (true) {
    i = i + 1;
  }

  while (42) { /* Should be boolean */
    i = i + 1;
  }
```

```
}
```

Expected error:

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

B.2.50 fail-while2.mc

```
int main()
{
  int i;

  while (true) {
    i = i + 1;
  }

  while (true) {
    foo(); /* foo undefined */
  }

}
```

Expected error:

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