|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Shanqi Lu | Jiafei Song | Zihan Jiao | Yanan Zhang | Hyoyoon Kate Kim |
| sl4017 | js4984 | zj2203 | yz3054 | hk2870 |

*Liva*

*A Lite Version of Java*

Table of Contents

[Chapter1 Introduction 3](#_Toc458714006)

[Chapter2 Language Tutorial 3](#_Toc458714007)

[2.1 Environment Setup 3](#_Toc458714008)

[2.2 Run and test 3](#_Toc458714009)

[2.3 Hello World 5](#_Toc458714010)

[Chapter3 Language Manual 6](#_Toc458714011)

[3.1 INTRODUCTION 6](#_Toc458714012)

[3.2 LEXICAL CONVENTIONS 6](#_Toc458714013)

[3.2.1 White Space 7](#_Toc458714014)

[3.2.2 Comments 7](#_Toc458714015)

[3.2.3 Identifiers 7](#_Toc458714016)

[3.2.4 Keyword 7](#_Toc458714017)

[3.2.5 Literals 7](#_Toc458714018)

[3.2.6 Separators 9](#_Toc458714019)

[3.2.7 Operators 9](#_Toc458714020)

[3.3 Types 9](#_Toc458714021)

[3.3.1 Primitive Types 9](#_Toc458714022)

[3.3.2 Reference Types 10](#_Toc458714023)

[3.4 Expressions 11](#_Toc458714024)

[3.4.1 Evaluation, Denotation, and Result 11](#_Toc458714025)

[3.4.2 Type of an Expression 12](#_Toc458714026)

[3.4.3 Evaluation Order 13](#_Toc458714027)

[3.4.4 Lexical Literals 13](#_Toc458714028)

[3.4.5 The Arithmetic Operations 13](#_Toc458714029)

[3.4.6 The Relational Operations 14](#_Toc458714030)

[3.4.7 The Bitwise and Conditional Operations: 15](#_Toc458714031)

[3.4.8 Method Invocation Expressions 15](#_Toc458714032)

[3.4.9 Array Access Expressions 15](#_Toc458714033)

[3.4.10 Assignment 16](#_Toc458714034)

[3.5 Statements 16](#_Toc458714035)

[3.5.1 Expression Statements 16](#_Toc458714036)

[16](#_Toc458714037)

[3.5.2 Declaration Statements 16](#_Toc458714038)

[3.5.3 Control Flow Statements 17](#_Toc458714039)

[3.5.4 Method Creation and Method Call 19](#_Toc458714040)

[3.5.5 Print to Console 20](#_Toc458714041)

[3.5.6 Empty Statement 21](#_Toc458714042)

[3.6 Classes 21](#_Toc458714043)

[3.6.1 Class Declarations 21](#_Toc458714044)

[3.6.2 Object Instantiation 22](#_Toc458714045)

[3.6.3 Inheritance 22](#_Toc458714046)

[4 Project Plan 23](#_Toc458714047)

[4.1 Plan 23](#_Toc458714048)

[4.2 Timeline 23](#_Toc458714049)

[4.3 Roles and Responsibilities. 24](#_Toc458714050)

[4.4 Software Environment 24](#_Toc458714051)

[4.5 Project Prolog 24](#_Toc458714052)

[5. Architecture Design 25](#_Toc458714053)

[5.1 Overview 25](#_Toc458714054)

[5.2 The scanner 25](#_Toc458714055)

[5.3 The Parser 25](#_Toc458714056)

[5.4 The Semantic Checker 26](#_Toc458714057)

[5.5 The Code Generator 27](#_Toc458714058)

[Chapter6 TEST PLAN 27](#_Toc458714059)

[6.1 Reason for chosing test cases 28](#_Toc458714060)

[6.2Unit testing 28](#_Toc458714061)

[6.3Integration testing 28](#_Toc458714062)

[6.4Automation 28](#_Toc458714063)

[6.5Representative source language programs 28](#_Toc458714064)

# Chapter 1 Introduction

Liva is a general-purpose programming language and a lite version of Java. Having realized that the focus of this project should be applying compiler design theories in practice rather than innovation, we decided to develop a language that has the similar syntax and abstract data types in Java. Past projects related to general-purpose languages all have their own interesting features like Cpi (2013) implemented a data type called “struct” and Dice (2015) developed their own version of inheritance mechanism. Our preference is to develop some basic features and object-oriented model that resembles Java. This language is compiled down to LLVM.

Unlike domain-specific programming languages are designed for specific fields, our language is designed for general purpose, hence serving as a portable language that runs on many platforms as long as LLVM is runnable. Programs written in Liva will look like Java in many ways including variable declaration and class declaration. Common algorithms like GCD can be easily implemented using our language.

# Chapter 2 Language Tutorial

## 

## 2.1 Environment Setup

This compiler has been built and tested on Ubuntu 16.04 and OSX. Prerequisites include OCaml, LLVM, Opam, etc. Instructions on how to install the necessary packages under Ubuntu 16.04 is shown below:

sudo apt-get install -y ocaml m4 llvm opam

opam init

opam install llvm.3.6 ocamlfind

eval `opam config env`

For installation under OS X, please refer to the MicroC compiler’s README file which can be found at <http://www.cs.columbia.edu/~sedwards/classes/2016/4115-summer/index.html>

## 2.2 Run and test

First, make sure that you are in the “Liva” root directory. To test the compiler using the test suit, follow the instructions below:

$ make liva

-ocamlfind ocamlopt -c -package llvm ast.ml

-ocamlfind ocamlopt -c -package llvm sast.ml

-ocamlyacc parser.mly

-ocamlc -c ast.ml

-ocamlc -c parser.mli

-ocamlfind ocamlopt -c -package llvm parser.ml

-ocamllex scanner.mll

127 states, 6605 transitions, table size 27182 bytes

-ocamlfind ocamlopt -c -package llvm scanner.ml

-ocamlfind ocamlopt -c -package llvm semant.ml

-ocamlfind ocamlopt -c -package llvm codegen.ml

-ocamlfind ocamlopt -c -package llvm liva.ml

-ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis ast.cmx sast.cmx parser.cmx -scanner.cmx semant.cmx codegen.cmx liva.cmx -o liva

You may need to run “chmod +x testall.sh” to unlock the shell script.

$ ./testall.sh

-n test-add...

OK

-n test-and...

OK

...

-n fail-sub...

OK

-n fail-while1...

OK

## 2.3 Hello World

To test a simple “Hello World” program, you need to:

1 compile the compiler (this have already been done using “make liva” in the previous step)

2 output generated LLVM IR code to a “\*.ll” file

3 run “\*.ll” file

hello\_world.liva:

class test {

void main () {

print ("Hello World!\n");

}

}

Copy the above code to an empty file named “hello\_world.liva”, then run:

$ ./liva < hello\_world.liva > hello\_world.ll

$ lli hello\_world.ll

This should print out the desired result “Hello World!”.

## 2.4 Another small program

This is an example to show that Liva supports object-oriented paradigm. Write another small program and test it.

test.liva:

class calculator{

int addition(int x, int y){

int z;

z = x + y;

return(z);

}

}

class test {

void main(){

int result;

class calculator obj = new calculator();

result = obj.addition(31,79);

print ("result=",result,"\n");

}

}

$ ./liva < test.liva > test.ll

$ lli test.ll

result=110

# Chapter 3 Language Manual

## 3.1 INTRODUCTION

Liva is a general purpose programming language and a lite version of Java. It is designed to let programmers who are familiar with class-based languages to feel comfortable with developing common algorithms like GCD. It is lite in the sense that it maintains some but not all features in Java. It has the similar syntax and abstract data types in Java and supports object-oriented paradigm and inheritance. However, generics and nested classes are beyond the scope of this project, hence they are not to be implemented.

The Liva programming language is strongly typed. The compiler checks whether arguments passed to a function match expected types and return an error if not. It is a portable language and compiled down to LLVM.

This language reference manual is organized as follows:

* Section 3.2 describes the lexical conventions of the Liva programming language.
* Section 3.3 describes types. Types are divided into two categories: primitive types and reference types.
* Section 3.4 describes classes including class declarations and inheritance.
* Section 3.5 describes statements.
* Section 3.6 describes expressions.

## 3.2 LEXICAL CONVENTIONS

This chapter specifies the lexical conventions of Liva programming language. A compiler takes a program which consists of a sequence of characters and reduce it to a sequence of elements, which are tokens, white space and comments. The tokens are identifiers, keywords, literals, separators, and operators.

*Element:*

*White Space| Comment| Token*

*Token:*

*Identifier| Keyword| Literal| Separator| Operator*

### 3.2.1 White Space

White space in Liva is defined as space character, tab character, form feed character(page-breaking) and line terminator character. White space characters are ignored by a compiler except as they serve to separate tokens.

### 3.2.2 Comments

There is one kind of comments:

*/\* text \*/*

All characters from “/\*” to “\*/” are ignored.

### 3.2.3 Identifiers

An identifier is a sequence of letters, digits and underscore ‘\_’. It can only begin with a letter. Identifiers are the names of variables, methods and classes. They are case-sensitive.

### 3.2.4 Keyword

Keywords are reserved and cannot be used as identifiers.

* *Keyword:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| for | *new* | *if* | *boolean* | *this* | *break* |
| *double* | *implements* | *else* | *import* | *return* | *extends* |
| *int* | *char* | *void* | *class* | *float* | *while* |

### 3.2.5 Literals

Literals are syntactic representations of numeric, character, boolean or string data. They are used for representing values in programs.

#### 3.2.5.1 Boolean Literals

There are two boolean literals:

* **true** represents a true Boolean value
* **false** represents a false Boolean value

#### 3.2.5.2 Integer Literals

Integer numbers in Liva are in decimal format. Negative decimal numbers such as -10 are actually expressions consisting of the operator ‘-’ and integer literal. The primitive type of integer literal is **int**.

#### 3.2.5.3 Floating Point Literals

Floating point numbers are expressed as decimal fractions and consist of:

* an optional ‘+’ or ‘-’ sign; if omitted, the value is positive,
* one of the following formats

|  |  |  |  |
| --- | --- | --- | --- |
| Format | | | Example |
| integer digits | **.** |  | 7. |
| integer digits | **.** | integer digits | 17.31 |
|  | **.** | integer digits | .56 |

#### 3.2.5.4 Character Literals

Character literals are surrounded by single quotes: **'a'**, **'#'**

#### 3.2.5.5 String Literals

String literals begin with a double quote character **"**, followed by zero or more characters and a terminating double quote **"**

Within string literals, there can be escape sequences but not unescaped newline.

#### 3.2.5.6 Escape Sequences for Character and String Literals

An escape sequence is used to represent a special character. It begins with a backslash character (\), which indicates that the following character should be treated specially. Escape sequences are listed in the table below.

|  |  |
| --- | --- |
| Name | Character |
| TAB | \t |
| newline | \n |
| double quote | \” |
| single quote | \’ |
| backslash | \\ |

### 3.2.6 Separators

Separators are symbols used for separating tokens.

{ } ( ) ; , .

### 3.2.7 Operators

The expression section of this manual will explain behaviors of these operators. Here is a list of all the supported operators.

= > < ! == >=  
<= != & | + -  
\* \ %

## 3.3 Types

The Liva programming language supports two kinds of types: primitive types and reference types. There are also two kinds of data values: primitive values and reference values accordingly. There is also a special null type.

Primitive types are boolean types and numeric types. Reference types are class types and array types.

### 3.3.1 Primitive Types

Primitive types are predefined by the Liva programming language. Their names are reserved keywords.

#### 3.3.1.1 Integral Types

The integral types are int and char.

The integer data type is a 32-bit sequence of digits, which has a minimum value of -2^31 and a maximum value of 2^31-1. An integer literal is a sequence of digits preceded by an optional negative sign. A zero value cannot be preceded by a negative sign.

**int x** = 10;  
**int y** = -50;  
**int z** = 0;

#### 3.3.1.2 Char Types

The char data type is a single 8-bit ASCII character. Their values range from ‘0x00’ to ‘0x7F’.

**char x** = **'a'**;

#### 3.3.1.3 Floating-Point Types

The floating-point data type is a signed-precision 32-bit format values.

**float x** = 1.5;  
**float y** = -5.1;  
**float z** = 2.0;

#### 3.3.1.4 The Boolean Type

The boolean data type has two possible values: true and false. A boolean is its own type and cannot be compared to a non-boolean variable. Therefore, expression “true == 1” would lead to an error.

**boolean x** = **true**;  
**boolean y** = **false**;

### 3.3.2 Reference Types

There are two kinds of reference types: class types and array types.

#### 3.3.2.1 Class types

A class type consists of an identifier which is optionally followed by parameters. A class is an extensible template for creating objects. See Chapter 6 to get more explanation about class and object.

#### 3.3.2.2 String

String is a build-in class in Liva programming language. A string is a sequence of characters like the word “Hello”. Section 6. will introduce the String class.

#### 3.3.2.3 Array types

Array is a special type. An array contains a number of elements with primitive types. All elements in an array must have the same types.

**int**[] **ai**;   
**char**[] **ac** = { **'a'**, **'b'**, **'c'**, **' '** };

#### 3.3.2.4 Void type

The void type is used to indicate an empty return value from a function call. For example, a main function does not have a return value. It is declared in the following way.

**void** main{ }

## 3.4 Expressions

Much of the work in a program is done by evaluating *expressions*, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both. This chapter specifies the meanings of expressions and the rules for their evaluation.

### 3.4.1 Evaluation, Denotation, and Result

Liva evaluates a larger expression by evaluating smaller parts of it. When an expression in a program is *evaluated* (*executed*), the result denotes one of three things:

* A variable

int a = 1;

int b;

b = a + 1; /\* returns variable \*/

* A value

int foo (int a){

return a + 1;

}

foo(3); /\*returns a value\*/

* Nothing (for void functions and methods)

int x = 7;

void increase (int a) {

x = x + a;

}

increase(2); /\* returns nothing \*/

An expression denotes nothing if and only if it is a method invocation that invokes a method that does not return a value, that is, a method declared void. Such an expression can be used only as an expression statement (in statement chapter), because every other context in which an expression can appear requires the expression to denote something.

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, if the expression denotes a variable or a value, we may speak simply of the *value* of the expression. In this way, we may say each expression denotes a value in a certain type.

### 3.4.2 Type of an Expression

For an expression that denotes to a variable, the value stored in a variable is always compatible with the type of the variable. In other words, the value of an expression whose type is T is always suitable for assignment to a variable of type T.

The rules for determining the type of an expression that denotes to a value are explained separately below for each kind of expression. Including arithmetic operations, relation operations, bitwise/conditional operations, assignment.

### 3.4.3 Evaluation Order

Liva guarantees that the operands of operators are evaluated in a specific *evaluation order*, namely, from left to right.

#### 3.4.3.1 Left-Hand Operand First

The left-hand operand of a binary operator is fully evaluated before any part of the right-hand operand is evaluated.

#### 3.4.3.2 Evaluate Operands before Operation

Liva guarantees that every operand of an operator (except the conditional operators &, |) is fully evaluated before any part of the operation itself is performed.

For example, in an assignment expression, the assignment will not be evaluated until the right hand operands (if it is another expression) is evaluated.

#### 3.4.3.3 Evaluation Respects Parentheses and Precedence

Liva respects the order of evaluation indicated explicitly by parentheses and implicitly by operator precedence.

### 3.4.4 Lexical Literals

A literal denotes a fixed, unchanging value. This kind of expression could be evaluated without being broken into small expressions.

The type of a literal is determined has been defined previously. Evaluation of a lexical literal always completes normally.

### 3.4.5 The Arithmetic Operations

The operators +, -, \*, /, and % are called the arithmetic operators. An expression concatenated by an arithmetic operator is called an arithmetic expression. The value of an arithmetic expression is numeric (int or double, depends on the operands, defined previously).They have precedence of two level: \*, / and % are higher than + and -. They are syntactically left-associative (they group left-to-right). The type of the arithmetic expression is the promoted type of its operands.

The type of each of the operands of arithmetic operators must be a type that is convertible to a primitive numeric type, or a compile-time error occurs. For example: adding two objects of a user-defined class with “+” is prohibited.

|  |  |
| --- | --- |
| **+** | Adds values on either side of the operator |
| **-** | Subtracts right hand operand from left hand operand |
| **\*** | Multiplies values on either side of the operator |
| **/** | Divides left hand operand by right hand operand |
| **%** | Divides left hand operand by right hand operand and returns remainder |

### 3.4.6 The Relational Operations

The value of an equality expression is always boolean.The equality operators may be used to compare two operands that are convertible to numeric int type, or two operands of type boolean. Liva does not support character comparison. All other cases result in a compile-time error.

|  |  |
| --- | --- |
| **==** | Checks if the values of two operands are equal or not, if values are equal then condition becomes true. |
| **!=** | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. |
| **>** | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. |
| **<** | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. |
| **>=** | Checks if the value of left operand is greater than or equal to (no less than) the value of right operand, if yes then condition becomes true. |
| **<=** | Checks if the value of left operand is less than or equal to (no greater than) the value of right operand, if yes then condition becomes true. |

### 3.4.7 The Bitwise and Conditional Operations:

Unlike Java, Liva uses &, |, ^, ~ for both bitwise and conditional operations. That depends on the operands. The operands of a Bitwise/Conditional Operation should both be int or boolean.

|  |  |
| --- | --- |
| & | Binary AND if both operands are int type / Logic AND if both operands are boolean type |
| | | Binary OR if both operands are int type / Logic OR if both operands are boolean type |
| ^ | Binary XOR if both operands are int type / Logic XOR if both operands are boolean type |
| ~ | Binary Complement if both operands are int type / Logic INVERT if both operands are boolean type |

### 3.4.8 Method Invocation Expressions

A method/function invocation expression is used to invoke an instance method (explanation about instance and method are in the class section). The result type of the chosen method is determined as follows: If the chosen method/function is declared with a return type of void, then the result is void. Otherwise, the result type is the method/function's declared return type.

### 3.4.9 Array Access Expressions

An array access expression contains two subexpressions, the *array reference expression* (before the left bracket) and the *index expression* (within the brackets).

a[10]

/\* if a is an int array, then a[10] is an array access expression that returns a variable with an int value. \*/

Note that the array reference expression may be a name or any primary expression that is not an array creation expression. The type of the array reference expression must be an array type. For the index expression, the promoted type must be int, or a compile-time error occurs.

The result of an array access expression is a variable of type T or an object, namely the variable or the object within the array selected by the value of the index expression.

### 3.4.10 Assignment

In Liva, the only assignment operator is “=”.

The result of the first operand of an assignment operator must be a variable. This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access or an array access (defined previously).

int a;

a = 1; /\* variable assignment \*/

class foo {

int value;

constructor(int x) {

this.value = x /\* class field assignment \*/

}

}

class foo test = new foo (2);

test.value = 3; /\* object properties assignment \*/

## 3.5 Statements

A statement forms a complete unit of execution. All expressions of statements are explained in the following. Except as indicated, statements are executed in sequence.

### 3.5.1 Expression Statements

Most statements are expression statements and expression statements are usually assignments or function calls. A function call is a call to a function along with its formal arguments

/\* Object creation expressions \*/

class Student e1 = new Student ();

/\* Assignment expressions \*/

c = 8933.234;

### 3.5.2 Declaration Statements

A declaration statement declares a variable by specifying its data type and name. It also could initialize the variable during the declaring.

*/\* declare a variable with data type and name \*/***char a**;  
**int b** =10;  
**float c**;

**int [] array1** = **new int [**10**]** ;  
**float n**= 3.5;

**boolean isMatch** = **false**;

### 3.5.3 Control Flow Statements

#### 3.5.3.1 If-then and If-then-else

There are two forms of conditional statements.

For the first If-then case, the conditional expression that is evaluated is enclosed in balanced parentheses. The section of code that is conditionally executed is specified as a sequence of statements enclosed in balanced braces. If the conditional expression evaluates to false, control jumps to the end of the if-then statement.

if (expression) {

statement1

statement2

...

}

In the second If-then-else case, the second sub-statement is executed if the expression is false. In liva, for a single statement, the brackets are optional.

if (expression) {

statement1

} else {

Statement2

}

The following example shows how to use if-then else statement:

**if** (**true**){

print(42);

print (100);

} **else** {

print(8);

}

#### 3.5.3.2 Looping: for

The ‘for’ condition will also run in a loop so long as the condition specified in the ‘for’ statement is true. The ‘for’ statement has the following format:

for (expression1; expression2; expression 3) {

statement

}

The first expression specifies initialization for the loop and it is executed once at the beginning of the 'for' statement (liva could not accept a statement for expression1 such as int i=0); the second specifies a Boolean expression, made before each iteration, such that the loop is terminated when the expression becomes false; the third expression specifies an operation which is performed after each iteration.

The following example uses a ‘for’ statement to print the numbers from 0 to 10:

**int** num;

**for** (num=0; num < 11; num ++) {

print(num);

}

#### 3.5.3.3 Looping: while

The ‘while’ statement has the form:

while(expression) {

statement

}

The ‘while’ statement will be executed in a loop as long as the specified condition in the while statement is true. The expression must have type boolean, or a compile-time error occurs.

* If the value for expression is true, then the contained statement is executed
* If execution of the statement completes normally, then the entire ‘while’ statement is executed again, beginning by re-evaluating the expression*.*
* If the value of the expression is false, no further action is taken and the ‘while’ statement completes normally.

The following example shows how to use ‘while’ statement:

**int** **i**=10;

**while** (i > 0) {

print(i);

i = i - 1;

}

#### 3.5.3.4 Return

A ‘return’ returns to its caller by means of the ‘return’ statement, which has one of the forms:

return;

return(expression);

In the first case no value is returned when a method is declared void. For the first case, the users could specify no return statement for simplification. In the second case, simply put the value (or an expression that calculates the value) after the return Keyword, then the value of the expression is returned to the caller of the ‘return’.

### 3.5.4 Method Creation and Method Call

The user could write the user-defined methods.

returnType nameOfMethod (Parameter List) {

/\*method body\*/

}

* returnType: Method may return a value.
* nameOfMethod: This is the method name. The method signature consists of the method name and the parameter list.
* Parameter List: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters. A usual format for the parameter list is (dataType Id, dataType Id …, dataType Id )
* method body: The method body defines what the method does with statements.

For using a method, it should be called. There are two ways in which a method is returned i.e. method returns a value or returning nothing (no return value). The method should be defined in a class.

Following is the example to demonstrate how to define a method and how to call it with a returned value in a class:

**class** myclass{

**int** calc (**int** x, **int** y){

**int** z;

z = x + y;

**return** (z);

}

}

**class** test {

**void** main(){

**int** x = 9;

**float** y = 6.0;

**float** z;

**class** myclass obj = **new** myclass();

z = obj.calc(x, y);

print (**"z=",** z);

}

}

### 3.5.5 Print to Console

The *print()* function takes one or more parameters and prints them one by one to standard output. The parameter type may be string, number. It is in the following form:

print (parameters);

Here is an example to accept an *int* and print the *int* to the console.

print (1);

Another example to accept a string and print the string to the console:

print (“CS4115 is fun!”)

### 3.5.6 Empty Statement

An empty statement does nothing and has the following form:

;

## 3.6 Classes

Classes and objects are highly related. A real world object has states and behaviors. For instance, a dog has states — name, breed, color as well as behaviors — barking, eating. An object in Liva also has states — stored in fields and behaviors — shown via methods. Fields are local variables stored inside a class and methods are functions that may be invoked by instances of that class. So class bodies consist of fields and methods. Actually, there is a special kind of methods called constructors which will be discussed in following sections. In a word, a class can be defined as a template that describes the states and behaviors that objects of its type support.

### 3.6.1 Class Declarations

A class declaration defines a new reference type and how it is implemented. A body of a class declaration contains field declarations and method declarations. Following is an example of how to define a class. Use the keyword “class” to indicate the start of a class declaration. In this example, “myclass” is the class name. It has two fields, one is an integer and the other is a character. One method “fuction1” is define and it simply returns integer 1.

**class** myclass {  
 **int field1**;  
 **char field2**;  
 **int** function1(){  
 **return** (1);  
 }  
}

### 

### 3.6.2 Object Instantiation

The ‘new’ keyword is used to instantiate a new object by allocating memory for it. The ‘new’ requires a single argument: a *constructor method* for the object to be created. The constructor method is responsible for initializing the new object. In Liva, the name of constructor method for one class is defined as “constructor”. The user could define a constructor in the class. liva would create a default constructor for the class if the users do not explicitly define a constructor for a class. It is essentially a non-parameterized constructor without any arguments. The function of default constructor is to call the super class constructor and initialize all instance variables. When the keyword ‘this’ is used, it is replaced by an instance of the containing object at runtime.

The following example shows how to define a constructor method and initialize a new object:

*/\* User defined constructor\*/*

**class** myclass{

**int** **a**;

constructor(**int** x){

**this**.**a** = x;

}

}

**class** test {

**void** main(){

**class** myclass obj = **new** myclass(10);

print (**"a="**, obj.a);

}

}

### 3.6.3 Inheritance

Inheritance is that a subclass acquires all the behaviors and properties of a super class. A subclass inherits fields and methods from its parent class. The keyword “extends” is used to indicate a relationship between a subclass and a parent class like the following declaration.

**class** subclass **extends** myclass {  
  
}

A subclass may override the methods of its parent class. If a method is overridden, an instance of the subclass can only access the new version but not the original method from the super class.

**class** myclass{  
 **int** calc (**int** x, **int** y){  
 **int** z;  
 z = x + y;  
 **return** (z);  
 }  
}  
**class** subclass **extends** myclass{  
 **int** calc (**int** x, **int** y){  
 **int** z;  
 z = x + y + 1;  
 **return** (z);  
 }  
}

# 4 Project Plan

## 4.1 Plan

Our group members worked collaboratively on Github. At the beginning, we studied the code of the MicroC languge to get more understanding about compiler design. We have met our TA every week to discuss our project and ask questions. After we got the “Hello Word” program successfully compiled, we worked together to add more features and fix bugs. Finally, in the last week, we asked our TA about the priority of the features we planned to add and implemented some of them with high priorities.

## 4.2 Timeline

|  |  |
| --- | --- |
| July 6th - 11th | Brainstorm for language design and the proposal |
| July 11th – 20th | Determine language syntax and scope. Write Language Reference Manual. |
| July 20th – July 25th | Scanner and Parser. |
| July 25th – August 1st | Expressions, Print function and main function. |
| August 1st | “Hello World !”. |
| August 1st – August 8th | Add object-oriented features and array. Fix bugs |
| August 8th - August 11th | Report and presentation. |

## 4.3 Roles and Responsibilities.

In our team, the role of each member was quite flexible. We worked together very often coding and fixing bugs. We have solved many problems together these days. Zihan and Shanqi mainly concentrated on implementing the code generator. Jiafei and Yanan focused on semantic check. Kate joined us from the middle of our project and she did some testing and document work.

## 4.4 Software Environment

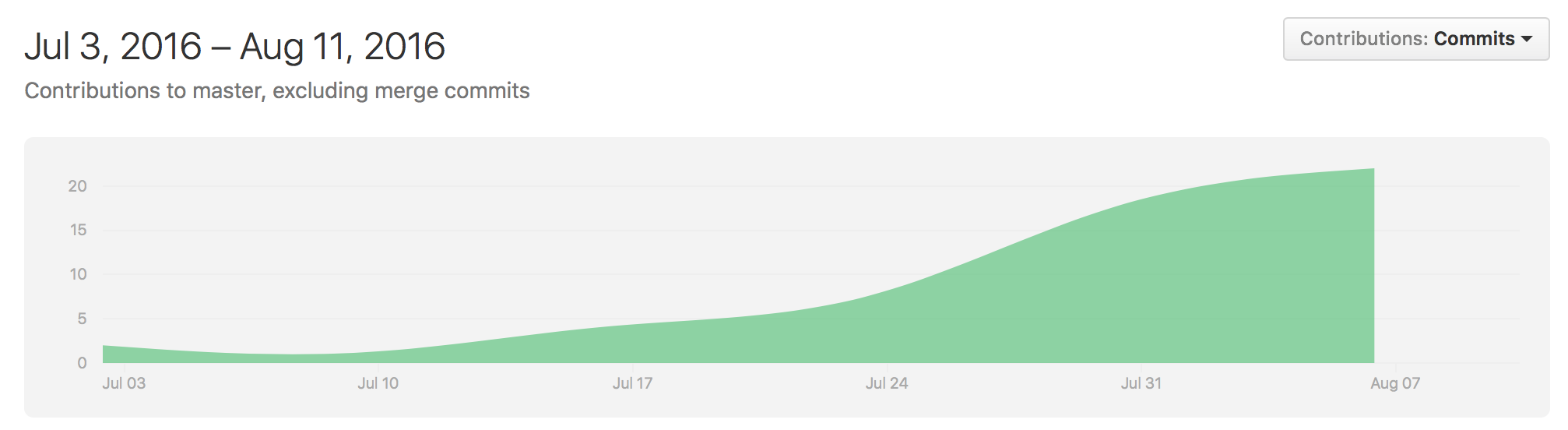
**Ubuntu 16.04** – All our team members work on Ubuntu 16.4, the latest version to avoid unnecessary environment conflict. Our Ubuntu are powered by PD11 and VMware 12.

**OS X** – Meanwhile, we also try our language on OS X to ensure that LIVA is portable.

**LLVM 3.6** - All our team members work with LLVM 3.6

**Github** – We work on the same branch in our Github repository.

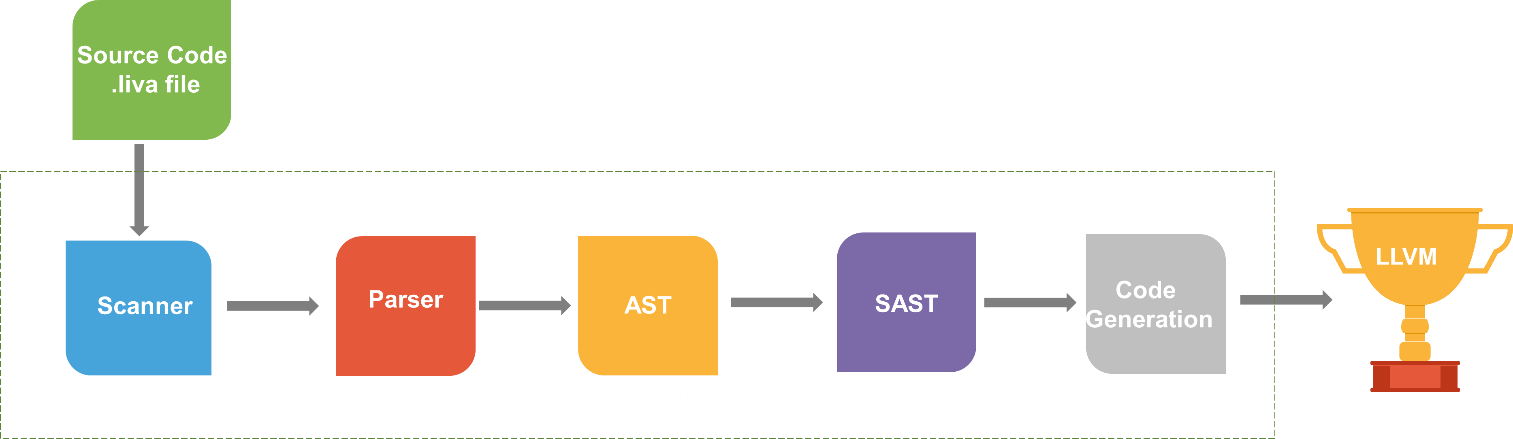
## 4.5 Project Prolog



The commit curve from July 3 to Aug 11. It shows our work distributed evenly during the 30-day period. All our team members involved heavily in the development of LIVA.

# 5. Architecture Design

## 5.1 Overview



**Figure 5.1 Overview of compiler architecture design**

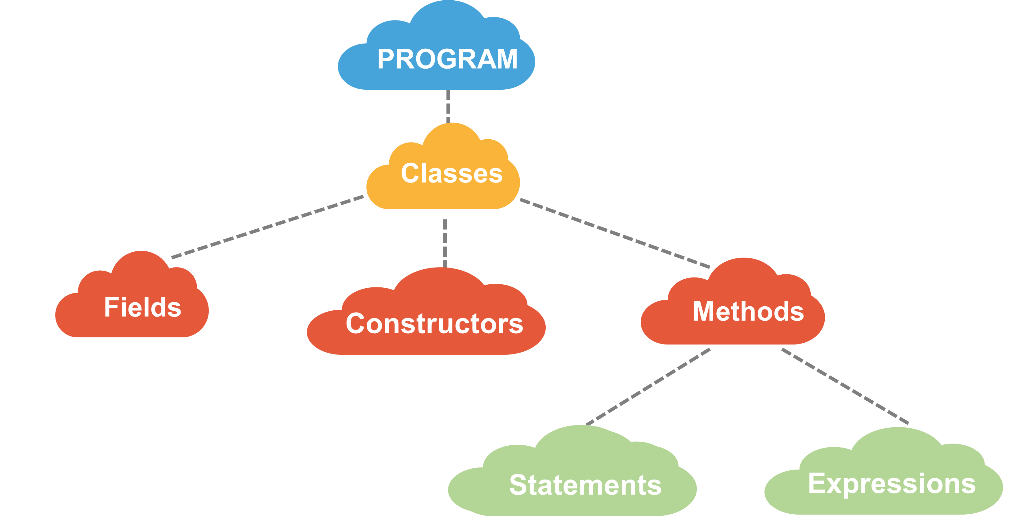
The compiler architectural design of Liva is shown in Figure 5.1. Overall, Liva follows a traditional compiler architecture design with a lexical scanner and parser at the front end, followed by generation of a Semantically Checked and Typed Abstract Syntax Tree (SAST) from an abstract syntax tree (AST) and finally LLVM IR code generation. We have a total of 5 modules which are codegen.ml, liva.ml, semant.ml, parser.mly, scanner.mll and 2 interfaces which are ast.ml and sast.ml.

## 5.2 The scanner

As the start of the front end, scanner reads a source file and convert it into tokens, and at the same time it checks whether each tokens is valid, if not, it will report the illegal character. Besides, it is also responsible for ignoring white space and comments which are not useful for Liva program.

## 5.3 The Parser

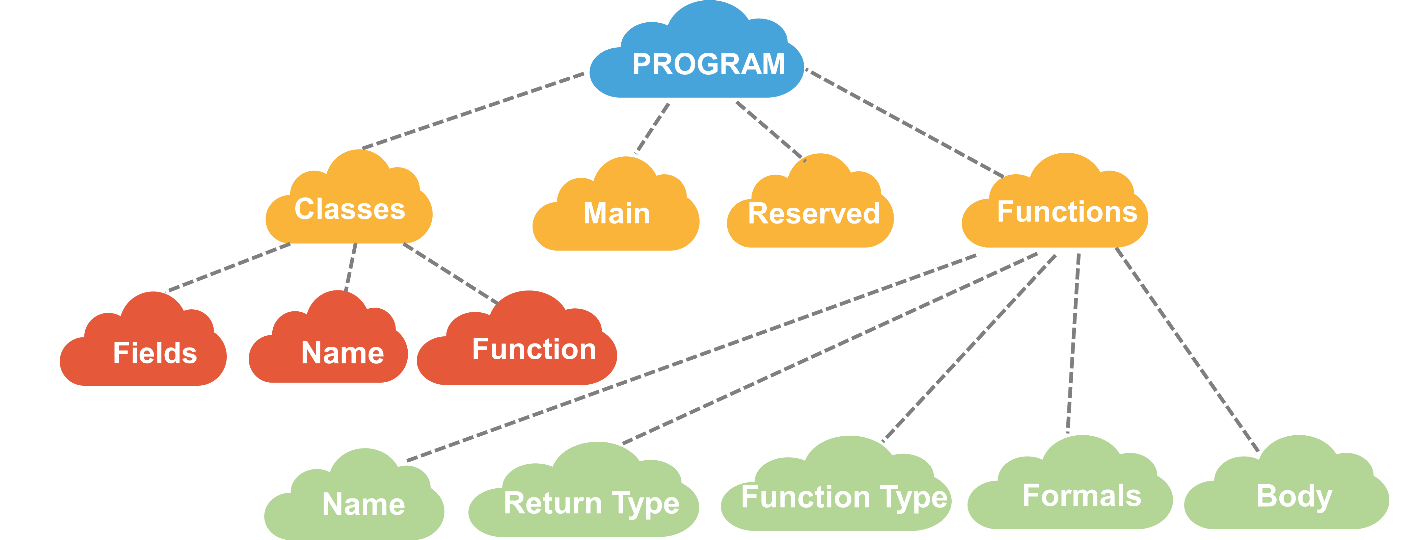
the tokens passed by the scanner are interpreted by the parser according to the precedence rules of Liva language and constructs an abstract syntax tree based on the definitions provided and the input tokens are constructed. The parser’s main goal is to organize the tokens of the program into class declarations. The top level of the abstract syntax tree is a structure called Program which contains all classes. The fields, constructors and methods are declared within the classes. Specific to the method declarations record is the creation of an AST of functions from groups of statements, statements evaluating the results of expressions, and expressions formed from operations and assignments of variables, references and constants. The Parser produces the abstract syntax tree (AST) which is shown in Figure 5.1.

****

**Figure 5.2 AST program representation.**

## 5.4 The Semantic Checker

There are four kinds of work a semantic checker is responsible for. To begin with, it adds reserved functions as a part of the SAST shown in figure5.3, and these reserved functions can also help to check whether there are any functions use the same name as reserved ones which is not allowed in Liva. Next, semantic checker do some work concerning statistic semantic checking. It checks whether the source code is semantically correct from various aspects including whether there are duplicated fields or methods in one class, whether there are duplicated classes in one program and adds default constructor if there isn’t user-defined constructor in one class. Thirdly, on the basis of the first two semantic checker deals with inheritance. Semantic checker finds all the inheritance relationship by looking through all the classes, if there is a inheritance relationship between two classes, semantic checker will add the fields and the methods of super class to subclass, but it the subclass declares fields or methods which share the same name as those of super class, subclass will override those fields and methods of super class. Finally, semantic checker converts its input, AST, to SAST which is helpful for code generator to generate LLVM IR code. Semantic checker separates the methods from classes, separates main method which is the entrance of Liva program from methods and add types to all the statements and expressions, in the meanwhile, it also do some work related with statistic semantic checking, including whether names or identifiers are defined before they are referred to, whether names or identifiers are used correctly, whether types are consistent and so on to make it as smooth as possible for code generator to generate code.

****

**Figure 5.3 SAST representation.**

## 5.5 The Code Generator

The main function of the compiler is to convert the abstract syntax tree into LLVM IR. After the semantic abstract syntax tree is generated, the semantic checker sends it to the code generator which construct the LLVM IR file which contains the final instructions for the program.

This code generation is written using the OCaml LLVM library, which uses OCaml functions to produce the desired LLVM code with the static variables used during code generation. Codegen.the\_module is the top-level structure that the LLVM IR uses to contain code and it contains all of the functions and global variables in a chunk of code. The Codegen.builder object is a object that keeps track of the current place to insert instructions and has methods to create new instructions. The Codegen.named\_values map keeps track of which values are defined in the current scope and what their LLVM representation is. After all of above is setup, the code generator iterates through the entire semantic abstract syntax tree and produces the necessary LLVM code for each function, statement, and expression.

# Chapter 6 TEST PLAN

All the test cases used to test Liva are put into a folder named tests. The test cases which consist of unit tests, integration tests are designed to test all the features of Liva, from both positive way and negative way. All these various testing methods are used to create a robust testing environment for Liva language.

## 6.1 Reason for chosing test cases

All the tests were added as new language features were added, therefore, the language features decided the test cases we picked out to a large extent, and most of these tests are aimed at testing every aspect of Liva.

## 6.2 Unit testing

Unit testing was used to check whether small pieces of our language could behave as defined. The tests can be divided into two types: tests are meant to pass, tests are meant to fail, thereby utilizing positive and negative testing. Negative testing ensures that invalid input is not accepted, i.e. Liva is able to properly reject invalid input, while positive testing on the other hand allows us to assess whether Liva is able to behave and work out the result as defined.

## 6.3 Integration testing

Once smaller tests were verified to pass, they would be integrated into larger programs, so as to ensure whether Liva can manage to behave properly in more complex program. These integrated tests are the basis of our final interesting program.

## 6.4 Automation

Automation of testing becomes more and more necessary as the project moves forward, and it is an extremely useful tool for developing our language. Automation allows us to make sure that new added language features would not obstruct other features which have passed the tests. We created an automated regression test suite largely borrowed from the MicroC Compiler. The test cases meant to pass are written out using the notation ‘test-.liva’ and the corresponding output as ‘test-.out’, while the corresponding output of test cases meat to fail as ‘test-.err’. The test script is executed with the ./testall.sh command which will then display a list of tests that pass, fail, or produce a printed output that differs from the desired printed output.

## 6.5 Representative source language programs

1.

test-inheritance2.liva

class calculator {

int add(int x, int y){

int z = x + y;

return(z);

}

}

class my\_calculator extends calculator{

}

class test {

void main(){

int x;

int y;

int z;

x = 66;

y = 98;

class my\_calculator obj = new my\_calculator();

z = obj.add(x,y);

print ("z=",z);

}

}

test-inheritance2.ll

; ModuleID = 'Liva'

%test = type <{ i32 }>

%my\_calculator = type <{ i32 }>

%calculator = type <{ i32 }>

@tmp = private unnamed\_addr constant [3 x i8] c"z=\00"

@tmp.1 = private unnamed\_addr constant [5 x i8] c"%s%d\00"

declare i32 @printf(i8\*, ...)

declare i8\* @malloc(i32)

define i64\* @lookup(i32 %c\_index, i32 %f\_index) {

entry:

%tmp = alloca i64\*\*, i32 3

%tmp1 = alloca i64\*, i32 0

%tmp2 = getelementptr i64\*\*, i64\*\*\* %tmp, i32 2

store i64\*\* %tmp1, i64\*\*\* %tmp2

%tmp3 = alloca i64\*

%tmp4 = getelementptr i64\*, i64\*\* %tmp3, i32 0

store i64\* bitcast (i32 (%my\_calculator\*, i32, i32)\* @my\_calculator.add to i64\*), i64\*\* %tmp4

%tmp5 = getelementptr i64\*\*, i64\*\*\* %tmp, i32 1

store i64\*\* %tmp3, i64\*\*\* %tmp5

%tmp6 = alloca i64\*

%tmp7 = getelementptr i64\*, i64\*\* %tmp6, i32 0

store i64\* bitcast (i32 (%calculator\*, i32, i32)\* @calculator.add to i64\*), i64\*\* %tmp7

%tmp8 = getelementptr i64\*\*, i64\*\*\* %tmp, i32 0

store i64\*\* %tmp6, i64\*\*\* %tmp8

%tmp9 = getelementptr i64\*\*, i64\*\*\* %tmp, i32 %c\_index

%tmp10 = load i64\*\*, i64\*\*\* %tmp9

%tmp11 = getelementptr i64\*, i64\*\* %tmp10, i32 %f\_index

%tmp12 = load i64\*, i64\*\* %tmp11

ret i64\* %tmp12

}

define %test\* @test.constructor() {

entry:

%this = alloca %test

%tmp = call i8\* @malloc(i32 ptrtoint (i1\*\* getelementptr (i1\*, i1\*\* null, i32 1) to i32))

%tmp1 = bitcast i8\* %tmp to %test\*

%tmp2 = load %test, %test\* %tmp1

store %test %tmp2, %test\* %this

%.key = getelementptr inbounds %test, %test\* %this, i32 0, i32 0

store i32 2, i32\* %.key

ret %test\* %this

}

define i32 @my\_calculator.add(%my\_calculator\* %this, i32 %x, i32 %y) {

entry:

%z = alloca i32

%addtmp = add i32 %x, %y

store i32 %addtmp, i32\* %z

%z1 = load i32, i32\* %z

ret i32 %z1

}

define %my\_calculator\* @my\_calculator.constructor() {

entry:

%this = alloca %my\_calculator

%tmp = call i8\* @malloc(i32 ptrtoint (i1\*\* getelementptr (i1\*, i1\*\* null, i32 1) to i32))

%tmp1 = bitcast i8\* %tmp to %my\_calculator\*

%tmp2 = load %my\_calculator, %my\_calculator\* %tmp1

store %my\_calculator %tmp2, %my\_calculator\* %this

%.key = getelementptr inbounds %my\_calculator, %my\_calculator\* %this, i32 0, i32 0

store i32 1, i32\* %.key

ret %my\_calculator\* %this

}

define i32 @calculator.add(%calculator\* %this, i32 %x, i32 %y) {

entry:

%z = alloca i32

%addtmp = add i32 %x, %y

store i32 %addtmp, i32\* %z

%z1 = load i32, i32\* %z

ret i32 %z1

}

define %calculator\* @calculator.constructor() {

entry:

%this = alloca %calculator

%tmp = call i8\* @malloc(i32 ptrtoint (i1\*\* getelementptr (i1\*, i1\*\* null, i32 1) to i32))

%tmp1 = bitcast i8\* %tmp to %calculator\*

%tmp2 = load %calculator, %calculator\* %tmp1

store %calculator %tmp2, %calculator\* %this

%.key = getelementptr inbounds %calculator, %calculator\* %this, i32 0, i32 0

store i32 0, i32\* %.key

ret %calculator\* %this

}

define i32 @main() {

entry:

%this = alloca %test

%tmp = call i8\* @malloc(i32 ptrtoint (i1\*\* getelementptr (i1\*, i1\*\* null, i32 1) to i32))

%tmp1 = bitcast i8\* %tmp to %test\*

%tmp2 = load %test, %test\* %tmp1

store %test %tmp2, %test\* %this

%.key = getelementptr inbounds %test, %test\* %this, i32 0, i32 0

store i32 2, i32\* %.key

%x = alloca i32

%y = alloca i32

%z = alloca i32

store i32 66, i32\* %x

store i32 98, i32\* %y

%obj = alloca %my\_calculator

%tmp3 = call %my\_calculator\* @my\_calculator.constructor()

%tmp4 = load %my\_calculator, %my\_calculator\* %tmp3

store %my\_calculator %tmp4, %my\_calculator\* %obj

%cindex = getelementptr inbounds %my\_calculator, %my\_calculator\* %obj, i32 0, i32 0

%cindex5 = load i32, i32\* %cindex

%fptr = call i64\* @lookup(i32 %cindex5, i32 0)

%my\_calculator.add = bitcast i64\* %fptr to i32 (%my\_calculator\*, i32, i32)\*

%x6 = load i32, i32\* %x

%y7 = load i32, i32\* %y

%tmp8 = call i32 %my\_calculator.add(%my\_calculator\* %obj, i32 %x6, i32 %y7)

store i32 %tmp8, i32\* %z

%z9 = load i32, i32\* %z

%tmp10 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([5 x i8], [5 x i8]\* @tmp.1, i32 0, i32 0), i8\* getelementptr inbounds ([3 x i8], [3 x i8]\* @tmp, i32 0, i32 0), i32 %z9)

ret i32 0

}

2.

test-while\_for\_nest.liva

class test {

void main(){

int i = 1;

int j;

while(i < 10)

{

j = 11;

for(j = 11; j < 13; j = j + 1)

print("i = ", i, " ", "j = ", j, "\n");

i = i + 1;

}

print("\n\n");

for(i = 1; i < 10; i = i + 1)

{

j = 11;

while(j < 13)

{

print("i = ", i, " ", "j = ", j, "\n");

j = j + 1;

}

}

}

}

test-while\_for\_nest.ll

; ModuleID = 'Liva'

%test = type <{ i32 }>

@tmp = private unnamed\_addr constant [5 x i8] c"i = \00"

@tmp.1 = private unnamed\_addr constant [2 x i8] c" \00"

@tmp.2 = private unnamed\_addr constant [5 x i8] c"j = \00"

@tmp.3 = private unnamed\_addr constant [2 x i8] c"\0A\00"

@tmp.4 = private unnamed\_addr constant [13 x i8] c"%s%d%s%s%d%s\00"

@tmp.5 = private unnamed\_addr constant [3 x i8] c"\0A\0A\00"

@tmp.6 = private unnamed\_addr constant [3 x i8] c"%s\00"

@tmp.7 = private unnamed\_addr constant [5 x i8] c"i = \00"

@tmp.8 = private unnamed\_addr constant [2 x i8] c" \00"

@tmp.9 = private unnamed\_addr constant [5 x i8] c"j = \00"

@tmp.10 = private unnamed\_addr constant [2 x i8] c"\0A\00"

@tmp.11 = private unnamed\_addr constant [13 x i8] c"%s%d%s%s%d%s\00"

declare i32 @printf(i8\*, ...)

declare i8\* @malloc(i32)

define i64\* @lookup(i32 %c\_index, i32 %f\_index) {

entry:

%tmp = alloca i64\*\*

%tmp1 = alloca i64\*, i32 0

%tmp2 = getelementptr i64\*\*, i64\*\*\* %tmp, i32 0

store i64\*\* %tmp1, i64\*\*\* %tmp2

ret i64\* null

}

define %test\* @test.constructor() {

entry:

%this = alloca %test

%tmp = call i8\* @malloc(i32 ptrtoint (i1\*\* getelementptr (i1\*, i1\*\* null, i32 1) to i32))

%tmp1 = bitcast i8\* %tmp to %test\*

%tmp2 = load %test, %test\* %tmp1

store %test %tmp2, %test\* %this

%.key = getelementptr inbounds %test, %test\* %this, i32 0, i32 0

store i32 0, i32\* %.key

ret %test\* %this

}

define i32 @main() {

entry:

%this = alloca %test

%tmp = call i8\* @malloc(i32 ptrtoint (i1\*\* getelementptr (i1\*, i1\*\* null, i32 1) to i32))

%tmp1 = bitcast i8\* %tmp to %test\*

%tmp2 = load %test, %test\* %tmp1

store %test %tmp2, %test\* %this

%.key = getelementptr inbounds %test, %test\* %this, i32 0, i32 0

store i32 0, i32\* %.key

%i = alloca i32

store i32 1, i32\* %i

%j = alloca i32

br label %cond

loop: ; preds = %cond

store i32 11, i32\* %j

store i32 11, i32\* %j

br label %cond5

loop3: ; preds = %cond5

%i7 = load i32, i32\* %i

%j8 = load i32, i32\* %j

%tmp9 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([13 x i8], [13 x i8]\* @tmp.4, i32 0, i32 0), i8\* getelementptr inbounds ([5 x i8], [5 x i8]\* @tmp, i32 0, i32 0), i32 %i7, i8\* getelementptr inbounds ([2 x i8], [2 x i8]\* @tmp.1, i32 0, i32 0), i8\* getelementptr inbounds ([5 x i8], [5 x i8]\* @tmp.2, i32 0, i32 0), i32 %j8, i8\* getelementptr inbounds ([2 x i8], [2 x i8]\* @tmp.3, i32 0, i32 0))

br label %step4

step4: ; preds = %loop3

%j10 = load i32, i32\* %j

%addtmp = add i32 %j10, 1

store i32 %addtmp, i32\* %j

br label %cond5

cond5: ; preds = %step4, %loop

%j11 = load i32, i32\* %j

%lesstmp = icmp slt i32 %j11, 13

br i1 %lesstmp, label %loop3, label %afterloop6

afterloop6: ; preds = %cond5

%i12 = load i32, i32\* %i

%addtmp13 = add i32 %i12, 1

store i32 %addtmp13, i32\* %i

br label %step

step: ; preds = %afterloop6

br label %cond

cond: ; preds = %step, %entry

%i14 = load i32, i32\* %i

%lesstmp15 = icmp slt i32 %i14, 10

br i1 %lesstmp15, label %loop, label %afterloop

afterloop: ; preds = %cond

%tmp16 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([3 x i8], [3 x i8]\* @tmp.6, i32 0, i32 0), i8\* getelementptr inbounds ([3 x i8], [3 x i8]\* @tmp.5, i32 0, i32 0))

store i32 1, i32\* %i

br label %cond19

loop17: ; preds = %cond19

store i32 11, i32\* %j

br label %cond23

loop21: ; preds = %cond23

%i25 = load i32, i32\* %i

%j26 = load i32, i32\* %j

%tmp27 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([13 x i8], [13 x i8]\* @tmp.11, i32 0, i32 0), i8\* getelementptr inbounds ([5 x i8], [5 x i8]\* @tmp.7, i32 0, i32 0), i32 %i25, i8\* getelementptr inbounds ([2 x i8], [2 x i8]\* @tmp.8, i32 0, i32 0), i8\* getelementptr inbounds ([5 x i8], [5 x i8]\* @tmp.9, i32 0, i32 0), i32 %j26, i8\* getelementptr inbounds ([2 x i8], [2 x i8]\* @tmp.10, i32 0, i32 0))

%j28 = load i32, i32\* %j

%addtmp29 = add i32 %j28, 1

store i32 %addtmp29, i32\* %j

br label %step22

step22: ; preds = %loop21

br label %cond23

cond23: ; preds = %step22, %loop17

%j30 = load i32, i32\* %j

%lesstmp31 = icmp slt i32 %j30, 13

br i1 %lesstmp31, label %loop21, label %afterloop24

afterloop24: ; preds = %cond23

br label %step18

step18: ; preds = %afterloop24

%i32 = load i32, i32\* %i

%addtmp33 = add i32 %i32, 1

store i32 %addtmp33, i32\* %i

br label %cond19

cond19: ; preds = %step18, %afterloop

%i34 = load i32, i32\* %i

%lesstmp35 = icmp slt i32 %i34, 10

br i1 %lesstmp35, label %loop17, label %afterloop20

afterloop20: ; preds = %cond19

ret i32 0

}

Test suites

fail-add.liva:

class arith {

void main()

{

int i;

i = "1" + 1;

print(i);

}

}

------------------------------------------------------------------------------------------------------

fail-array\_access.liva:

class test {

void main() {

char b = 'a';

float[] a = new float[10];

print(a[b]);

}

}

------------------------------------------------------------------------------------------------------

fail-array\_access2.liva:

class test {

void main() {

float[] a = new float[10];

print(a[1][1]);

}

}

------------------------------------------------------------------------------------------------------

fail-array\_init.liva:

class test {

void main() {

float[] a = new float[10.0];

}

}

------------------------------------------------------------------------------------------------------

fail-diff.liva:

class arith {

void main()

{

int i;

i = "1" - 1;

print(i);

}

}

------------------------------------------------------------------------------------------------------

fail-div.liva:

class arith {

void main()

{

int i;

i = "4" / 2;

print(i);

}

}

------------------------------------------------------------------------------------------------------

fail-equal1.liva:

class test {

void main(){

float i = 1.0;

if (i == 1.0) print(42);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

fail-equal2.liva:

class test {

void main(){

float i = "123";

if (i == true) print(42);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

fail-for1.liva:

class test {

void main(){

int i;

for (i = 0 ; i = 10 ; i = i + 1) {

print(i);

}

}

}

------------------------------------------------------------------------------------------------------

fail-function.liva:

class myclass{

int calc (int x, int y){

int z;

z = x + y;

return (z);

}

}

class test {

void main(){

int x = 9;

int y = 6;

int z;

class myclass obj = new myclass();

z = obj.ca\_lc(x, y);

print ("z=",z);

}

}

------------------------------------------------------------------------------------------------------

fail-function2.liva:

class myclass{

int calc (int x, int y){

int z;

z = x + y;

return (z);

}

}

class test {

void main(){

int x = 9;

int y = 6;

int z;

class myclass obj = new myclass();

z = obj.calc(x, x, y);

print ("z=",z);

}

}

------------------------------------------------------------------------------------------------------

fail-function3.liva:

class myclass{

int calc (int x, int y){

int z;

z = x + y;

return (z);

}

}

class test {

void main(){

int x = 9;

float y = 6.0;

float z;

class myclass obj = new myclass();

z = obj.calc(x, y);

print ("z=",z);

}

}

------------------------------------------------------------------------------------------------------

fail-hello.liva:

class test {

print ("Hello World!");

void main(){

}

}

------------------------------------------------------------------------------------------------------

fail-hello2.liva:

class test {

void main(){

int a ;

int b ;

a=1.1;

b=3;

print ("multiple ", "params!", "\n", a, "\n" ,b, "\n");

}

}

------------------------------------------------------------------------------------------------------

fail-if1.liva:

class test {

void main(){

if ("123") print(42);

}

}

------------------------------------------------------------------------------------------------------

fail-mod.liva:

class arith {

void main()

{

int i;

i = "4" % 3;

print(i);

}

}

------------------------------------------------------------------------------------------------------

fail-mul.liva:

class arith {

void main()

{

int i;

i = "15" \* 5;

print(i);

}

}

------------------------------------------------------------------------------------------------------

fail-not.liva:

class test {

void main(){

int i = 1;

boolean j;

j = !(i + 1);

}

}

------------------------------------------------------------------------------------------------------

fail-obj\_access.liva:

class test {

void main() {

int a;

a.amethod;

}

}

------------------------------------------------------------------------------------------------------

fail-obj\_access2.liva:

class test {

void main() {

int a;

(1+1).amethod;

}

}

------------------------------------------------------------------------------------------------------

fail-obj\_access3.liva:

class myclass{

int a;

constructor(int x){

this.a = x;

}

}

class test {

void main(){

class myclass obj = new myclass(10);

print ("b=",obj.b);

}

}

------------------------------------------------------------------------------------------------------

fail-sub.liva:

class test {

void main(){

int j;

j = -(true);

print(j);

}

}

------------------------------------------------------------------------------------------------------

fail-while1.liva:

class test {

void main() {

int i;

i = 5;

while (i = 1) {

print(i);

i = i - 1;

}

print(42);

}

}

------------------------------------------------------------------------------------------------------

test-add.liva:

class arith {

void main()

{

int i;

i = 1 + 1;

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-and.liva:

class test {

void main(){

int i = 1;

int j = 3;

if (i == 1 & j == 3)

{

print(i, " ", j, "\n");

}

}

}

------------------------------------------------------------------------------------------------------

test-arith.liva:

class arith {

void main()

{

int i;

i = 1 + 3 \* 4 % 7 - 4 / 2;

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-array.liva:

class test {

void main() {

float[] a = new float[10];

int[] b = new int[10];

int i;

a[0] = 1.0;

b[0] = 1;

for(i = 1; i < 10; i = i + 1)

{

a[i] = a[i - 1] + 1.0;

b[i] = b[i - 1] + 1;

}

for(i = 0; i < 10; i = i + 1)

print("a[",i,"]"," = ",a[i]," , ", "b[",i,"]"," = ",b[i], "\n");

}

}

------------------------------------------------------------------------------------------------------

test-array\_object.liva:

class calculator{

int g;

int addition(int x, int y){

this.g =9;

int z;

z = x + y;

return(z);

}

}

class test {

void main() {

class calculator c = new calculator();

class calculator[] a = new class calculator[10];

a[0] = c;

print(a[0].addition(1,1));

}

}

------------------------------------------------------------------------------------------------------

test-comments.liva:

class test {

void main(){

float i = 1.111;

/\*HAHAHAHA

/\* print(i);\*/

BOOOO!%$#$^%^&^%g)\_\_\*%^#@...

\*/

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-constructor.liva:

class myclass{

int a;

constructor(int x){

this.a = x;

}

}

class test {

void main(){

class myclass obj = new myclass(10);

print ("a=",obj.a);

}

}

------------------------------------------------------------------------------------------------------

test-diff.liva:

class arith {

void main()

{

float i;

i = 1.3 - 1.0;

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-div.liva:

class arith {

void main()

{

int i;

i = 4 / 2;

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-equal.liva:

class test {

void main(){

int i = 1;

if (i == 1) print(42);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

test-fib.liva:

class test {

int fib (int x){

int z;

if (x <2) z=1;

else z= this.fib(x-1) + this.fib(x-2);

return (z);

}

void main(){

int x;

int y;

int z;

int m;

x = 5;

y = 6;

z = this.fib (x);

print (z);

}

}

------------------------------------------------------------------------------------------------------

test-for1.liva:

class test {

void main(){

int i;

for (i = 0 ; i < 10 ; i = i + 1) {

print(i);

}

}

}

------------------------------------------------------------------------------------------------------

test-for\_nest.liva:

class test {

void main(){

int i;

int j;

for(i = 0; i < 10; i = i + 1)

for(j = 11; j < 13; j = j + 1)

print("i = ", i, " ", "j = ", j, "\n");

}

}

------------------------------------------------------------------------------------------------------

test-function.liva:

class myclass{

int calc (int x, int y){

int z;

z = x + y;

return (z);

}

}

class test {

void main(){

int x;

int y;

int z;

x = 9;

y = 6;

class myclass obj = new myclass();

z = obj.calc(x, y);

print ("z=",z);

}

}

------------------------------------------------------------------------------------------------------

test-gcd.liva:

class gcd {

void main(){

int x;

int y;

int z;

x = 66;

y = 98;

while(x != y){

if(x > y){

x = x - y;

}

else{

y = y - x;

}

}

print ("gcd=",x);

}

}

------------------------------------------------------------------------------------------------------

test-geq.liva:

class test {

void main(){

int i = 1;

int j = 1;

if (i >= j) print("yes");

else print("no");

}

}

------------------------------------------------------------------------------------------------------

test-gt.liva:

class test {

void main(){

int i = 4;

int j = 1;

if (i > j) print(4);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

test-hello.liva:

class test {

void main(){

print ("Hello World!");

}

}

------------------------------------------------------------------------------------------------------

test-hello2.liva:

class test {

void main(){

int a ;

int b ;

a=1;

b=3;

print ("multiple ", "params!", "\n", a, "\n" ,b, "\n");

}

}

------------------------------------------------------------------------------------------------------

test-if1.liva:

class test {

void main(){

print (100);

if (true) print(42);

else print(8);

print(17);

}

}

------------------------------------------------------------------------------------------------------

test-if\_nest.liva:

class test {

void main(){

int i = 1;

int j = 3;

if (true)

{

if(i == 1)

{

if(i < j)

{

print(j);

}

}

}

}

}

------------------------------------------------------------------------------------------------------

test-inheritance.liva:

class myclass{

int a;

constructor(int x){

this.a = x;

}

}

class subclass extends myclass{

constructor(int x){

this.a = x;

}

}

class test {

void main(){

class subclass obj = new subclass(10);

print ("a=",obj.a);

}

}

------------------------------------------------------------------------------------------------------

test-inheritance2.liva:

class calculator {

int add(int x, int y){

int z = x + y;

return(z);

}

}

class my\_calculator extends calculator{

}

class test {

void main(){

int x;

int y;

int z;

x = 66;

y = 98;

class my\_calculator obj = new my\_calculator();

z = obj.add(x,y);

print ("z=",z);

}

}

------------------------------------------------------------------------------------------------------

test-leq.liva:

class test {

void main(){

int i = 1;

int j = 1;

if (i <= j) print(4);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

test-lt.liva:

class test {

void main(){

int i = 3;

int j = 1;

if (i < j) print(4);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

test-mod.liva:

class arith {

void main()

{

int i;

i = 4 % 3;

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-mul.liva:

class arith {

void main()

{

int i;

i = 15 \* 5;

print(i);

}

}

------------------------------------------------------------------------------------------------------

test-nequal.liva:

class test {

void main(){

int i = 3;

if (i != 1) print(4);

else print(8);

}

}

------------------------------------------------------------------------------------------------------

test-not.liva:

class test {

void main(){

int i = 1;

int j = 3;

if (!(i != 1))

{

print("BOOO!");

}

}

}

------------------------------------------------------------------------------------------------------

test-obj.liva:

class myclass{

}

class test {

void main(){

class myclass obj = new myclass();

print ("obj\n");

}

}

------------------------------------------------------------------------------------------------------

test-or.liva:

class test {

void main(){

int i = 1;

int j = 3;

if (i != 1 | j == 3)

{

print(i, " ", j, "\n");

}

}

}

------------------------------------------------------------------------------------------------------

test-override.liva:

class myclass{

int a;

int calc (int x, int y){

int z;

z = x + y;

return (z);

}

}

class subclass extends myclass{

int calc (int x, int y){

int z;

z = x - y;

return (z);

}

}

class test {

void main(){

int x;

int y;

int z;

x = 9;

y = 6;

class subclass obj = new subclass();

z = obj.calc(x, y);

print ("z=",z);

}

}

------------------------------------------------------------------------------------------------------

test-sub.liva:

class test {

void main(){

int i = 3;

int j;

j = -(-(i) + i \* i);

print(j);

}

}

------------------------------------------------------------------------------------------------------

test-while1.liva:

class test {

void main() {

int i;

i = 5;

while (i > 0) {

print(i);

i = i - 1;

}

print(42);

}

}

------------------------------------------------------------------------------------------------------

test-while\_for\_nest.liva:

class test {

void main(){

int i = 1;

int j;

while(i < 10)

{

j = 11;

for(j = 11; j < 13; j = j + 1)

print("i = ", i, " ", "j = ", j, "\n");

i = i + 1;

}

print("\n\n");

for(i = 1; i < 10; i = i + 1)

{

j = 11;

while(j < 13)

{

print("i = ", i, " ", "j = ", j, "\n");

j = j + 1;

}

}

}

}

# 7 Lessons Learned

**Shanqi:**

Start early and do not surrender to difficult bugs. There is not much online community supports for OCaml Llvm API. I found this website helpful <https://llvm.moe/ocaml-3.7/Llvm.html> . However, this is not enough and you may want to refer to related past projects for some help. At the beginning, we all hated OCaml and found nothing worth in it. Now, I think that the most valuable thing about OCaml is the pattern matching. I cannot image how many switch statements would be used if we coded the compiler in Java.

First thing to do is to make sure that you understand the basic functions of the makefile and test shell in MicroC. They are useful during the development process. MicroC is a great example for beginners. After you understand it, you will know what is the basic steps of compiling a language. The parser file and the ast file are two highly related things. They only do one thing — reduce tokens according to the rules in parser to a single root element. In most cases, we call the root “program”.

Semantic check and code generator are the most time-consuming parts. All difficult bugs appear here. Some of your group members may focus on semantic check and others on code generator. Ocaml has the magic “let rec…” binding which makes the following bindings start with “and let…” a connected block. A good software architecture can save time. Before starting your project, you may figure out what is the general structure of your semantic check and code generator. We have blocks to process statements and expressions and other sub-blocks as helpers.

Start early and keep going.

**Jiafei:**

This is my first time I have been in New York and studied in Columbia University as a visiting student. Frankly speaking, I have gained a lot from this course. First of all, I gained a deep understanding towards how languages and their features were implemented. Before this class, I always complain about the language I use about why it could not be more convenient for programmers to use, now I feel appreciate to those who designed these languages and respect these languages from the bottom of my heart. Besides, this is also my first time I have studied and used functional programming language, which offer me a precious opportunity to broaden my programming thinking. Additionally, this project made me realize the importance of a general view of the whole project. I was anxious to get down to coding as soon as I thought I was able to do something for the project, however, after several hours working I just deleted all my codes because I found them useless. After talking with one of the group member, I realized that I was lack of a general view of the whole project, which resulted in my difficulty in getting start in detail. I believe the general thinking I learned from this project will be of great importance to my future study.

There are two suggestion I would like to offer to future group. First, make a plan early and get to start early. This is significant to not just avoid last-minute crunch but complete a satisfying project and get the most out of this course. Secondly, teamwork is the next thing I would like to emphasize. At first, we had group meeting once a week, but later we found it was not enough and we had meeting several times a week, and in fact this increased our efficiency a lot. Whenever you encounter with difficulties, you could ask your group members for help, because the group can provide an important resource for the whole project, including asking questions as well as deciding on the best plan for implementing language features.

**Zihan:**

The most important thing I learned from this project is how to use Github to work together. I've never done such a large project like LIVA on Github. I think Gitbub is really a clear and efficient tool for team coding.

In addition, having a consistent coding style and naming convention is really important for a project team. Block comments above function definitions are critical. Wear sunscreen. Variable and function names should tell the reader what it is used for.

Ocaml is also an interesting programming language for me. Functional programming is impressive.

**Yanan:**

1. It is always better to start the project as early as you can. Time is every tight for summer session and one small issue could cost you a whole day to figure out. It is very important to set up a suitable project plan and the deadline for each milestone. Follow up the timeline strictly and don’t delay the subtask implementation.
2. Spend enough time in designing a suitable architecture at early stages. This could save a lot of time to avoid modifying the architecture for new features added in.
3. It is a good way to divide the total project into subtasks and assign the subtasks to the group members. But make sure to make active communication with other group members and have each member some familiarity to each part, making it easier to make changes iteratively.
4. GitHub is a good software development tool and it could improve the efficiency if we learn how to better use it. However, there could also be a big waste of time if we have a wrong operation such as pushing before commit or forgetting about pushing. We have met some problems with pushing new stuff to GitHub remotely. Some updates are missing and we were not sure what happened for this problem.
5. Avoid waste time on doing the duplicated work! Discuss with the group members and know each other’s progress on the same task.
6. Trust our group members sincerely but also keep alert on every possible problem during the project development process.
7. I will not choose to compile our language to LLVM IR since there is so few reference on teaching how to use OCaml to write a LLVM IR code generator. It took us a lot of time to search for the good and useful references for this project.

**Kate:**

My most important lesson this term is to meet good teammates, for this was the first group project I participated in other than lab groups. To have good teammates, it is crucial to be the good teammate yourself, and despite my difficult situation, my teammates accepted me and helped me out greatly.

# Chapter 8 Appendix

scanner.mll

{

open Parser

let depth = ref 0

let unescape s =

Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)

}

let whitespace = [' ' '\t' '\r' '\n']

let ascii = ([' '-'!' '#'-'[' ']'-'~'])

let alpha = ['a'-'z' 'A'-'Z']

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

let id = alpha (alpha | digit | '\_')\*

let int = digit+

let float = (digit+) '.' (digit+)

let char = ''' ( ascii) '''

let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']

let escape\_char = ''' (escape) '''

let string = '"' ( (ascii | escape)\* as s) '"'

rule token = parse

whitespace { token lexbuf }

| "/\*" { incr depth; comment lexbuf }

(\* separator \*)

| '(' { LPAREN }

| ')' { RPAREN }

| '{' { LBRACE }

| '}' { RBRACE }

| ';' { SEMI }

| ',' { COMMA }

| '.' { DOT }

(\* Operators \*)

| '+' { PLUS }

| '-' { MINUS }

| '\*' { TIMES }

| '/' { DIVIDE }

| '%' { MODULO }

| '=' { ASSIGN }

| "==" { EQ }

| "!=" { NEQ }

| '<' { LT }

| "<=" { LEQ }

| ">" { GT }

| ">=" { GEQ }

| "&" { AND }

| "|" { OR }

| "!" { NOT }

| '[' { LBRACKET }

| ']' { RBRACKET }

(\* Branch Control \*)

| "if" { IF }

| "else" { ELSE }

| "for" { FOR }

| "while" { WHILE }

| "return" { RETURN }

(\* Data Types \*)

| "int" { INT }

| "float" { FLOAT }

| "boolean" { BOOLEAN }

| "char" { CHAR }

| "void" { VOID }

| "null" { NULL }

| "true" { TRUE }

| "false" { FALSE }

(\* Classes \*)

| "class" { CLASS }

| "constructor" { CONSTRUCTOR }

| "extends" { EXTENDS }

| "this" { THIS }

| "new" { NEW }

| int as lxm { INT\_LITERAL(int\_of\_string lxm) }

| float as lxm { FLOAT\_LITERAL(float\_of\_string lxm) }

| char as lxm { CHAR\_LITERAL(String.get lxm 1) }

| escape\_char as lxm { CHAR\_LITERAL(String.get (unescape lxm) 1) }

| string { STRING\_LITERAL(unescape s) }

| id as lxm { ID(lxm) }

| eof { EOF }

| \_ as illegal { raise (Failure("illegal character " ^ Char.escaped illegal )) }

and comment = parse

"\*/" { decr depth; if !depth > 0 then comment lexbuf else token lexbuf }

| "/\*" { incr depth; comment lexbuf }

| \_ { comment lexbuf }

ast.ml

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Not | Or | Mod

type primitive = Int\_t | Float\_t | Void\_t | Bool\_t | Char\_t | String\_t | Objecttype of string | ConstructorType | Null\_t

type datatype = Arraytype of primitive \* int | Datatype of primitive |Any

type extends = NoParent | Parent of string

type fname = Constructor | FName of string

type formal = Formal of datatype \* string | Many of datatype

type expr =

Int\_Lit of int

| Boolean\_Lit of bool

| Float\_Lit of float

| String\_Lit of string

| Char\_Lit of char

| This (\*need to be implemented\*)

| Id of string

| Binop of expr \* op \* expr

| Assign of expr \* expr

| Noexpr

| ArrayCreate of datatype \* expr list

| ArrayAccess of expr \* expr list

| ObjAccess of expr \* expr

| Call of string \* expr list

| ObjectCreate of string \* expr list

| Unop of op \* expr

| Null

type stmt =

Block of stmt list

| Expr of expr

| Return of expr

| If of expr \* stmt \* stmt

| For of expr \* expr \* expr \* stmt

| While of expr \* stmt

| Local of datatype \* string \* expr

type field = Field of datatype \* string

type func\_decl = {

fname : fname;

returnType : datatype;

formals : formal list;

body : stmt list;

overrides : bool;

rootcname : string option;

}

type cbody = {

fields : field list;

constructors : func\_decl list;

methods : func\_decl list;

}

type class\_decl = {

cname : string;

extends : extends;

cbody: cbody;

}

type program = Program of class\_decl list

(\*get function name,tell constructor from ordinary functions\*)

let string\_of\_fname = function

Constructor -> "constructor"

| FName(s) -> s

let string\_of\_primitive = function (\*primitive type\*)

Int\_t -> "int"

| Float\_t -> "float"

| Void\_t -> "void"

| Bool\_t -> "bool"

| Char\_t -> "char"

| Objecttype(s) -> "class" ^ " " ^ s

| ConstructorType -> "constructor"

| Null\_t -> "null"

| String\_t -> "String"

let string\_of\_object = function

Datatype(Objecttype(s)) -> s

| \_ -> ""

let rec print\_brackets = function

1 -> "[]"

| a -> "[]" ^ print\_brackets (a - 1)

let string\_of\_expr e = "remain to be completed"

let string\_of\_datatype = function (\*datatype\*)

Arraytype(p, i) -> (string\_of\_primitive p) ^ (print\_brackets i)

| Datatype(p) -> (string\_of\_primitive p)

| Any -> "Any"

let string\_of\_op = function(\*operator\*)

Add -> "+"

| Sub -> "-"

| Mult -> "\*"

| Div -> "/"

| Equal -> "=="

| Neq -> "!="

| Less -> "<"

| Leq -> "<="

| Greater -> ">"

| Geq -> ">="

| And -> "and"

| Not -> "not"

| Or -> "or"

| Mod -> "%"

let string\_of\_boolean b = match b with

true -> "true"

| false -> "false"

parser.mly

%{ open Ast %}

%token CLASS EXTENDS CONSTRUCTOR DOT THIS

%token INT FLOAT BOOLEAN CHAR VOID NULL TRUE FALSE

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA

%token AND NOT OR PLUS MINUS TIMES DIVIDE ASSIGN MODULO

%token EQ NEQ LT LEQ GT GEQ

%token RETURN IF ELSE FOR WHILE NEW

%token <int> INT\_LITERAL

%token <float> FLOAT\_LITERAL

%token <string> STRING\_LITERAL

%token <string> ID

%token <char> CHAR\_LITERAL

%token EOF

%nonassoc NOELSE

%nonassoc ELSE

%right ASSIGN

%left AND OR

%left EQ NEQ

%left LT GT LEQ GEQ

%left PLUS MINUS

%left TIMES DIVIDE MODULO

%right NOT

%right RBRACKET

%left LBRACKET

%right DOT

%start program

%type <Ast.program> program

%%

program:

cdecls EOF { Program($1) }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CLASSES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

cdecls:

cdecl\_list { List.rev $1 }

cdecl\_list:

cdecl { [$1] }

| cdecl\_list cdecl { $2::$1 }

cdecl:

CLASS ID LBRACE cbody RBRACE { {

cname = $2;

extends = NoParent;

cbody = $4

} }

| CLASS ID EXTENDS ID LBRACE cbody RBRACE { {

cname = $2;

extends = Parent($4);

cbody = $6

} }

cbody:

/\* nothing \*/ { {

fields = [];

constructors = [];

methods = [];

} }

| cbody field { {

fields = $2 :: $1.fields;

constructors = $1.constructors;

methods = $1.methods;

} }

| cbody constructor { {

fields = $1.fields;

constructors = $2 :: $1.constructors;

methods = $1.methods;

} }

| cbody fdecl { {

fields = $1.fields;

constructors = $1.constructors;

methods = $2 :: $1.methods;

} }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CONSTRUCTORS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

constructor:

CONSTRUCTOR LPAREN formals\_opt RPAREN LBRACE stmt\_list RBRACE {

{

fname = Constructor;

returnType = Datatype(ConstructorType);

formals = $3;

body = List.rev $6;

overrides = false;

rootcname = None;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FIELDS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* public UserObj name; \*/

field:

datatype ID SEMI { Field($1, $2) }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

METHODS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

fname:

ID { $1 }

fdecl:

datatype fname LPAREN formals\_opt RPAREN LBRACE stmt\_list RBRACE

{

{

fname = FName($2);

returnType = $1;

formals = $4;

body = List.rev $7;

overrides = false;

rootcname = None;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FORMALS/PARAMETERS & VARIABLES & ACTUALS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

formals\_opt:

/\* nothing \*/ { [] }

| formal\_list { List.rev $1 }

formal\_list:

formal { [$1] }

| formal\_list COMMA formal { $3 :: $1 }

formal:

datatype ID { Formal($1, $2) }

actuals\_opt:

/\* nothing \*/ { [] }

| actuals\_list { List.rev $1 }

actuals\_list:

expr { [$1] }

| actuals\_list COMMA expr { $3 :: $1 }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DATATYPES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

primitive:

INT { Int\_t }

| FLOAT { Float\_t }

| CHAR { Char\_t }

| BOOLEAN { Bool\_t }

| VOID { Void\_t }

name:

CLASS ID { Objecttype($2) }

type\_tag:

primitive { $1 }

| name { $1 }

array\_type:

type\_tag LBRACKET brackets RBRACKET { Arraytype($1, $3) }

datatype:

type\_tag { Datatype($1) }

| array\_type { $1 }

brackets:

/\* nothing \*/ { 1 }

| brackets RBRACKET LBRACKET { $1 + 1 }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EXPRESSIONS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

stmt\_list:

/\* nothing \*/ { [] }

| stmt\_list stmt { $2 :: $1 }

stmt:

expr SEMI { Expr($1) }

| RETURN expr SEMI { Return($2) }

| RETURN SEMI { Return(Noexpr) }

| LBRACE stmt\_list RBRACE { Block(List.rev $2) }

| IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([Expr(Noexpr)])) }

| IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }

| FOR LPAREN expr\_opt SEMI expr\_opt SEMI expr\_opt RPAREN stmt

{ For($3, $5, $7, $9) }

| WHILE LPAREN expr RPAREN stmt { While($3, $5) }

| datatype ID SEMI { Local($1, $2, Noexpr) }

| datatype ID ASSIGN expr SEMI { Local($1, $2, $4) }

expr\_opt:

/\* nothing \*/ { Noexpr }

| expr { $1 }

expr:

literals { $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 MODULO expr { Binop($1, Mod, $3)}

| NOT expr { Unop (Not, $2) }

| expr OR expr { Binop($1, Or, $3) }

| expr DOT expr { ObjAccess($1, $3) }

| expr ASSIGN expr { Assign($1, $3) }

| MINUS expr { Unop (Sub, $2) }

| ID LPAREN actuals\_opt RPAREN { Call($1, $3) }

| NEW ID LPAREN actuals\_opt RPAREN { ObjectCreate($2, $4) }

| NEW type\_tag bracket\_args RBRACKET { ArrayCreate(Datatype($2), List.rev $3) }

| expr bracket\_args RBRACKET { ArrayAccess($1, List.rev $2) }

| LPAREN expr RPAREN { $2 }

bracket\_args:

LBRACKET expr { [$2] }

| bracket\_args RBRACKET LBRACKET expr { $4 :: $1 }

literals:

INT\_LITERAL { Int\_Lit($1) }

| FLOAT\_LITERAL { Float\_Lit($1) }

| TRUE { Boolean\_Lit(true) }

| FALSE { Boolean\_Lit(false) }

| STRING\_LITERAL { String\_Lit($1) }

| CHAR\_LITERAL { Char\_Lit($1) }

| THIS { This }

| ID { Id($1) }

| NULL { Null }

sast.ml

open Ast

type sexpr =

SInt\_Lit of int

| SBoolean\_Lit of bool

| SFloat\_Lit of float

| SString\_Lit of string

| SChar\_Lit of char

| SId of string \* datatype

| SBinop of sexpr \* op \* sexpr \* datatype

| SAssign of sexpr \* sexpr \* datatype

| SNoexpr

| SArrayCreate of datatype \* sexpr list \* datatype

| SArrayAccess of sexpr \* sexpr list \* datatype

| SObjAccess of sexpr \* sexpr \* datatype

| SCall of string \* sexpr list \* datatype \* int

| SObjectCreate of string \* sexpr list \* datatype

| SArrayElements of sexpr list \* datatype

| SUnop of op \* sexpr \* datatype

| SNull

type sstmt =

SBlock of sstmt list

| SExpr of sexpr \* datatype

| SReturn of sexpr \* datatype

| SIf of sexpr \* sstmt \* sstmt

| SFor of sexpr \* sexpr \* sexpr \* sstmt

| SWhile of sexpr \* sstmt

| SLocal of datatype \* string \* sexpr

type func\_type = User | Reserved

type sfunc\_decl = {

sfname : fname;

sreturnType : datatype;

sformals : formal list;

sbody : sstmt list;

functype : func\_type;

source : string;

overrides : bool;

}

type sclass\_decl = {

scname : string;

sfields : field list;

sfuncs: sfunc\_decl list;

}

(\* Class Declarations | All method declarations | Main entry method \*)

type sprogram = {

classes : sclass\_decl list;

functions : sfunc\_decl list;

main : sfunc\_decl;

reserved : sfunc\_decl list;

}

semant.ml

(\* Semantic checking for the Liva compiler \*)

open Ast

open Sast

(\* Semantic checking of a program. Returns Sast if successful,

throws an exception if something is wrong. \*)

(\*global variables and helpfer functions\*)

module StringMap = Map.Make(String)

module StringSet = Set.Make (String)

module SS = Set.Make(

struct

let compare = Pervasives.compare

type t = datatype

end )

type classMap = {

fieldMap : Ast.field StringMap.t;

functionMap : Ast.func\_decl StringMap.t;

constructorMap : Ast.func\_decl StringMap.t;

builtFuncMap : sfunc\_decl StringMap.t;

cdecl : Ast.class\_decl;

}

type env ={

envClassMaps: classMap StringMap.t;

envName: string;

envClassMap: classMap;

envLocals: datatype StringMap.t;

envParams: Ast.formal StringMap.t;

envReturnType:datatype;

envInFor: bool;

envInWhile: bool;

envBuiltIn:sfunc\_decl list;

}

let updateEnv env envName =

{

envClassMaps = env.envClassMaps;

envName = envName;

envClassMap = env.envClassMap;

envLocals = env.envLocals;

envParams = env.envParams;

envReturnType = env.envReturnType;

envInFor = env.envInFor;

envInWhile = env.envInWhile;

envBuiltIn = env.envBuiltIn;

}

let strucIndexes: (string, int) Hashtbl.t = Hashtbl.create 10

let inheritanceRelation:(string, string list) Hashtbl.t = Hashtbl.create 10

let createStructIndexes cdecls=

let classHandler index cdecl=

Hashtbl.add strucIndexes cdecl.cname index in

List.iteri classHandler cdecls

let defaultC =

{

fname = Ast.Constructor;

returnType = Datatype(ConstructorType);

formals = [];

body = [];

overrides = false;

rootcname = None;

}

let getName cname fdecl = (\*get the name of function,cname.constructor-> constructor / cname.xxx-> normal\_function / main\*)

let name = string\_of\_fname fdecl.fname (\*tell constructor from normal function\*)

in

match name with

"main" -> "main"

| \_ -> cname ^ "." ^ name

let typOFSexpr = function(\*get the type of sexpression\*)

SInt\_Lit(\_) -> Datatype(Int\_t)

| SBoolean\_Lit(\_) -> Datatype(Bool\_t)

| SFloat\_Lit(\_) -> Datatype(Float\_t)

| SString\_Lit(\_) -> Arraytype(Char\_t, 1)

| SChar\_Lit(\_) -> Datatype(Char\_t)

| SId(\_, d) -> d

| SBinop(\_, \_, \_, d) -> d

| SAssign(\_, \_, d) -> d

| SNoexpr -> Datatype(Void\_t)

| SArrayCreate(\_, \_, d) -> d

| SArrayAccess(\_, \_, d) -> d

| SObjAccess(\_, \_, d) -> d

| SCall(\_, \_, d,\_) -> d

| SObjectCreate(\_, \_, d) -> d

| SArrayElements(\_, d) -> d

| SUnop(\_, \_, d) -> d

| SNull -> Datatype(Null\_t)

(\*\*\*\* Entry point for translating Ast to Sast \*\*\*\*\*)

let check program =

(\* add reserved built-in functions\*)

let storeBuiltinFunctions =

let i32\_t = Datatype(Int\_t) and

void\_t = Datatype(Void\_t) and

str\_t = Arraytype( Char\_t, 1)

in

let mf t s = Formal(t, s)

in

let builtinStub fname returnType formals =

{

sfname = FName (fname);

sreturnType = returnType;

sformals= formals;

functype= Sast.Reserved;

sbody=[];

overrides = false;

source= "NA"

}

in

let builtinFunctions =[

builtinStub "print" (void\_t) ([Many(Any)]);

builtinStub "malloc" (str\_t) ([mf i32\_t "size"]);

builtinStub "cast" (Any) ([mf Any "in"]);

]

in builtinFunctions

in

let builtinFunctions = storeBuiltinFunctions

in

(\* create class maps\*)

let getConstructorName cname fdecl =

let params = List.fold\_left

(fun s f -> match f with

Formal(t, \_) -> s ^ "." ^ string\_of\_datatype t

| \_ -> "" ) "" fdecl.formals

in

let name = string\_of\_fname fdecl.fname

in cname ^ "." ^ name ^ params

in

let mappingClass builtinFunctions cdecls =

let builtFuncMap =

List.fold\_left (fun mp sfunc -> StringMap.add (string\_of\_fname sfunc.sfname) sfunc mp) StringMap.empty builtinFunctions

in

let assistant mp cdecl =

let fieldpart mp = function Field(d,n) ->

if (StringMap.mem n mp)

then raise (Failure ("Duplicated Field: " ^ n))

else (StringMap.add n (Field(d, n)) mp)

in

let constructorpart condecl =

if List.length condecl > 1

then raise (Failure ("Duplicated Constructor"))

else if List.length condecl = 0 (\*default constructor\*)

then StringMap.add (getConstructorName cdecl.cname defaultC) defaultC StringMap.empty

else

StringMap.add (getConstructorName cdecl.cname (List.hd condecl)) (List.hd condecl) StringMap.empty

in

let funcpart m fdecl =

let funname = getName cdecl.cname fdecl

in

if (StringMap.mem funname mp)

then raise (Failure ("Duplicated Function: " ^ funname))

else

let strfunname = string\_of\_fname fdecl.fname

in

if (StringMap.mem strfunname builtFuncMap)

then raise (Failure ("Cannot use the reserved buit-in function name: " ^ strfunname))

else (StringMap.add (getName cdecl.cname fdecl) fdecl m)

in

(if (StringMap.mem cdecl.cname mp)

then raise (Failure ("Duplicated class name: " ^ cdecl.cname))

else

StringMap.add cdecl.cname

{

fieldMap = List.fold\_left fieldpart StringMap.empty cdecl.cbody.fields;

constructorMap = constructorpart cdecl.cbody.constructors;

functionMap = List.fold\_left funcpart StringMap.empty cdecl.cbody.methods;

builtFuncMap = builtFuncMap;

cdecl = cdecl

} mp)

in List.fold\_left assistant StringMap.empty cdecls

in

match program with

Program (classes) -> ignore (createStructIndexes classes);

let classMaps = mappingClass builtinFunctions classes

in

(\* convert statement in Ast to statement in Sast\*)

let rec exprToSexpr env = function

Int\_Lit i -> SInt\_Lit(i), env

| Boolean\_Lit b -> SBoolean\_Lit(b), env

| Float\_Lit f -> SFloat\_Lit(f), env

| String\_Lit s -> SString\_Lit(s), env

| Char\_Lit c -> SChar\_Lit(c), env

| This -> SId("this", Datatype(Objecttype(env.envName))), env

| Id s -> SId(s, getIDType env s), env

| Null -> SNull, env

| Noexpr -> SNoexpr, env

| ObjAccess(e1, e2) -> checkObjAccess env e1 e2, env

| ObjectCreate(s, el) -> checkConstructor env s el, env

| Call(s, el) -> checkCallType env s el, env

| ArrayCreate(d, el) -> checkArrayInitialize env d el, env

| ArrayAccess(e, el) -> checkArrayAccess env e el, env

| Assign(e1, e2) -> checkAssign env e1 e2, env

| Unop(op, e) -> checkUnop env op e, env

| Binop(e1, op, e2) -> checkBinop env e1 op e2, env

and exprsToSexprs env el = (\*convert expression list to sexpression list\*)

let envref = ref env

in

let rec assistant = function

h :: t -> let newh, env = exprToSexpr !envref h

in(

envref := env;

newh :: (assistant t))

| [] -> []

in

(assistant el), !envref

and getIDType env s =

try

StringMap.find s env.envLocals

with

| Not\_found -> try let formal = StringMap.find s env.envParams

in (function Formal(t, \_) -> t

| Many t -> t) formal

with | Not\_found -> raise (Failure ("ID is undefined: " ^ s))

and checkArrayInitialize env d el =

let arraySize = List.length el(\*get the dimention of array\*)

in

let checkIndexType e = (\*check whether the type of index is int\*)

let sexpr, \_ = exprToSexpr env e (\*convert expression to sexpression\*)

in

let typ= typOFSexpr sexpr (\*get the type of sexpression\*)

in

if typ = Datatype(Int\_t)

then sexpr

else raise (Failure ("Invalid index type for array initialization: " ^ string\_of\_datatype typ))

in

let checkTyp = function(\*check whether the type can be array type\*)

Datatype(x) -> Arraytype(x, arraySize)

| \_ as t -> raise (Failure ("Invalid array type: " ^ (string\_of\_datatype t)))

in

let typ = checkTyp d

in

let sel = List.map checkIndexType el

in

SArrayCreate(d, sel, typ)

and checkArrayAccess env e el =

let arraySize = List.length el (\*get the size of array\*)

in

let checkIndexType arg = (\*check whether the type of index is int\*)

let sexpr, \_ = exprToSexpr env arg (\*convert expression to sexpression\*)

in

let typ = typOFSexpr sexpr (\*get the type of sexpression\*)

in

if typ = Datatype(Int\_t)

then sexpr

else raise (Failure ("Invalid index type for array access: " ^ string\_of\_datatype typ))

in

let se, \_ = exprToSexpr env e (\*convert expression to sexpression\*)

in

let typ = typOFSexpr se (\*get the type of sexpression\*)

in

let checkArraySize num = function

Arraytype(t, n) -> if num = n

then Datatype(t)

else raise (Failure ("Invalid demention for array access: " ^ (string\_of\_int num) ^ " > " ^ (string\_of\_int n)))

| \_ as t -> raise (Failure ("Invalid type for array access: " ^ (string\_of\_datatype t)))

in

let typ = checkArraySize arraySize typ

in

let sel = List.map checkIndexType el

in SArrayAccess(se, sel, typ)

and checkObjAccess env lhs rhs =

let checkLHS = function(\*check the expression before ‘.’ and get sexpression\*)

This -> SId("this", Datatype(Objecttype(env.envName)))

| Id s -> SId(s, getIDType env s)

| ArrayAccess(e, el) -> checkArrayAccess env e el

| \_ -> raise (Failure ("LHS of object access must be an instance of certain class"))

in

let getCname lhsTyp = match lhsTyp with (\*get the type of the expression before ‘.’, i.e. class name\*)

Datatype(Objecttype(name)) -> name

| \_ as d -> raise (Failure ("Object access must have ObjectType: " ^ string\_of\_datatype d))

in

let rec checkRHS (env) lhsTyp=

let classname = getCname lhsTyp (\*get the class name\*)

in

let search\_classfield env (id) cname=

let cmap = StringMap.find cname env.envClassMaps (\*get the class map of current class\*)

in

let match\_field = function Field(d, \_) -> d (\*get datatype of the expression after ‘.’\*)

in

try match\_field (StringMap.find id cmap.fieldMap)

with | Not\_found -> raise (Failure ("Unknown field identifier for class: " ^ id ^ " -> " ^ cname))

in

function

Id s -> SId(s, (search\_classfield env s classname )), env (\* Check fields\*)

| Call(fname, el) -> let env = updateEnv env classname (\* Check functions\*)

in checkCallType env fname el, env

| \_ as e -> raise (Failure ("Invalid object access: " ^ string\_of\_expr e))

in

let slhs= checkLHS lhs in

let slhsTyp = typOFSexpr slhs in

let lcname = getCname slhsTyp in

let lhsenv = updateEnv env lcname in

let srhs, \_ = checkRHS lhsenv slhsTyp (\*env\*) rhs in

let srhsTyp = typOFSexpr srhs in

SObjAccess(slhs, srhs, srhsTyp )

and checkCallType env fname el =

let sel, env = exprsToSexprs env el(\*convert expression list to sexpression list\*)

in

let cmap = try StringMap.find env.envName env.envClassMaps (\*check whether the class has been defined\*)

with | Not\_found -> raise (Failure ("Undefined class: " ^ env.envName))

in(\*check type\*)

let check\_pa\_onebyone formal param = (\*check parameter according to type\*)

let ftyp = match formal with

Formal(d, \_) -> d

| \_ -> Datatype(Void\_t)

in

let ptyp = typOFSexpr param(\*get the type of actual parameter\*)

in

if ftyp = ptyp

then param

else raise (Failure ("Incompatible type for function: " ^ fname ^ " " ^ string\_of\_datatype ptyp ^ " -> " ^ string\_of\_datatype ftyp))

in

let getIndex func funcName =

let cdecl = cmap.cdecl in

let fns = List.rev cdecl.cbody.methods in

let rec find x lst =

match lst with

| [] -> raise (Failure ("Could not find " ^ fname))

| fdecl :: t ->

let searchName = (getName env.envName func) in

if x = searchName then 0

else if searchName = "main" then find x t

else 1 + find x t

in

find funcName fns

in

let checkParams formals params = match formals, params with (\*check parameter according to amount\*)

[Many(Any)], \_ -> params

| [], [] -> []

| \_ -> if List.length formals <> List.length params

then raise (Failure ("Incorrect argument number for function: " ^ fname))

else List.map2 check\_pa\_onebyone formals sel

in

try

let func = StringMap.find fname cmap.builtFuncMap

in

let actuals = checkParams func.sformals sel

in SCall(fname, actuals, func.sreturnType,0)

with | Not\_found -> let sfname = env.envName ^ "." ^ fname

in

try let f = StringMap.find sfname cmap.functionMap

in

let actuals = checkParams f.formals sel in

let index = getIndex f sfname in

SCall(sfname, actuals, f.returnType, index)

with | Not\_found -> raise (Failure ("Function is not found: " ^ sfname))

and checkConstructor env s el =

let sel, env = exprsToSexprs env el

in

let params = List.fold\_left

(fun s e -> s ^ "." ^ (string\_of\_datatype (typOFSexpr e))) "" sel

in

let constructorName = s ^ "." ^ "constructor" ^ params

in

let objectTyp = Datatype(Objecttype(s)) in

SObjectCreate(constructorName, sel, objectTyp)

and checkAssign env e1 e2 =

let se1, env = exprToSexpr env e1(\*convert expression to sexpression\*)

in

let se2, env = exprToSexpr env e2

in

let type1 = typOFSexpr se1(\*get the type of sexpression\*)

in

let type2 = typOFSexpr se2

in

match (type1, se2) with

Datatype(Objecttype(\_)), SNull -> SAssign(se1, se2, type1)

| \_ ->

match type1, type2 with

Datatype(Objecttype(d)), Datatype(Objecttype(t)) ->

if d = t

then SAssign(se1, se2, type1)

else raise (Failure ("Assignment types are mismatched: " ^ string\_of\_datatype type1 ^ " <-> " ^ string\_of\_datatype type2))

| \_ -> if type1 = type2

then SAssign(se1, se2, type1)

else raise (Failure ("Assignment types are mismatched: " ^ string\_of\_datatype type1 ^ " <-> " ^ string\_of\_datatype type2))

and checkUnop env op e =

let checkNum t = function(\*operator for number\*)

Sub -> t

| \_ as o -> raise (Failure ("Invalid unary operation: " ^ string\_of\_op o))

in

let checkBool = function(\*operator for bool\*)

Not -> Datatype(Bool\_t)

| \_ as o -> raise (Failure ("Invalid unary operation: " ^ string\_of\_op o))

in

let se, env = exprToSexpr env e (\*convert expression to sexpression\*)

in

let st = typOFSexpr se (\*get the type of sexpression\*)

in

match st with (\*check the type of operand\*)

Datatype(Int\_t)

| Datatype(Float\_t) -> SUnop(op, se, checkNum st op)

| Datatype(Bool\_t) -> SUnop(op, se, checkBool op)

| \_ as o -> raise (Failure ("Invalid operant type for unary operation: " ^ string\_of\_datatype o))

and checkBinop env e1 op e2 =

let getequal type1 type2 se1 se2 op =

if (type1 = Datatype(Float\_t) || type2 = Datatype(Float\_t)) (\*unqualified types\*)

then raise (Failure ("Equality operation is not supported for Float types"))

else

match type1, type2 with (\*qualified types\*)

Datatype(Objecttype(\_)), Datatype(Null\_t)

| Datatype(Null\_t), Datatype(Objecttype(\_)) -> SBinop(se1, op, se2, Datatype(Bool\_t))

| \_ -> if type1 = type2

then SBinop(se1, op, se2, Datatype(Bool\_t))

else raise (Failure ("Invalid equality operator for these types: " ^ (string\_of\_datatype type1) ^ " <-> " ^ (string\_of\_datatype type2)))

in

let getlogic type1 type2 se1 se2 op =(\*check operants and conver to sbinop\*)

match type1, type2 with

Datatype(Bool\_t), Datatype(Bool\_t) -> SBinop(se1, op, se2, Datatype(Bool\_t))

| \_ -> raise (Failure ("Invalid type for logical operator" ^ (string\_of\_datatype type1) ^ "<->" ^ (string\_of\_datatype type2)))

in

let getcomp type1 type2 se1 se2 op =

match type1, type2 with

Datatype(Int\_t), Datatype(Float\_t)

| Datatype(Float\_t), Datatype(Int\_t) -> SBinop(se1, op, se2, Datatype(Bool\_t))

| \_ -> if type1 = type2

then SBinop(se1, op, se2, Datatype(Bool\_t))

else raise (Failure ("Invalid type for comparison operator: " ^ (string\_of\_datatype type1) ^ "<->" ^ (string\_of\_datatype type2)))

in

let getarith type1 type2 se1 se2 op =

match type1, type2 with (\*qualified combination of operant type\*)

Datatype(Int\_t), Datatype(Float\_t)

| Datatype(Float\_t), Datatype(Int\_t)

| Datatype(Float\_t), Datatype(Float\_t) -> SBinop(se1, op, se2, Datatype(Float\_t))

| Datatype(Int\_t), Datatype(Int\_t) -> SBinop(se1, op, se2, Datatype(Int\_t))

| \_ -> raise (Failure ("Invalid type for arithmetic operator: " ^ (string\_of\_datatype type1) ^ "<->" ^ (string\_of\_datatype type2)))

in

let se1, env = exprToSexpr env e1 (\*convert expression to sexpression\*)

in

let se2, env = exprToSexpr env e2 (\*convert expression to sexpression\*)

in

let type1 = typOFSexpr se1(\*get the type of sexpression\*)

in

let type2 = typOFSexpr se2(\*get the type of sexpression\*)

in

match op with(\*check and convert binopexpression according to binopexpression type\*)

Equal | Neq -> getequal type1 type2 se1 se2 op

| And | Or -> getlogic type1 type2 se1 se2 op

| Less | Leq | Greater | Geq -> getcomp type1 type2 se1 se2 op

| Add | Mult | Sub | Div | Mod -> getarith type1 type2 se1 se2 op

| \_ -> raise (Failure ("Invalid binop operator: " ^ (string\_of\_op op)))

in

let rec convertStmtsToSstmts env stmts =

let envref = ref(env) in

let rec iter = function

h::t -> let newh, newenv = checkStmt !envref h in

(envref := newenv; newh::(iter t))

| [] -> []

in

let sstmts = (iter stmts), !envref in

sstmts

and checkExprStmt e env =

let se, env = exprToSexpr env e

in

let typ = typOFSexpr se

in

SExpr(se, typ), env

and checkIfStmt e s1 s2 env =

let se, \_ = exprToSexpr env e and

ifbody, \_ = checkStmt env s1 and

elsebody, \_ = checkStmt env s2

in

let typ = typOFSexpr se

in

match typ with

Datatype(Bool\_t) -> SIf(se, ifbody, elsebody), env

| \_ -> raise (Failure("invalid if type"))

and checkForStmt e1 e2 e3 s env =

let se1, \_ = exprToSexpr env e1 and

se2, \_ = exprToSexpr env e2 and

se3, \_ = exprToSexpr env e3 and

forBodyStmt, env = checkStmt env s

in

let cond = typOFSexpr se2

in

match cond with

Datatype(Bool\_t) -> SFor(se1, se2, se3, forBodyStmt), env

| Datatype(Void\_t) -> SFor(se1, se2, se3, forBodyStmt), env

| \_ -> raise (Failure("Invalid for statement type"))

and checkWhileStmt e s env =

let se, \_ = exprToSexpr env e and

sstmt, \_ = checkStmt env s

in

let typ = typOFSexpr se

in

match typ with

Datatype(Bool\_t) -> SWhile(se, sstmt), env

| Datatype(Void\_t) -> SWhile(se, sstmt), env

| \_ -> raise (Failure("Invalid while Statement Type"))

and checkSblock sl env = match sl with

[] -> SBlock([SExpr(SNoexpr, Datatype(Void\_t))]), env

| \_ -> let sl, \_ = convertStmtsToSstmts env sl

in SBlock(sl), env

and checkReturn e env =

let se, env = exprToSexpr env e in

let typ = typOFSexpr se

in

match typ, env.envReturnType with

Datatype(Null\_t), Datatype(Objecttype(\_)) -> SReturn(se, typ), env

| \_ ->

if typ = env.envReturnType

then SReturn(se, typ), env

else raise (Failure ("Return type is mismatched!"))

and checkLocal d s e env =

if StringMap.mem s env.envLocals

then raise (Failure ("Duplicate Local variable defined: " ^ s))

else

let se, env = exprToSexpr env e

in

let typ = typOFSexpr se

in

let update\_env = {

envClassMaps = env.envClassMaps;

envName = env.envName;

envClassMap = env.envClassMap;

envLocals = StringMap.add s d env.envLocals; (\* add new locals \*)

envParams = env.envParams;

envReturnType = env.envReturnType;

envInFor = env.envInFor;

envInWhile = env.envInWhile;

envBuiltIn = env.envBuiltIn;

}

in

if typ = Datatype(Void\_t) || typ = Datatype(Null\_t) || typ = d

then

match d with

Datatype(Objecttype(x)) ->

if not (StringMap.mem (Ast.string\_of\_object d) env.envClassMaps)

then raise (Failure ("Undefined Class: " ^ string\_of\_object d ))

else

let local = SLocal(d, s, se)

in local, update\_env

| \_ -> SLocal(d, s, se), update\_env

else

raise (Failure("Local assignment type mismatch: " ^ Ast.string\_of\_datatype d ^ " <-> " ^ Ast.string\_of\_datatype typ))

and checkStmt env = function

Expr e -> checkExprStmt e env

| If(e, s1, s2) -> checkIfStmt e s1 s2 env

| While(e, s) -> checkWhileStmt e s env

| For(e1, e2, e3, e4) -> checkForStmt e1 e2 e3 e4 env

| Block sl -> checkSblock sl env

| Return e -> checkReturn e env

| Local(d, s, e) -> checkLocal d s e env

in

(\* about inheritance\*)

let rec manageInheritance classes classMaps =

let inheritanceMap = getInheritanceMap classes classMaps in(\*forest: father class name -> [son class name]\*)

let allClassesM = getClassesForM classes inheritanceMap in(\*all classes including inherited classes which have been dealed with according to methods\*)

let classMethodMap = getClassMethodMap allClassesM in

let allClassmapsF = getClassmapForF classMaps inheritanceMap in(\* classmap including inherited classes which have been dealed with according to field \*)

let finalMap = getFinalMap allClassmapsF allClassesM classMethodMap in

finalMap, allClassesM

and getInheritanceMap cdecls cmap =

let handler a cdecl =

match cdecl.extends with

Parent(s) ->

let new\_list = if (StringMap.mem s a) then

cdecl.cname::(StringMap.find s a)

else

[cdecl.cname]

in

Hashtbl.add inheritanceRelation s new\_list;

(StringMap.add s new\_list a)

| NoParent -> a

in

let forest = List.fold\_left handler StringMap.empty cdecls in

let handler key value =

if not (StringMap.mem key cmap) then

raise (Failure("undefined class"))

in

ignore(StringMap.iter handler forest);

forest

and getClassesForM cdecls inheritanceMap =

let cDataBase = List.fold\_left (fun a litem -> StringMap.add litem.cname litem a) StringMap.empty cdecls (\*class name -> class declaration\*)

in

let seperateInheritanceMap fathers sons sets =

let fatherSet = StringSet.add fathers (fst sets) in

let addSonList sndSet son = StringSet.add son sndSet in

let sonSet = List.fold\_left addSonList (snd sets) sons in

(fatherSet, sonSet)

in

let sSet = StringSet.empty in

let fatherAndson = StringMap.fold seperateInheritanceMap inheritanceMap (sSet, sSet) in

let noFather = StringSet.diff (fst fatherAndson) (snd fatherAndson) in

let rec getNewClasses newMap father sons =

let assistant newMap oneson =

let fatherCdecl = StringMap.find father newMap in (\* class declaration of father\*)

let sonCdecl = StringMap.find oneson cDataBase in (\* class declaration of one son of father \*)

let newSonCdecl = getNewSonCdecl fatherCdecl sonCdecl in

let newNewMap = (StringMap.add oneson newSonCdecl newMap) in

if (StringMap.mem oneson inheritanceMap) then

let sonsOFOneSon = StringMap.find oneson inheritanceMap in

getNewClasses newNewMap oneson sonsOFOneSon

else newNewMap

in

List.fold\_left assistant newMap sons

in

let newClassMap =

let assistant noFather mp =

let sonOfSameFather = StringMap.find noFather inheritanceMap in (\*list: son classes for a certain father class\*)

let noFatherMap = StringMap.add noFather (StringMap.find noFather cDataBase) mp in(\*stringmap: root class name -> class declaration\*)

getNewClasses noFatherMap noFather sonOfSameFather

in

StringSet.fold assistant noFather StringMap.empty

in

let completeClasses cdecl mp =

let halfCompletedCMp =

try StringMap.find cdecl.cname newClassMap

with | Not\_found -> cdecl

in

halfCompletedCMp::mp

in

let completedClassesM = List.fold\_right completeClasses cdecls [] in

completedClassesM

and getNewSonCdecl fatherCdecl sonCdecl =

let sonCBody =

{

fields = fatherCdecl.cbody.fields @ sonCdecl.cbody.fields;

constructors = sonCdecl.cbody.constructors;

methods = getFatherMethods fatherCdecl.cname fatherCdecl.cbody.methods sonCdecl.cbody.methods

}

in

{

cname = sonCdecl.cname;

extends = sonCdecl.extends;

cbody = sonCBody

}

and getFatherMethods father fatherMethods sonMethods =

let checkMethod sonMethod lists =

let newSonMethods =

getNewSonMethod father (fst lists) sonMethod

in

if (fst lists) = newSonMethods

then ((fst lists), sonMethod::(snd lists))

else (newSonMethods, (snd lists))

in

let allMethod =

List.fold\_right checkMethod sonMethods (fatherMethods, [])

in

(fst allMethod) @ (snd allMethod)

and getNewSonMethod father fatherMethods sonMethod =

let replace fatherMethod sonMethodList =

let getClassName = function

None -> Some(father)

| Some(x) -> Some(x)

in

let newSonMethod =

{

fname = sonMethod.fname;

returnType = sonMethod.returnType;

formals = sonMethod.formals;

body = sonMethod.body;

overrides = true;

rootcname = getClassName fatherMethod.rootcname;

}

in

if (getMethodName fatherMethod) = (getMethodName sonMethod)

then newSonMethod::sonMethodList

else fatherMethod::sonMethodList

in

List.fold\_right replace fatherMethods []

and getMethodName fdecl =

let params = List.fold\_left

(fun s ->

(function Formal(t, \_) -> s ^ "." ^ Ast.string\_of\_datatype t

| \_ -> "" ))

"" fdecl.formals

in

let name = Ast.string\_of\_fname fdecl.fname in

let ret\_type = Ast.string\_of\_datatype fdecl.returnType in

ret\_type ^ "." ^ name ^ "." ^ params

and getClassMethodMap allClasses =

let getMethodMap cdecl = (\*stringmap: method name -> function declaration\*)

let addMethod mp fdecl = StringMap.add (getName cdecl.cname fdecl) fdecl mp in

List.fold\_left addMethod StringMap.empty cdecl.cbody.methods

in

let addClassMethodMap mp cdecl = StringMap.add cdecl.cname (getMethodMap cdecl) mp in (\*stringmap: class name -> function map (upper)\*)

List.fold\_left addClassMethodMap StringMap.empty allClasses

and getClassmapForF classMaps inheritanceMap =

let seperateInheritanceMap fathers sons sets =

let fatherSet = StringSet.add fathers (fst sets) in

let addSonList sndSet son = StringSet.add son sndSet in

let sonSet = List.fold\_left addSonList (snd sets) sons in

(fatherSet, sonSet)

in

let sSet = StringSet.empty in

let fatherAndson = StringMap.fold seperateInheritanceMap inheritanceMap (sSet, sSet) in

let noFather = StringSet.diff (fst fatherAndson) (snd fatherAndson) in

let rec getNewClassMap oldMap father sons =

let assistant oldMap son =

let fatherFieldMap = (StringMap.find father oldMap).fieldMap in (\* field\_map of father \*)

let sonFieldMap = (StringMap.find son oldMap).fieldMap in (\* field\_map son \*)

let newSonFieldMap = getNewSonFieldMap fatherFieldMap sonFieldMap in

let newMap = getNewMap newSonFieldMap son oldMap in

if (StringMap.mem son inheritanceMap) then

let sonOfSon = StringMap.find son inheritanceMap in

getNewClassMap newMap son sonOfSon

else newMap

in

List.fold\_left assistant oldMap sons

in

let result = StringSet.fold (fun ns clp -> getNewClassMap clp ns (StringMap.find ns inheritanceMap)) noFather classMaps

in result

and getNewSonFieldMap fatherFieldMap sonFieldMap = (\* add father's fields to son \*)

StringMap.fold (fun fa fi sonmp -> StringMap.add fa fi sonmp) fatherFieldMap sonFieldMap

and getNewMap sonFieldMap son oldMap =

let assistant m =

{

fieldMap = sonFieldMap;

functionMap = m.functionMap;

constructorMap = m.constructorMap;

builtFuncMap = m.builtFuncMap;

cdecl = m.cdecl;

}

in

let sonMap = StringMap.find son oldMap in

let newSonMap = assistant sonMap in

let newMap = StringMap.add son newSonMap oldMap in

newMap

and getFinalMap allClassmapsF allClassesM classMethodMap =

let getCdecl cname =

try List.find (fun cdecl -> cdecl.cname = cname) allClassesM

with | Not\_found -> raise (Failure("Class not found!")) (\*impossible, has been checked before\*)

in

let assistant cname cmap =

let mMap = StringMap.find cname classMethodMap in

let cdecl = getCdecl cname in

{

fieldMap = cmap.fieldMap;

functionMap = mMap;

constructorMap = cmap.constructorMap;

builtFuncMap = cmap.builtFuncMap;

cdecl = cdecl;

}

in

let updateCmap cname cmap mp = StringMap.add cname (assistant cname cmap) mp in

StringMap.fold updateCmap allClassmapsF StringMap.empty

in

let classMaps, cdecls = manageInheritance classes classMaps

in

let appendConstructor fbody cname returnType =

let key = Hashtbl.find strucIndexes cname

in

let thisInit = [SLocal(

returnType,

"this",

SCall( "cast",

[SCall("malloc",

[

SCall("sizeof", [SId("ignore", returnType)], Datatype(Int\_t),0)

],

Arraytype(Char\_t, 1),0)

],

returnType,

0

)

);

SExpr(

SAssign(

SObjAccess(

SId("this", returnType),

SId(".key", Datatype(Int\_t)),

Datatype(Int\_t)

),

SInt\_Lit(key),

Datatype(Int\_t)

),

Datatype(Int\_t)

)

]

in

let returnThis =

[

SReturn(

SId("this", returnType),

returnType

)

]

in

thisInit @ fbody @ returnThis

in

let convertFuncToSfunc classMaps reserved classMap cname func=

let appendMain fbodyStmt cname returnType =

let key = Hashtbl.find strucIndexes cname in

let thisInit = [SLocal(

returnType,

"this",

SCall( "cast",

[SCall("malloc",

[

SCall("sizeof", [SId("ignore", returnType)], Datatype(Int\_t), 0)

],

Arraytype(Char\_t, 1), 0)

],

returnType, 0

)

);

SExpr(

SAssign(

SObjAccess(

SId("this", returnType),

SId(".key", Datatype(Int\_t)),

Datatype(Int\_t)

),

SInt\_Lit(key),

Datatype(Int\_t)

),

Datatype(Int\_t)

)

]

in

thisInit @ fbodyStmt

in

let rootClassName = match func.rootcname with

Some(x) -> x

| None -> cname

in

let classFormal =

if func.overrides then

Ast.Formal(Datatype(Objecttype(rootClassName)), "this")

else

Ast.Formal(Datatype(Objecttype(cname)), "this")

in

let envAssistant m fname = match fname with

Formal(d, s) -> (StringMap.add s fname m)

| \_ -> m

in

let env\_params = List.fold\_left envAssistant StringMap.empty (classFormal :: func.formals) in

let env = {

envClassMaps = classMaps;

envName = cname;

envClassMap = classMap;

envLocals = StringMap.empty;

envParams = env\_params;

envReturnType = func.returnType;

envInFor = false;

envInWhile = false;

envBuiltIn = reserved;

}

in

let fbody = fst (convertStmtsToSstmts env func.body) in

let funcName = (getName cname func) in

let fbody = if funcName= "main"

then (appendMain fbody cname (Datatype(Objecttype(cname))))

else fbody

in

{

sfname = Ast.FName (getName cname func);

sreturnType = func.returnType;

sformals = classFormal :: func.formals;

sbody = fbody;

functype = Sast.User;

overrides = func.overrides;

source = cname;

}

in

let convertConstructorToSfunc classMaps reserved classMap cname constructor =

let env = {

envClassMaps = classMaps;

envName = cname;

envClassMap = classMap;

envLocals = StringMap.empty;

envParams = List.fold\_left (fun m f -> match f with Formal(d, s) -> (StringMap.add s f m) | \_ -> m) StringMap.empty constructor.formals;

envReturnType = Datatype(Objecttype(cname));

envInFor = false;

envInWhile = false;

envBuiltIn = reserved;}

in

let fbody = fst (convertStmtsToSstmts env constructor.body)

in

{

sfname = Ast.FName(getConstructorName cname constructor);

sreturnType = Datatype(Objecttype(cname));

sformals = constructor.formals;

sbody = appendConstructor fbody cname (Datatype(Objecttype(cname)));

functype = Sast.User;

overrides = false;

source = "NA";

}

in

let converttosast classMaps builtinFunctions cdecls =

let deConstructorBody cname =

let retyp = Datatype(Objecttype(cname)) in

let fbody = [] in

appendConstructor fbody cname retyp

in

let defaultSc cname =

{

sfname = Ast.FName (cname ^ "." ^ "constructor");

sreturnType = Datatype(Objecttype(cname));

sformals = [];

sbody = deConstructorBody cname;

functype = Sast.User;

overrides = false;

source = "NA";

}

in

let convertClassToSast sfuncs cdecl =

{scname =cdecl.cname;

sfields =cdecl.cbody.fields;

sfuncs= sfuncs;

}

in

let isMain fdecl = fdecl.sfname = FName("main")

in

let checkMain fdecls =

let mainFuncs = (List.filter isMain fdecls) in

if List.length mainFuncs > 1

then raise (Failure("Multiple main functions are defined!"))

else if List.length mainFuncs < 1

then raise (Failure("Main function is not defined!"))

else List.hd mainFuncs

in

let removeMainFunc funcs = List.filter (fun func -> not(isMain func)) funcs

in

let handleClass cdecl =

let classMap =StringMap.find cdecl.cname classMaps

in

let sConstructors = match cdecl.cbody.constructors with

[] -> (defaultSc cdecl.cname) :: [](\*no user defined constructor\*)

| \_ -> List.map (convertConstructorToSfunc classMaps builtinFunctions classMap cdecl.cname) cdecl.cbody.constructors

in

let funcs = List.fold\_left (fun l f -> (convertFuncToSfunc classMaps builtinFunctions classMap cdecl.cname f):: l) [] cdecl.cbody.methods

in

let sfuncs = removeMainFunc funcs

in

let scdecl = convertClassToSast sfuncs cdecl

in

(scdecl, funcs @sConstructors)

in

let loopClass t c =

let scdecl =handleClass c

in (fst scdecl::fst t, snd scdecl @ snd t)

in

let scdecls, funcs =List.fold\_left loopClass ([],[]) cdecls

in

let mainFunc = checkMain funcs in

let funcs= removeMainFunc funcs in

{

classes = scdecls;

functions = funcs;

main = mainFunc;

reserved = builtinFunctions;

}

in

let sast = converttosast classMaps builtinFunctions cdecls in

sast

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/

\*)

open Llvm

open Hashtbl

open Ast

open Semant

open Sast

module L = Llvm

let context = L.global\_context ()

let the\_module = L.create\_module context "Liva"

let builder = L.builder context

let i1\_t = L.i1\_type context

let i8\_t = L.i8\_type context;;

let i32\_t = L.i32\_type context;;

let i64\_t = L.i64\_type context;;

let f\_t = L.double\_type context;;

let str\_t = L.pointer\_type i8\_t;;

let void\_t = L.void\_type context;;

let arr\_type = Arraytype(Char\_t, 1)

let local\_var\_table:(string, llvalue) Hashtbl.t = Hashtbl.create 100

let formals\_table:(string, llvalue) Hashtbl.t = Hashtbl.create 100

let struct\_typ\_table:(string, lltype) Hashtbl.t = Hashtbl.create 100

let struct\_field\_idx\_table:(string, int) Hashtbl.t = Hashtbl.create 100

(\*~~~~~~~~~~~~~~~ global functions: code generator utils ~~~~~~~~~~~~~~~~~ \*)

let rec get\_llvm\_type datatype = match datatype with

Datatype(Int\_t) -> i32\_t

| Datatype(Float\_t) -> f\_t

| Datatype(Bool\_t) -> i1\_t

| Datatype(Char\_t) -> i8\_t

| Datatype(Void\_t) -> void\_t

| Datatype(Null\_t) -> i32\_t

| Datatype(Objecttype(name)) -> L.pointer\_type(find\_llvm\_struct\_type name)

| Arraytype(t, i) -> get\_arr\_llvm\_type (Arraytype(t, (i)))

| \_ -> raise(Failure ("Invalid DataType"))

and find\_llvm\_struct\_type name =

try Hashtbl.find struct\_typ\_table name

with | Not\_found -> raise(Failure ("undeclared struct"^ name))

and get\_arr\_llvm\_type datatype = match datatype with

Arraytype(t, 0) -> get\_llvm\_type (Datatype(t))

| Arraytype(t, 1) -> L.pointer\_type (get\_llvm\_type (Datatype(t)))

| Arraytype(t, i) -> L.pointer\_type (get\_arr\_llvm\_type (Arraytype(t, (i-1))))

| \_ -> raise(Failure ("Invalid Array Pointer Type"))

let string\_of\_fname = function

Constructor -> "constructor"

| FName(s) -> s

let find\_func\_in\_module fname =

match (L.lookup\_function fname the\_module) with

None -> raise (Failure("Function NotFound in module: " ^ fname))

| Some f -> f

(\*~~~~~~~~~~~~~~~~~~~~~~~~~ code generator top level ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ \*)

let translate sast =

let classes = sast.classes in

let functions = sast.functions in

let main = sast.main in

let util\_func () =

let printf\_typ = L.var\_arg\_function\_type i32\_t [| pointer\_type i8\_t |] in

let malloc\_typ = L.function\_type (str\_t) [| i32\_t |] in

let lookup\_typ = L.function\_type (pointer\_type i64\_t) [| i32\_t; i32\_t |] in

let \_ = L.declare\_function "printf" printf\_typ the\_module in

let \_ = L.declare\_function "malloc" malloc\_typ the\_module in

let \_ = L.define\_function "lookup" lookup\_typ the\_module in

()

in

let \_ = util\_func () in

(\*~~~~~~~~~~~~~~~~~~~~~~~~~ Generate class struct ~~~~~~~~~~~~~~~~~~~~~~~~~~~~\*)

let add\_struct\_typ\_table cls =

let struct\_typ = L.named\_struct\_type context cls.scname in

Hashtbl.add struct\_typ\_table cls.scname struct\_typ

in

let \_ = List.map add\_struct\_typ\_table classes in

let class\_struct\_gen s =

let struct\_t = Hashtbl.find struct\_typ\_table s.scname in

let type\_list = List.map (function Field(d, \_) -> get\_llvm\_type d) s.sfields in

let name\_list = List.map (function Field(\_, s) -> s) s.sfields in

let type\_list = i32\_t :: type\_list in

let name\_list = ".key" :: name\_list in

let type\_array = (Array.of\_list type\_list) in

List.iteri (

fun i f ->

let n = s.scname ^ "." ^ f in

Hashtbl.add struct\_field\_idx\_table n i;

)

name\_list;

L.struct\_set\_body struct\_t type\_array true

in

let \_ = List.map class\_struct\_gen classes in

(\*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ define functions ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\*)

let func\_define sfdecl=

let fname = sfdecl.sfname in

let is\_var\_arg =ref false in

let parameters = List.rev ( List.fold\_left

(fun l ->

(function

Formal (t,\_)-> get\_llvm\_type t::l

| \_ -> ignore(is\_var\_arg = ref true); l

)

)

[] sfdecl.sformals)

in

let fty =

if !is\_var\_arg

then L.var\_arg\_function\_type (get\_llvm\_type sfdecl.sreturnType )

(Array.of\_list parameters)

else L.function\_type (get\_llvm\_type sfdecl.sreturnType)

(Array.of\_list parameters)

in

L.define\_function (string\_of\_fname fname) fty the\_module

in

let \_ = List.map func\_define functions in

(\*~~~~~~~~~~~~~~~~~~~~~ function generation utils ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\*)

(\*statement gengeration\*)

let rec stmt\_gen llbuilder = function

SBlock sl -> List.hd (List.map (stmt\_gen llbuilder) sl)

| SLocal (d, s, e) ->

let local\_gen datatype var\_name expr llbuilder =

let t = match datatype with

Datatype(Objecttype(name)) -> find\_llvm\_struct\_type name

| \_ -> get\_llvm\_type datatype

in

let alloca = L.build\_alloca t var\_name llbuilder in

Hashtbl.add local\_var\_table var\_name alloca;

let lhs = SId(var\_name, datatype) in

match expr with

SNoexpr -> alloca

| \_ -> assign\_gen lhs expr datatype llbuilder

in

local\_gen d s e llbuilder

| SReturn (e, d) ->

let return\_gen d expr llbuilder =

match expr with

SId(name, d) ->

(match d with

| Datatype(Objecttype(\_)) -> build\_ret (id\_gen false false name d llbuilder) llbuilder

| \_ -> build\_ret (id\_gen true true name d llbuilder) llbuilder)

| SObjAccess(e1, e2, d) -> build\_ret (obj\_access\_gen true e1 e2 d llbuilder) llbuilder

| SNoexpr -> build\_ret\_void llbuilder

| \_ -> build\_ret (expr\_gen llbuilder expr) llbuilder

in

return\_gen d e llbuilder

| SExpr (se, \_) -> expr\_gen llbuilder se

(\*control flow\*)

| SIf (e, s1, s2) ->

let if\_gen exp then\_stmt else\_stmt llbuilder =

let condition= expr\_gen llbuilder exp in

let start\_block = L.insertion\_block llbuilder in

let parent\_function = L.block\_parent start\_block in

let then\_block = L.append\_block context "then" parent\_function in

L.position\_at\_end then\_block llbuilder;

let then\_val = stmt\_gen llbuilder then\_stmt in

let new\_then\_block = L.insertion\_block llbuilder in

let else\_block = L.append\_block context "else" parent\_function in

L.position\_at\_end else\_block llbuilder;

let else\_val= stmt\_gen llbuilder else\_stmt in

let new\_else\_block = L.insertion\_block llbuilder in

let merge\_block = L.append\_block context "ifcont" parent\_function in

L.position\_at\_end merge\_block builder;

let incoming = [(then\_val, new\_then\_block); (else\_val, new\_else\_block)] in

let phi = L.build\_phi incoming "iftmp" builder in

L.position\_at\_end start\_block llbuilder;

ignore (build\_cond\_br condition then\_block else\_block llbuilder);

position\_at\_end new\_then\_block llbuilder; ignore (build\_br merge\_block llbuilder);

position\_at\_end new\_else\_block llbuilder; ignore (build\_br merge\_block llbuilder);

(\* set the builder to the end of the merge block \*)

L.position\_at\_end merge\_block llbuilder;

phi

in

if\_gen e s1 s2 llbuilder

| SFor (se1, se2, se3, s) -> for\_gen se1 se2 se3 s llbuilder

| SWhile (se, s) -> while\_gen se s llbuilder

(\*expression generation\*)

and expr\_gen llbuilder = function

SInt\_Lit (i) -> L.const\_int i32\_t i

| SBoolean\_Lit (b) -> if b then L.const\_int i1\_t 1 else L.const\_int i1\_t 0

| SFloat\_Lit (f) -> L.const\_float f\_t f

| SChar\_Lit (c) -> L.const\_int i8\_t (Char.code c)

| SString\_Lit (s) -> L.build\_global\_stringptr s "tmp" llbuilder

| SId (id, d) -> id\_gen true false id d llbuilder

| SBinop (e1, op, e2, d) ->

let binop\_gen e1 op e2 d llbuilder =

let type1 = Semant.typOFSexpr e1 in

let type2 = Semant.typOFSexpr e2 in

let e1 = expr\_gen llbuilder e1 in

let e2 = expr\_gen llbuilder e2 in

let float\_ops op e1 e2 =

match op with

Add -> L.build\_fadd e1 e2 "flt\_addtmp" llbuilder

| Sub -> L.build\_fsub e1 e2 "flt\_subtmp" llbuilder

| Mult -> L.build\_fmul e1 e2 "flt\_multmp" llbuilder

| Div -> L.build\_fdiv e1 e2 "flt\_divtmp" llbuilder

| Mod -> L.build\_frem e1 e2 "flt\_sremtmp" llbuilder

| Equal -> L.build\_fcmp Fcmp.Oeq e1 e2 "flt\_eqtmp" llbuilder

| Neq -> L.build\_fcmp Fcmp.One e1 e2 "flt\_neqtmp" llbuilder

| Less -> L.build\_fcmp Fcmp.Ult e1 e2 "flt\_lesstmp" llbuilder

| Leq -> L.build\_fcmp Fcmp.Ole e1 e2 "flt\_leqtmp" llbuilder

| Greater -> L.build\_fcmp Fcmp.Ogt e1 e2 "flt\_sgttmp" llbuilder

| Geq -> L.build\_fcmp Fcmp.Oge e1 e2 "flt\_sgetmp" llbuilder

| \_ -> raise(Failure("Invalid operator for floats"))

in

let int\_ops op e1 e2 =

match op with

Add -> L.build\_add e1 e2 "addtmp" llbuilder

| Sub -> L.build\_sub e1 e2 "subtmp" llbuilder

| Mult -> L.build\_mul e1 e2 "multmp" llbuilder

| Div -> L.build\_sdiv e1 e2 "divtmp" llbuilder

| Mod -> L.build\_srem e1 e2 "sremtmp" llbuilder

| Equal -> L.build\_icmp Icmp.Eq e1 e2 "eqtmp" llbuilder

| Neq -> L.build\_icmp Icmp.Ne e1 e2 "neqtmp" llbuilder

| Less -> L.build\_icmp Icmp.Slt e1 e2 "lesstmp" llbuilder

| Leq -> L.build\_icmp Icmp.Sle e1 e2 "leqtmp" llbuilder

| Greater -> L.build\_icmp Icmp.Sgt e1 e2 "sgttmp" llbuilder

| Geq -> L.build\_icmp Icmp.Sge e1 e2 "sgetmp" llbuilder

| And -> L.build\_and e1 e2 "andtmp" llbuilder

| Or -> L.build\_or e1 e2 "ortmp" llbuilder

| \_ -> raise(Failure("Invalid operator for integers"))

in

let binop\_type\_cast lhs rhs lhsType rhsType llbuilder =

match (lhsType, rhsType) with

Datatype(Int\_t), Datatype(Int\_t) -> (lhs, rhs), Datatype(Int\_t)

| Datatype(Int\_t), Datatype(Char\_t) -> (build\_uitofp lhs i8\_t "tmp" llbuilder, rhs), Datatype(Char\_t)

| Datatype(Int\_t), Datatype(Float\_t) -> (build\_sitofp lhs f\_t "tmp" llbuilder, rhs), Datatype(Float\_t)

| Datatype(Char\_t), Datatype(Int\_t) -> (lhs, build\_uitofp rhs i8\_t "tmp" llbuilder), Datatype(Char\_t)

| Datatype(Char\_t), Datatype(Char\_t) -> (lhs, rhs), Datatype(Char\_t)

| Datatype(Bool\_t), Datatype(Bool\_t) -> (lhs, rhs), Datatype(Bool\_t)

| Datatype(Float\_t), Datatype(Int\_t) -> (lhs, build\_sitofp rhs f\_t "tmp" llbuilder), Datatype(Float\_t)

| Datatype(Float\_t), Datatype(Float\_t) -> (lhs, rhs), Datatype(Float\_t)

| Datatype(Objecttype(d)), Datatype(Null\_t) -> (lhs, rhs), lhsType

| Datatype(Null\_t), Datatype(Objecttype(d)) -> (rhs, lhs), rhsType

| Arraytype(d, s), Datatype(Null\_t) -> (lhs, rhs), lhsType

| Datatype(Null\_t), Arraytype(d, s) -> (rhs, lhs), rhsType

| \_ -> raise (Failure("binop type not supported"))

in

let (e1, e2), d = binop\_type\_cast e1 e2 type1 type2 llbuilder in

let type\_handler d = match d with

Datatype(Float\_t) -> float\_ops op e1 e2

| Datatype(Int\_t)

| Datatype(Bool\_t)

| Datatype(Char\_t) -> int\_ops op e1 e2

| Datatype(Objecttype(\_))

| \_ -> raise (Failure("Invalid binop type"))

in

type\_handler d

in

binop\_gen e1 op e2 d llbuilder

| SUnop (op, e, d) ->

let unop\_gen op e d llbuilder =

let e\_typ = Semant.typOFSexpr e in

let e = expr\_gen llbuilder e in

let unops op e\_typ e = match (op, e\_typ) with

(Sub, Datatype(Int\_t)) -> L.build\_neg e "int\_unoptmp" llbuilder

| (Sub, Datatype(Float\_t)) -> L.build\_fneg e "flt\_unoptmp" llbuilder

| (Not, Datatype(Bool\_t)) -> L.build\_not e "bool\_unoptmp" llbuilder

| \_ -> raise (Failure("unop not supported")) in

let unop\_type\_handler d = match d with

Datatype(Float\_t)

| Datatype(Int\_t)

| Datatype(Bool\_t) -> unops op e\_typ e

| \_ -> raise (Failure("invalid unop type "))

in

unop\_type\_handler d

in

unop\_gen op e d llbuilder

| SAssign (e1, e2, d) -> assign\_gen e1 e2 d llbuilder

| SCall (fname, expr\_list, d, \_) ->

let reserved\_func\_gen llbuilder d expr\_list = function

"print" -> print\_func\_gen expr\_list llbuilder

| "sizeof" -> sizeof\_func\_gen expr\_list llbuilder

| "cast" -> cast\_func\_gen expr\_list d llbuilder

| "malloc" -> malloc\_func\_gen "malloc" expr\_list d llbuilder

| \_ as call\_name -> raise(Failure("function call not found: "^ call\_name))

in

reserved\_func\_gen llbuilder d expr\_list fname

| SObjectCreate (id, el, d) ->

let create\_obj\_gen fname el d llbuilder =

let f = find\_func\_in\_module fname in

let params = List.map (expr\_gen llbuilder) el in

let obj = L.build\_call f (Array.of\_list params) "tmp" llbuilder in

obj

in

create\_obj\_gen id el d llbuilder

| SObjAccess (e1, e2, d) -> obj\_access\_gen true e1 e2 d llbuilder

| SArrayCreate (t, el, d) ->

let arr\_create\_gen llbuilder t expr\_type el =

if(List.length el > 1) then raise(Failure("array not supported"))

else

match expr\_type with

Arraytype(Char\_t, 1) ->

let e = List.hd el in

let size = (expr\_gen llbuilder e) in

let t = get\_llvm\_type t in

let arr = L.build\_array\_malloc t size "tmp" llbuilder in

let arr = L.build\_pointercast arr (pointer\_type t) "tmp" llbuilder in

arr

| \_ ->

let e = List.hd el in

let t = get\_llvm\_type t in

let size = (expr\_gen llbuilder e) in

let size\_t = L.build\_intcast (size\_of t) i32\_t "tmp" llbuilder in

let size = L.build\_mul size\_t size "tmp" llbuilder in

let size\_real = L.build\_add size (const\_int i32\_t 1) "arr\_size" llbuilder in

let arr = L.build\_array\_malloc t size\_real "tmp" llbuilder in

let arr = L.build\_pointercast arr (pointer\_type t) "tmp" llbuilder in

let arr\_len\_ptr = L.build\_pointercast arr (pointer\_type i32\_t) "tmp" llbuilder in

ignore(build\_store size\_real arr\_len\_ptr llbuilder);

let init\_array arr arr\_len init\_val start\_pos llbuilder =

let new\_block label =

let f = block\_parent (insertion\_block llbuilder) in

append\_block (global\_context ()) label f

in

let bbcurr = insertion\_block llbuilder in

let bbcond = new\_block "array.cond" in

let bbbody = new\_block "array.init" in

let bbdone = new\_block "array.done" in

ignore (L.build\_br bbcond llbuilder);

position\_at\_end bbcond llbuilder;

let counter = L.build\_phi [const\_int i32\_t start\_pos, bbcurr] "counter" llbuilder in

add\_incoming ((L.build\_add counter (const\_int i32\_t 1) "tmp" llbuilder), bbbody) counter;

let cmp = L.build\_icmp Icmp.Slt counter arr\_len "tmp" llbuilder in

ignore (L.build\_cond\_br cmp bbbody bbdone llbuilder);

position\_at\_end bbbody llbuilder;

let arr\_ptr = L.build\_gep arr [| counter |] "tmp" llbuilder in

ignore (L.build\_store init\_val arr\_ptr llbuilder);

ignore (L.build\_br bbcond llbuilder);

position\_at\_end bbdone llbuilder

in

init\_array arr\_len\_ptr size\_real (const\_int i32\_t 0) 0 llbuilder;

arr

in

arr\_create\_gen llbuilder t d el

| SArrayAccess (e, el, d) -> arr\_access\_gen false e el d llbuilder

| SNoexpr -> L.build\_add (L.const\_int i32\_t 0) (L.const\_int i32\_t 0) "nop" llbuilder

| \_ -> raise(Failure("expression not match"))

and print\_func\_gen expr\_list llbuilder =

let printf = find\_func\_in\_module "printf" in

let tmp\_count = ref 0 in

let incr\_tmp = fun x -> incr tmp\_count in

let map\_expr\_to\_printfexpr expr =

let exprType = Semant.typOFSexpr expr in

match exprType with

Datatype(Bool\_t) ->

incr\_tmp ();

let tmp\_var = "tmp" ^ (string\_of\_int !tmp\_count) in

let trueStr = SString\_Lit("true") in

let falseStr = SString\_Lit("false") in

let id = SId(tmp\_var, arr\_type) in

ignore(stmt\_gen llbuilder (SLocal(arr\_type, tmp\_var, SNoexpr)));

ignore(

stmt\_gen llbuilder

(

SIf(

expr, SExpr(SAssign(id, trueStr, arr\_type), arr\_type),

SExpr(SAssign(id, falseStr, arr\_type), arr\_type)

)

)

);

expr\_gen llbuilder id

| \_ -> expr\_gen llbuilder expr

in

let params = List.map map\_expr\_to\_printfexpr expr\_list in

let param\_types = List.map (Semant.typOFSexpr) expr\_list in

let map\_param\_to\_string = function

Arraytype(Char\_t, 1) -> "%s"

| Datatype(Int\_t) -> "%d"

| Datatype(Float\_t) -> "%f"

| Datatype(Bool\_t) -> "%s"

| Datatype(Char\_t) -> "%c"

| \_ -> raise (Failure("Print invalid type"))

in

let const\_str = List.fold\_left (fun s t -> s ^ map\_param\_to\_string t) "" param\_types in

let s = expr\_gen llbuilder (SString\_Lit(const\_str)) in

let zero = const\_int i32\_t 0 in

let s = L.build\_in\_bounds\_gep s [| zero |] "tmp" llbuilder in

L.build\_call printf (Array.of\_list (s :: params)) "tmp" llbuilder

and sizeof\_func\_gen el llbuilder =

let type\_of\_sexpr = Semant.typOFSexpr (List.hd el) in

let type\_of\_sexpr = get\_llvm\_type type\_of\_sexpr in

let size\_of\_typ = L.size\_of type\_of\_sexpr in

L.build\_intcast size\_of\_typ i32\_t "tmp" llbuilder

and cast\_func\_gen el d llbuilder =

let cast\_malloc\_to\_objtype lhs currType newType llbuilder = match newType with

Datatype(Objecttype(x)) ->

let obj\_type = get\_llvm\_type (Datatype(Objecttype(x))) in

L.build\_pointercast lhs obj\_type "tmp" llbuilder

| \_ -> raise (Failure("cannot cast"))

in

let expr = List.hd el in

let t = Semant.typOFSexpr expr in

let lhs = match expr with

| Sast.SId(id, d) -> id\_gen false false id d llbuilder

| SObjAccess(e1, e2, d) -> obj\_access\_gen false e1 e2 d llbuilder

| \_ -> expr\_gen llbuilder expr

in

cast\_malloc\_to\_objtype lhs t d llbuilder

and malloc\_func\_gen fname el d llbuilder =

let f = find\_func\_in\_module fname in

let params = List.map (expr\_gen llbuilder) el in

match d with

Datatype(Void\_t) -> L.build\_call f (Array.of\_list params) "" llbuilder

| \_ -> L.build\_call f (Array.of\_list params) "tmp" llbuilder

and id\_gen deref checkParam id d llbuilder =

if deref then

try Hashtbl.find formals\_table id

with | Not\_found ->

try let \_val = Hashtbl.find local\_var\_table id in

build\_load \_val id llbuilder

with | Not\_found -> raise (Failure("unknown variable id " ^ id))

else

try Hashtbl.find local\_var\_table id

with | Not\_found ->

try

let \_val = Hashtbl.find formals\_table id in

if checkParam then raise (Failure("cannot assign"))

else \_val

with | Not\_found -> raise (Failure("unknown variable id "^ id))

and assign\_gen lhs rhs d llbuilder =

let rhs\_t = Semant.typOFSexpr rhs in

let lhs, isObjAccess = match lhs with

| Sast.SId(id, d) -> id\_gen false false id d llbuilder, false

| SObjAccess(e1, e2, d) -> obj\_access\_gen false e1 e2 d llbuilder, true

| SArrayAccess(se, sel, d) -> arr\_access\_gen true se sel d llbuilder, true

| \_ -> raise (Failure("Left hand side must be assignable"))

in

let rhs = match rhs with

| Sast.SId(id, d) -> id\_gen false false id d llbuilder

| SObjAccess(e1, e2, d) -> obj\_access\_gen true e1 e2 d llbuilder

| \_ -> expr\_gen llbuilder rhs

in

let rhs = match d with

Datatype(Objecttype(\_)) ->

if isObjAccess then rhs

else build\_load rhs "tmp" llbuilder

| Datatype(Null\_t) -> L.const\_null (get\_llvm\_type d)

| \_ -> rhs

in

let rhs = match d, rhs\_t with

Datatype(Char\_t), Datatype(Int\_t) -> L.build\_uitofp rhs i8\_t "tmp" llbuilder

| Datatype(Int\_t), Datatype(Char\_t) -> L.build\_uitofp rhs i32\_t "tmp" llbuilder

| \_ -> rhs

in

ignore(L.build\_store rhs lhs llbuilder);

rhs

and for\_gen start cond step body llbuilder =

let preheader\_bb = L.insertion\_block llbuilder in

let the\_function = L.block\_parent preheader\_bb in

let \_ = expr\_gen llbuilder start in

let loop\_bb = L.append\_block context "loop" the\_function in

let step\_bb = L.append\_block context "step" the\_function in

let cond\_bb = L.append\_block context "cond" the\_function in

let after\_bb = L.append\_block context "afterloop" the\_function in

ignore (L.build\_br cond\_bb llbuilder);

L.position\_at\_end loop\_bb llbuilder;

ignore (stmt\_gen llbuilder body);

let bb = L.insertion\_block llbuilder in

L.move\_block\_after bb step\_bb;

L.move\_block\_after step\_bb cond\_bb;

L.move\_block\_after cond\_bb after\_bb;

ignore(L.build\_br step\_bb llbuilder);

L.position\_at\_end step\_bb llbuilder;

let \_ = expr\_gen llbuilder step in

ignore(L.build\_br cond\_bb llbuilder);

L.position\_at\_end cond\_bb llbuilder;

let cond\_val = expr\_gen llbuilder cond in

ignore (L.build\_cond\_br cond\_val loop\_bb after\_bb llbuilder);

L.position\_at\_end after\_bb llbuilder;

const\_null f\_t

and while\_gen pred body\_stmt llbuilder =

let null\_sexpr = SInt\_Lit(0) in

for\_gen null\_sexpr pred null\_sexpr body\_stmt llbuilder

and obj\_access\_gen is\_assign lhs rhs d llbuilder =

let obj\_func\_gen param\_ty fptr parent\_expr el d llbuilder =

let match\_sexpr se = match se with

SId(id, d) ->

let deref = match d with

Datatype(Objecttype(\_)) -> false

| \_ -> true

in

id\_gen deref false id d llbuilder

| se -> expr\_gen llbuilder se

in

let parent\_expr = build\_pointercast parent\_expr param\_ty "tmp" llbuilder in

let params = List.map match\_sexpr el in

match d with

Datatype(Void\_t) -> L.build\_call fptr (Array.of\_list (parent\_expr :: params)) "" llbuilder

| \_ -> L.build\_call fptr (Array.of\_list (parent\_expr :: params)) "tmp" llbuilder

in

let check\_lhs = function

SId(s, d) -> id\_gen false false s d llbuilder

| SArrayAccess(e, el, d) -> arr\_access\_gen false e el d llbuilder

| \_ -> raise (Failure("check lhd error"))

in

let rec check\_rhs parent\_expr parent\_type =

let parent\_str = Ast.string\_of\_object parent\_type in

function

SId(field, d) ->

let search\_term = (parent\_str ^ "." ^ field) in

let field\_index = Hashtbl.find struct\_field\_idx\_table search\_term in

let \_val = build\_struct\_gep parent\_expr field\_index field llbuilder in

let \_val = match d with

Datatype(Objecttype(\_)) ->

if not is\_assign then \_val

else build\_load \_val field llbuilder

| \_ ->

if not is\_assign then

\_val

else

build\_load \_val field llbuilder

in

\_val

| SCall(fname, el, d, index) ->

let index = const\_int i32\_t index in

let c\_index = build\_struct\_gep parent\_expr 0 "cindex" llbuilder in

let c\_index = build\_load c\_index "cindex" llbuilder in

let lookup\_func = find\_func\_in\_module "lookup" in

let fptr = L.build\_call lookup\_func [| c\_index; index |] "fptr" llbuilder in

let fptr2 = find\_func\_in\_module fname in

let f\_ty = type\_of fptr2 in

let param1 = param fptr2 0 in

let param\_ty = type\_of param1 in

let fptr = L.build\_pointercast fptr f\_ty fname llbuilder in

let ret = obj\_func\_gen param\_ty fptr parent\_expr el d llbuilder in

let ret = ret in

ret

| SObjAccess(e1, e2, \_) ->

let e1\_type = Semant.typOFSexpr e1 in

let e1 = check\_rhs parent\_expr parent\_type e1 in

let e2 = check\_rhs e1 e1\_type e2 in

e2

| \_ -> raise (Failure("invalid access"))

in

let lhs\_type = Semant.typOFSexpr lhs in

match lhs\_type with

Arraytype(\_, \_) ->

let lhs = expr\_gen llbuilder lhs in

let \_val = build\_gep lhs [| (const\_int i32\_t 0) |] "tmp" llbuilder in

build\_load \_val "tmp" llbuilder

| \_ ->

let lhs = check\_lhs lhs in

let rhs = check\_rhs lhs lhs\_type rhs in

rhs

and arr\_access\_gen is\_assign e el d llbuilder =

let index = expr\_gen llbuilder (List.hd el) in

let index = match d with

Datatype(Char\_t) -> index

| \_ -> L.build\_add index (const\_int i32\_t 1) "tmp" llbuilder

in

let arr = expr\_gen llbuilder e in

let \_val = L.build\_gep arr [| index |] "tmp" llbuilder in

if is\_assign

then \_val

else build\_load \_val "tmp" llbuilder

in

(\*~~~~~~~~~~~~~~~~~~~~~~~~~ function generation top-level ~~~~~~~~~~~~~~~~~~~~~~~~~~~~\*)

let build\_func sfdecl =

Hashtbl.clear local\_var\_table;

Hashtbl.clear formals\_table;

let fname = string\_of\_fname sfdecl.sfname in

let f =find\_func\_in\_module fname in

let llbuilder = L.builder\_at\_end context (L.entry\_block f) in

let init\_formals f sformals =

let sformals = Array.of\_list (sformals) in

Array.iteri (

fun i a ->

let formal = sformals.(i) in

let string\_of\_formal\_name = function

Formal(\_, s) -> s

| \_ -> " "

in

let formal\_name = string\_of\_formal\_name formal in

L.set\_value\_name formal\_name a;

Hashtbl.add formals\_table formal\_name a;

)

(params f)

in

let \_ = init\_formals f sfdecl.sformals in

let \_ = if sfdecl.overrides then

let this\_param = Hashtbl.find formals\_table "this" in

let source = Datatype(Objecttype(sfdecl.source)) in

let casted\_param = L.build\_pointercast this\_param (get\_llvm\_type source) "casted" llbuilder in

Hashtbl.replace formals\_table "this" casted\_param;

in

let \_ = stmt\_gen llbuilder (SBlock (sfdecl.sbody)) in

if sfdecl.sreturnType = Datatype (Void\_t)

then ignore (L.build\_ret\_void llbuilder);

()

in

let \_ = List.map build\_func functions in

(\*~~~~~~~~~~~~~~~~~~~~ main function generation top-level ~~~~~~~~~~~~~~~~~~~~~~~~~~~~\*)

let build\_main main =

Hashtbl.clear local\_var\_table;

Hashtbl.clear formals\_table;

let fty = L.function\_type i32\_t[||] in

let f = L.define\_function "main" fty the\_module in

let llbuilder = L.builder\_at\_end context (L.entry\_block f) in

let \_ = stmt\_gen llbuilder (SBlock (main.sbody)) in

L.build\_ret (L.const\_int i32\_t 0) llbuilder

in

let \_ = build\_main main in

(\*~~~~~~~~~~~~~~~ virtual function table generation top-level ~~~~~~~~~~~~~~~~~~~~~~~~~~\*)

let build\_vftable scdecls =

let rt = L.pointer\_type i64\_t in

let void\_pt = L.pointer\_type i64\_t in

let void\_ppt = L.pointer\_type void\_pt in

let f = find\_func\_in\_module "lookup" in

let llbuilder = L.builder\_at\_end context (entry\_block f) in

let len = List.length scdecls in

let total\_len = ref 0 in

let scdecl\_llvm\_arr = L.build\_array\_alloca void\_ppt (const\_int i32\_t len) "tmp" llbuilder in

let handle\_scdecl scdecl =

let index = Hashtbl.find Semant.strucIndexes scdecl.scname in

let len = List.length scdecl.sfuncs in

let sfdecl\_llvm\_arr = L.build\_array\_alloca void\_pt (const\_int i32\_t len) "tmp" llbuilder in

let handle\_fdecl i sfdecl =

let fptr = find\_func\_in\_module (Ast.string\_of\_fname sfdecl.sfname) in

let fptr = L.build\_pointercast fptr void\_pt "tmp" llbuilder in

let ep = L.build\_gep sfdecl\_llvm\_arr [| (const\_int i32\_t i) |] "tmp" llbuilder in

ignore(L.build\_store fptr ep llbuilder);

in

List.iteri handle\_fdecl scdecl.sfuncs;

total\_len := !total\_len + len;

let ep = L.build\_gep scdecl\_llvm\_arr [| (const\_int i32\_t index) |] "tmp" llbuilder in

ignore(build\_store sfdecl\_llvm\_arr ep llbuilder);

in

List.iter handle\_scdecl scdecls;

let c\_index = param f 0 in

let f\_index = param f 1 in

L.set\_value\_name "c\_index" c\_index;

L.set\_value\_name "f\_index" f\_index;

if !total\_len == 0 then

L.build\_ret (const\_null rt) llbuilder

else

let vtbl = L.build\_gep scdecl\_llvm\_arr [| c\_index |] "tmp" llbuilder in

let vtbl = L.build\_load vtbl "tmp" llbuilder in

let fptr = L.build\_gep vtbl [| f\_index |] "tmp" llbuilder in

let fptr = L.build\_load fptr "tmp" llbuilder in

L.build\_ret fptr llbuilder

in

let \_ = build\_vftable classes in

the\_module;

liva.ml

(\* Top-level of the Liva compiler: scan & parse the input,

check the resulting AST, generate LLVM IR, and dump the module \*)

open Ast

open Sast

type action = Ast | LLVM\_IR | Compile

let \_ =

let action = if Array.length Sys.argv > 1 then

List.assoc Sys.argv.(1) [ ("-a", Ast); (\* Print the AST only \*)

("-l", LLVM\_IR); (\* Generate LLVM, don't check \*)

("-c", Compile) ] (\* Generate, check LLVM IR \*)

else Compile in

let lexbuf = Lexing.from\_channel stdin in

let ast = Parser.program Scanner.token lexbuf in

let sast = Semant.check ast in ();

match action with

Ast -> print\_string ("not implemented")

| 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)

Makefile

# Make sure ocamlbuild can find opam-managed packages: first run

#

# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : liva.native

liva.native :

ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \

liva.native

# "make clean" removes all generated files

.PHONY : clean

clean :

ocamlbuild -clean

rm -rf testall.log \*.diff liva scanner.ml parser.ml parser.mli

rm -rf \*.cmx \*.cmi \*.cmo \*.cmx \*.o

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx sast.cmx parser.cmx scanner.cmx semant.cmx codegen.cmx liva.cmx

liva : $(OBJS)

ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS) -o liva

scanner.ml : scanner.mll

ocamllex scanner.mll

parser.ml parser.mli : parser.mly

ocamlyacc parser.mly

%.cmo : %.ml

ocamlc -c $<

%.cmi : %.mli

ocamlc -c $<

%.cmx : %.ml

ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep \*.ml \*.mli" after building scanner.ml and parser.ml

ast.cmo :

ast.cmx :

sast.cmo :

sast.cmx :

codegen.cmo : ast.cmo sast.cmo

codegen.cmx : ast.cmx ast.cmx

liva.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo sast.cmo

liva.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx sast.cmx

parser.cmo : ast.cmo sast.cmo parser.cmi

parser.cmx : ast.cmx sast.cmx parser.cmi

scanner.cmo : parser.cmi

scanner.cmx : parser.cmx

semant.cmo : ast.cmo sast.cmo

semant.cmx : ast.cmx sast.cmx

parser.cmi : ast.cmo

# Building the tarball

TESTS = add1 arith1 arith2 arith3 fib for1 for2 func1 func2 func3 \

func4 func5 func6 func7 func8 gcd2 gcd global1 global2 global3 \

hello if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1 var2 \

while1 while2

FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1 for2 \

for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 func8 \

func9 global1 global2 if1 if2 if3 nomain return1 return2 while1 \

while2

TESTFILES = $(TESTS:%=test-%.mc) $(TESTS:%=test-%.out) \

$(FAILS:%=fail-%.mc) $(FAILS:%=fail-%.err)

TARFILES = asat.ml ast.ml codegen.ml Makefile liva.ml parser.mly README scanner.mll \

semant.ml testall.sh $(TESTFILES:%=test/%)

liva-llvm.tar.gz : $(TARFILES)

cd .. && tar czf liva-llvm/liva-llvm.tar.gz \

$(TARFILES:%=liva-llvm/%)

testall.sh

#!/bin/sh

# Regression testing script for Liva

# Step through a list of files

# Compile, run, and check the output of each expected-to-work test

# Compile and check the error of each expected-to-fail test

# Path to the LLVM interprete

LLI="lli"

#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the liva compiler. Usually "./liva.native"

# Try "\_build/liva.native" if ocamlbuild was unable to create a symbolic link.

#LIVA="./liva.native"

LIVA="./liva"

# Set time limit for all operations

ulimit -t 30

globallog=testall.log

rm -f $globallog

error=0

globalerror=0

keep=0

Usage() {

echo "Usage: testall.sh [options] [.liva files]"

echo "-k Keep intermediate files"

echo "-h Print this help"

exit 1

}

SignalError() {

if [ $error -eq 0 ] ; then

echo "FAILED"

error=1

fi

echo " $1"

}

# Compare <outfile> <reffile> <difffile>

# Compares the outfile with reffile. Differences, if any, written to difffile

Compare() {

generatedfiles="$generatedfiles $3"

echo diff -b $1 $2 ">" $3 1>&2

diff -b "$1" "$2" > "$3" 2>&1 || {

SignalError "$1 differs"

echo "FAILED $1 differs from $2" 1>&2

}

}

# Run <args>

# Report the command, run it, and report any errors

Run() {

echo $\* 1>&2

eval $\* || {

SignalError "$1 failed on $\*"

return 1

}

}

# RunFail <args>

# Report the command, run it, and expect an erro

RunFail() {

echo $\* 1>&2

eval $\* && {

SignalError "failed: $\* did not report an error"

return 1

}

return 0

}

Check() {

error=0

basename=`echo $1 | sed 's/.\*\\///

s/.liva//'`

reffile=`echo $1 | sed 's/.liva$//'`

basedir="`echo $1 | sed 's/\/[^\/]\*$//'`/."

echo -n "$basename..."

echo 1>&2

echo "###### Testing $basename" 1>&2

generatedfiles=""

generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&

Run "$LIVA" "<" $1 ">" "${basename}.ll" &&

Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&

Compare ${basename}.out ${reffile}.out ${basename}.diff

# Report the status and clean up the generated files

if [ $error -eq 0 ] ; then

if [ $keep -eq 0 ] ; then

rm -f $generatedfiles

fi

echo "OK"

echo "###### SUCCESS" 1>&2

else

echo "###### FAILED" 1>&2

globalerror=$erro

fi

}

CheckFail() {

error=0

basename=`echo $1 | sed 's/.\*\\///

s/.liva//'`

reffile=`echo $1 | sed 's/.liva$//'`

basedir="`echo $1 | sed 's/\/[^\/]\*$//'`/."

echo -n "$basename..."

echo 1>&2

echo "###### Testing $basename" 1>&2

generatedfiles=""

generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&

RunFail "$LIVA" "<" $1 "2>" "${basename}.err" ">>" $globallog &&

Compare ${basename}.err ${reffile}.err ${basename}.diff

# Report the status and clean up the generated files

if [ $error -eq 0 ] ; then

if [ $keep -eq 0 ] ; then

rm -f $generatedfiles

fi

echo "OK"

echo "###### SUCCESS" 1>&2

else

echo "###### FAILED" 1>&2

globalerror=$erro

fi

}

while getopts kdpsh c; do

case $c in

k) # Keep intermediate files

keep=1

;;

h) # Help

Usage

;;

esac

done

shift `expr $OPTIND - 1`

LLIFail() {

echo "Could not find the LLVM interpreter \"$LLI\"."

echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"

exit 1

}

which "$LLI" >> $globallog || LLIFail

if [ $# -ge 1 ]

then

files=$@

else

files="tests/test-\*.liva tests/fail-\*.liva"

fi

for file in $files

do

case $file in

\*test-\*)

Check $file 2>> $globallog

;;

\*fail-\*)

CheckFail $file 2>> $globallog

;;

\*)

echo "unknown file type $file"

globalerror=1

;;

esac

done

exit $globalerro