Espresso Programming Language Final Report

Somdeep Dey, Rohit Gurunath, Jianfeng Qian, Oliver Willens

December 21, 2016

Contents

1 Introduction

1.1 Overview

Espresso is a hybrid object oriented language and functional language. The ultimate goal of the project is to practice design of a simple language. The compiler is written in OCaml and compiles to Low Level Virtual Machine (LLVM), a multi-platform enabled runtime environment. The basic operators, conditional statements, loops and data types that are found in Java will be supported with Object-oriented behavior and some new lambda related features. Related former projects include Dice(2015), Liva(2016) and Scala(2016).

1.2 Background

Object-oriented programming is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. In Object-oriented programming, computer programs are designed by making them out of objects that interact with one another.There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.

LLVM is a particularly powerful compiler infrastructure that provides a base to build a quick and optimal language.

OCaml, originally known as Objective Caml, is the main implementation of the Caml programming language. OCaml's toolset includes an interactive top-level interpreter, a bytecode compiler, a reversible debugger, a package manager (OPAM), and an optimizing native code compiler. It has a large standard library as well as robust modular and object-oriented programming constructs that make it applicable for large-scale software engineering.

Our goal is to use OCaml and LLVM to utilize this paradigm and model in the process of producing a syntactically Java-like language. Simultaneously, we seek to offer a solution that is simple and easy to use. Implementing inheritance and objects in a c-like context like LLVM allows for fine control over the code.

1.3 Goals of Espresso

Intuition. The overarching goal of Espresso is intuitiveness. We want Espresso programmers to be able to focus on creating cool products, rather than spending a hefty amount oNf time on syntax. This was the inspiration for modeling Espresso on Java, a language that is both intuitive and flexible. We sought to further this attribute by taking aspects of Java that we thought were either overly redundant or unnecessarily long, e.g. `System.out.print()`.

Transparency. The LLVM IR allows the user to witness and understand

what the compiler is doing and ultimately the program they wrote in Espresso is doing.

Broad Application. Our team sought to create a language for general purpose use, rather than choosing a domain-specific language for a certain field or task. The language is portable, running on any platform that enables LLVM. Espresso could potentially serve as a strong platform for learning the essentials of Object-Oriented Programming.

2 Language Tutorial

2.1 Environment Configuration

Espresso requires the OCaml LLVM library which is most easily installed using `opam`.

2.1.1 Ubuntu 15.10 Installation

Run the following commands in a bash session to install LLVM and its development libraries, the m4 macro preprocessor, and opam, and then use opam to install LLVM with the following terminal commands.

```
sudo apt-get install -y ocaml m4 llvm opam
opam init
opam install llvm.3.6 ocamlfind
eval `opam config env`

make
./testall.sh
```

2.1.2 OS X Installation

Install Homebrew and Verify installation

```
ruby -e "$(curl -fsSL
↪  https://raw.githubusercontent.com/Homebrew/install/master/install)"
 brew doctor
```

Install Opam and Conifgure

```
brew install opam
opam init
```

Install LLVM

```
brew install llvm
```

Take note of where brew places the LLVM executables. It will show you the path to them under the CAVEATS section of the post-install terminal output. Also take note of the LLVM version installed.

Set Up Opam Enviornment

```
eval `opam config env`
```

Install OCaml LLVM library

```
opam install llvm.3.6
```

Ensure that the version of LLVM you install here matches the version you installed via brew.

2.2 Run & Test

Position yourself in the root of the **Espresso** directory. First, try and test the compiler using the entire test suite. Start by typing

```
$make espresso
```

Next, execute our pre-fabricated test suite.

```
$./testall.sh
```

2.3 Hello World!

You might start programming in Espresso with the classic "Hello World!" program.
hello.es

```
class hello {

int main() {
        print_string("hello world!");
    return 0;
   }

}
```

This program illustrates the declaration of a class, declaration of a main method, as well as the use of `print_string()`, where in Java a programmer would use `System.out.print()`.

To execute:

1. Compile Espresso, and save **hello.es**.
2. Send the generated LLVM IR code to a **.ll** file.

   ```
   $./espresso.native -l < hello.es > hello.ll
   ```

3. Run the **.ll** file with **lli**.

   ```
   $lli hello.ll
   ```

3 Language Reference Manual

3.1 Primitive Types

3.1.1 int

Integer is a type that stores a supplied value in 4 bytes. Intuitively, the type should be used to store whole decimal number values. Integers can store decimal values ranging from -2-31 to 231-1. Initialization is shown below.

```
int latte;
int mocha;
int cappuccino;

latte = 4;
mocha = 2147483647;
cappuccino = -9;
```

3.1.2 float

Float is a type that stores a supplied value in 4 bytes. Intuitively, the type should be used to store all real numbers to which the Integer type is insufficient. In practical terms, this will mean fractional numbers and numbers of greater magnitude than is supported by Integers. Initialization is shown below.

```
float latte;
float mocha;

latte = 4.5;
mocha = -9.0;
```

3.1.3 bool

The Boolean type is a binary indicator that can be either True or False. Booleans can also be null. Additionally, a boolean can be compared to or initialized as an Integer that is assigned as 0 or 1, or assigned as a 1 or 0.

```
bool latte;
bool coffee;
bool mug;

latte = true;
coffee = false;
mug = (coffee == false); //mug is True
```

3.1.4 char

The Character type is a single alphabetic ASCII character between single quotes. The range is ‘a’ - ‘z’, ‘A’ - ‘Z’. Initialization is shown below.

```
char roast;
roast = 'c';
```

3.1.5 void

The void type is used as a placeholder to imply that a method will not return a value. All methods have to have a return type, so void must be used if there is no concrete return desired. Below is a sample method declaration:

```
void methodName (<formals_opt>) {
 //method body
 //reading comments again??
}
```

3.2 Reference Types

3.2.1 Arrays

Arrays are datatypes that store into memory 0 or more items of a specific type in an indexed manner. Sample array declarations are shown below.

```
int[10] float_arr; //An array of ten Integers
bool[1] bool_arr; //An array of one Boolean value
```

Arrays should be be declared and initialized desperately. A example of this is shown below.

```
int [3] int_arr;
int i;
for(i=0;i<3;i = i + 1)
          int_arr[i] = i;
```

A user can also store values at a specific index later on.

```
int_arr[2] = 5; //The array is now {0, 1, 5}
```

3.2.2 Strings

A string is a class. An instance of a string object contains an array of the primitive datatype char. Espresso supports three methods to manipulate strings: substring(), charAt(), length().

```
String sen;
sen = "Fresh cup of coffee.";
String short_sen;
short_sen = sen.substring(3,6); //short_sen holds "sh c"
//note that substring() is inclusive on both parameters
char myChar;
myChar = sen.charAt(0); //char holds 'F'
int numLetters;
numLetters = sen.length(); //numLetters holds 20
```

3.3 Lexical Conventions

3.3.1 Comments

Supported comments are of two types :

- Single Line Comments

```
//Start typing here
```

- Multi-line Comments

```
/*
None of this matters.
Nothing really matters.
This code, like you, is an insignificant speck.
*/
```

3.3.2 White Space

White spaces in Espresso are comprised of single space characters, tab spaces, page breaks and line ending characters. These are ignored by the Espresso compiler (we'll call it *bru*) with the primary purpose being that of acting as a separator for tokens. That is, one space serves the same purpose as several lines of space.

3.3.3 Reserved Words

Keywords are reserved and cannot be used as regular identifier names.

continue	true	for	float
if	else	int	String
char	break	void	return
else	while	new	class
bool	extends	false	new
this	lambda	break	continue

3.3.4 Identifiers

An identifier is a sequence of letters, digits, and underscores. It can only begin with a lowercase letter. Identifiers are essentially the names of variables, methods, and classes. They are case-sensitive.

3.3.5 Literals

Literals are syntactic depictions of the values of integers, characters, booleans or strings. They indicate the actual representation of values with the program context.

- Boolean literals - Two possible boolean literals :
  - true

  - false

- Integer literals

  These are of the primitive type `int`. They are numeric values that do not comprise any decimal component.

- Floating Point Literals

  These are expressed as decimal fractions and consist of types like 0.56, 1.23, etc.

- Character Literals

  Character literals are contained between a pair of single quotes.

  ```
  'a'
  ```

- String Literals

  String literals start with " , followed by any number of characters and end with " . No newline character can occur with the string unless correctly escaped.

  ```
  "a cup of joe."
  ```

3.3.6 Operators

Operators will include relational, boolean and logical operators, described in greater detail in the expressions section.

3.4 Expressions

A large component of the work in Espresso is done in the form of evaluation of expressions. An example of this is the evaluation of variable assignments of the following type:

```
int a;
a = 10;
```

3.4.1 Evaluation & Result Computation

When an expression is evaluated, the eventual result will be one of the following:

- A variable
- A value or a component of a larger expression
- void - in the instance of void functions, for example.

The single case in which an expression can be nothing (or void) is in the case of its utilization as a return type for a particular method/function that has no return type as it does not return any value on completion of execution. On the other hand, if the expression evaluates to a variable (which falls under the subcategory of an identifier), then within the overall evaluation of the expression, the value of the variable is applied. As such, for both values and variables, expressions evaluate to values , that in themselves may be the final result or may be a component of a larger expression, depending on how they are nested.

3.4.2 Expression Type

Expressions are often used in the form of assignment operations of the following type :

```
x = a + 1;
```

Hence, the evaluation of expressions in Espresso is such that the result is of the same type as the variable it is assigned to. Often the range of operations that can be incorporated in specific expressions are only possible given that they are semantically valid for the types of values/variables that are present in the expression. Example:

```
int  a;
a = 5 % "e";
```

The above code snippet makes no sense in the context of Espresso, as the ‘ % ’ operator holds no meaning in relation to the String “e” , even though ordinarily it would indeed have valid context in relation to a and 5.

3.4.3 Evaluation Order

The evaluation order that is internally employed by Espresso is universally **left-to-right.**

3.4.4 Lexical Literals

Lexical literals indicate the actual representation of fixed, unchanging values that are the smallest unbroken unit that can be evaluated as one, and as such can not be further diluted into expressions. The mapping between the various literals and their corresponding literals are as follows :

- Integer literals map to the **int** datatype.
- Floating literals map to the **float** datatype.
- Boolean literals map to the **bool** datatype.
- Character literals map to the **char** datatype.
- String literals map to the **String** datatype.

Evaluation of a literal is essentially a direct mapping.

3.4.5 Arithmetic Operations

Arithmetic operators include the following :

+	Adds values on both sides
-	Subtracts value on the right from the value on the left
*	Multiplies values on either side
/	Divides left hand operand by the right hand operand
%	Divides left hand operand by the right hand operand and computes the remainder

These are only possible when both operands are primitive types like int or float. They are all binary operands and follow left to right associativity.

3.4.6 Relational Operations

The value on evaluation of any relational expression always results to Boolean. Equality comparisons can only be between similar types.

==	Returns true if both operands are equal
!=	Returns true if the two operands are not equal
>	Returns true if left operand is greater than the right
<	Returns true if right operand is greater than the left
>=	Returns true if left operand greater than or equal to right
<=	Returns true if left operand lesser than or equal to right

3.4.7 Logical Operations

&&	And
||	Or
!	Not

3.4.8 Method invocation

If the invoked method has a return type void, then void (no result) is returned and no return type is expected. Else, the value with the datatype specified in the method signature, is returned. A return statement is expected at the end of the invoked method that has to return a value.

3.4.9 Array Access

Array access is done in the form of an array reference (a variable or identifier) followed by an index (or position) enclosed in square brackets. **The index must be of type int , if not, Espresso will automatically redirect it to the closest integer value less than the specified index (flooring to avoid possibility of hitting an index out of bounds greater than array size).**

3.4.10 Assignment

The assignment operator is '='. This is the only type of assignment supported in Espresso.

3.5 Classes

3.5.1 Class Declaration

A class declaration defines how it is implemented. The declaration has fields and methods. Class Declarations. An Example is shown below.

```
Class  A{
}
```

3.5.2 Constructor Declaration

Espresso does not support traditional constructors. The programmer can only declare fields inside a class, and cannot assign to a field outside of a method.

3.5.3 Field Declaration

Field declaration is an expression.

3.5.4 Method Declaration

```
Class BankCount{
        int saving;
        String name;
        BankCount(class BankCount self, String n, int a){
        self.name = n;
        self.Saving = a;
    }
    bool withdraw(class BankCount self, int a){
            if (a < 0){
                return false;
        }
        else if(self.saving > a){
        self.saving -= a;
        return true;
        }
        else{
        return false;
        }
    }
    bool deposit(class BankCountint a){
    if (a < 0){
            return false;
            }
    self.saving += a;
    return true;
    }
}
```

3.5.5 Lambda Functions

Lambdas are anonymous functions that can be treated like objects. See an example below:

```
class work
{
        int a;
        void main()
        {

                int b;
                int c;
                int d;
                int[10] arr;
                this.a = 100;
                class animal an;
                lambda : char lfunc(char a) { return a; }
                print_char (an.getChar(lfunc));
        }
}
class animal
{
        char b;
        bool x;

        char getChar(lambda lfunc) {
             return #lfunc('a');
        }

        int perform()
        {
                int i;
                i = 5;
                i = 1;
                return i*2;
        }
}
```

3.6 Statements

3.6.1 Expression Statements

An expression statement contains an expression, and ends with a semicolon.

```
expression;
```

3.6.2 Declaration Statement

Declaration Statements can declare a basic type array or class;

```
int a;
float b;
String sentence;
boolean flag;
String [5] sentences;
int [10] datas;
Class  A;
```

3.6.3 Control Flow Statements

If-else
If else flow control can have an else or not.

```
if (expr1) { expr2 };
```

```
if (expr1){
        expr2;
} else{
        expr3;
}
```

Two longer examples:

```
bool  gt (int a, int b){
        if (a >= b)
                return true;
            return   false;
}

Int max (int a, int b){
        if (a >= b)
                 return  a;
     else return  b;
```

Looping: for
For loops work as such:

```
for ( expr1 ; expr2 ; expr3 ){
        expr4;
}
```

An example that prints numbers 0 to 9:

```
int i;
for (int i;i < 10 ;i = i + 1){
         print_int (i);
}
```

An example to print an array.

```
int[5] data;
int i;
for(i=0; i < 5; i = i+1){
        data[i]=i;
    }
for(int item: data ){
        print_int (item);
}
```

Looping: while
While loops work as such:

```
while ( expr1 ){
 Expr2;
}
```

An example that prints 0 to 9:

```
        int i = 0;
        while (i < 10){
                print_int(i);
                i = i + 1;
}
```

3.6.4 Branching

Break
Break interrupts the closest looping. An example: only print the connected positive number for each array. The output should 1,2,5,6, each for one line.

```
int[2][4] m;
int  i;
int j;
for(i = 0; i < 2; i = i + 1)
        for(j=0;j<4;j = j + 1)
       m[i][j] = i * 2 + j;
while (i < 2){
        foreach (int item from m[i]){
                if (item > 0)
                        print_int (item);
            else
                        Break;
        }
    i = i + 1;
}
```

Continue
Continue will pass current the expression of looping. An example prints all positive number for an array.

```
int[4] data;
data = {1, 2,-3,4};
int i;
for ( i = 0; i < 4;i++){
        if(data[i] < 0)
            continue;
         print_int(data[i]);
}
```

Return

Return will return the function - it can return nothing or an expression.

```
return;
return expression;
```

An example which returns the first positive number of an array.

```
int  firstPosiviteNumber(int [] data){
        int i;
        for (int i; i < 4; i++){
                if(data[i] > 0)
                        return data[i];
     return -1;
}
```

3.6.5 Method

The Method statement works as such:

```
returntype functionname(typ a, typ b){
        exprs;
         return returntype;
}
```

3.7 This Pointer

Espresso utilizes pointers that are only accessible with in a class, pointing to the object for which a member function is called. See an example below;

```
class Test {

    int field1;

    int method1() {

    }

    int method2(class Test self) {

    }
```

```
    int method3(class Test self) {
        method2(self);
        print_int(self.field1);
    }
}

class MainClass {

    int field;

    int func1(class MainClass self) {

    }

    int func2() {

    }


    int main() {
        class Test t;
        // method 1 cannot modify any state in Test
        t.method1();

        // method 2 has a context object to
        t.method2(t);

       // this -> pointer to the class where main is defined
        func1(this);
        print_int(this.field);
        func2();
    }
}
```

3.8 Library

Espresso provides basic libraries, including string_buffer, array_list and tree.

3.8.1 string_buffer

string_buffer works like the String of Java. We provide functions such as append(), concat(), and get_length().

```
class string_buffer sb;
class string_buffer sb2;
String str;
String str2;
str = "hell";
```

```
str2 = "world!";

sb.init(sb, str);
print_string("\n");
sb.append(sb, 'o');

sb2.init(sb2, str2);
sb.concat(sb, sb2);
```

3.8.2 array_list

array_list works like the ArrayList of Java. We provide functions such as add() and remove().

```
class int_array_list al;
          int i;
          int count;
          count = 5;
          int [5] arr;

          al.init(al, arr);
          for(i = 0; i < count; i = i + 1) {
                  al.add(al, i);
          }
```

3.9 Empty Statement

Empty Statement is nothing but just semicolon.

```
;
```

3.10 Grammar

Below is an entire grammar listing for Espresso. This is very similar to what is implemented in our scanner. The following are the list of tokens and their associated regexes:

```
(* Ocamllex scanner for Espresso *)

{ open Parser

let unescape s =
            Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun
             ↪ x -> x)
}
let digit =  ['0'-'9']
let charl = ''' ( _? ) '''
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let string = '"' ( (ascii | escape)* as s) '"'
```

```
rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace
  ↪ *)
| "/*"      { comment lexbuf }           (* Comments *)
| "//"            { sincom lexbuf }                (*
↪ Single-Line comments *)
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LSQUARE }                          (* Square
↪ brackets for Arrays *)
| ']'       { RSQUARE }
| ';'       { SEMI }
| ':'       { COLON }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '='       { ASSIGN }
| '%'            { MODULUS }
| '.'      { DOT }
| '#'      { POUND }
| "**"           { POWER }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| '>'       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "foreach" { FOREACH }                        (*
↪ Foreach loop *)
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "void"   { VOID }
| "String" { STRING }
| "float"  { FLOAT }
| "char"   { CHAR }
```

```
| "true"    { TRUE }
| "false"   { FALSE }
| "break"   { BREAK }
| "continue" { CONTINUE }
| "hashmap"{HASHMAP}
| "class"   { CLASS }
| "this"    { THIS }
| "lambda"   { LAMBDA }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as
↪  lxm { ID(lxm) }
| string { STRLIT(unescape s) }
| charl as lxm { CHARLIT(String.get lxm 1) }
| ['0'-'9']+['.']['0'-'9']+ as lxm {
↪  FLOATLIT(float_of_string lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
↪  Char.escaped char)) }

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


and sincom = parse
  "\n" { token lexbuf }
| _    { sincom lexbuf }
```

4 Project Plan

4.1 Process

4.1.1 Planning

Our team focused on regular team meetings twice weekly. We'd meet every Tuesday as team, as well as every Thursday afternoon with our TA, Graham Gobieski. Meetings with Graham were especially helpful to gauge our process - at times it was made clear that we were doing well in terms of time line, and it times it was clear that we should pick up the pace.

4.1.2 Specification

We created an initial specification of the features we hoped to implement in Espresso. From there, we logically devised the building blocks that would be needed to build an Espresso compiler. We knew from the beginning that our language would be heavily based on Java. Initial conversations with Graham Gobieski and Steven Edwards helped us drill down on what features should be focused upon. We all agreed that we wanted arrays, objects, and classes to be the key focuses of our language. During the building of the scanner and parser,

we were able to put a tangible front on the implementation of our language, which solidified our plan.

4.1.3 Development

The path for the compiler's development was always quite clear. The compiler follows a well-defined pipeline, starting from an espresso file through 6-8 programs, and culminating in LLVM IR code. The first hump after submitting the language reference manual was building the scanner and parser. After that, we planned out several weeks for the semantic checker, and in the last month focused on the code generation, test suite, and standard library.

4.1.4 Testing

Each incremental bit of progress in our language was to be tested in the moment. This was unavoidable, and creating a small file to check the most recently implemented feature. Large projects are build on small steps forward. That said, we always knew that we would need to flesh out a comprehensive test suite as we neared the end of our project as to ensure the robustness of our language.

4.2 Style Guide

We agreed upon the following style conventions:

- We used one tab for indentation.
- When writing in OCaml, we used the *in* keyword at the beginning of a line, and usually on its own line
- We would use a multiline comment at the beginning of a large block of OCaml code, in order to segment and understand our code in digestible fragments.
- We put pattern matches on the same indent with respect to one another.
- We kept all lines of code under 120 characters.
- Use underscores for naming variables and methods, as opposed to capital letters within one word.
- We made an effort for meaningful variable and method names that would enhance understanding for the reader.
- We attempted to avoid repetition, never making the same computation twice.

4.3 Project Time-line

Date	Progress
September 28th	Espresso Language Proposal Submitted
October 13th	Git Repository Created, Initial Commit
October 26th	1st edition of Language Reference Manual submitted
November 20th	"Hello World!" Demonstration
November 25th	SAST Created and functional
December 11th	Functions Complete, Classes Complete
December 20th	Final Demonstration Final Submission

4.4 Roles & Responsibilities

The table below shows a generalization of contributions. However, work was completed as a group and each teammate made a large contribution.

Team member	Responsibility
Somdeep Dey	Compiler Front-End, Semantic Check, Code Generator, Standard Library
Rohit Gurunath	Compiler Front-End, Semantic Check, Code Generator, Standard Library
Jianfeng Qian	Test Suite, Standard Library, Final Report
Oliver Willens	Documentation, Standard Library, LRM, Final Report, Latex

4.5 Software Development Environment

We used the following software development environment:

1. **Language:** Ocaml 4.02.1. For the purpose of the compiler's front end, we used Ocamlyacc and Ocamllex, generators for parsing and lexing, respectively. These tools are standard extensions and part of the normal Ocaml distribution.

2. **Operating System:** We all worked on the 64-bit, Ubuntu 15.10 distribution of Linux. Somdeep, Rohit, and Oliver used a virtual machine image provided by the instructors of COMS W4115, while Jianfeng use his native OS, as he already used Ubuntu on his laptop.

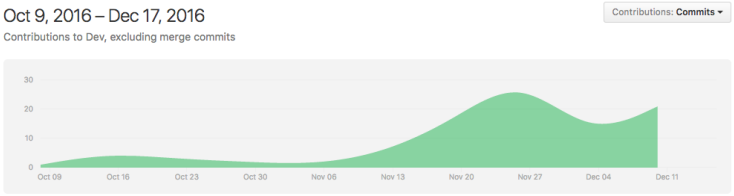
3. **Text Editing:** We all used our favorite text editing application; namely Sublime Text 2, Microsoft's Visual Studio Code, and Vim.

4. **Version Control:** We quickly agreed upon the use of Github.

5. **Typesetting:** We used the LaTeX document preparation system for reporting and documentation. Clean and organized writings helped keep the project on track.

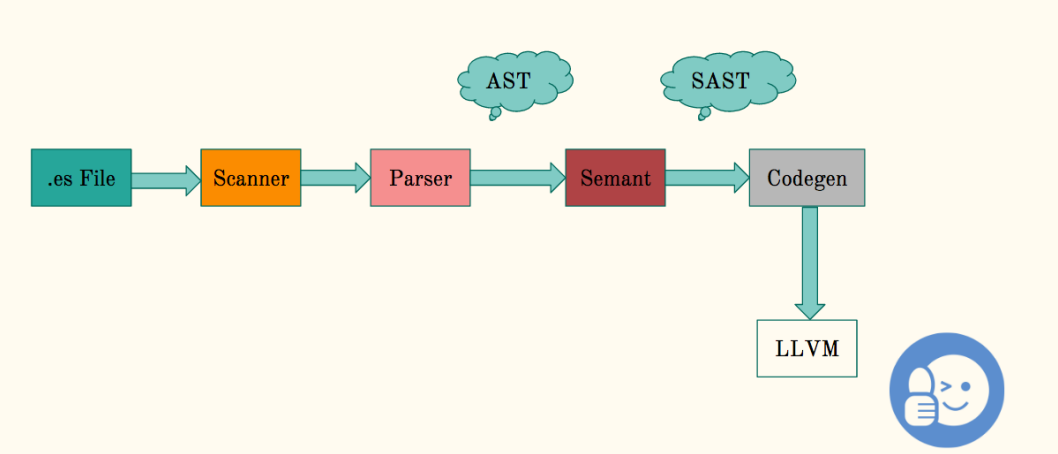
4.6 Project Log

We have included a timeline of our git commits over the course of the semester.



5 Architectural Design

5.1 Overview



The figure above illustrates our compiler's pipeline. Overall, Espresso 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, espresso.ml, semant.ml, parser.mly, scanner.mll and 2 interfaces which are ast.ml and sast.ml. The espresso.ml file is the dance caller, controlling the flow of the compiler.

5.2 scanner.mll

As the start of the front end, the lexical scanner reads an Espresso source file and converts it into lexical tokens, and at the same time it checks whether each token is valid. If the token is invalid, it will report the illegal character. Furthermore, it is also responsible for ignoring white space and comments which are not useful for the execution of an Espresso program. The scanner is built with OCamllex, a lexer generation utility which is part of the OCaml distribution.

5.3 parser.mly

The parser is responsible for producing an Abstract Syntax Tree (AST). The tokens passed by the scanner are interpreted by the parser according to the precedence rules of the Espresso language. The AST is constructed based on the definitions provided and the input tokens are constructed. One of the parser's primary goals 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. If a program passes through the parser successfully, we know that the action-segments in the source

are grammatically correct, although they may still be semantically incorrect. The parser is built with OCamlyacc, a parser generation utility that is part of the OCaml distribution.

5.4 semant.ml

The semantic checker is written in OCaml. It takes in the AST as input and produces a SAST which is type-checked. Semant traverseds the AST recursively, taking each AST node and producing corresponding SAST nodes. We then have, in the SAST, a semantically verified abstract syntax tree. Some of the main tasks of the semantic checker are to:

- Adding reserved functions (such as printing) to the SAST and checking for proper use of reserved functions
- Checking for duplicated fields and methods within the same class
- Checking that the return type of a method matches with its declaration
- Check that an assigned value matches the type of variable it is assigned to
- Check for exactly one main class

5.5 codegen.ml

After the semantic checker generates the semantic abstract syntax tree, control is sent to the the code generator. The high-level purpose here is to take the SAST and produce LLVM immediate representation (IR) code. The IR represents various analogous functions, statements and expressions written by the Espresso programmer. We utilized the LLVM OCaml library, which is comprehensively documented at https://llvm.moe/ocaml/Llvm.html.

6 Test Plan

Our modus operandi for testing followed a similar path to groups that have come before us. The development of our testing suite was two pronged as such:

- **Unit Testing.** Our overall ethos was test-driven development. A test provides a goal. Throughout the process, we built small to medium sized test programs to benchmark the most recently implemented feature.
- **Integration Testing.** As we neared completion of the Espresso compiler, we built up a comprehensive test suite

6.1 Example Programs

We've included five programs that were used to test and demo our language when it was near completion. Also included are the llvm IR code for each program.

6.1.1 99bottles.es

```
class Beer {
      void print_line1(int num_bottles) {
                  if (num_bottles >= 2 && num_bottles <=
                   ↪ 99) {
               print_int(num_bottles);
               print_string(" bottles of beer on the
                ↪ wall,");
               print_int(num_bottles);
               print_string(" bottles of beer.\n");
               return ;
            }
            if (num_bottles == 1) {
               print_int(num_bottles);
               print_string(" bottle of beer on the
                ↪ wall,");
               print_int(num_bottles);
               print_string(" bottle of beer.\n");
               return ;
            }
            if (num_bottles == 0) {
               print_string("No more bottles of beer on
                ↪ the wall, no more bottles of
                ↪ beer.\n");
               return ;
            }
      }

      void print_line2(int num_bottles) {
                  if (num_bottles == 99 ) {
               print_string("Go to the store and buy
                ↪ some more, 99 bottles of beer on the
                ↪ wall.\n");
            } else {
              print_string ("Take one down and pass it
               ↪ around, ");
              if (num_bottles == 1) {
                        print_int(num_bottles);
                        print_string(" bottle of beer on
                         ↪ the wall.\n");
                         return ;
              }
              if (num_bottles == 0) {
                 print_string(" no more bottles of beer
                  ↪ on the wall.\n");
                 return ;
              }
```

```
            if (num_bottles >= 2 && num_bottles <= 98)
             ↪ {
                    print_int(num_bottles);
                    print_string(" bottles of beer on
                     ↪ the wall.\n");
            }
         }
      }

      int main() {
            int num_bottles;
          num_bottles = 99;
          while(num_bottles >= 0) {
             print_line1(num_bottles);
             num_bottles = num_bottles - 1;
             if (num_bottles < 0) {
                    num_bottles = 99;
             }
             print_line2(num_bottles);
             if (num_bottles == 99) {
                    break;
             }
             print_string("\n");
          }
             return 0;
      }
}
```

6.1.2 99bottles.ll

```
; ModuleID = 'Espresso Codegen'

%Lambda = type <{ i32 }>
%Beer = type <{ i32 }>

@tmp = private unnamed_addr constant [3 x i8] c"%d\00"
@tmp.1 = private unnamed_addr constant [30 x i8] c"
 ↪ bottles of beer on the wall,\00"
@tmp.2 = private unnamed_addr constant [3 x i8]
 ↪ c"%s\00"
@tmp.3 = private unnamed_addr constant [3 x i8]
 ↪ c"%d\00"
@tmp.4 = private unnamed_addr constant [19 x i8] c"
 ↪ bottles of beer.\0A\00"
@tmp.5 = private unnamed_addr constant [3 x i8]
 ↪ c"%s\00"
@tmp.6 = private unnamed_addr constant [3 x i8]
 ↪ c"%d\00"
```

```
@tmp.7 = private unnamed_addr constant [29 x i8] c"
↪ bottle of beer on the wall,\00"
@tmp.8 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.9 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.10 = private unnamed_addr constant [18 x i8] c"
↪ bottle of beer.\0A\00"
@tmp.11 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.12 = private unnamed_addr constant [63 x i8] c"No
↪ more bottles of beer on the wall, no more bottles
↪ of beer.\0A\00"
@tmp.13 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.14 = private unnamed_addr constant [68 x i8] c"Go
↪ to the store and buy some more, 99 bottles of beer
↪ on the wall.\0A\00"
@tmp.15 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.16 = private unnamed_addr constant [35 x i8]
↪ c"Take one down and pass it around, \00"
@tmp.17 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.18 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.19 = private unnamed_addr constant [30 x i8] c"
↪ bottle of beer on the wall.\0A\00"
@tmp.20 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.21 = private unnamed_addr constant [39 x i8] c" no
↪ more bottles of beer on the wall.\0A\00"
@tmp.22 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.23 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.24 = private unnamed_addr constant [31 x i8] c"
↪ bottles of beer on the wall.\0A\00"
@tmp.25 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.26 = private unnamed_addr constant [2 x i8]
↪ c"\0A\00"
@tmp.27 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"

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

define void @Beer.print_line1(i32 %num_bottles) {
```

```
entry:
  %lambda_obj = alloca %Lambda
  %sgetmp = icmp sge i32 %num_bottles, 2
  %leqtmp = icmp sle i32 %num_bottles, 99
  %andtmp = and i1 %sgetmp, %leqtmp
  br i1 %andtmp, label %then, label %else

then:                                             ;
 ↪ preds = %entry
  %tmp = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp, i32 0, i32
   ↪ 0), i32 %num_bottles)
  %tmp1 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.2, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([30 x i8], [30 x
   ↪ i8]* @tmp.1, i32 0, i32 0))
  %tmp2 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.3, i32 0, i32
   ↪ 0), i32 %num_bottles)
  %tmp3 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.5, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([19 x i8], [19 x
   ↪ i8]* @tmp.4, i32 0, i32 0))
  ret void
  br label %ifcont

else:                                             ;
 ↪ preds = %entry
  br label %ifcont

ifcont:                                           ;
 ↪ preds = %else, %then
  %eqtmp = icmp eq i32 %num_bottles, 1
  br i1 %eqtmp, label %then4, label %else9

then4:                                            ;
 ↪ preds = %ifcont
  %tmp5 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.6, i32 0, i32
   ↪ 0), i32 %num_bottles)
  %tmp6 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.8, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([29 x i8], [29 x
   ↪ i8]* @tmp.7, i32 0, i32 0))
  %tmp7 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.9, i32 0, i32
   ↪ 0), i32 %num_bottles)
```

```
  %tmp8 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.11, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([18 x i8], [18 x
   ↪ i8]* @tmp.10, i32 0, i32 0))
  ret void
  br label %ifcont10

else9:                                            ;
 ↪ preds = %ifcont
  br label %ifcont10

ifcont10:                                         ;
 ↪ preds = %else9, %then4
  %eqtmp11 = icmp eq i32 %num_bottles, 0
  br i1 %eqtmp11, label %then12, label %else14

then12:                                           ;
 ↪ preds = %ifcont10
  %tmp13 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.13, i32 0, i32 0), i8* getelementptr
   ↪ inbounds ([63 x i8], [63 x i8]* @tmp.12, i32 0,
   ↪ i32 0))
  ret void
  br label %ifcont15

else14:                                           ;
 ↪ preds = %ifcont10
  br label %ifcont15

ifcont15:                                         ;
 ↪ preds = %else14, %then12
  ret void
}

define void @Beer.print_line2(i32 %num_bottles) {
entry:
  %lambda_obj = alloca %Lambda
  %eqtmp = icmp eq i32 %num_bottles, 99
  br i1 %eqtmp, label %then, label %else

then:                                             ;
 ↪ preds = %entry
  %tmp = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.15, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([68 x i8], [68 x
   ↪ i8]* @tmp.14, i32 0, i32 0))
  br label %ifcont17
```

```
else:                                             ;
↪ preds = %entry
  %tmp1 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.17, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([35 x i8], [35 x
   ↪ i8]* @tmp.16, i32 0, i32 0))
  %eqtmp2 = icmp eq i32 %num_bottles, 1
  br i1 %eqtmp2, label %then3, label %else6

then3:                                            ;
↪ preds = %else
  %tmp4 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.18, i32 0, i32
   ↪ 0), i32 %num_bottles)
  %tmp5 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.20, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([30 x i8], [30 x
   ↪ i8]* @tmp.19, i32 0, i32 0))
  ret void
  br label %ifcont

else6:                                            ;
↪ preds = %else
  br label %ifcont

ifcont:                                           ;
↪ preds = %else6, %then3
  %eqtmp7 = icmp eq i32 %num_bottles, 0
  br i1 %eqtmp7, label %then8, label %else10

then8:                                            ;
↪ preds = %ifcont
  %tmp9 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.22, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([39 x i8], [39 x
   ↪ i8]* @tmp.21, i32 0, i32 0))
  ret void
  br label %ifcont11

else10:                                           ;
↪ preds = %ifcont
  br label %ifcont11

ifcont11:                                         ;
↪ preds = %else10, %then8
  %sgetmp = icmp sge i32 %num_bottles, 2
  %leqtmp = icmp sle i32 %num_bottles, 98
```

```
  %andtmp = and i1 %sgetmp, %leqtmp
  br i1 %andtmp, label %then12, label %else15

then12:                                           ;
↪  preds = %ifcont11
  %tmp13 = call i32 (i8*, ...) @printf(i8*
   ↪  getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪  @tmp.23, i32 0, i32 0), i32 %num_bottles)
  %tmp14 = call i32 (i8*, ...) @printf(i8*
   ↪  getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪  @tmp.25, i32 0, i32 0), i8* getelementptr
   ↪  inbounds ([31 x i8], [31 x i8]* @tmp.24, i32 0,
   ↪  i32 0))
  br label %ifcont16

else15:                                           ;
↪  preds = %ifcont11
  br label %ifcont16

ifcont16:                                         ;
↪  preds = %else15, %then12
  br label %ifcont17

ifcont17:                                         ;
↪  preds = %ifcont16, %then
  ret void
}

define i32 @main() {
entry:
  %this = alloca %Beer
  %lambda_obj = alloca %Lambda
  %num_bottles = alloca i32
  store i32 99, i32* %num_bottles
  br label %cond

loop:                                             ;
↪  preds = %cond
  %num_bottles1 = load i32, i32* %num_bottles
  call void @Beer.print_line1(i32 %num_bottles1)
  %num_bottles2 = load i32, i32* %num_bottles
  %subtmp = sub i32 %num_bottles2, 1
  store i32 %subtmp, i32* %num_bottles
  %num_bottles3 = load i32, i32* %num_bottles
  %lesstmp = icmp slt i32 %num_bottles3, 0
  br i1 %lesstmp, label %then, label %else
```

```
then:                                                             ;
↪ preds = %loop
  store i32 99, i32* %num_bottles
  br label %ifcont

else:                                                             ;
↪ preds = %loop
  br label %ifcont

ifcont:                                                           ;
↪ preds = %else, %then
  %num_bottles4 = load i32, i32* %num_bottles
  call void @Beer.print_line2(i32 %num_bottles4)
  %num_bottles5 = load i32, i32* %num_bottles
  %eqtmp = icmp eq i32 %num_bottles5, 99
  br i1 %eqtmp, label %then6, label %else7

then6:                                                            ;
↪ preds = %ifcont
  br label %afterloop
  br label %ifcont8

else7:                                                            ;
↪ preds = %ifcont
  br label %ifcont8

ifcont8:                                                          ;
↪ preds = %else7, %then6
  %tmp = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.27, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([2 x i8], [2 x
   ↪ i8]* @tmp.26, i32 0, i32 0))
  br label %inc

inc:                                                              ;
↪ preds = %ifcont8
  br label %cond

cond:                                                             ;
↪ preds = %inc, %entry
  %num_bottles9 = load i32, i32* %num_bottles
  %sgetmp = icmp sge i32 %num_bottles9, 0
  br i1 %sgetmp, label %loop, label %afterloop

afterloop:                                                        ;
↪ preds = %cond, %then6
  ret i32 0
  ret i32 0
```

```
}
```

6.1.3 array_list.es

```
class int_array_list {
      // datastructure used to represent the arraylist
      int [20] arr;

      // maintains the number of elements currently
      ↪ stored in the arraylist
      int count;

      // maintains the actual size reserved for the
      ↪ arraylist
      int size;

      // constructor - call this method explicitly
      ↪ before invoking any other methods
      void init(class int_array_list self, int [10]
      ↪ array) {
                  self.count = 0;
            self.size = 20;
            self.arr = array;
      }

      // add an element to the array list
      void add(class int_array_list self, int val) {
                  if (self.count == self.size) {
               // resize array -  (since we used 'int'
               ↪ for the arraytype declaration in the
               ↪ parser, we are forced to use a
               ↪ constant here :(
               int [1024] new_arr;

               // copy all elements
               int i;
               for(i = 0; i < self.count; i = i + 1) {
                           new_arr[i] = self.arr[i];
               }

               // switch to the new array
               self.arr = new_arr;
            }

            // at this point, we know that there is
            ↪ space to hold at least one value
            self.arr[self.count] = val;
            self.count = self.count + 1;
      }
```

```
        // remove the element val from the array list -
        ↪ removes only the first occurence
        int remove(class int_array_list self, int val) {
                int i;
            int pos;
            int removed_val;

            pos = -1;
            for(i = 0; i < self.count; i = i + 1) {
                    if (self.arr[i] == val) {
                    pos = i;
                    break;
                }
            }

            if (pos == -1) {
               return -1;
            }
            removed_val = self.arr[pos];

                // fill the every slot with the value
                ↪ from the next adjacent slot
            for(i = pos; i < self.count; i = i + 1) {
                    self.arr[i] = self.arr[i + 1];
            }

            self.count = self.count - 1;
            return removed_val;
        }

}


class Test {

        int main() {
                class int_array_list al;
            int i;
            int count;
            count = 5;
            int [5] arr;

            al.init(al, arr);
            for(i = 0; i < count; i = i + 1) {
                    al.add(al, i);
            }
```

```
        print_string("******** printing from array
        ↪ list ***********\n");
        for(i = 0; i < count; i = i + 1) {
                print_int(al.arr[i]);
              print_string(" ");
        }
        print_string("\n******** removing elements
        ↪ from array list **************\n");

        for(i = 0; i < count; i = i + 1) {
                int removed_val;
              removed_val = al.remove(al, i);
              if (removed_val == i) {
                 print_int(removed\_val);
                 print_string(" ");
              } else {
                print_string("error removing: ");
                printt_int(i);
                print_string("\n");
              }
        }

        print_string("\n");

        return 0;
    }
```

6.1.4 array_list.ll

```
%int_array_list = type <{ i32, i32, i32, i32* }>
%Lambda = type <{ i32 }>
%Test = type <{ i32 }>

@tmp = private unnamed_addr constant [48 x i8]
↪ c"******** printing from array list
↪ ***********\0A\00"
@tmp.1 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.2 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.3 = private unnamed_addr constant [2 x i8] c" \00"
@tmp.4 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.5 = private unnamed_addr constant [60 x i8]
↪ c"\0A******** removing elements from array list
↪ **************\0A\00"
@tmp.6 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
```

```
@tmp.7 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.8 = private unnamed_addr constant [2 x i8] c" \00"
@tmp.9 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.10 = private unnamed_addr constant [17 x i8]
↪ c"error removing: \00"
@tmp.11 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.12 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.13 = private unnamed_addr constant [2 x i8]
↪ c"\0A\00"
@tmp.14 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.15 = private unnamed_addr constant [2 x i8]
↪ c"\0A\00"
@tmp.16 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"

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

define void @int_array_list.init(%int_array_list*
↪ %self, i32* %array) {
entry:
  %lambda_obj = alloca %Lambda
  %count = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 2
  store i32 0, i32* %count
  %size = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 1
  store i32 20, i32* %size
  %arr = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 3
  store i32* %array, i32** %arr
  ret void
}

define void @int_array_list.add(%int_array_list* %self,
↪ i32 %val) {
entry:
  %lambda_obj = alloca %Lambda
  %count = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 2
  %count1 = load i32, i32* %count
  %size = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 1
  %size2 = load i32, i32* %size
```

```
  %eqtmp = icmp eq i32 %count1, %size2
  br i1 %eqtmp, label %then, label %else

then:                                             ;
↪  preds = %entry
  %malloccall = tail call i8* @malloc(i32 mul (i32 add
  ↪  (i32 mul (i32 ptrtoint (i32* getelementptr (i32,
  ↪  i32* null, i32 1) to i32), i32 1024), i32 1), i32
  ↪  ptrtoint (i32* getelementptr (i32, i32* null, i32
  ↪  1) to i32)))
  %tmp = bitcast i8* %malloccall to i32*
  %new_arr = alloca i32*
  store i32* %tmp, i32** %new_arr
  %i = alloca i32
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪  preds = %cond
  %i3 = load i32, i32* %i
  %tmp4 = add i32 %i3, 1
  %new_arr5 = load i32*, i32** %new_arr
  %tmp6 = getelementptr i32, i32* %new_arr5, i32 %tmp4
  %i7 = load i32, i32* %i
  %tmp8 = add i32 %i7, 1
  %arr = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 3
  %arr9 = load i32*, i32** %arr
  %tmp10 = getelementptr i32, i32* %arr9, i32 %tmp8
  %tmp11 = load i32, i32* %tmp10
  store i32 %tmp11, i32* %tmp6
  br label %inc

inc:                                              ;
↪  preds = %loop
  %i12 = load i32, i32* %i
  %addtmp = add i32 %i12, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                             ;
↪  preds = %inc, %then
  %i13 = load i32, i32* %i
  %count14 = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 2
  %count15 = load i32, i32* %count14
  %lesstmp = icmp slt i32 %i13, %count15
  br i1 %lesstmp, label %loop, label %afterloop
```

```
afterloop:                                        ;
↪  preds = %cond
  %arr16 = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 3
  %new_arr17 = load i32*, i32** %new_arr
  store i32* %new_arr17, i32** %arr16
  br label %ifcont

else:                                             ;
↪  preds = %entry
  br label %ifcont

ifcont:                                           ;
↪  preds = %else, %afterloop
  %count18 = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 2
  %count19 = load i32, i32* %count18
  %tmp20 = add i32 %count19, 1
  %arr21 = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 3
  %arr22 = load i32*, i32** %arr21
  %tmp23 = getelementptr i32, i32* %arr22, i32 %tmp20
  store i32 %val, i32* %tmp23
  %count24 = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 2
  %count25 = getelementptr inbounds %int_array_list,
  ↪  %int_array_list* %self, i32 0, i32 2
  %count26 = load i32, i32* %count25
  %addtmp27 = add i32 %count26, 1
  store i32 %addtmp27, i32* %count24
  ret void
}

define i32 @int_array_list.remove(%int_array_list*
↪  %self, i32 %val) {
entry:
  %lambda_obj = alloca %Lambda
  %i = alloca i32
  %pos = alloca i32
  %removed_val = alloca i32
  store i32 -1, i32* %pos
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪  preds = %cond
  %i1 = load i32, i32* %i
```

```
  %tmp = add i32 %i1, 1
  %arr = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 3
  %arr2 = load i32*, i32** %arr
  %tmp3 = getelementptr i32, i32* %arr2, i32 %tmp
  %tmp4 = load i32, i32* %tmp3
  %eqtmp = icmp eq i32 %tmp4, %val
  br i1 %eqtmp, label %then, label %else

then:                                                ;
↪ preds = %loop
  %i5 = load i32, i32* %i
  store i32 %i5, i32* %pos
  br label %afterloop
  br label %ifcont

else:                                                ;
↪ preds = %loop
  br label %ifcont

ifcont:                                              ;
↪ preds = %else, %then
  br label %inc

inc:                                                 ;
↪ preds = %ifcont
  %i6 = load i32, i32* %i
  %addtmp = add i32 %i6, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                                ;
↪ preds = %inc, %entry
  %i7 = load i32, i32* %i
  %count = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 2
  %count8 = load i32, i32* %count
  %lesstmp = icmp slt i32 %i7, %count8
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                           ;
↪ preds = %cond, %then
  %pos9 = load i32, i32* %pos
  %eqtmp10 = icmp eq i32 %pos9, -1
  br i1 %eqtmp10, label %then11, label %else12

then11:                                              ;
↪ preds = %afterloop
```

```
  ret i32 -1
  br label %ifcont13

else12:                                           ;
↪ preds = %afterloop
  br label %ifcont13

ifcont13:                                         ;
↪ preds = %else12, %then11
  %pos14 = load i32, i32* %pos
  %tmp15 = add i32 %pos14, 1
  %arr16 = getelementptr inbounds %int_array_list,
   ↪ %int_array_list* %self, i32 0, i32 3
  %arr17 = load i32*, i32** %arr16
  %tmp18 = getelementptr i32, i32* %arr17, i32 %tmp15
  %tmp19 = load i32, i32* %tmp18
  store i32 %tmp19, i32* %removed_val
  %pos20 = load i32, i32* %pos
  store i32 %pos20, i32* %i
  br label %cond23

loop21:                                           ;
↪ preds = %cond23
  %i25 = load i32, i32* %i
  %tmp26 = add i32 %i25, 1
  %arr27 = getelementptr inbounds %int_array_list,
   ↪ %int_array_list* %self, i32 0, i32 3
  %arr28 = load i32*, i32** %arr27
  %tmp29 = getelementptr i32, i32* %arr28, i32 %tmp26
  %i30 = load i32, i32* %i
  %addtmp31 = add i32 %i30, 1
  %tmp32 = add i32 %addtmp31, 1
  %arr33 = getelementptr inbounds %int_array_list,
   ↪ %int_array_list* %self, i32 0, i32 3
  %arr34 = load i32*, i32** %arr33
  %tmp35 = getelementptr i32, i32* %arr34, i32 %tmp32
  %tmp36 = load i32, i32* %tmp35
  store i32 %tmp36, i32* %tmp29
  br label %inc22

inc22:                                            ;
↪ preds = %loop21
  %i37 = load i32, i32* %i
  %addtmp38 = add i32 %i37, 1
  store i32 %addtmp38, i32* %i
  br label %cond23
```

```
cond23:                                           ;
↪ preds = %inc22, %ifcont13
  %i39 = load i32, i32* %i
  %count40 = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 2
  %count41 = load i32, i32* %count40
  %lesstmp42 = icmp slt i32 %i39, %count41
  br i1 %lesstmp42, label %loop21, label %afterloop24

afterloop24:                                      ;
↪ preds = %cond23
  %count43 = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 2
  %count44 = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %self, i32 0, i32 2
  %count45 = load i32, i32* %count44
  %subtmp = sub i32 %count45, 1
  store i32 %subtmp, i32* %count43
  %removed_val46 = load i32, i32* %removed_val
  ret i32 %removed_val46
}

declare noalias i8* @malloc(i32)

define i32 @main() {
entry:
  %this = alloca %Test
  %lambda_obj = alloca %Lambda
  %al = alloca %int_array_list
  %i = alloca i32
  %count = alloca i32
  store i32 5, i32* %count
  %malloccall = tail call i8* @malloc(i32 mul (i32 add
  ↪ (i32 mul (i32 ptrtoint (i32* getelementptr (i32,
  ↪ i32* null, i32 1) to i32), i32 5), i32 1), i32
  ↪ ptrtoint (i32* getelementptr (i32, i32* null, i32
  ↪ 1) to i32)))
  %tmp = bitcast i8* %malloccall to i32*
  %arr = alloca i32*
  store i32* %tmp, i32** %arr
  %arr1 = load i32*, i32** %arr
  call void @int_array_list.init(%int_array_list* %al,
  ↪ i32* %arr1)
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪ preds = %cond
```

```
  %i2 = load i32, i32* %i
  call void @int_array_list.add(%int_array_list* %al,
  ↪ i32 %i2)
  br label %inc

inc:                                              ;
↪ preds = %loop
  %i3 = load i32, i32* %i
  %addtmp = add i32 %i3, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                             ;
↪ preds = %inc, %entry
  %i4 = load i32, i32* %i
  %count5 = load i32, i32* %count
  %lesstmp = icmp slt i32 %i4, %count5
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                        ;
↪ preds = %cond
  %tmp6 = call i32 (i8*, ...) @printf(i8* getelementptr
  ↪ inbounds ([3 x i8], [3 x i8]* @tmp.1, i32 0, i32
  ↪ 0), i8* getelementptr inbounds ([48 x i8], [48 x
  ↪ i8]* @tmp, i32 0, i32 0))
  store i32 0, i32* %i
  br label %cond9

loop7:                                            ;
↪ preds = %cond9
  %i11 = load i32, i32* %i
  %tmp12 = add i32 %i11, 1
  %arr13 = getelementptr inbounds %int_array_list,
  ↪ %int_array_list* %al, i32 0, i32 3
  %arr14 = load i32*, i32** %arr13
  %tmp15 = getelementptr i32, i32* %arr14, i32 %tmp12
  %tmp16 = load i32, i32* %tmp15
  %tmp17 = call i32 (i8*, ...) @printf(i8*
  ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
  ↪ @tmp.2, i32 0, i32 0), i32 %tmp16)
  %tmp18 = call i32 (i8*, ...) @printf(i8*
  ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
  ↪ @tmp.4, i32 0, i32 0), i8* getelementptr inbounds
  ↪ ([2 x i8], [2 x i8]* @tmp.3, i32 0, i32 0))
  br label %inc8

inc8:                                             ;
↪ preds = %loop7
```

```
  %i19 = load i32, i32* %i
  %addtmp20 = add i32 %i19, 1
  store i32 %addtmp20, i32* %i
  br label %cond9

cond9:                                            ;
↪ preds = %inc8, %afterloop
  %i21 = load i32, i32* %i
  %count22 = load i32, i32* %count
  %lesstmp23 = icmp slt i32 %i21, %count22
  br i1 %lesstmp23, label %loop7, label %afterloop10

afterloop10:                                      ;
↪ preds = %cond9
  %tmp24 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.6, i32 0, i32 0), i8* getelementptr inbounds
   ↪ ([60 x i8], [60 x i8]* @tmp.5, i32 0, i32 0))
  store i32 0, i32* %i
  br label %cond27

loop25:                                           ;
↪ preds = %cond27
  %removed_val = alloca i32
  %i29 = load i32, i32* %i
  %tmp30 = call i32
   ↪ @int_array_list.remove(%int_array_list* %al, i32
   ↪ %i29)
  store i32 %tmp30, i32* %removed_val
  %removed_val31 = load i32, i32* %removed_val
  %i32 = load i32, i32* %i
  %eqtmp = icmp eq i32 %removed_val31, %i32
  br i1 %eqtmp, label %then, label %else

then:                                             ;
↪ preds = %loop25
  %removed_val33 = load i32, i32* %removed_val
  %tmp34 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.7, i32 0, i32 0), i32 %removed_val33)
  %tmp35 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.9, i32 0, i32 0), i8* getelementptr inbounds
   ↪ ([2 x i8], [2 x i8]* @tmp.8, i32 0, i32 0))
  br label %ifcont

else:                                             ;
↪ preds = %loop25
```

```
  %tmp36 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.11, i32 0, i32 0), i8* getelementptr
   ↪ inbounds ([17 x i8], [17 x i8]* @tmp.10, i32 0,
   ↪ i32 0))
  %i37 = load i32, i32* %i
  %tmp38 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.12, i32 0, i32 0), i32 %i37)
  %tmp39 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.14, i32 0, i32 0), i8* getelementptr
   ↪ inbounds ([2 x i8], [2 x i8]* @tmp.13, i32 0, i32
   ↪ 0))
  br label %ifcont

ifcont:                                           ;
 ↪ preds = %else, %then
  br label %inc26

inc26:                                            ;
 ↪ preds = %ifcont
  %i40 = load i32, i32* %i
  %addtmp41 = add i32 %i40, 1
  store i32 %addtmp41, i32* %i
  br label %cond27

cond27:                                           ;
 ↪ preds = %inc26, %afterloop10
  %i42 = load i32, i32* %i
  %count43 = load i32, i32* %count
  %lesstmp44 = icmp slt i32 %i42, %count43
  br i1 %lesstmp44, label %loop25, label %afterloop28

afterloop28:                                      ;
 ↪ preds = %cond27
  %tmp45 = call i32 (i8*, ...) @printf(i8*
   ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
   ↪ @tmp.16, i32 0, i32 0), i8* getelementptr
   ↪ inbounds ([2 x i8], [2 x i8]* @tmp.15, i32 0, i32
   ↪ 0))
  ret i32 0
  ret i32 0
}
```

6.1.5 lambda.es

```
class Test {
```

```
        void print_int_array(int [5] arr, int len) {
                int i;
            for(i = 0; i < len; i = i + 1) {
                    print_int(arr[i]);
                print_string(" ");
            }
            print_string("\n");
        }

        int main() {
                class Sort s;
            int [5] arr;
            int [5] asc;
            int [5] desc;
            int i;

            lambda : bool comp (int x, int y) { if (x <
            ↪ y) { return true; } return false; }

            print_string("..... original array
            ↪ .....\n");
            for(i = 0; i < 5; i = i + 1) {
                    arr[i] = i % 3;
                print_int(arr[i]);
                print_string (" ");
            }

            print_string("\n.....sort
            ↪ descending.....\n");
            asc = s.sort(s, arr, comp);
            print_int_array(asc, 5);


                return 0;
        }

}


class Sort {
        int [5] a;
        int [5] sort(class Sort self, int [5] nums,
        ↪ lambda comp) {
                self.a = nums;
            int i;
            int j;
            int temp;
            int x;
```

```
            int y;

            // bubble sort
            for(i = 0; i < 5; i = i + 1) {
                    for(j = 0; j < 4 - i; j = j + 1) {
                        bool val;
                        x = self.a[j];
                        y = self.a[j + 1];
                        if (#comp(x,y) == true) {
                                temp = self.a[j];
                           self.a[j] = self.a[j + 1];
                           self.a[j + 1] = temp;
                        }
                    }
            }

            return self.a;
        }

}
```

6.1.6 lambda.ll

```
%Sort = type <{ i32, i32* }>
%Lambda = type <{ i32 }>
%Test = type <{ i32 }>

@tmp = private unnamed_addr constant [3 x i8] c"%d\00"
@tmp.1 = private unnamed_addr constant [2 x i8] c" \00"
@tmp.2 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.3 = private unnamed_addr constant [2 x i8]
↪ c"\0A\00"
@tmp.4 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.5 = private unnamed_addr constant [39 x i8]
↪ c".......... original array ..........\0A\00"
@tmp.6 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.7 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.8 = private unnamed_addr constant [2 x i8] c" \00"
@tmp.9 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
@tmp.10 = private unnamed_addr constant [40 x i8]
↪ c"\0A.........sort descending............\0A\00"
@tmp.11 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"
```

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

define i32* @Sort.sort(%Sort* %self, i32* %nums,
↪  %Lambda* %comp) {
entry:
  %lambda_obj = alloca %Lambda
  %a = getelementptr inbounds %Sort, %Sort* %self, i32
  ↪  0, i32 1
  store i32* %nums, i32** %a
  %i = alloca i32
  %j = alloca i32
  %temp = alloca i32
  %x = alloca i32
  %y = alloca i32
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪  preds = %cond
  store i32 0, i32* %j
  br label %cond3

loop1:                                            ;
↪  preds = %cond3
  %val = alloca i1
  %j5 = load i32, i32* %j
  %tmp = add i32 %j5, 1
  %a6 = getelementptr inbounds %Sort, %Sort* %self, i32
  ↪  0, i32 1
  %a7 = load i32*, i32** %a6
  %tmp8 = getelementptr i32, i32* %a7, i32 %tmp
  %tmp9 = load i32, i32* %tmp8
  store i32 %tmp9, i32* %x
  %j10 = load i32, i32* %j
  %addtmp = add i32 %j10, 1
  %tmp11 = add i32 %addtmp, 1
  %a12 = getelementptr inbounds %Sort, %Sort* %self,
  ↪  i32 0, i32 1
  %a13 = load i32*, i32** %a12
  %tmp14 = getelementptr i32, i32* %a13, i32 %tmp11
  %tmp15 = load i32, i32* %tmp14
  store i32 %tmp15, i32* %y
  %x16 = load i32, i32* %x
  %y17 = load i32, i32* %y
  %tmp18 = call i1 @lambda_comp(i32 %x16, i32 %y17)
  %eqtmp = icmp eq i1 %tmp18, true
  br i1 %eqtmp, label %then, label %else
```

```
then:                                                ;
↪ preds = %loop1
  %j19 = load i32, i32* %j
  %tmp20 = add i32 %j19, 1
  %a21 = getelementptr inbounds %Sort, %Sort* %self,
  ↪ i32 0, i32 1
  %a22 = load i32*, i32** %a21
  %tmp23 = getelementptr i32, i32* %a22, i32 %tmp20
  %tmp24 = load i32, i32* %tmp23
  store i32 %tmp24, i32* %temp
  %j25 = load i32, i32* %j
  %tmp26 = add i32 %j25, 1
  %a27 = getelementptr inbounds %Sort, %Sort* %self,
  ↪ i32 0, i32 1
  %a28 = load i32*, i32** %a27
  %tmp29 = getelementptr i32, i32* %a28, i32 %tmp26
  %j30 = load i32, i32* %j
  %addtmp31 = add i32 %j30, 1
  %tmp32 = add i32 %addtmp31, 1
  %a33 = getelementptr inbounds %Sort, %Sort* %self,
  ↪ i32 0, i32 1
  %a34 = load i32*, i32** %a33
  %tmp35 = getelementptr i32, i32* %a34, i32 %tmp32
  %tmp36 = load i32, i32* %tmp35
  store i32 %tmp36, i32* %tmp29
  %j37 = load i32, i32* %j
  %addtmp38 = add i32 %j37, 1
  %tmp39 = add i32 %addtmp38, 1
  %a40 = getelementptr inbounds %Sort, %Sort* %self,
  ↪ i32 0, i32 1
  %a41 = load i32*, i32** %a40
  %tmp42 = getelementptr i32, i32* %a41, i32 %tmp39
  %temp43 = load i32, i32* %temp
  store i32 %temp43, i32* %tmp42
  br label %ifcont

else:                                                ;
↪ preds = %loop1
  br label %ifcont

ifcont:                                              ;
↪ preds = %else, %then
  br label %inc2

inc2:                                                ;
↪ preds = %ifcont
  %j44 = load i32, i32* %j
```

```
  %addtmp45 = add i32 %j44, 1
  store i32 %addtmp45, i32* %j
  br label %cond3

cond3:                                            ;
↪  preds = %inc2, %loop
  %j46 = load i32, i32* %j
  %i47 = load i32, i32* %i
  %subtmp = sub i32 4, %i47
  %lesstmp = icmp slt i32 %j46, %subtmp
  br i1 %lesstmp, label %loop1, label %afterloop4

afterloop4:                                       ;
↪  preds = %cond3
  br label %inc

inc:                                              ;
↪  preds = %afterloop4
  %i48 = load i32, i32* %i
  %addtmp49 = add i32 %i48, 1
  store i32 %addtmp49, i32* %i
  br label %cond

cond:                                             ;
↪  preds = %inc, %entry
  %i50 = load i32, i32* %i
  %lesstmp51 = icmp slt i32 %i50, 5
  br i1 %lesstmp51, label %loop, label %afterloop

afterloop:                                        ;
↪  preds = %cond
  %a52 = getelementptr inbounds %Sort, %Sort* %self,
   ↪  i32 0, i32 1
  %a53 = load i32*, i32** %a52
  ret i32* %a53
}

define void @Test.print_int_array(i32* %arr, i32 %len)
↪  {
entry:
  %lambda_obj = alloca %Lambda
  %i = alloca i32
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪  preds = %cond
  %i1 = load i32, i32* %i
```

```
  %tmp = add i32 %i1, 1
  %tmp2 = getelementptr i32, i32* %arr, i32 %tmp
  %tmp3 = load i32, i32* %tmp2
  %tmp4 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp, i32 0, i32
   ↪ 0), i32 %tmp3)
  %tmp5 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.2, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([2 x i8], [2 x
   ↪ i8]* @tmp.1, i32 0, i32 0))
  br label %inc

inc:                                              ;
 ↪ preds = %loop
  %i6 = load i32, i32* %i
  %addtmp = add i32 %i6, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                             ;
 ↪ preds = %inc, %entry
  %i7 = load i32, i32* %i
  %lesstmp = icmp slt i32 %i7, %len
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                        ;
 ↪ preds = %cond
  %tmp8 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.4, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([2 x i8], [2 x
   ↪ i8]* @tmp.3, i32 0, i32 0))
  ret void
}

define i1 @lambda_comp(i32 %x, i32 %y) {
entry:
  %lambda_obj = alloca %Lambda
  %lesstmp = icmp slt i32 %x, %y
  br i1 %lesstmp, label %then, label %else

then:                                             ;
 ↪ preds = %entry
  ret i1 true
  br label %ifcont

else:                                             ;
 ↪ preds = %entry
  br label %ifcont
```

```
ifcont:                                           ;
 ↪  preds = %else, %then
  ret i1 false
}

define i32 @main() {
entry:
  %this = alloca %Test
  %lambda_obj = alloca %Lambda
  %s = alloca %Sort
  %malloccall = tail call i8* @malloc(i32 mul (i32 add
   ↪  (i32 mul (i32 ptrtoint (i32* getelementptr (i32,
   ↪  i32* null, i32 1) to i32), i32 5), i32 1), i32
   ↪  ptrtoint (i32* getelementptr (i32, i32* null, i32
   ↪  1) to i32)))
  %tmp = bitcast i8* %malloccall to i32*
  %arr = alloca i32*
  store i32* %tmp, i32** %arr
  %malloccall1 = tail call i8* @malloc(i32 mul (i32 add
   ↪  (i32 mul (i32 ptrtoint (i32* getelementptr (i32,
   ↪  i32* null, i32 1) to i32), i32 5), i32 1), i32
   ↪  ptrtoint (i32* getelementptr (i32, i32* null, i32
   ↪  1) to i32)))
  %tmp2 = bitcast i8* %malloccall1 to i32*
  %asc = alloca i32*
  store i32* %tmp2, i32** %asc
  %malloccall3 = tail call i8* @malloc(i32 mul (i32 add
   ↪  (i32 mul (i32 ptrtoint (i32* getelementptr (i32,
   ↪  i32* null, i32 1) to i32), i32 5), i32 1), i32
   ↪  ptrtoint (i32* getelementptr (i32, i32* null, i32
   ↪  1) to i32)))
  %tmp4 = bitcast i8* %malloccall3 to i32*
  %desc = alloca i32*
  store i32* %tmp4, i32** %desc
  %i = alloca i32
  %comp = alloca %Lambda*
  %tmp5 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪  inbounds ([3 x i8], [3 x i8]* @tmp.6, i32 0, i32
   ↪  0), i8* getelementptr inbounds ([39 x i8], [39 x
   ↪  i8]* @tmp.5, i32 0, i32 0))
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
 ↪  preds = %cond
  %i6 = load i32, i32* %i
  %tmp7 = add i32 %i6, 1
```

```
  %arr8 = load i32*, i32** %arr
  %tmp9 = getelementptr i32, i32* %arr8, i32 %tmp7
  %i10 = load i32, i32* %i
  %sremtmp = srem i32 %i10, 3
  store i32 %sremtmp, i32* %tmp9
  %i11 = load i32, i32* %i
  %tmp12 = add i32 %i11, 1
  %arr13 = load i32*, i32** %arr
  %tmp14 = getelementptr i32, i32* %arr13, i32 %tmp12
  %tmp15 = load i32, i32* %tmp14
  %tmp16 = call i32 (i8*, ...) @printf(i8*
  ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
  ↪ @tmp.7, i32 0, i32 0), i32 %tmp15)
  %tmp17 = call i32 (i8*, ...) @printf(i8*
  ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
  ↪ @tmp.9, i32 0, i32 0), i8* getelementptr inbounds
  ↪ ([2 x i8], [2 x i8]* @tmp.8, i32 0, i32 0))
  br label %inc

inc:                                              ;
↪ preds = %loop
  %i18 = load i32, i32* %i
  %addtmp = add i32 %i18, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                             ;
↪ preds = %inc, %entry
  %i19 = load i32, i32* %i
  %lesstmp = icmp slt i32 %i19, 5
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                        ;
↪ preds = %cond
  %tmp20 = call i32 (i8*, ...) @printf(i8*
  ↪ getelementptr inbounds ([3 x i8], [3 x i8]*
  ↪ @tmp.11, i32 0, i32 0), i8* getelementptr
  ↪ inbounds ([40 x i8], [40 x i8]* @tmp.10, i32 0,
  ↪ i32 0))
  %arr21 = load i32*, i32** %arr
  %comp22 = load %Lambda*, %Lambda** %comp
  %tmp23 = call i32* @Sort.sort(%Sort* %s, i32* %arr21,
  ↪ %Lambda* %comp22)
  store i32* %tmp23, i32** %asc
  %asc24 = load i32*, i32** %asc
  call void @Test.print_int_array(i32* %asc24, i32 5)
  ret i32 0
  ret i32 0
```

```
}

declare noalias i8* @malloc(i32)
```

6.1.7 Strings.es

```
class string_buffer {

        char[10] arr;
        int len;
        int size;

        void init(class string_buffer self, String str) {
                    self.arr = str;
             self.len = 0;
             char c;
             c = 'a' - 'a';
             while(self.arr[self.len] != c) {
                     self.len = self.len + 1;
             }
             self.size = self.len;
        }

        int get_length(class string_buffer self) {
                   return self.len;
        }

        void append(class string_buffer self, char ch) {
                    if (self.size == self.len) {
                // increase buffer size
                char[1024] new_arr;
                int i;
                for(i = 0; i < self.len; i = i + 1) {
                            new_arr[i] = self.arr[i];
                }
                new_arr[i] = new_arr[i] - new_arr[i];
                self.arr = new_arr;
             }

             self.arr[self.len] = ch;
             self.len = self.len + 1;
        }

        void print_string_buffer(class string_buffer
        ↪  self) {
                    int i;
             for(i = 0; i < self.len; i = i + 1) {
                      print_char(self.arr[i]);
             }
```

```
            print_string("\n");
        }

        void concat(class string_buffer self, class
        ↪  string_buffer st) {

            int i;
            for(i = 0; i < st.get_length(st); i = i + 1)
            ↪  {
                    append(self, st.arr[i]);
            }
        }

}

class Test {

        int main() {
                class string_buffer sb;
            class string_buffer sb2;
            String str;
            String str2;
            str = "hell";
            str2 = "world!";

            sb.init(sb, str);
                print_int(sb.get_length(sb));
            print_string("\n");
            sb.append(sb, 'o');
            sb.print_string_buffer(sb);

            sb2.init(sb2, str2);
            sb.concat(sb, sb2);
            sb.print_string_buffer(sb);

                return 0;
        }
```

6.1.8 Strings.ll

```
%string_buffer = type <{ i32, i32, i32, i8* }>
%Lambda = type <{ i32 }>
%Test = type <{ i32 }>

@tmp = private unnamed_addr constant [3 x i8] c"%c\00"
@tmp.1 = private unnamed_addr constant [2 x i8]
↪  c"\0A\00"
@tmp.2 = private unnamed_addr constant [3 x i8]
↪  c"%s\00"
```

```
@tmp.3 = private unnamed_addr constant [5 x i8]
↪ c"hell\00"
@tmp.4 = private unnamed_addr constant [7 x i8]
↪ c"world!\00"
@tmp.5 = private unnamed_addr constant [3 x i8]
↪ c"%d\00"
@tmp.6 = private unnamed_addr constant [2 x i8]
↪ c"\0A\00"
@tmp.7 = private unnamed_addr constant [3 x i8]
↪ c"%s\00"

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

define void @string_buffer.init(%string_buffer* %self,
↪ i8* %str) {
entry:
  %lambda_obj = alloca %Lambda
  %arr = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 3
  store i8* %str, i8** %arr
  %len = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  store i32 0, i32* %len
  %c = alloca i8
  store i8 0, i8* %c
  br label %cond

loop:                                             ;
↪ preds = %cond
  %len1 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len2 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len3 = load i32, i32* %len2
  %addtmp = add i32 %len3, 1
  store i32 %addtmp, i32* %len1
  br label %inc

inc:                                              ;
↪ preds = %loop
  br label %cond

cond:                                             ;
↪ preds = %inc, %entry
  %len4 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len5 = load i32, i32* %len4
```

```
  %arr6 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 3
  %arr7 = load i8*, i8** %arr6
  %tmp = getelementptr i8, i8* %arr7, i32 %len5
  %tmp8 = load i8, i8* %tmp
  %c9 = load i8, i8* %c
  %neqtmp = icmp ne i8 %tmp8, %c9
  br i1 %neqtmp, label %loop, label %afterloop

afterloop:                                        ;
↪ preds = %cond
  %size = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 1
  %len10 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len11 = load i32, i32* %len10
  store i32 %len11, i32* %size
  ret void
}

define i32 @string_buffer.get_length(%string_buffer*
↪ %self) {
entry:
  %lambda_obj = alloca %Lambda
  %len = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len1 = load i32, i32* %len
  ret i32 %len1
}

define void @string_buffer.append(%string_buffer*
↪ %self, i8 %ch) {
entry:
  %lambda_obj = alloca %Lambda
  %size = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 1
  %size1 = load i32, i32* %size
  %len = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len2 = load i32, i32* %len
  %eqtmp = icmp eq i32 %size1, %len2
  br i1 %eqtmp, label %then, label %else

then:                                             ;
↪ preds = %entry
  %tmp = tail call i8* @malloc(i32 mul (i32 ptrtoint
  ↪ (i8* getelementptr (i8, i8* null, i32 1) to i32),
  ↪ i32 1024))
```

```
  %new_arr = alloca i8*
  store i8* %tmp, i8** %new_arr
  %i = alloca i32
  store i32 0, i32* %i
  br label %cond

loop:                                                ;
↪  preds = %cond
  %i3 = load i32, i32* %i
  %new_arr4 = load i8*, i8** %new_arr
  %tmp5 = getelementptr i8, i8* %new_arr4, i32 %i3
  %i6 = load i32, i32* %i
  %arr = getelementptr inbounds %string_buffer,
   ↪  %string_buffer* %self, i32 0, i32 3
  %arr7 = load i8*, i8** %arr
  %tmp8 = getelementptr i8, i8* %arr7, i32 %i6
  %tmp9 = load i8, i8* %tmp8
  store i8 %tmp9, i8* %tmp5
  br label %inc

inc:                                                 ;
↪  preds = %loop
  %i10 = load i32, i32* %i
  %addtmp = add i32 %i10, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                                ;
↪  preds = %inc, %then
  %i11 = load i32, i32* %i
  %len12 = getelementptr inbounds %string_buffer,
   ↪  %string_buffer* %self, i32 0, i32 2
  %len13 = load i32, i32* %len12
  %lesstmp = icmp slt i32 %i11, %len13
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                           ;
↪  preds = %cond
  %i14 = load i32, i32* %i
  %new_arr15 = load i8*, i8** %new_arr
  %tmp16 = getelementptr i8, i8* %new_arr15, i32 %i14
  %i17 = load i32, i32* %i
  %new_arr18 = load i8*, i8** %new_arr
  %tmp19 = getelementptr i8, i8* %new_arr18, i32 %i17
  %tmp20 = load i8, i8* %tmp19
  %i21 = load i32, i32* %i
  %new_arr22 = load i8*, i8** %new_arr
  %tmp23 = getelementptr i8, i8* %new_arr22, i32 %i21
```

```
  %tmp24 = load i8, i8* %tmp23
  %subtmp = sub i8 %tmp20, %tmp24
  store i8 %subtmp, i8* %tmp16
  %arr25 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 3
  %new_arr26 = load i8*, i8** %new_arr
  store i8* %new_arr26, i8** %arr25
  br label %ifcont

else:                                             ;
↪ preds = %entry
  br label %ifcont

ifcont:                                           ;
↪ preds = %else, %afterloop
  %len27 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len28 = load i32, i32* %len27
  %arr29 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 3
  %arr30 = load i8*, i8** %arr29
  %tmp31 = getelementptr i8, i8* %arr30, i32 %len28
  store i8 %ch, i8* %tmp31
  %len32 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len33 = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len34 = load i32, i32* %len33
  %addtmp35 = add i32 %len34, 1
  store i32 %addtmp35, i32* %len32
  ret void
}

define void
↪ @string_buffer.print_string_buffer(%string_buffer*
↪ %self) {
entry:
  %lambda_obj = alloca %Lambda
  %i = alloca i32
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪ preds = %cond
  %i1 = load i32, i32* %i
  %arr = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 3
  %arr2 = load i8*, i8** %arr
```

```
  %tmp = getelementptr i8, i8* %arr2, i32 %i1
  %tmp3 = load i8, i8* %tmp
  %tmp4 = call i32 (i8*, ...) @printf(i8* getelementptr
  ↪ inbounds ([3 x i8], [3 x i8]* @tmp, i32 0, i32
  ↪ 0), i8 %tmp3)
  br label %inc

inc:                                              ;
↪ preds = %loop
  %i5 = load i32, i32* %i
  %addtmp = add i32 %i5, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                             ;
↪ preds = %inc, %entry
  %i6 = load i32, i32* %i
  %len = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %self, i32 0, i32 2
  %len7 = load i32, i32* %len
  %lesstmp = icmp slt i32 %i6, %len7
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                        ;
↪ preds = %cond
  %tmp8 = call i32 (i8*, ...) @printf(i8* getelementptr
  ↪ inbounds ([3 x i8], [3 x i8]* @tmp.2, i32 0, i32
  ↪ 0), i8* getelementptr inbounds ([2 x i8], [2 x
  ↪ i8]* @tmp.1, i32 0, i32 0))
  ret void
}

define void @string_buffer.concat(%string_buffer*
↪ %self, %string_buffer* %st) {
entry:
  %lambda_obj = alloca %Lambda
  %i = alloca i32
  store i32 0, i32* %i
  br label %cond

loop:                                             ;
↪ preds = %cond
  %i1 = load i32, i32* %i
  %arr = getelementptr inbounds %string_buffer,
  ↪ %string_buffer* %st, i32 0, i32 3
  %arr2 = load i8*, i8** %arr
  %tmp = getelementptr i8, i8* %arr2, i32 %i1
  %tmp3 = load i8, i8* %tmp
```

```
  call void @string_buffer.append(%string_buffer*
  ↪ %self, i8 %tmp3)
  br label %inc

inc:                                              ;
↪ preds = %loop
  %i4 = load i32, i32* %i
  %addtmp = add i32 %i4, 1
  store i32 %addtmp, i32* %i
  br label %cond

cond:                                             ;
↪ preds = %inc, %entry
  %i5 = load i32, i32* %i
  %tmp6 = call i32
  ↪ @string_buffer.get_length(%string_buffer* %st)
  %lesstmp = icmp slt i32 %i5, %tmp6
  br i1 %lesstmp, label %loop, label %afterloop

afterloop:                                        ;
↪ preds = %cond
  ret void
}

declare noalias i8* @malloc(i32)

define i32 @main() {
entry:
  %this = alloca %Test
  %lambda_obj = alloca %Lambda
  %sb = alloca %string_buffer
  %sb2 = alloca %string_buffer
  %str = alloca i8*
  %str2 = alloca i8*
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
  ↪ @tmp.3, i32 0, i32 0), i8** %str
  store i8* getelementptr inbounds ([7 x i8], [7 x i8]*
  ↪ @tmp.4, i32 0, i32 0), i8** %str2
  %str1 = load i8*, i8** %str
  call void @string_buffer.init(%string_buffer* %sb,
  ↪ i8* %str1)
  %tmp = call i32
  ↪ @string_buffer.get_length(%string_buffer* %sb)
  %tmp2 = call i32 (i8*, ...) @printf(i8* getelementptr
  ↪ inbounds ([3 x i8], [3 x i8]* @tmp.5, i32 0, i32
  ↪ 0), i32 %tmp)
```

```
  %tmp3 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪ inbounds ([3 x i8], [3 x i8]* @tmp.7, i32 0, i32
   ↪ 0), i8* getelementptr inbounds ([2 x i8], [2 x
   ↪ i8]* @tmp.6, i32 0, i32 0))
  call void @string_buffer.append(%string_buffer* %sb,
   ↪ i8 111)
  call void
   ↪ @string_buffer.print_string_buffer(%string_buffer*
   ↪ %sb)
  %str24 = load i8*, i8** %str2
  call void @string_buffer.init(%string_buffer* %sb2,
   ↪ i8* %str24)
  call void @string_buffer.concat(%string_buffer* %sb,
   ↪ %string_buffer* %sb2)
  call void
   ↪ @string_buffer.print_string_buffer(%string_buffer*
   ↪ %sb)
  ret i32 0
  ret i32 0
}
```

6.1.9 tree.es

```
class Node
{
        class Node left;
        class Node right;
        int val;

        void init(class Node self,class Node null)
        {
                self.left = null;
                self.right = null;
                self.val = 1;

        }
}


class Tree
{


        void main()
        {
                class Node root;
                class Node null;
                null.val = -1;
                root.init(root,null);
```

```
        root.val = 1;
        //print_int(root.val);
        class Node root_left;
        root_left.init(root_left,null);
        root.left = root_left;
        class Node root_right;
        root_right.init(root_right,null);
        root.right = root_right;
        root_left.val = 2;
        root_right.val = 3;
        class Node root_far_left;
        class Node root_far_right;
        root_far_left.init(root_far_left,null);
        root_far_right.init(root_far_right,null);
        root_far_left.val = 4;
        root_far_right.val = 5;
        root_left.left = root_far_left;
        root_right.right = root_far_right;
        //print_int(root_left.val);
        print_string("\n");
        this.traverse(root);

        class Node test;
        test = root_left;
        /*if(test == root.left)
        {
                print_string("Equal");
        }        */

}


void traverse(class Node root)
{
        class Node x;
        x=root;

        if(x.val == -1)
        {
                return;
        }
        traverse(x.left);
        print_int(x.val);
        traverse(x.right);
}
```

```
}
```

6.1.10 tree.ll

```
%Node = type <{ i32, i32, %Node*, %Node* }>
%Lambda = type <{ i32 }>
%Tree = type <{ i32 }>

@tmp = private unnamed_addr constant [3 x i8] c"%d\00"
@tmp.1 = private unnamed_addr constant [2 x i8]
 ↪  c"\0A\00"
@tmp.2 = private unnamed_addr constant [3 x i8]
 ↪  c"%s\00"

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

define void @Tree.traverse(%Node* %root) {
entry:
  %lambda_obj = alloca %Lambda
  %x = alloca %Node
  %tmp = load %Node, %Node* %root
  store %Node %tmp, %Node* %x
  %val = getelementptr inbounds %Node, %Node* %x, i32
   ↪  0, i32 1
  %val1 = load i32, i32* %val
  %eqtmp = icmp eq i32 %val1, -1
  br i1 %eqtmp, label %then, label %else

then:                                             ;
 ↪  preds = %entry
  ret void
  br label %ifcont

else:                                             ;
 ↪  preds = %entry
  br label %ifcont

ifcont:                                           ;
 ↪  preds = %else, %then
  %left = getelementptr inbounds %Node, %Node* %x, i32
   ↪  0, i32 3
  %left2 = load %Node*, %Node** %left
  call void @Tree.traverse(%Node* %left2)
  %val3 = getelementptr inbounds %Node, %Node* %x, i32
   ↪  0, i32 1
  %val4 = load i32, i32* %val3
  %tmp5 = call i32 (i8*, ...) @printf(i8* getelementptr
   ↪  inbounds ([3 x i8], [3 x i8]* @tmp, i32 0, i32
   ↪  0), i32 %val4)
```

```
  %right = getelementptr inbounds %Node, %Node* %x, i32
  ↪ 0, i32 2
  %right6 = load %Node*, %Node** %right
  call void @Tree.traverse(%Node* %right6)
  ret void
}

define void @Node.init(%Node* %self, %Node* %null) {
entry:
  %lambda_obj = alloca %Lambda
  %left = getelementptr inbounds %Node, %Node* %self,
  ↪ i32 0, i32 3
  store %Node* %null, %Node** %left
  %right = getelementptr inbounds %Node, %Node* %self,
  ↪ i32 0, i32 2
  store %Node* %null, %Node** %right
  %val = getelementptr inbounds %Node, %Node* %self,
  ↪ i32 0, i32 1
  store i32 1, i32* %val
  ret void
}

define i32 @main() {
entry:
  %this = alloca %Tree
  %lambda_obj = alloca %Lambda
  %root = alloca %Node
  %null = alloca %Node
  %val = getelementptr inbounds %Node, %Node* %null,
  ↪ i32 0, i32 1
  store i32 -1, i32* %val
  call void @Node.init(%Node* %root, %Node* %null)
  %val1 = getelementptr inbounds %Node, %Node* %root,
  ↪ i32 0, i32 1
  store i32 1, i32* %val1
  %root_left = alloca %Node
  call void @Node.init(%Node* %root_left, %Node* %null)
  %left = getelementptr inbounds %Node, %Node* %root,
  ↪ i32 0, i32 3
  store %Node* %root_left, %Node** %left
  %root_right = alloca %Node
  call void @Node.init(%Node* %root_right, %Node*
  ↪ %null)
  %right = getelementptr inbounds %Node, %Node* %root,
  ↪ i32 0, i32 2
  store %Node* %root_right, %Node** %right
  %val2 = getelementptr inbounds %Node, %Node*
  ↪ %root_left, i32 0, i32 1
```

```
  store i32 2, i32* %val2
  %val3 = getelementptr inbounds %Node, %Node*
  ↪ %root_right, i32 0, i32 1
  store i32 3, i32* %val3
  %root_far_left = alloca %Node
  %root_far_right = alloca %Node
  call void @Node.init(%Node* %root_far_left, %Node*
  ↪ %null)
  call void @Node.init(%Node* %root_far_right, %Node*
  ↪ %null)
  %val4 = getelementptr inbounds %Node, %Node*
  ↪ %root_far_left, i32 0, i32 1
  store i32 4, i32* %val4
  %val5 = getelementptr inbounds %Node, %Node*
  ↪ %root_far_right, i32 0, i32 1
  store i32 5, i32* %val5
  %left6 = getelementptr inbounds %Node, %Node*
  ↪ %root_left, i32 0, i32 3
  store %Node* %root_far_left, %Node** %left6
  %right7 = getelementptr inbounds %Node, %Node*
  ↪ %root_right, i32 0, i32 2
  store %Node* %root_far_right, %Node** %right7
  %tmp = call i32 (i8*, ...) @printf(i8* getelementptr
  ↪ inbounds ([3 x i8], [3 x i8]* @tmp.2, i32 0, i32
  ↪ 0), i8* getelementptr inbounds ([2 x i8], [2 x
  ↪ i8]* @tmp.1, i32 0, i32 0))
  call void @Tree.traverse(%Node* %root)
  %test = alloca %Node
  %tmp8 = load %Node, %Node* %root_left
  store %Node %tmp8, %Node* %test
  ret i32 0
}
```

6.2 Automation

Testing was made easy using a bash shell script, testall.sh, which is presented below

```
#!/bin/sh

# Regression testing script for ESPRESSO
# 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 interpreter
LLI="lli"
```

```
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the espresso compiler.  Usually
↪  "./espresso.native"
# Try "_build/expresso.native" if ocamlbuild was unable
↪  to create a symbolic link.
ESPRESSO="./espresso.native"
#ESPRESSO="_build/espresso.native"

# Set time limit for all operations
ulimit -t 30
TMP_DIR="_tmp2"
mkdir -p ${TMP_DIR}
#rm ${TMP_DIR}/*.*
globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.es 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 error
RunFail() {
    echo $* 1>&2
    eval $* && {
        SignalError "failed: $* did not report an
        ↪ error"
        return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.es$//'`
    reffile=`echo $1 | sed 's/.es$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""
    generatedfiles="$generatedfiles
    ↪ ${TMP_DIR}/${basename}.ll
    ↪ ${TMP_DIR}/${basename}.out" &&
    Run "$ESPRESSO" "-l <" $1 ">"
    ↪ "${TMP_DIR}/${basename}.ll" &&
    Run "$LLI" "${TMP_DIR}/${basename}.ll" ">"
    ↪ "${TMP_DIR}/${basename}.out" &&
    Compare ${TMP_DIR}/${basename}.out ${reffile}.out
    ↪ ${TMP_DIR}/${basename}.diff

    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=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.es$//'`
    reffile=`echo $1 | sed 's/.es$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."


    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles
    ↪ ${TMP_DIR}/${basename}.err
    ↪ ${TMP_DIR}/${basename}.diff" &&
    RunFail "$ESPRESSO" "<" $1 "2>"
    ↪ "${TMP_DIR}/${basename}.err" ">>" $globallog &&
    Compare ${TMP_DIR}/${basename}.err ${reffile}.err
    ↪ ${TMP_DIR}/${basename}.diff

    # Report the status and clean up the generated
    ↪ files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "###### SUCCESS" 1>&2
    else
        echo "###### FAILED" 1>&2
        globalerror=$error
    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_*.es tests/fail_*.es"
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 $globalerror
```

6.3 Test suites

6.3.1 test_arith_int.es

```
class Test {

int main(int a) {
    int x;
    x = 2 + 5 * 4 % 9 - 8 / 2;
        print_int(x);
    return 0;
   }
}
```

6.3.2 test_arith_float.es

```
class Test {

int main(int a) {
    float x;
    x = 2.0 + 5.0 * 4.0 - 6.0 / 2.0;
        print_float(x);
    return 0;
   }
}
```

6.3.3 test_array.es

```
class Test {

  int main(int y) {
    int [10]arr;
    arr[0]=0;
    print_int(arr[0]);
    return 0;
   }
}
```

6.3.4 test_array_assign2array.es

```
class Test {

  int main(int y) {
    int [10]arr;
        arr[5]=5;
    int [10]arr2;
        arr2 = arr;
    print_int(arr[5]);
    return 0;
```

```
    }
}
```

6.3.5 test_break.es

```
class test_return
{
        int abc()
        {
                int j;
                for(j=1;j<10;j=j+1)
                {
                        int i;
                        if (j == 5) {
                           break;
                        }
                }
                return j;
        }

        int main(int a)
        {
                class test_return obj;
                print_int(obj.abc());
                return 0;
        }
}
```

6.3.6 test_comments.es

```
class Test {

int main(int a) {
    int x;
    /*Expresso
          Times up!@#$%^&*()
      print_int(0);
        */
    print_int(0);
        return 0;
   }
}
```

6.3.7 test_comments2.es

```
class Test {

int main(int a) {
    //print_int(0);
```

```
    print_int(0);
        return 0;
    }
}
```

6.3.8 test_equal.es

```
class Test {

int main(int y) {
    int b;
        b = 10;
        int z ;
        z = 1;
        if(b == 10)
                z = 2;
        print_int(z);
    return 0;
    }
}
```

6.3.9 test_float_add.es

```
class Test {

  int main(int y) {
    float x;
    float y;
    x = 1.0;
    y = 2.0;
    float z;
    z = x+y;
        print_float(z);
    return 0;
    }
}
```

6.3.10 test_for.es

```
class test_return
{
        int abc()
        {
                int j;
                for(j=1;j<10;j=j+1)
                {
                        int i;
                        if (j == 5) {
                           break;
```

```
                        }
                }
                return j;
        }

        int main()
        {
                int a;
                class test_return obj;
                print_int(obj.abc());
                return 0;
        }
}
```

6.3.11 test_for_nest_while.es

```
class Test {

int main(int a) {
    int x;
    int y;
        for(x =0; x <10;x = x +1) {
            x = x +2;
            y = 10;
            while(y < 20){
                    if((x==2)&&(y == 15))
                            print_int(x+y);
                    y = y + 1 ;
            }
        }
    return 0;
   }
}
```

6.3.12 test_for_nested.es

```
class Test {

int main(int a) {
    int x;
    int y;
    int z;
    z = 0;
        for(x =0; x <10;x = x + 1) {
            for(y = 0; y < 5 ; y = y+1){
                    z = z + 1 ;
            }
        }
        print_int(z);
```

```
    return 0;
  }
}
```

6.3.13 test_function.es

```
class Test {
        int plus(int x, int y){
                return x+y;
        }
          int main(int y) {
                  int z;
                  z = plus(2,3);
                print_int(z);
                  return 0;
  }
}
```

6.3.14 test_gcd.es

```
class Test {

 int gcd(int x, int y){
         if(x < y)
                  return gcd(y,x);
         if( y == 0)
                  return x;
         int z;
         z = x % y;
         return gcd(y,z);
}

int main(int a) {
    int x;
    int y;
    x = 12;
    y = 8;
        print_int(gcd(x,y));
        return 0;
  }
}
```

6.3.15 test_geq.es

```
class Test {

int main(int y) {

    int b;
```

```
        b = 10;
        int z ;
        z = 1;
        if(b >= 10)
                z = 2;
    print_int(z);
    return 0;
   }
}
```

6.3.16 test_geq2.es

```
class Test {

int main(int y) {

    int b;
        b = 10;
        int z ;
        z = 1;
        if(10 >= b)
                z = 2;
    return 0;
   }
}
```

6.3.17 test_gt.es

```
class Test {

int main(int y) {

    int b;
        b = 10;
        int z ;
        z = 1;
        if(11 > b)
                z = 2;
        print_int(z);
    return 0;
   }
}
```

6.3.18 test_gt2.es

```
class Test {

int main(int y) {
```

```
    int b;
        b = 10;
        int z ;
        z = 1;
        if(b > 9)
                z = 2;
        print_int(z);
    return 0;
   }
}
```

6.3.19 test_hello_world.es

```
class Test {

int main() {
        String s;
        s = "hello,world";
        print_string(s);
    return 0;
   }
}
```

6.3.20 test_id_assign.es

```
class Test {
  int x;
  int main() {
     this.x = 0;
     print_int(this.x);
     return 0;
   }
}
```

6.3.21 test_if.es

```
class Test {

int main(int a) {
    bool x;
    x = true;
    int z;
    z = 1;
    if(x)
            z =2;
    print_int(z);
    return 0;
   }
}
```

6.3.22 test_if2.es

```
class Test {

int main(int a) {
    bool x;
    x = true;
    bool y;
    y = true;
    int z;
    z = 1;
    if(x)
            z = 2;
    if (y)
            z = 3;
    print_int(z);
    return 0;
   }
}
```

6.3.23 test_if_else.es

```
class Test {

int main(int a) {
    bool x;
    x = false;
    int z;
    z = 0;
    if(x)
            z = 1;
    else
            z = 2;
    print_int(z);
    return 0;
   }
}
```

6.3.24 test_if_nested.es

```
class Test {

int main(int a) {
    bool x;
    x = true;
    bool y;
    y = true;
    int z;
    z = 1;
```

```
    if(x){
            z = z+2;
            if (y) {
                    z = z+3;
            }
    }
    print_int(z);
    return 0;
   }
}
```

6.3.25 test_int_add.es

```
class Test {

int main(int a) {
    int x;
    int y;
    x = 1;
    y = 2;
    int z;
    z = x+y;
        print_int(z);
    return 0;
   }
}
```

6.3.26 test_lambda_call.es

```
class work
{
        int a;
        void main()
        {

                int b;
                int c;
                int d;
                int[10] arr;
                this.a = 100;
                class animal an;
                lambda : char lfunc(char a) { return a; }
                print_char (an.getChar(lfunc));
        }
}
class animal
{
        char b;
        bool x;
```

```
        char getChar(lambda lfunc) {
             return #lfunc('a');
        }

        int perform()
        {
                int i;
                i = 5;
                i = 1;
                return i*2;
        }
}
```

6.3.27 test_lt.es

```
class Test {

int main(int y) {
     int b;
         b = 10;
         int z ;
         z = 1;
         if(11 < b)
                 z = 2;
         print_int(z);
     return 0;
    }
}
```

6.3.28 test_lt2.es

```
class Test {

int main(int y) {

     int b;
         b = 10;
         int z ;
         z = 1;
         if(b < 11)
                 z = 2;
         print_int(z);
     return 0;
    }
}
```

6.3.29 test_main_obj_access.es

```
class Work {

       int work_field;
       class Test test_obj;
       int mem_func(char c) {
                  int x ;
                  x = 5;
              return x;
       }
}

class Test {
  int x;

  int main(int y) {
     class Work w;
     w.work_field =  w.mem_func('a');
           print_int(w.work_field);
     return 0;
   }
}
```

6.3.30 test_new_array.es

```
class Test {

  int main(int y) {
    int [10]arr;
        arr[0]=5;
    int [10]arr2;
        arr2 = arr;
    print_int(arr2[0]);
    return 0;
   }
}
```

6.3.31 test_op_and.es

```
class Test {

int main(int a) {
    int x;
    int y;
    x = 1;
    y = 2;
    bool z;
    if(x == 1 && y ==2)
```

```
                print_int(3);
        else
                print_int(0);
        return 0;
    }
}
```

6.3.32 test_op_not.es

```
class Test {

int main(int a) {
        int x;
        x = 1;
        if(x == 1)
                print_int(0);
        else
                print_int(1);
        return 0;
    }
}
```

6.3.33 test_op_or.es

```
class Test {

int main(int a) {
        int x;
        int y;
        x = 1;
        y = 2;
        bool z;
        if(x == 1 || y ==2)
                print_int(3);
        else
                print_int(0);
        return 0;
    }
}
```

6.3.34 test_return.es

```
class test_return
{
        int abc()
        {
                return 1;
        }
```

```
        int main()
        {
                int a;
                class test_return obj;
                print_string(a());
                return 0;
        }

        String a()
        {
                return "a";
        }
}
```

6.3.35 test_return1.es

```
class test_return
{
        int main()
        {
                int a;
                class work obj;
                print_int(obj.handle());
                return 0;
        }
}

class work
{
        int handle()
        {
                return 1;
        }
}
```

6.3.36 test_return2.es

```
class test_return
{
        int abc()
        {
                return 2;
        }

        int main()
        {
                int a;
                class test_return obj;
                print_int(obj.abc());
```

```
                return 0;
        }
}
```

6.3.37 test_return3.es

```
class test_return
{
        int abc()
        {
                return 2;
        }

        int main()
        {
                int a;

                print_int(this.abc());
                return 0;
        }

        String a()
        {
                return "a";
        }
}
```

6.3.38 test_while.es

```
class test_while
{
        int abc()
        {
                int i;
                i = 0;
                int sum;
                sum = 0;
                while(i<5)
                {
                        sum = sum + i;
                        i = i + 1;
                }
                return sum;
        }

        int main()
        {
                int a;
                class test_while obj;
```

```
                print_int(obj.abc());
                return 0;
        }
}
```

6.3.39 test_while_nest_for.es

```
class Test {

int main(int a) {
    int x;
    x = 0;
        while(x < 20) {
            x = x +2;
            int y;
            y =2;
            for(y =0; y <10;y = y +1){
                    if((x==2) && (y ==0))
                            print_int(x+y);
                        y = y + 1 ;

            }
        }
    return 0;
   }
}
```

6.3.40 test_while_nested.es

```
class Test {

int main(int a) {
    int x;
    x = 77;
    int y;
    y = 22;
        while (x > 0) {

            x = x - 8;
            while( y < 40){
                    if(x == 69 && y == 22)
                            print_int(x+y);
                    y = y - 5;
            }
        }
    return 0;
   }
}
```

6.4 Tests Results

The test cases results are as following:

test_arith_float...OK
test_arith_int...OK
test_array...OK
test_array_assign2array...OK
test_array_assign_values...OK
test_array_int_access...OK
test_assign...OK
test_break...OK
test_comments...OK
test_equal...OK
test_float_add...OK
test_for...OK
test_for_nest_while...OK
test_for_nested...OK
test_function...OK
test_gcd...OK
test_geq...OK
test_geq2...OK
test_gt...OK
test_gt2...OK
test_hello_world...OK
test_id_assign...OK
test_if...OK
test_if1...OK
test_if2...OK
test_if_else...OK
test_if_nested...OK
test_int_add...OK
test_lambda_call...OK
test_lt...OK
test_lt2...OK
test_main_obj_access...OK
test_new_array...OK
test_op_and...OK
test_op_not...OK
test_op_or...OK
test_return...OK
test_return1...OK
test_return2...OK
test_return3...OK
test_while...OK
test_while_nest_for...OK
test_while_nested...OK
fail_array...OK
fail_array_access...OK
fail_assign...OK

fail_assign_int2float...OK
fail_assign_not_declare...OK
fail_break...OK
fail_comments...OK
fail_foreach_1...OK
fail_foreach_2...OK
fail_function_wrongname...OK
fail_function_wrongparams...OK
fail_if_else...OK
fail_if_wrong_type...OK
fail_multi_array...OK
fail_op_and...OK
fail_op_not_char...OK
fail_op_not_int...OK
fail_op_or...OK
fail_op_string_div_int...OK
fail_op_string_minus_int...OK
fail_op_string_mod_int...OK
fail_op_string_multi_int...OK
fail_op_string_plus_int...OK
fail_plus_int_char...OK
fail_return...OK
fail_string_array...OK
fail_unop...OK
fail_while...OK

6.5 Individual Responsibility

Jianfeng was our primary testing guru, and can take most of the credit for our testing suite. That said, all member of the team participated in testing throughout the development process as a natural part of coding.

7 Lessons Learned

7.1 Oliver Willens

The hardest part of this class is coding in unfamiliar terrain. It was fitting that we were attempting to design an intuitive language whilst struggling with a nerve-inducing lack of intuition. I wish that I had taken time to be comfortable in Ocaml very early in the semester. I wish that I knew this LLVM module guide existed a bit earlier so that I could familiarize myself.

The other extremely difficult part of the project is group dynamics. A group of 4 or 5 students is bound to have significant differences in courseload, schedules, and coding ability. My previous group coding experiences had been in summer internships where the experience is vastly different - a team that sits next to each other every day for nine hours. This situation is different, some people get ahead which makes it hard to catch up, especially in the unfamiliar terrain of OCaml and LLVM. The only partial solution here is extremely consistent communication and a commitment to go above and beyond to find times to meet.

7.2 Jianfeng Qian

The project is great help to learn knowledge of compiler. Using a language to write code is easy, but to design a elegant language take much more efforts.

LLVM is powerful, but it is also hard to learn.

To start the project as soon as possible and make a detailed project plan, because there are a lot of work to do.

Last lesson is team members matter. I am lucky to have 3 great team members, they all did great job and we worked together to move the progress steadily.

7.3 Rohit Gurunath

This project has been a highly challenging and a rewarding one. One of the biggest challenges has been adapting to OCaml, its syntax and the general nature of functional programming, as well as learning LLVM, its complexities, the power and the risks associated with low-level programming in LLVM. The project helped me understand how Ocaml, with its pattern matching makes it easy to work with trees thereby making it suitable for compiler development. Working on features like lambdas helped me understand how semantic analysis can be a powerful tool that can be used to rewrite complex constructs like lambdas into existing constructs like functions, thereby making it simpler to generate code. One aspect that I would like to restructure or potentially revisit is the security aspect of Espresso, in my opinion the single most important aspect of any good programming language. Type safety is a good initial step but little changes like preventing stack overflows through features like stack cookies, control flow integrity checks etc would definitely go a long way to enhance the underlying security provided by Espresso.

7.4 Somdeep Dey

The most challenging part of Espresso has been the use of functional programming. Coding in OCaml put me in unfamiliar waters, after years of fine tuning my knowledge of OOP. What I wished I had done is taken several days at the beginning of the project to truly internalize the basics of OCaml - that would have smoothed out many bumps that we came across during the process. However, with practice I came to appreciate the benefits of OCaml - pattern matching, modular programming, and strong static type checking. I truly wish that there was more time to take this project further - we all have such a clear vision of how to improve our language (inheritance), but there is only so much time. With that in mind, I encourage future groups to recognize the scope of the project and start early.

8 Appendix

8.1 Source Code

8.1.1 parser.mly

```
/* Ocamlyacc parser for Espresso */

%{
open Ast
%}

%token CLASS
%token SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA
 ↪  COLON DOT THIS POUND
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT MODULUS POWER
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE FOREACH INT BOOL VOID STRING FLOAT
 ↪  CHAR BREAK HASHMAP LAMBDA CONTINUE
%token <int> LITERAL
%token <string> ID
%token <string> STRLIT
%token <char> CHARLIT
%token <float> FLOATLIT
%token EOF

%nonassoc NOELSE POUND
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULUS POWER
%right NOT SUB
%nonassoc LSQUARE
%right DOT

%start program
%type <Ast.program> program

%%

program:
  cdecls EOF { Program($1) }

cdecls:
cdecl_list       { List.rev $1 }

cdecl_list:
    cdecl       { [$1] }
|   cdecl_list cdecl    { $2::$1 }

cdecl:
   CLASS ID LBRACE cbody RBRACE
   { {
     cname = $2;
```

```
      cbody = $4
      } }

cbody:
       { {
        fields = [];
        methods = []
      } }
    | cbody vdecl { {
        fields = $2 :: $1.fields;
        methods = $1.methods
      } }
    | cbody func_decl { {
        fields = $1.fields;
        methods = $2 :: $1.methods
      } }


fname:
         ID { $1 }

func_decl:
   data_typ fname LPAREN formals_opt RPAREN LBRACE stmt_list
    ↪  RBRACE
     { { typ = $1;
         fname = $2;
         formals = $4;
         body = List.rev $7 } }

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

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

formal:
          data_typ ID {Formal($1,$2)}

data_typ:
    typ  { Datatype($1) }
  |   array_typ { $1 }
  |   hashmap_typ { $1 }
typ:
    INT { Int }
  | BOOL { Bool }
  | VOID { Void }
  | STRING { String }
  | FLOAT { Float }
  | CHAR { Char }
  | LAMBDA { Lambda }
  | CLASS ID {ObjTyp($2)}

hashmap_typ:
  HASHMAP LT typ COMMA typ GT {Hashmaptype($3,$5)}

array_typ:
```

```
    typ LSQUARE LITERAL RSQUARE {ArrayType($1, $3)}

/* This is only for the class data members  */
vdecl:
   data_typ ID SEMI { Vdecl($1, $2) }

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

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
  ↪  Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
    { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | FOREACH LPAREN data_typ ID COLON ID RPAREN stmt
    { Foreach($3, $4, $6, $8) }
  | LAMBDA COLON data_typ ID LPAREN formals_opt RPAREN stmt  {
  ↪  Lambda($3, $4, $6, $8) }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }
  | data_typ ID  SEMI { Local($1,$2) }
 /* | data_typ ID ASSIGN expr SEMI { Local($1, $2, $4) }
*/
expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
  | CHARLIT          { Charlit($1) }
  | STRLIT           { Strlit($1) }
  | FLOATLIT         { Floatlit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID               { Id($1) }
  | THIS             { This }
  | 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 MODULUS expr { Binop($1, Mod,   $3) }
  | expr POWER expr { Binop($1, Pow,   $3) }
  | expr EQ     expr { Binop($1, Eq, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Lt,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Gt, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec SUB { Unop(Sub, $2) }
```

```
  | NOT expr          { Unop(Not, $2) }
  | expr ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | POUND ID LPAREN actuals_opt RPAREN { LambdaCall($2, $4) }
  | LPAREN expr RPAREN { $2 }
  | expr LSQUARE expr RSQUARE { ArrayAccess($1, $3) }
  | ID LBRACE expr RBRACE { HashmapAccess($1, $3) }
  | expr DOT expr { ObjectAccess($1, $3) }


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

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

8.1.2 scanner.mll

```
(* Ocamllex scanner for Espresso *)

{ open Parser

let unescape s =
            Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}
let digit =  ['0'-'9']
let charl = ''' ( _? ) '''
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let string = '"' ( (ascii | escape)* as s) '"'

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| "//"           { sincom lexbuf }                (* Single-Line
↪  comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LSQUARE }                        (* Square brackets
↪  for Arrays *)
| ']'      { RSQUARE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| '%'           { MODULUS }
| '.'     { DOT }
| '#'     { POUND }
| "**"           { POWER }
| "=="     { EQ }
```

```
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| '>'       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "foreach" { FOREACH }                    (* Foreach loop *)
| "return"  { RETURN }
| "int"     { INT }
| "bool"    { BOOL }
| "void"    { VOID }
| "String"  { STRING }
| "float"   { FLOAT }
| "char"    { CHAR }
| "true"    { TRUE }
| "false"   { FALSE }
| "break"   { BREAK }
| "continue" { CONTINUE }
| "hashmap"{HASHMAP}
| "class"   { CLASS }
| "this"    { THIS }
| "lambda"   { LAMBDA }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm)
↪  }
| string { STRLIT(unescape s) }
| charl as lxm { CHARLIT(String.get lxm 1) }
| ['0'-'9']+['.']['0'-'9']+ as lxm { FLOATLIT(float_of_string lxm)
↪  }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
↪  char)) }

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


and sincom = parse
  "\n" { token lexbuf }
| _    { sincom lexbuf }
```

8.1.3 semant.ml

```
open Ast
open Sast

module StringMap = Map.Make (String)

type classMap = {
        field_map               : Ast.var_decl StringMap.t;
```

```
        func_map                        : Ast.func_decl
         ↪  StringMap.t;
        reserved_func_map                 : sfunc_decl
         ↪  StringMap.t;
        cdecl                                : Ast.cdecl;
}

let is_lambda = ref false
let lambda_count = ref 0
let lambda_funcs = ref [] ;;

let lambda_func_map = ref StringMap.empty ;;

type env = {
        env_class_maps: classMap StringMap.t;
        env_class_map : classMap;
        env_name      : string;
        env_locals    : typ StringMap.t;
        env_parameters: Ast.formal StringMap.t;
        env_return_type: typ;
        env_in_for    : bool;
        env_in_while  : bool;
    env_in_foreach: bool;
        env_reserved  : sfunc_decl list;
}

let update_env_name env env_name =
{
        env_class_maps = env.env_class_maps;
        env_class_map = env.env_class_map;
        env_name       = env_name;
        env_locals     = env.env_locals;
        env_parameters = env.env_parameters;
        env_return_type = env.env_return_type;
        env_in_for     = env.env_in_for;
        env_in_while   = env.env_in_while;
        env_in_foreach = env.env_in_foreach;
        env_reserved   = env.env_reserved;
}

let update_call_stack env in_for in_while in_foreach =
{
        env_class_maps = env.env_class_maps;
        env_class_map = env.env_class_map;
        env_name       = env.env_name;
        env_locals     = env.env_locals;
        env_parameters = env.env_parameters;
        env_return_type = env.env_return_type;
        env_in_for     = in_for;
        env_in_while   = in_while;
        env_in_foreach = in_foreach;
        env_reserved   = env.env_reserved;
}



(* get complete function name prepended with the class *)
let get_fully_qualified_name class_name fdecl =
```

```
    let func_name = fdecl.fname in
        match func_name with
            "main" -> "main"
            |   _ -> class_name ^ "." ^ func_name

let string_of_object = function
                Datatype(ObjTyp(s))         -> s
        |         _ -> ""

(* define all built-in functions supported by espresso *)
let get_reserved_funcs =
    let reserved_struct name return_type formals =
                {
                        sfname                          = name;
                        styp                  = return_type;
                        sformals                    = formals;
                        sbody                            = [];
                        sftype                       = Sast.Reserved;
                        scontext_class  = "Nil";
                        sthis_ptr       =
                         ↪  SId("Nil",Datatype(ObjTyp("Nil")))
                }
        in
    let reserved_functions = [
        reserved_struct "print_int" (Datatype(Void))
         ↪  ([Formal(Datatype(Int), "int_arg")]);
                reserved_struct "print_float" (Datatype(Void))
                 ↪  ([Formal(Datatype(Float), "float_arg")]);
                reserved_struct "print_char" (Datatype(Void))
                 ↪  ([Formal(Datatype(Char), "char_arg")]);
                reserved_struct "print_string" (Datatype(Void))
                 ↪  ([Formal(Datatype(String), "string_arg")]);
                reserved_struct "print_char_array"
                 ↪  (Datatype(Void)) ([Formal(ArrayType(Char, 1),
                 ↪  "char_arr_arg")]);
    ] in
    reserved_functions

let get_class_maps cdecls reserved_maps =
        let reserved_funcs = reserved_maps in
    let setup_class_map m cdecl =
        (* get all fields belonging to the class cdecl. Raise
         ↪  error if duplicates are found *)
        let field_maps m = function Vdecl(typ, name) ->
            if (StringMap.mem name m)
                then raise (Failure(" duplicate field name :" ^
                 ↪  name))
            else
                (StringMap.add name (Vdecl(typ, name)) m)
        in

        (* get all methods belonging to the class cdecl. Raise
         ↪  error if duplicates are found *)
        let func_maps m fdecl =
            let func_full_name = get_fully_qualified_name
             ↪  cdecl.cname fdecl
            in
            if (StringMap.mem func_full_name m)
```

```
                then raise (Failure ("duplicate function : " ^
                ↪  func_full_name))
            else
                                if (StringMap.mem fdecl.fname
                                ↪  reserved_funcs)
                                    then
                                    ↪  raise(Failure("function
                                    ↪  " ^ fdecl.fname ^ " is
                                    ↪  a reserved keyword!"))
                                    else (StringMap.add
                                    ↪  (func_full_name) fdecl
                                    ↪  m)
        in

        (* check for duplicate classes and add their fields,
        ↪  methods respectively *)
        (if (StringMap.mem cdecl.cname m)
            then raise(Failure ("Duplicate Class Name : " ^
            ↪  cdecl.cname ))
        else
            StringMap.add cdecl.cname
            {
                field_map = List.fold_left field_maps
                ↪  StringMap.empty cdecl.cbody.fields;
                func_map = List.fold_left func_maps
                ↪  StringMap.empty cdecl.cbody.methods;
                                reserved_func_map = reserved_maps;
                cdecl = cdecl;
            }
            m
        )
    in List.fold_left setup_class_map StringMap.empty cdecls
(*in*)

let get_scdecl_from_cdecl sfdecls (cdecl) =
        {
                scname = cdecl.cname;
        scbody = {sfields = cdecl.cbody.fields; smethods =
        ↪  sfdecls; }
        }



let rec get_sexpr_from_expr env expr = match expr with
        Literal i -> SLiteral(i), env
    |   Strlit s -> SStrlit(s), env
    |   Floatlit f -> SFloatlit(f), env
    |   BoolLit b -> SBoolLit(b), env
    |   Charlit c -> SCharlit(c), env
        |         Id id -> SId(id, (get_id_data_type env id)), env
        |   This  -> SId("this", Datatype(ObjTyp(env.env_name))),
        ↪  env
        |          Assign(expr1, expr2) -> check_assignment env
        ↪  expr1 expr2, env
        |         Binop(expr1, op, expr2) -> check_binop env expr1
        ↪  op expr2, env
        |         Unop(op, expr) -> check_unop env op expr, env
```

```
        |         ArrayAccess(id, expr) -> check_array_access env
        ↪  id expr, env
        |         HashmapAccess(id, expr) -> check_hashmap_access
        ↪  env id expr, env
        |         ObjectAccess(expr1, expr2) -> check_object_access
        ↪  env expr1 expr2, env
        |         Call(func_name, expr_list) -> check_call env
        ↪  func_name expr_list, env
        |          LambdaCall(func_name, expr_list) ->
        ↪  check_lambda_call env func_name expr_list, env
        |          Noexpr          ->          SNoexpr, env


(* Update this function whenever SAST's sexpr is updated *)
and get_type_from_sexpr sexpr = match sexpr with
        SLiteral(_) -> Ast.Datatype(Int)
    |   SStrlit(_) -> Ast.Datatype(String)
    |   SFloatlit(_) -> Ast.Datatype(Float)
    |   SBoolLit(_) -> Ast.Datatype(Bool)
    |   SCharlit(_) -> Ast.Datatype(Char)
        |   SId(_, t) -> t
    |   SBinop(_,_,_,t) -> t
    |   SUnop(_,_,t) -> t
    |   SAssign(_,_,t) -> t
    |   SCall(_,_,t) -> t
    |   SArrayAccess(_,_,t) -> t
        |   SHashmapAccess(_,_,t) -> t
        |         SObjectAccess(_,_,t) -> t
    |   SNoexpr -> Ast.Datatype(Void)

(* get a list of sexprs from a list of exprs *)
and get_sexprl_from_exprl env el =
  let env_ref = ref(env) in
  let rec helper = function
          head::tail ->
                let a_head, env = get_sexpr_from_expr !env_ref
                ↪  head in
                env_ref := env;
                a_head::(helper tail)
        | [] -> []
  in (helper el), !env_ref

(*semantically verify a block*)
and check_block env blk = match blk with
                [] -> SBlock([SExpr(SNoexpr,Datatype(Void))]),env
        |        _  ->
                let blk, _ = convert_stmt_list_to_sstmt_list env
                ↪  blk in
                SBlock(blk),env


(*semantically verify an Expression*)
and check_expr env expr =
        let sexpr,env = get_sexpr_from_expr env expr in
        let type_sexpr = get_type_from_sexpr sexpr in
        SExpr(sexpr, type_sexpr), env

(* semantically verify a return statement *)
```

```
and check_return env expr =
    let sexpr, _ = get_sexpr_from_expr env expr in
    let type_sexpr = get_type_from_sexpr sexpr in

        if !is_lambda = false
                then
            if type_sexpr = env.env_return_type
                then SReturn(sexpr, type_sexpr), env
            else
                raise (Failure ("Expected type " ^
                 ↪  Ast.string_of_datatype (env.env_return_type) ^
                 ↪  " but got " ^ Ast.string_of_datatype
                 ↪  (type_sexpr)))

        else
                SReturn(sexpr,type_sexpr),env

(* semantically verify an if statement *)
and check_if env expr st1 st2 =
        let sexpr,_ = get_sexpr_from_expr env expr in
        let type_sexpr = get_type_from_sexpr sexpr in
        let if_body,_ =        parse_stmt env st1 in
        let else_body,_ = parse_stmt env st2 in
        if type_sexpr = Datatype(Bool)
                then SIf(sexpr,if_body,else_body), env
                else raise(Failure ("Invalid If expression type,
                 ↪  must be Bool"))

 (* semantically verify local variable declaration *)
 and check_local env dt name =
         if StringMap.mem name env.env_locals
                 then raise (Failure ("Duplicate local
                  ↪  declaration"))
        else
                let new_env = {
                        env_class_maps = env.env_class_maps;
                        env_class_map = env.env_class_map;
                        env_name = env.env_name;
                        env_locals = StringMap.add name dt
                         ↪  env.env_locals;
                        env_parameters = env.env_parameters;
                        env_return_type = env.env_return_type;
                        env_in_for = env.env_in_for;
                        env_in_while = env.env_in_while;
                        env_in_foreach = env.env_in_foreach;
                        env_reserved = env.env_reserved;
                }in
                (match dt with
                        Datatype(ObjTyp(s)) ->
                                (if not (StringMap.mem
                                 ↪  (string_of_object dt)
                                 ↪  env.env_class_maps)
                                        then raise((Failure
                                         ↪  ("Class type not
                                         ↪  defined")))
                                 else
                                        SLocal(dt,name),new_env)
                |         _ -> SLocal(dt,name),new_env)
```

```
(* semantically verify a while statement *)
and check_while env expr st =
        let old_val = env.env_in_while in
        let env = update_call_stack env env.env_in_for true
        ↪  env.env_in_foreach in

        let sexpr,_ = get_sexpr_from_expr env expr in
        let type_sexpr = get_type_from_sexpr sexpr in
        let sstmt, _ = parse_stmt env st in
        let swhile =
                if(type_sexpr = Datatype(Bool) || type_sexpr =
                ↪  Datatype(Void))
                        then SWhile(sexpr,sstmt)
                        else raise        (Failure ("Invalid while
                        ↪  condition statement"))
        in
        let env = update_call_stack env env.env_in_for old_val
        ↪  env.env_in_foreach in
        swhile,env


(*semantically verify a for statement*)
(*MUST CONTAIN CONDITIONAL AS BOOLEAN*)
and check_for env exp1 exp2 exp3 st =
        let old_val = env.env_in_for in
        let env = update_call_stack env true env.env_in_while
        ↪  env.env_in_foreach in

        let sexpr1,_ = get_sexpr_from_expr env exp1 in
        let sexpr2,_ = get_sexpr_from_expr env exp2 in
        let sexpr3,_ = get_sexpr_from_expr env exp3 in
        let for_body,_ = parse_stmt env st in
        let conditional_type = get_type_from_sexpr sexpr2 in
        let st_for =
                if(conditional_type = Datatype(Bool))
                        then SFor(sexpr1,sexpr2,sexpr3,for_body)
                else raise (Failure ("Invalid For statement
                ↪  conditional"))
        in

        let env = update_call_stack env old_val env.env_in_while
        ↪  env.env_in_foreach in
        st_for, env


(*semantically check a foreach statement*)
and check_foreach env dt id1 id2 st =
        let old_val = env.env_in_foreach in
        let env = update_call_stack env env.env_in_for
        ↪  env.env_in_while true in

        if StringMap.mem id1 env.env_locals
                 then raise (Failure ("Duplicate local
                 ↪  declaration"))
        else

                let new_env = {
```

```
                        env_class_maps = env.env_class_maps;
                        env_class_map = env.env_class_map;
                        env_name = env.env_name;
                        env_locals = StringMap.add id1 dt
                        ↪  env.env_locals;
                        env_parameters = env.env_parameters;
                        env_return_type = env.env_return_type;
                        env_in_for = env.env_in_for;
                        env_in_while = env.env_in_while;
                        env_in_foreach = env.env_in_foreach;
                        env_reserved = env.env_reserved;
                }
                in

                let foreach_body,_ = parse_stmt new_env st in
                let type_id = get_id_data_type new_env id2 in

        let st_foreach =
                        if (dt = Datatype(Int) || dt =
                        ↪  Datatype(Char) || dt = Datatype(Bool)
                        ↪  || dt = Datatype(Float) || dt =
                        ↪  Datatype(String) || dt =
                        ↪  Datatype(Void))
                                then
                                ↪  SForeach(dt,id1,id2,foreach_body)
                        else raise(Failure ("Foreach only works on
                        ↪  primitives currently"))
        in

        let st_foreach_2 =
                        match type_id,dt with
                                ArrayType(t,_),Datatype(d) ->
                                        if(t = d)
                                                then (st_foreach)
                                        else
                                                raise(Failure
                                                ↪  ("Mismatch in
                                                ↪  array and
                                                ↪  iterator type
                                                ↪  for foreach"))
                        | _ ->          raise(Failure ("Need array
                        ↪  type to walkthrough in foreach"))
        in

        let env = update_call_stack env env.env_in_for
        ↪  env.env_in_while old_val in
        st_foreach_2,new_env



and check_continue env =
        if env.env_in_for || env.env_in_while ||
        ↪  env.env_in_foreach then
                SContinue, env
        else
                raise (Failure ("Continue cannot be called outside
                ↪  of a loop"))
```

```
(*semantically verify a break statement*)
and check_break env =
        if env.env_in_for || env.env_in_while ||
         ↪  env.env_in_foreach then
                SBreak,env
        else
                raise (Failure ("Break cannot be called outside of
                 ↪  a loop"))

(* check types in assignments *)
and check_assignment env expr1 expr2 =
        let sexpr1, _ = get_sexpr_from_expr env expr1 in
        let sexpr, _ = get_sexpr_from_expr env expr2 in
        let type_id = get_type_from_sexpr sexpr1 in match sexpr1
         ↪  with
                SId(_,_) | SArrayAccess(_,_,_) |
                 ↪  SHashmapAccess(_,_,_) | SObjectAccess(_,_,_)
                 ↪  ->
                        (let type_sexpr = get_type_from_sexpr
                         ↪  sexpr in match (type_id, type_sexpr)
                         ↪  with
                                Datatype(ObjTyp(t1)),
                                 ↪  Datatype(ObjTyp(t2)) ->
                               if t1 = t2
                                then SAssign(sexpr1, sexpr,
                                 ↪  type_id)
                                else raise (Failure ("illegal
                                 ↪  assignment from " ^
                                 ↪  (string_of_datatype
                                 ↪  type_sexpr) ^ " to " ^
                                 ↪  (string_of_datatype type_id)))
                               |  _,_ -> if type_id = type_sexpr
                                then SAssign(sexpr1, sexpr,
                                 ↪  type_id)
                                else match (type_id, type_sexpr)
                                 ↪  with
                                        ArrayType(p1,_),
                                         ↪  ArrayType(p2, _) -> if
                                         ↪  p1 = p2 then
                                         ↪  SAssign(sexpr1, sexpr,
                                         ↪  type_sexpr)
```

```
                                    | ArrayType(Char, _) ,
                                    ↪  Datatype(String) ->
                                    ↪  SAssign(sexpr1, sexpr,
                                    ↪  type_id)
                                        |         _,_ ->
                                                raise(Failure
                                                ↪  ("illegal
                                                ↪  assignment
                                                ↪  here from " ^
                                                ↪  (string_of_datatype
                                                ↪  type_sexpr) ^
                                                ↪  " to " ^
                                                ↪  (string_of_datatype
                                                ↪  type_id) ))

        )
        |         _ -> raise(Failure("lvalue required for
        ↪  assignment "))


(* semantically validate arithemtic operations *)
and check_arithmetic_ops sexpr1 sexpr2 op type1 type2 = match
 ↪  (type1, type2) with
(* Assuming that the lhs and rhs must have the same type *)
                (Datatype(Int), Datatype(Int)) -> SBinop(sexpr1,
                 ↪  op, sexpr2, Datatype(Int))
        |         (Datatype(Float), Datatype(Float)) ->
         ↪  SBinop(sexpr1, op, sexpr2, Datatype(Float))
        |         (Datatype(Char), Datatype(Char)) ->
         ↪  SBinop(sexpr1, op, sexpr2, Datatype(Char))
        |         _,_ -> raise(Failure("types " ^
         ↪  (string_of_datatype type1) ^ " and " ^
         ↪  (string_of_datatype type2) ^ " are incompatible for
         ↪  arithmetic operations "))

and check_relational_ops sexpr1 sexpr2 op type1 type2 = match
 ↪  (type1, type2) with
(* Assuming that the lhs and rhs must have the same type *)
                (Datatype(Int), Datatype(Int)) -> SBinop(sexpr1,
                 ↪  op, sexpr2, Datatype(Bool))
```

```
        |         (Datatype(Float), Datatype(Float)) ->
        ↪  SBinop(sexpr1, op, sexpr2, Datatype(Bool))
        |         (Datatype(Char), Datatype(Char)) ->
        ↪  SBinop(sexpr1, op, sexpr2, Datatype(Bool))
        |         _,_ -> raise(Failure("types " ^
        ↪  (string_of_datatype type1) ^ " and " ^
        ↪  (string_of_datatype type2) ^ " are incompatible for
        ↪  comparison operations "))

(* Assuming that the types on either side are equal - no implicit
 ↪  type casting/ type promotions *)
and check_equality_ops sexpr1 sexpr2 op type1 type2 = match
 ↪  (type1, type2) with
                (* we cast characters and integers based on the
                 ↪  lhs in the codegen *)
                Datatype(Char), Datatype(Int) | Datatype(Int) ,
                 ↪  Datatype(Char) -> SBinop(sexpr1, op, sexpr2,
                 ↪  Datatype(Bool))
        | _ ->
                if type1 = type2
                        then SBinop(sexpr1, op, sexpr2,
                         ↪  Datatype(Bool))
                        else raise(Failure("types " ^
                         ↪  (string_of_datatype type1) ^ " and " ^
                         ↪  (string_of_datatype type2) ^ " are
                         ↪  incompatible for equality operations
                         ↪  "))

(* supports only boolean types *)
and check_logical_ops sexpr1 sexpr2 op type1 type2 = match (type1,
 ↪  type2) with
                (Datatype(Bool), Datatype(Bool)) -> SBinop(sexpr1,
                 ↪  op, sexpr2, Datatype(Bool))
        |         _,_ -> raise(Failure("types " ^
        ↪  (string_of_datatype type1) ^ " and " ^
        ↪  (string_of_datatype type2) ^ " are incompatible for
        ↪  logical operations. Only boolean types are supported.
        ↪  "))

(* check binary Arithmetic, relational & logical operations on
 ↪  expressions *)
and check_binop env expr1 op expr2 =
        let sexpr1, _ = get_sexpr_from_expr env expr1 in
        let sexpr2, _ = get_sexpr_from_expr env expr2 in
        let type1 = get_type_from_sexpr sexpr1 in
        let type2 = get_type_from_sexpr sexpr2 in
        match op with
                Add | Sub | Mult | Div | Mod | Pow ->
                 ↪  check_arithmetic_ops sexpr1 sexpr2 op type1
                 ↪  type2
        |         Lt | Leq | Gt | Geq -> check_relational_ops
        ↪  sexpr1 sexpr2 op type1 type2
        |         Eq | Neq -> check_equality_ops sexpr1 sexpr2 op
        ↪  type1 type2
        |         And | Or -> check_logical_ops sexpr1 sexpr2 op
        ↪  type1 type2
        |         _ -> raise (Failure("unknown binary operator "))
```

```
and check_unop env op expr =
        let get_numeric_sunop oper sexpr typ_exp = match oper with
                        Sub -> SUnop(oper, sexpr, typ_exp)
                |        _ -> raise (Failure (" illegal unary
                 ↪  operator for numeric type " ^
                 ↪  (string_of_datatype typ_exp)))
        in
        let get_bool_sunop oper sexpr typ_expr = match oper with
                        Not -> SUnop(oper, sexpr, typ_expr)
                |        _ -> raise (Failure (" illegal unary
                 ↪  operator for boolean type "))
        in
        let sexpr, _ = get_sexpr_from_expr env expr in
        let type_sexpr = get_type_from_sexpr sexpr in
        match type_sexpr with
                Datatype(Int) | Datatype(Float) ->
                 ↪  get_numeric_sunop op sexpr type_sexpr
        |        Datatype(Bool) -> get_bool_sunop op sexpr
         ↪  type_sexpr
        |        _ -> raise(Failure("unary oparator can only be
         ↪  applied to Int, Float or Bool types "))

(* semantic check for array element access. Supporting only 1D
 ↪  arrays now *)
and check_array_access env id expr =
        let sid,_ = get_sexpr_from_expr env id in
        let sexpr, _ = get_sexpr_from_expr env expr in
        let type_sexpr = get_type_from_sexpr sexpr in match
         ↪  type_sexpr with
                Datatype(Int) -> (* check if id was declared as an
                 ↪  array type *)
                                                (let type_id =
                                                 ↪  get_type_from_sexpr
                                                 ↪  sid in match
                                                 ↪  type_id with
                                                     ArrayType(t,
                                                      ↪  _) ->
                                                      ↪  SArrayAccess(sid,
                                                      ↪  sexpr,
                                                      ↪  Datatype(t))
                                                |        _ ->
                                                 ↪  raise(Failure("
                                                 ↪  identifier " ^
                                                 ↪  string_of_expr
                                                 ↪  id ^ " does
                                                 ↪  not belong to
                                                 ↪  an ArrayType
                                                 ↪  "))
                                                )
        |        _ -> raise(Failure(" array index must be an
         ↪  integer "))

(* semantically check for hashmap element access. Supporint only
 ↪  primitive types for hashmaps *)
and check_hashmap_access env id expr =
        let sexpr, _ = get_sexpr_from_expr env expr in
        let type_sexpr = get_type_from_sexpr sexpr in
```

```
        let type_id = get_id_data_type env id in match (type_id,
        ↪  type_sexpr) with
                        (Hashmaptype(t1, t2), Datatype(prim)) ->
                        ↪  (if t1 = prim
                                        (* NOTE: we return type t2
                                        ↪  and not t1 as t2 ->
                                        ↪  type of the value *)
                                        then SHashmapAccess(id,
                                        ↪  sexpr, Datatype(t2))
                                        else
                                        ↪  raise(Failure("expected
                                        ↪  key of type " ^
                                        ↪  (string_of_datatype
                                        ↪  (Datatype(t1))) ^ "
                                        ↪  but got type " ^
                                        ↪  (string_of_datatype
                                        ↪  type_sexpr)  ))
                                )
                |         (_, Datatype(prim))  ->
                ↪  raise(Failure("identifier " ^ id ^ " is not a
                ↪  valid hashmap type "))
                |         (_, _) -> raise(Failure(" Hashmap
                ↪  currently supports only primitive "))

(* semantically check object access *)
and check_object_access env expr1 expr2 =
        (* verify that the invoking object is a valid identifier
        ↪  *)
        let check_obj_id expr = match expr with
                Id obj -> SId(obj, get_id_data_type env obj)
        |        This -> SId("this",
        ↪  Datatype(ObjTyp(env.env_name)))
        |        ArrayAccess(id, expr) -> check_array_access env
        ↪  id expr
        in

        (* get object's class name *)
        let get_class_name obj = match obj with
                Datatype(ObjTyp(class_name)) -> class_name
        |        _ -> raise(Failure(" expected object type!"))
        in

        let rec check_member lhs_env obj_type top_env mem_expr =
                let class_name = get_class_name obj_type in
                match mem_expr with
                        (* search list of member fields in the
                        ↪  class *)
                        Id id -> (let class_map = StringMap.find
                        ↪  class_name lhs_env.env_class_maps in
                                let match_field field = match
                                ↪  field with
                                        Vdecl(dt, field_name) ->
                                        ↪  SId(id, dt), lhs_env
                                in
                                try match_field (StringMap.find id
                                ↪  class_map.field_map)
```

```
                                with | Not_found ->
                                 ↪  raise(Failure("Unrecognized
                                 ↪  member")) )
                        |       Call(func_name, expr_list) ->        (*
                         ↪  handle member functions *)




                        |       ObjectAccess(e1, e2) -> (* handle
                         ↪  recursive object access patterns *)
                                (let old_env = lhs_env in
                                let lhs,new_lhs_env = check_member lhs_env
                                 ↪  obj_type top_env e1 in
                                let lhs_type = get_type_from_sexpr lhs in
                                let new_env = update_env_name new_lhs_env
                                 ↪  (get_class_name lhs_type) in

                                let rhs, _ = check_member new_env lhs_type
                                 ↪  lhs_env e2 in
                                let rhs_type = get_type_from_sexpr rhs in
                                SObjectAccess(lhs, rhs, rhs_type),
                                 ↪  old_env)

                        |       _ -> raise(Failure("Unrecognized
                         ↪  datatype! "))
                in

                (* check left-side of object access - it should resolve to
                 ↪  some identifier *)
                let sexpr1 = check_obj_id expr1 in
                let type_obj = get_type_from_sexpr sexpr1 in
                let env_obj = update_env_name env (get_class_name
                 ↪  type_obj) in

                (* check the right side - it should belong to the invoking
                 ↪  object's class *)
                let sexpr2,_ = check_member env_obj type_obj env expr2 in
                let type_member = get_type_from_sexpr sexpr2 in
                SObjectAccess(sexpr1, sexpr2, type_member)

(* check function call semantics *)
(* pass invoking object's environment in env if this is invoked by
 ↪  an object *)
and check_call env func_name expr_list = match env.env_name with
                (* get class in corresponding env *)
                "Lambda" ->
                 (
                        let sexpr_list, _ = get_sexprl_from_exprl env
                         ↪  expr_list in
                        (* check a given formal and actual parameter *)
                        let get_actual_param formal_param param =
                                let formal_type = match formal_param with
                                 ↪  Formal(t, _) -> t | _ ->
                                 ↪  Datatype(Void) in
                                let param_type = get_type_from_sexpr param
                                 ↪  in
```

```
                        if formal_type = param_type
                                then param
                                else raise (Failure("Type mismatch
                                 ↪  in lambda . Expected " ^
                                 ↪  string_of_datatype formal_type
                                 ↪  ^ " but got " ^
                                 ↪  string_of_datatype
                                 ↪  param_type))
                in

                (* check lengths of formal and passed parameters
                 ↪  and get actual parameters *)
                let get_actual_params formal_params params =
                        let formal_len = List.length formal_params
                         ↪  in
                        let param_len = List.length params in
                                if formal_len = param_len
                                        then List.map2
                                         ↪  get_actual_param
                                         ↪  formal_params params
                                        else raise(Failure("
                                         ↪  formal and actual
                                         ↪  parameters have
                                         ↪  unequal lengths "))
                in

                (* get the actual lambda prototype from the map *)
                let func_handle = StringMap.find func_name
                 ↪  !lambda_func_map in

                let actuals_list = get_actual_params
                 ↪  func_handle.sformals sexpr_list in
                SCall(func_handle.sfname, actuals_list,
                 ↪  func_handle.styp)
        )
        | _ ->
(
        let context_class_map = try StringMap.find env.env_name
         ↪  env.env_class_maps with
                |        Not_found -> raise (Failure ("class " ^
                 ↪  env.env_name ^ " was not found in the context
                 ↪  of this function call " ^ func_name ))
        in
        let sexpr_list, _ = get_sexprl_from_exprl env expr_list in
        (* check a given formal and actual parameter *)
        let get_actual_param formal_param param =
                let formal_type = match formal_param with
                 ↪  Formal(t, _) -> t | _ -> Datatype(Void) in
                let param_type = get_type_from_sexpr param in
                if formal_type = param_type
                        then param
                        else match (formal_type, param_type) with
                                ArrayType(p1,_),ArrayType(p2,_) ->
                                 ↪  if p1 = p2 then param
```

```
                        | _,_ -> raise (Failure("Type mismatch.
                        ↪  Expected " ^ string_of_datatype
                        ↪  formal_type ^ " but got " ^
                        ↪  (string_of_datatype param_type)))
        in

        (* check lengths of formal and passed parameters and get
        ↪  actual parameters *)
        let get_actual_params formal_params params =
                let formal_len = List.length formal_params in
                let param_len = List.length params in
                        if param_len = formal_len
                                then List.map2 get_actual_param
                                ↪  formal_params params
                                else raise(Failure(" formal and
                                ↪  actual parameters have unequal
                                ↪  lengths "))
        in

        let func_full_name = env.env_name ^ "." ^ func_name in
        (* look for the function in the list of reserved
        ↪  functions. If it is not found there
                look at the list of member functions of the
↪  context_class *)
        try let func_handle = StringMap.find func_name
        ↪  context_class_map.reserved_func_map in
                let actuals_list = get_actual_params
                ↪  func_handle.sformals sexpr_list in
                SCall(func_name, actuals_list, func_handle.styp)
                ↪  with
        |        Not_found ->
                (* search the list of member functions *)
                try let func_handle = StringMap.find
                ↪  func_full_name context_class_map.func_map in
                        let actuals_list = get_actual_params
                        ↪  func_handle.formals sexpr_list in
                        SCall(func_full_name, actuals_list,
                        ↪  func_handle.typ) with
                |        Not_found -> raise(Failure("function " ^
                ↪  func_name ^ " was not found "))
```

```
        )

and check_lambda_call env func_name expr_list =

        (* check if func_name is in params or locals *)
        let sfunc_name = try let ftype = StringMap.find func_name
        ↪  env.env_parameters in
                match ftype with Formal(_, name) -> name
               with | Not_found ->
                       (
                               try let _ = StringMap.find
                               ↪  func_name env.env_locals in
                               func_name
                               with | Not_found ->
                               ↪  raise(Failure("identifier " ^
                               ↪  func_name ^ " was not
                               ↪  found!"))
                       )
        in
        let lambda_obj = Id("lambda_obj") in
        (* actual lambda name is retrieved from the map in
        ↪  check_call *)
        let lambda_call = Call(sfunc_name, expr_list) in
        check_object_access env lambda_obj lambda_call


(*semantic check for lambda functions*)
and check_lambda env id formals st ret_typ=

                let return_present_type lamb_body =
                        match lamb_body with
                SBlock(stlist)          ->
                        (
        let leng = List.length stlist in
                if((leng) = 0) then
                        raise (Failure ("empty Lambda statement,
                        ↪  must atleast contain a return"))

                else
                let last_stmt = List.hd (List.rev stlist) in
                match last_stmt with
                        SReturn(_,typ) -> typ
                | _ -> raise(Failure ("Lambda function must end
                ↪  with a return statement"))
                        )
                | _         ->raise (Failure "Lambda must be
                ↪  enclosed in a block")
                in

                if StringMap.mem id env.env_locals
                        then raise (Failure ("Duplicate local
                        ↪  declaration"))
                else
                let old_env = env in
                let new_env = env in

                (*yet to add clash of old params with lambda
                ↪  params, assuming no clash, adding directly*)
```

```
(*not allowing globals in lambdas*)

let get_params_map m formal_node = match
 ↪  formal_node with
                Formal(data_type, formal_name) ->
                 ↪  (StringMap.add formal_name
                 ↪  formal_node m)
        |          _ -> m
in

let env_params = List.fold_left get_params_map
 ↪  StringMap.empty formals in

let env = {
        env_class_maps = env.env_class_maps;
        env_class_map = env.env_class_map;
        env_name = env.env_name;
        env_locals = StringMap.empty;
        env_parameters = env_params;
        env_return_type = env.env_return_type;
        env_in_for = env.env_in_for;
        env_in_while = env.env_in_while;
        env_in_foreach = env.env_in_foreach;
        env_reserved = env.env_reserved;

}
in

is_lambda := true;
let sstmt, _ = parse_stmt env st in
is_lambda        := false;

let actual_ret_typ = return_present_type sstmt in
if actual_ret_typ = ret_typ
        then (
                (*Restoring old environment*)
                let old_env = {
                        env_class_maps =
                         ↪  old_env.env_class_maps;
                        env_class_map =
                         ↪  old_env.env_class_map;
                        env_name =
                         ↪  old_env.env_name;
                        env_locals = StringMap.add
                         ↪  id
                         ↪  (Ast.Datatype(Lambda))
                         ↪  old_env.env_locals;
                        env_parameters =
                         ↪  old_env.env_parameters;
                        env_return_type =
                         ↪  old_env.env_return_type;
                        env_in_for =
                         ↪  old_env.env_in_for;
                        env_in_while =
                         ↪  old_env.env_in_while;
                        env_in_foreach =
                         ↪  old_env.env_in_foreach;
```

```
                                        env_reserved =
                                         ↪  old_env.env_reserved;
                                    }
                                    in
                                    (* generate an sfunc_decl object
                                     ↪  to transform the lamda into a
                                     ↪  function *)

                                    let lambda_sfdecl = {
                        styp = ret_typ;
                        sfname = "lambda_" ^ id;
                        sformals = formals;
                        sbody = [ sstmt ];
                        sftype = Sast.Udf;
                        scontext_class =  "Lambda";
                        sthis_ptr =
                         ↪  SId("Lambda",Datatype(ObjTyp("Lambda")));
                                    } in
                                    (* add this lambda to the global
                                     ↪  list of lambda functions *)
                                    lambda_funcs := (lambda_sfdecl ::
                                     ↪  !lambda_funcs) ;
                                    SLocal(Ast.Datatype(Lambda), id),
                                     ↪  old_env
                            )
                                            else
                                             ↪  raise(Failure("expected
                                             ↪  " ^
                                             ↪  (string_of_datatype
                                             ↪  ret_typ) ^ " but got "
                                             ↪  ^  (string_of_datatype
                                             ↪  actual_ret_typ) ^ "
                                             ↪  for lambda " ^ id))
```

```
(* Parse a single statement by matching with different forms that
 ↪  a statement
    can take, and generate appropriate SAST node *)
and parse_stmt env stmt = match stmt with
                Ast.Block blk -> check_block env blk
          |      Ast.Expr expr -> check_expr        env expr
          |      Ast.Return expr -> check_return env expr
          |       Ast.If(expr,st1,st2) -> check_if env expr st1
          ↪  st2
          |      Ast.While(expr,st) -> check_while env expr st
          |        Ast.For(exp1,exp2,exp3,st) -> check_for env
          ↪  exp1 exp2 exp3 st
          |      Ast.Foreach(dt,exp1,exp2,st) -> check_foreach env
          ↪  dt exp1 exp2 st
          |      Ast.Lambda(ret_typ, id,formals, st)        ->
          ↪  check_lambda env id formals st ret_typ
          |      Ast.Break -> check_break env
          |      Ast.Continue -> check_continue env
          |       Ast.Local(dt, name) -> check_local env dt name
```

```
(* Process the list of statements and return a list of sstmt nodes
 ↪  *)
and convert_stmt_list_to_sstmt_list env stmt_list =
        let env_reference = ref(env) in
        let rec get_sstmt = function
          hd::tl ->
                let sstmt, env = parse_stmt !env_reference hd in
                env_reference := env;
                sstmt::(get_sstmt tl)
        | [] -> []
        in
        let sstmts = (get_sstmt stmt_list), !env_reference in
        sstmts


and get_id_data_type env id = match id with
        (* search local variables *)
                "lambda_obj" -> Datatype(ObjTyp("Lambda"))
        | _ -> (
                try StringMap.find id env.env_locals
                with | Not_found -> (* search function arguments
                 ↪  *)
                try let param = StringMap.find id
                 ↪  env.env_parameters in
                        (function Formal(t, _) -> t) param
                with | Not_found -> (* search field members *)
                try let var_decl = StringMap.find id
                 ↪  env.env_class_map.field_map in
                        (function Vdecl(t, _) -> t) var_decl
                with | Not_found -> raise (Failure ("undefined
                 ↪  identifier " ^ id))
        )


(* Checks for the presence of a return statement when the
 ↪  signature indicates a
        non-void return type *)
let is_return_present func_name func_body func_return_type =
    let leng = List.length func_body in
    if ((leng) = 0) then
        (*if (func_return_type != Datatype(Void)) then (raise
         ↪  (Failure("No Statement"))) else (raise(Failure ("Blah
         ↪  blah"))))*)
        match func_return_type with
          Datatype(Void) -> ()
        | _ ->        raise(Failure ("Empty function body where
         ↪  return was expected"))
    else
    let last_stmt = List.hd (List.rev func_body) in match
     ↪  last_stmt, func_return_type with
            _,Datatype(Void) -> ()
        |   SReturn(_,_), _ -> ()   (* There is a return statement
         ↪  *)
```

```
        |   _ -> raise(Failure "non-void function does not have a
        ↪  return statement\n")


(* Function that converts func_decl into sfunc_decl *)
let convert_fdecl_to_sfdecl class_maps reserved class_map cname
↪  fdecl =
    let get_params_map m formal_node = match formal_node with
                        Formal(data_type, formal_name) ->
                        ↪  (StringMap.add formal_name formal_node
                        ↪  m)
                |          _ -> m
        in
        let this_ptr = Formal(Datatype(ObjTyp(cname)), "this") in
        let env_params = List.fold_left get_params_map
        ↪  StringMap.empty (fdecl.formals) in
    let env = {
                env_class_maps         = class_maps;
                env_class_map = class_map;
                env_name               = cname;
                env_locals              = StringMap.empty;
                env_parameters         = env_params;
                env_return_type         = fdecl.typ;
                env_in_for                  = false;
                env_in_while           = false;
        env_in_foreach = false;
                env_reserved           = reserved;
        } in
    (* the function's body is a list of statements. Semanticallu
    ↪  check each statement
       and generate the Sast node *)
    let fbody = fst (convert_stmt_list_to_sstmt_list env
    ↪  fdecl.body) in
    let fname = (get_fully_qualified_name cname fdecl) in
        ignore(is_return_present fname fbody fdecl.typ);
    {
                sfname              = (get_fully_qualified_name
                ↪  cname fdecl);
                styp               = fdecl.typ;
                sformals     = fdecl.formals;
                sbody                   = fbody;
                sftype                  = Sast.Udf;
                scontext_class = cname;
                sthis_ptr = SId("this", Datatype(ObjTyp(cname)));
        }



(* FUNCTION FOR GENERATING SAST  *)
let get_sast class_maps reserved cdecls =

        (* look through SAST functions *)
        let find_main = (fun f -> match f.sfname with "main" ->
        ↪  true | _ -> false) in
        let remove_main funcs =
                List.filter (fun func -> not (find_main func))
                ↪  funcs
        in
```

```
        let get_main func_list =
                let mains = (List.find_all find_main func_list) in
                if List.length mains < 1 then
                raise (Failure ("Main not Defined"))
                else if List.length mains > 1 then
                raise (Failure ("too many mains defined"))
                else List.hd mains
        in

        let handle_cdecl cdecl =
                let class_map = StringMap.find cdecl.cname
                 ↪  class_maps in
                 (* apply convert_fdecl_to_sfdecl on each method
                  ↪  from the class and accumulate the
                  ↪  corresponding sfdecls in the list *)
        let sfunc_list_with_main = List.fold_left (fun ls f ->
         ↪  (convert_fdecl_to_sfdecl class_maps reserved class_map
         ↪  cdecl.cname f) :: ls) [] cdecl.cbody.methods in
                let scdecl = get_scdecl_from_cdecl
                 ↪  sfunc_list_with_main cdecl in
                (scdecl, sfunc_list_with_main)
        in

        let iter_cdecls t c =
        let scdecl = handle_cdecl c in
        (fst scdecl :: fst t, snd scdecl @ snd t)
        in

        let scdecl_list, sfunctions_list = List.fold_left
         ↪  iter_cdecls ([], []) cdecls in


        let slambda_body = {
                 sfields = [];
                 smethods = !lambda_funcs;
        } in
        let slambda_class = {
                  scname = "Lambda";
                  scbody = slambda_body;
        } in

        let main = get_main sfunctions_list in
        {
                classes = (scdecl_list @ [slambda_class]);
                functions = ((remove_main sfunctions_list) @
                 ↪  !lambda_funcs ); (* Should we remove main from
                 ↪  this ? *)
                main = main;
                reserved = reserved;
        }



let handle_lambdas cdecl =
        let methods = cdecl.cbody.methods in

        (* parse a lambda expression from a given statement *)
        let parse_lambda_from_stmt stmt = match stmt with
```

```
                        Lambda(ret_typ, id,formals, st) ->
                        (
                        let lambda_prototype = {
                                styp = ret_typ;
                                sfname = "lambda_" ^ id;
                                sformals = formals;
                                sbody = [ ];
                                sftype = Sast.Udf;
                                scontext_class =  "Lambda";
                                sthis_ptr =
                                 ↪  SId("Lambda",Datatype(ObjTyp("Lambda")));
                        } in
                        ignore(lambda_func_map := StringMap.add id
                         ↪  lambda_prototype !lambda_func_map) ;
                        "done"
                        )
                |         _ -> "idk"
        in

        let handle_method func =
                (* identify lambda expressions from a given list
                 ↪  of statements *)
                let rec handle_statements stmts = match stmts with
                                hd::tl ->
                                        ignore(parse_lambda_from_stmt
                                         ↪  hd);
                                        "1" :: handle_statements
                                         ↪  tl
                                | [] -> []
                in
                handle_statements func.body

        in
        ignore(List.map handle_method methods);
        []



let check pgm = match pgm with(*function*)
        Program(cdecls) ->

        (* generate reserved functions and obtain their map *)
        let reserved_functions = get_reserved_funcs in
        let reserved_maps = List.fold_left (fun m func ->
         ↪  StringMap.add func.sfname func m) StringMap.empty
         ↪  reserved_functions in

        (* get class_map for the given class *)
              let class_maps = get_class_maps cdecls reserved_maps
               ↪  in

                (* make one pass over all functions to extract
                 ↪  lambda declarations and add the prototypes to
                 ↪  the lambda_func hashtable  *)
                let _ = List.map handle_lambdas cdecls in

        (* perform semantic checking of all fields and methods.
         ↪  Generate an SAST *)
```

```
            let sast = get_sast class_maps reserved_functions
             ↪  cdecls in


        sast
```

8.1.4 ast.ml

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

type op = Add | Sub | Mult | Div | Eq | Neq | Lt | Leq | Gt | Geq
 ↪   | Mod | Pow |
          And | Or | Not

type uop = Sub | Not

(* These are the primitive datatypes supported by espresso, along
 ↪  with Object *)
type primitive = Int | Bool | Void | String | Float | Char |
 ↪  Lambda | ObjTyp of string
type typ = Datatype of primitive | Hashmaptype of primitive *
 ↪  primitive | ArrayType of primitive * int

(*type data_typ =  Datatype of typ | Any *)

type formal = Formal of typ * string

type expr =
    Literal of int
  | Strlit of string
  | Floatlit of float
  | BoolLit of bool
  | Charlit of char
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | Call of string * expr list
  | LambdaCall of string * expr list
  | ArrayAccess of expr * expr
  | HashmapAccess of string * expr
  | ObjectAccess of expr * expr
  | Noexpr
  | This

type var_decl = Vdecl of typ * string

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
  | Foreach of typ * string * string * stmt
  | Lambda of typ * string * formal list * stmt
  | Break
  | Continue
```

```
  | Local of typ * string

type func_decl = {
    typ : typ;
    fname : string;
    formals : formal list;
    body : stmt list;
  }


type cbody = {
  fields : var_decl list;
  methods : func_decl list;

}

type cdecl = {
  cname : string;
  cbody : cbody;
}

type program = Program of cdecl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Pow -> "**"
  | Eq -> "=="
  | Neq -> "!="
  | Lt -> "<"
  | Leq -> "<="
  | Gt -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
    Sub -> "-"
  | Not -> "!"

let string_of_primitive = function
    Int -> "int" ^ " "
  | Bool -> "bool" ^ " "
  | Void -> "void" ^ " "
  | String -> "String" ^ " "
  | Float -> "float" ^ " "
  | Char -> "char" ^ " "
  | Lambda -> "lambda" ^ " "
  | ObjTyp(s) -> "class " ^ s ^ " "


(* Helper function to pretty print datatypes*)
let string_of_datatype = function
```

```
                ArrayType(p, sz)        -> (string_of_primitive p)
                 ↪  ^ "[" ^ (string_of_int sz) ^ "]"
              | Datatype(p) -> string_of_primitive p
              | Hashmaptype(p1, p2) -> "hashmap <" ^
              ↪  string_of_primitive p1 ^ "," ^
              ↪  string_of_primitive p2 ^ ">"

        (*|     Any                     -> "Any" *)

(*let string_of_typ = function
  Datatype(p) -> string_of_primitive p
  | _ -> ""*)


let string_of_vdecl (var_decl) = match var_decl with
    Vdecl (t, id) -> string_of_datatype t ^ " " ^ id ^ ";\n"


let string_of_object = function
                Datatype(ObjTyp(s))        -> s
        |         _ -> ""

(* Helper function to pretty print formal arguments *)
let string_of_formal = function
        Formal(t, name) -> (string_of_datatype t) ^ " " ^ name
    |   _               -> ""

let string_of_formal_name = function
        Formal(t, name) -> name
    |   _ -> ""

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Charlit(l) -> "'" ^ (String.make 1 l) ^ "'"
  | Strlit(s) -> s
  | Floatlit(s) -> string_of_float s
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
      ↪  string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr
  ↪  e2
  | ArrayAccess(v, e) -> string_of_expr v ^ "[" ^ string_of_expr e
  ↪  ^ "]"
  | HashmapAccess(v, e) -> v ^ "<" ^ string_of_expr e ^ ">"
  | ObjectAccess(e1, e2) -> string_of_expr e1 ^ "." ^
  ↪  string_of_expr e2
  | This -> "this"
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
      ↪  ")"
  | LambdaCall(f, el) ->
      "#" ^ f ^ "(" ^ String.concat ", " (List.map string_of_expr
      ↪  el) ^ ")"
  | Noexpr -> ""
```

```
let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
      ↪  "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
  ↪  string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
    string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
    "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ "
    ↪  ; " ^
    string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
  ↪  string_of_stmt s
  | Foreach(t,e1,e2,s) -> "foreach (" ^ string_of_datatype t ^ e1
  ↪  ^
   " : " ^  e2 ^ ")\n" ^ string_of_stmt s
  | Lambda(ret_typ, str,fml,stmt) -> "lambda :" ^
  ↪  (string_of_datatype ret_typ) ^ str ^ "(" ^ String.concat ",
  ↪  " (List.map string_of_formal fml) ^ ")" ^ string_of_stmt
  ↪  stmt
  | Break -> "break;\n"
  | Local(t,s) -> string_of_datatype t ^  s ^ ";\n"



let string_of_func_decl func_decl =
  string_of_datatype func_decl.typ ^ " " ^
  func_decl.fname ^ "(" ^ String.concat ", " (List.map
  ↪  string_of_formal func_decl.formals) ^
  ")\n{\n" ^
 (* String.concat "" (List.map string_of_vdecl func_decl.locals) ^
 ↪  *)
  String.concat "" (List.map string_of_stmt func_decl.body) ^
  "}\n"

let string_of_class class_decl =
    "class " ^ class_decl.cname ^ " {\n" ^
    String.concat "" (List.map string_of_vdecl
    ↪  class_decl.cbody.fields) ^ "\n" ^
    String.concat "\n" (List.map string_of_func_decl
    ↪  class_decl.cbody.methods) ^
    "}\n"


let string_of_program program = match program with
  Program cdecls ->
    String.concat "" (List.map (string_of_class) cdecls)
```

8.1.5 sast.ml

```
open Ast

type sexpr =
    SLiteral of int
```

```
  | SStrlit of string
  | SFloatlit of float
  | SBoolLit of bool
  | SCharlit of char
  | SId of string * typ
  | SBinop of sexpr * op * sexpr * typ
  | SUnop of uop * sexpr * typ
  | SAssign of sexpr * sexpr * typ
  | SCall of string * sexpr list * typ
  | SArrayAccess of sexpr * sexpr * typ
  | SHashmapAccess of string * sexpr * typ
  | SObjectAccess of sexpr * sexpr * typ
  | SNoexpr

(*type var_decl = Vdecl of typ * string*)


type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr * typ
  | SReturn of sexpr * typ
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SForeach of typ * string * string * sstmt
  | SBreak
  | SContinue
  | SLocal of typ * string

type ftype = Reserved | Udf


type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : formal list;
    sbody : sstmt list;
    sftype : ftype;
    scontext_class :  string;
    sthis_ptr : sexpr;
  }


type scbody = {
  sfields : var_decl list;
  smethods : sfunc_decl list;

}

type scdecl = {
  scname : string;
  scbody : scbody;
}

type sprogram =  {
        classes : scdecl list;
        functions : sfunc_decl list;
        main : sfunc_decl;
```

```
        reserved : sfunc_decl list;
}
```

8.1.6 codegen.ml

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

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

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

Detailed documentation on the OCaml LLVM library:

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

*)

open Sast

module L = Llvm
module A = Ast
module Sem = Semant
module Hash = Hashtbl

open Llvm.MemoryBuffer
open Llvm_bitreader
module StringMap = Map.Make(String)
let values:(string, L.llvalue) Hash.t = Hash.create 50
let params:(string, L.llvalue) Hash.t = Hash.create 50
let class_types:(string, L.lltype) Hash.t = Hash.create 10
let class_field_indexes:(string, int) Hash.t = Hash.create 50
let class_fields:(string, L.llvalue) Hash.t = Hash.create 50
let class_this:(string,L.llvalue) Hash.t = Hash.create 50



let context = L.global_context ()
let the_module = L.create_module context "Espresso Codegen"
let builder = L.builder context

let i32_t = L.i32_type context;;
let i8_t = L.i8_type context;;
let f_t = L.double_type context;;
let i1_t = L.i1_type context;;
let str_t = L.pointer_type i8_t;;
let i64_t = L.i64_type context;;
let void_t = L.void_type context;;

let is_loop = ref false

let (br_block) = ref (L.block_of_value (L.const_int i32_t 0))
let (cont_block) = ref (L.block_of_value (L.const_int i32_t 0))



(*Code generation for a string*)
```

```
let rec string_gen llbuilder s =
  L.build_global_stringptr s "tmp" llbuilder

(*Recursively return pointer type for array based on size*)
let rec get_ptr_type dt = match dt with
    A.ArrayType(t,0) -> get_llvm_type (A.Datatype(t))
  | A.ArrayType(t,i)  -> L.pointer_type (get_llvm_type
  ↪  (A.Datatype(t)))
  (*| A.ArrayType(t,i)  -> L.pointer_type (get_ptr_type
  ↪  (A.ArrayType(t,i-1)))*)
  | _ -> raise (Failure ("Invalid Array Pointer Type"))

(*return corresponding llvm types for Ast datatype - get_type*)
and get_llvm_type (dt : A.typ) = match dt with
  A.Datatype(Int) -> i32_t
| A.Datatype(Float) -> f_t
| A.Datatype(Bool)  -> i1_t
| A.Datatype(Char)  ->  i8_t
| A.Datatype(Void)  ->  void_t
| A.Datatype(String)  -> str_t
| A.Datatype(A.Lambda) -> L.pointer_type(find_class "Lambda")
| A.Datatype(ObjTyp(name))  ->  L.pointer_type(find_class name)
| A.ArrayType(prim,i) ->  get_ptr_type (A.ArrayType(prim,(i)))
| _ -> raise (Failure ("llvm type not yet supported"))


(*Find out if a class/struct in llvm name exists, during object
↪  declaration*)
and find_class name =
  try Hash.find class_types name
  with | Not_found  ->  raise(Failure ("Invalid class name"))

(*Code generation for any expression that is an id*)
let rec id_gen llbuilder id dt isderef checkparam =
  if isderef then
    try
      (* try parameters *)
        Hash.find params id
    with | Not_found ->
      (* try local variables *)
      try let _val = Hash.find values id in
      L.build_load _val id llbuilder
      with Not_found -> raise (Failure ("Unknown variable there "
      ↪  ^ id))
  else
    try Hash.find values id
    with | Not_found ->
    try
        let _val = Hash.find params id in
        if checkparam then raise (Failure ("Cannot assign to a
        ↪  parameter"))
        else _val
    with | Not_found -> raise (Failure ("Unknown variable here " ^
    ↪  id))

and binop_gen llbuilder expr1 op expr2 dt =
  let le1 = sexpr_gen llbuilder expr1 in
  let le2 = sexpr_gen llbuilder expr2 in
```

```
let type1 = Sem.get_type_from_sexpr expr1 in
let type2 = Sem.get_type_from_sexpr expr2 in

let int_ops e1 op e2 = match op with
                A.Add                   -> L.build_add e1 e2
                 ↪  "addtmp" llbuilder
        |       A.Sub                   -> L.build_sub e1 e2
        ↪  "subtmp" llbuilder
        |       A.Mult                   -> L.build_mul e1 e2
        ↪  "multtmp" llbuilder
        |       A.Div                   -> L.build_sdiv e1 e2
        ↪  "divtmp" llbuilder
        |       A.Mod                   -> L.build_srem e1 e2
        ↪  "sremtmp" llbuilder
        |       A.Eq                    -> L.build_icmp L.Icmp.Eq
        ↪  e1 e2 "eqtmp" llbuilder
        |       A.Neq                   -> L.build_icmp L.Icmp.Ne
        ↪  e1 e2 "neqtmp" llbuilder
        |       A.Lt                    -> L.build_icmp
        ↪  L.Icmp.Slt e1 e2 "lesstmp" llbuilder
        |       A.Leq                   -> L.build_icmp L.Icmp.Sle
        ↪  e1 e2 "leqtmp" llbuilder
        |       A.Gt                    -> L.build_icmp L.Icmp.Sgt
        ↪  e1 e2 "sgttmp" llbuilder
        |       A.Geq                   -> L.build_icmp L.Icmp.Sge
        ↪  e1 e2 "sgetmp" llbuilder
        |       A.And                   -> L.build_and e1 e2
        ↪  "andtmp" llbuilder
        |       A.Or                          -> L.build_or  e1
        ↪  e2 "ortmp" llbuilder
        |       _                          -> raise
        ↪  (Failure("unsupported operator for integer arguments
        ↪  "))
       in

let float_ops e1 op e2 = match op with
                A.Add                   -> L.build_fadd e1 e2
                 ↪  "flt_addtmp" llbuilder
        |       A.Sub                   -> L.build_fsub e1 e2
        ↪  "flt_subtmp" llbuilder
        |       A.Mult                   -> L.build_fmul e1 e2
        ↪  "flt_multtmp" llbuilder
        |       A.Div                   -> L.build_fdiv e1 e2
        ↪  "flt_divtmp" llbuilder
        |       A.Mod                   -> L.build_frem e1 e2
        ↪  "flt_sremtmp" llbuilder
        |       A.Eq              -> L.build_fcmp L.Fcmp.Oeq e1
        ↪  e2 "flt_eqtmp" llbuilder
        |       A.Neq                   -> L.build_fcmp L.Fcmp.One
        ↪  e1 e2 "flt_neqtmp" llbuilder
        |       A.Lt                    -> L.build_fcmp
        ↪  L.Fcmp.Ult e1 e2 "flt_lesstmp" llbuilder
        |       A.Leq                   -> L.build_fcmp L.Fcmp.Ole
        ↪  e1 e2 "flt_leqtmp" llbuilder
        |       A.Gt                    -> L.build_fcmp L.Fcmp.Ogt
        ↪  e1 e2 "flt_sgttmp" llbuilder
        |       A.Geq                   -> L.build_fcmp L.Fcmp.Oge
        ↪  e1 e2 "flt_sgetmp" llbuilder
```

```
        |          _                              -> raise
         ↪  (Failure("unsupported operation for floating point
         ↪  arguments"))
        in

  (* handle object comparisons *)
        let non_primitive_types e1 op e2 = match op with
                      A.Eq -> L.build_is_null e1 "tmp" llbuilder
                |         A.Neq -> L.build_is_not_null e1 "tmp"
                 ↪  llbuilder
                |          _          -> raise (Failure("unsupported
                 ↪  operator for objects "))
        in

  let match_types dt = match dt with
    A.Datatype(Float) -> float_ops le1 op le2
  | A.Datatype(Int) | A.Datatype(Bool) | A.Datatype(Char)
   ↪  ->int_ops le1 op le2
  | A.Datatype(ObjTyp(_)) | A.ArrayType(_,_) | A.Hashmaptype(_,_)
   ↪  ->  non_primitive_types le1 op le2
  | _ -> raise(Failure("Unrecognized datatype! "))
  in
  match_types dt


and unop_gen llbuilder op expr dt =
  let unop_type = Sem.get_type_from_sexpr expr in
  let unop_llvalue = sexpr_gen llbuilder expr in
  let build_unop op utype exp_llval = match (op, utype) with
    (A.Sub, A.Datatype(Int)) -> L.build_neg exp_llval
     ↪  "int_unop_tmp" llbuilder
  | (A.Sub, A.Datatype(Float)) -> L.build_fneg exp_llval
   ↪  "float_unop_tmp" llbuilder
  | (A.Not, A.Datatype(Bool)) -> L.build_not exp_llval
   ↪  "bool_unop_tmp" llbuilder
  | _ -> raise(Failure("unsupported operator " ^ (A.string_of_uop
   ↪  op) ^ " and type " ^ (A.string_of_datatype utype) ^ " for
   ↪  unop"))
  in
  let handle_unop_type dt = match dt with
      A.Datatype(Int) | A.Datatype(Float) | A.Datatype(Bool) ->
       ↪  build_unop op dt unop_llvalue
  |    _ -> raise(Failure("invalid type for unop" ^
   ↪  (A.string_of_datatype dt)))
  in
  handle_unop_type dt

(*Code generation for Object Access*)
and obj_access_gen llbuilder lhs rhs d isAssign =

  let check_lhs lhs =
    match lhs with
      SId(s,d)  ->  id_gen llbuilder s d false false

    | SArrayAccess(_,_,_)  ->  raise (Failure ("yet to do : array
     ↪  as lhs of object invocation"))
    | _ -> raise (Failure ("LHS of object access must be object"))
```

```
  in

  let rec check_rhs isLHS par_exp par_type rhs=
    let par_str = A.string_of_object par_type in
    match rhs with

      SId(field,d)  ->
        let search_t = (par_str ^ "." ^ field) in
        let field_index = Hash.find class_field_indexes search_t
        ↪  in
        let _val = L.build_struct_gep par_exp field_index field
        ↪  llbuilder in
        let _val = match d with
          Datatype(ObjTyp(_)) ->
            if not isAssign then _val
            else L.build_load _val field llbuilder
          | _ ->
            if not isAssign then _val
            else L.build_load _val field llbuilder
        in
        _val
    | SCall(func_name, expr_list, ftype) ->   call_gen llbuilder
    ↪  func_name expr_list ftype

    | SObjectAccess(e1, e2, d)  ->
        let e1_type = Sem.get_type_from_sexpr e1 in
        let e1 = check_rhs true par_exp par_type e1 in
        let e2 = check_rhs true e1 e1_type e2 in
        e2

    (*| SObjectAccess(obj_name, exp, dt) ->
            let obj_typ = Semant.get_type_from_sexpr obj_name in
            let obj_val = check_rhs isAssign par_exp par_type
↪   obj_name in
            let mem_val = check_rhs isAssign obj_name obj_typ exp
↪   in
            mem_val *)
    | _ ->  raise(Failure ("yet to do : rhs types in object access
    ↪  codegen"))
  in

  let lhs_type = Sem.get_type_from_sexpr lhs in
  (*yet to do : treating arrays as objects? for length*)

  let lhs = check_lhs lhs in
  let rhs = check_rhs true lhs lhs_type rhs in
  rhs

(*Code generation for assign*)
and assign_gen llbuilder lhs rhs dt =
  let rhs_type = Sem.get_type_from_sexpr rhs in

  (*code generation for the lhs expression*)
  let lhs, isObjacc = match lhs with
  | Sast.SId(id,dt) ->  id_gen llbuilder id dt false false,false
  | SArrayAccess(st,exp,dt) -> array_access_gen llbuilder st exp
  ↪  dt true, false
```

```
  | SObjectAccess(se, sel, d) -> obj_access_gen llbuilder se sel d
   ↪  false,true
  | _ ->  raise (Failure ("LHS of an assignment must be
   ↪  stand-alone"))
  in

  let rhs = match rhs with
  | Sast.SId(id,dt) -> ( match dt with
                       | A.Datatype(ObjTyp(_)) -> id_gen llbuilder
                        ↪  id dt false false
                       | _ -> id_gen llbuilder id dt true false
                        )
  | Sast.SObjectAccess(e1,e2,d) ->  obj_access_gen llbuilder e1 e2
   ↪  d true
  | _ ->  sexpr_gen llbuilder rhs
  in

  let rhs = match dt with
    A.Datatype(ObjTyp(_)) ->
      if isObjacc then rhs
      else L.build_load rhs "tmp" llbuilder
  | _ ->  rhs
  in
  let rhs = match dt,rhs_type with
    A.Datatype(Char),A.Datatype(Int)  ->  L.build_uitofp rhs i8_t
     ↪  "tmp" llbuilder
  | A.Datatype(Int),A.Datatype(Char)  ->  L.build_uitofp rhs i32_t
   ↪  "tmp" llbuilder
  | _   -> rhs
  in

  ignore(L.build_store rhs lhs llbuilder);
  rhs



(*Code generation for array access*)
and array_access_gen llbuilder st exp dt isAssign =
  let index = sexpr_gen llbuilder exp in
  let index = match dt with
    A.Datatype(Char)  -> index
  | _ -> L.build_add index (L.const_int i32_t 1) "tmp" llbuilder
  in
  (*let  arr = id_gen llbuilder st dt true false in*)
  let arr = sexpr_gen llbuilder st in
  (*ignore(raise (Failure (L.string_of_llvalue index))); *)
  let _val = L.build_gep arr [| index |] "tmp" llbuilder in
  if isAssign
    then _val
    else L.build_load _val "tmp" llbuilder


(*Codegen for initialising an array*)
and array_init llbuilder arr arr_len init_val start_pos =
  let new_block label =
    let f = L.block_parent (L.insertion_block llbuilder) in
    L.append_block (L.global_context ()) label f
  in
```

```
  let bbcurr = L.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);
  L.position_at_end bbcond llbuilder;

  (*manage counter for length of array*)
  let counter = L.build_phi [L.const_int i32_t start_pos, bbcurr]
   ↪  "counter" llbuilder in
  L.add_incoming ((L.build_add counter (L.const_int i32_t 1) "tmp"
   ↪  llbuilder), bbbody) counter;
  let cmp = L.build_icmp L.Icmp.Slt counter arr_len "tmp"
   ↪  llbuilder in
  ignore(L.build_cond_br cmp bbbody bbdone llbuilder);
  L.position_at_end bbbody llbuilder;

  (*Assign array position to init_val*)
  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);
  L.position_at_end bbdone llbuilder



(*Code generation for array creation, allocating space for the
 ↪  array*)
and array_create_gen llbuilder t exp_t el =
  match exp_t with
    A.ArrayType(A.Char,_) ->
      let e = el in
      let size = (sexpr_gen llbuilder (SLiteral(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 (L.pointer_type t) "tmp"
       ↪  llbuilder in
      arr
  | _ ->
    let e = el in
    let t = get_llvm_type t in
    let size = (sexpr_gen llbuilder (SLiteral(e))) in
    let size_t = L.build_intcast (L.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 (L.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 (L.pointer_type t) "tmp"
     ↪  llbuilder in

    (*let arr_len_ptr = L.build_pointercast arr (L.pointer_type
     ↪  i32_t) "tmp" llbuilder in

    (*Store the length of the array*)
    ignore(L.build_store size_real arr_len_ptr llbuilder);
```

```
    array_init llbuilder arr_len_ptr  size_real (L.const_int i32_t
 ↪  0) 0;*)
    arr



(*Code generation for an expression*)
and sexpr_gen llbuilder = function
    SLiteral(i) ->  L.const_int i32_t i
  | SBoolLit(b) -> if b then L.const_int i1_t 1 else L.const_int
   ↪  i1_t 0
  | SFloatlit(f)  ->  L.const_float f_t f
  | SStrlit(s)  ->   string_gen llbuilder s
  | SCharlit(c) ->  L.const_int i8_t  (Char.code c)
  | SId(name,dt)  ->  id_gen llbuilder name dt true false
  | SBinop(expr1, op, expr2, dt) -> binop_gen llbuilder expr1 op
   ↪  expr2 dt
  | SUnop(op, e, dt) -> unop_gen llbuilder op e dt
  | SAssign(exp1,exp2,dt) ->  assign_gen llbuilder exp1 exp2 dt
  | SCall(name, expr_list, dt) -> call_gen llbuilder name
   ↪  expr_list dt
  | SArrayAccess(name,exp,dt) ->  array_access_gen llbuilder name
   ↪  exp dt false
  | SObjectAccess(e1,e2,d) ->  obj_access_gen llbuilder e1 e2 d
   ↪  true
  | SNoexpr       -> L.build_add (L.const_int i32_t 0) (L.const_int
   ↪  i32_t 0) "nop" llbuilder

  | _ -> raise (Failure "Not supported in codegen yet")

and call_gen  llbuilder func_name expr_list dt =
  let match_sexpr se = match se with
      SId(id, dt) -> let is_deref = match dt with
          Datatype(ObjTyp(_)) -> false
      | _ -> true
      in id_gen llbuilder id dt is_deref false
  |   se -> sexpr_gen llbuilder se
  in
  match func_name with
    "print_int" | "print_char"
    | "print_float" | "print_string"
    | "print_char_array" -> print_gen llbuilder expr_list
    | _ -> (let params = List.map match_sexpr expr_list in
            (* func_name is unique since it is prepended with
             ↪  class_name always *)
           match dt with
               Datatype(Void) -> L.build_call (func_lookup
                ↪  func_name) (Array.of_list params) "" llbuilder
             | _ -> L.build_call (func_lookup func_name)
              ↪  (Array.of_list params) "tmp" llbuilder
           )
  (*| _ -> raise(Failure("function " ^ func_name ^ " did not match
   ↪  any known function!"))*)

and func_lookup fname = match (L.lookup_function fname the_module)
 ↪  with
    None -> raise (Failure("function " ^ fname ^ " was not
     ↪  found!"))
```

```
  | Some func -> func


and print_gen llbuilder expr_list =
    (* currently we don't support boolean types *)
    (* generate llvm code for the expression list *)
    let params = List.map (fun expr -> sexpr_gen llbuilder expr)
    ↪  expr_list in
    let param_types = List.map (Semant.get_type_from_sexpr)
    ↪  expr_list in
    let get_format_string dt = match dt with
            A.ArrayType(Char, _)          -> "%s"
                |          A.Datatype(Int)                  -> "%d"
                |          A.Datatype(Float)          -> "%f"
                |          A.Datatype(String)          -> "%s"
                |          A.Datatype(Char)                  -> "%c"
                |          _
                ↪
                ↪  -> raise (Failure("Datatype not supported by
                ↪  codegen!"))
    in
    let fmt_str = List.fold_left (fun s t -> s ^ get_format_string
    ↪  t) "" param_types in
    let s = sexpr_gen llbuilder (SStrlit(fmt_str)) in
          let zero = L.const_int i32_t 0 in
          let s = L.build_in_bounds_gep s [| zero |] "tmp"
          ↪  llbuilder in
          L.build_call (func_lookup "printf") (Array.of_list (s ::
          ↪  params)) "tmp" llbuilder

(*Code generation for if statement*)
and if_stmt_gen llbuilder exp then_st (else_st:Sast.sstmt) =
    let cond_val = sexpr_gen llbuilder exp in

    (*Write the first block, initial part, jump to relevant else
    ↪  parts as well*)
    let start_bb = L.insertion_block llbuilder in
    let the_func = L.block_parent start_bb in

    let then_bb = L.append_block context "then" the_func in

    (*Push out the 'then' output result/value*)
    L.position_at_end then_bb llbuilder;
    let _ = stmt_gen llbuilder then_st in


    (*codegen of then block modifies current block *)
    let new_then_bb = L.insertion_block llbuilder in

    (*push out else block in new location of llvm block code*)
    let else_bb = L.append_block context "else" the_func in
    L.position_at_end else_bb llbuilder;
    let _ = stmt_gen llbuilder else_st in


    let new_else_bb = L.insertion_block llbuilder in

    let merge_bb = L.append_block context "ifcont" the_func in
```

```
    L.position_at_end merge_bb llbuilder;

    let else_bb_val = L.value_of_block new_else_bb in

    L.position_at_end start_bb llbuilder;
    ignore(L.build_cond_br cond_val then_bb else_bb llbuilder);

    L.position_at_end new_then_bb llbuilder;
    ignore(L.build_br merge_bb llbuilder);
    L.position_at_end new_else_bb llbuilder;
    ignore(L.build_br merge_bb llbuilder);

    L.position_at_end merge_bb llbuilder;

    else_bb_val

(*Code generation for a for statement*)
and for_gen llbuilder init_st cond_st inc_st body_st =
  let old_val = !is_loop in
  is_loop := true;

  let the_func = L.block_parent (L.insertion_block llbuilder) in

  (*emit initialization code first*)
  let _ = sexpr_gen llbuilder init_st in

  (*Basically create the associated blocks for llvm : loop, inc,
   ↪  cond, afterloop*)
  let loop_bb = L.append_block context "loop" the_func in
  let inc_bb = L.append_block context "inc" the_func in
  let cond_bb = L.append_block context "cond" the_func in
  let after_bb = L.append_block context "afterloop" the_func in

  let _ = if not old_val then
    cont_block := inc_bb;
    br_block := after_bb;
  in

  (*hit the condition statement with a jump*)
  ignore (L.build_br cond_bb llbuilder);

  L.position_at_end loop_bb llbuilder;

  (*emit the code generated for the body of statements for the
   ↪  current loop*)
  ignore (stmt_gen llbuilder body_st);

  let bb = L.insertion_block llbuilder in
  L.move_block_after bb inc_bb;
  L.move_block_after inc_bb cond_bb;
  L.move_block_after cond_bb after_bb;
  ignore (L.build_br inc_bb llbuilder);

  (*Start physical insertion at inc*)
  L.position_at_end inc_bb llbuilder;

  (*emit the block of inc generated code*)
  let _ = sexpr_gen llbuilder inc_st in
```

```
  ignore (L.build_br cond_bb llbuilder);

  L.position_at_end cond_bb llbuilder;

  let cond_val = sexpr_gen llbuilder cond_st in
  ignore (L.build_cond_br cond_val loop_bb after_bb llbuilder);

  L.position_at_end after_bb llbuilder;

  is_loop := old_val;

  L.const_null f_t

(*Code generation for a while statement*)
and while_gen llbuilder cond_ body_ =
  let null_se = SLiteral(0) in
  for_gen llbuilder null_se cond_ null_se body_


(*Code generation for a return statement*)
and return_gen llbuilder exp typ =
  match exp with
    SNoexpr -> L.build_ret_void llbuilder
  | _ -> L.build_ret (sexpr_gen llbuilder exp) llbuilder


(*Code generation for local declaration*)
and local_gen llbuilder dt st  =
  let arr,t,flag = match dt with
          A.Datatype(A.ObjTyp(name)) -> (L.build_add (L.const_int
           ↪  i32_t 0) (L.const_int i32_t 0) "nop"
           ↪  llbuilder),find_class name,false
        (*| A.ArrayType(A.(),i) ->  array_create_gen llbuilder
        ↪  prim i st*)
        | A.ArrayType(prim,len)  ->  (array_create_gen llbuilder
        ↪  (A.Datatype(prim)) dt len),get_llvm_type dt,true
        | _ -> (L.build_add (L.const_int i32_t 0) (L.const_int
        ↪  i32_t 0) "nop" llbuilder),get_llvm_type dt,false
  in


  let alloc = L.build_alloca t st llbuilder in
  Hash.add values st alloc;

  if flag = false
    then  alloc

  else
    (*let lhs = SId(st,dt) in*)
    let generated_lhs = id_gen llbuilder st dt false false in
    ignore(L.build_store arr generated_lhs llbuilder);
    alloc

and break_gen llbuilder =
  let bblock = fun () -> !br_block in
  L.build_br (bblock ()) llbuilder

and continue_gen llbuilder =
```

```
  let bblock = fun () -> !cont_block in
  L.build_br (bblock ()) llbuilder

(*Codegen for stmt*)
and stmt_gen llbuilder = function
  SBlock st ->  List.hd(List.map (stmt_gen llbuilder) st)
| SExpr(exp,dt) -> sexpr_gen llbuilder exp
| SReturn(exp,typ) -> return_gen llbuilder exp typ
| SIf(exp,st1,st2)  ->  if_stmt_gen llbuilder exp st1 st2
| SFor(exp1,exp2,exp3,st) ->  for_gen llbuilder exp1 exp2 exp3 st
| SWhile(e,s) ->  while_gen llbuilder e s
| SLocal(dt,st) ->  local_gen llbuilder dt st
| SBreak -> break_gen llbuilder
| SContinue -> continue_gen llbuilder
|  _ -> raise (Failure ("unknown statement"))


let setup_this_pointer llbuilder class_name =
  let class_type = Hash.find class_types class_name in
  let alloc = L.build_malloc class_type (class_name ^
  ↪  "_heap_this") llbuilder in

  ignore(Hash.add class_this class_name alloc);
  alloc


let init_params func formals =
    let formals = Array.of_list (formals) in

    Array.iteri (fun i v ->
      let name = formals.(i) in
      let name = A.string_of_formal_name name in
      L.set_value_name name v;
      Hash.add params name v;
    ) (L.params func)

(* function prototypes are declared here in llvm. This is used
↪  later to generate Call instructions *)
let func_stub_gen sfunc_decl =
  let params_types = List.rev (List.fold_left (fun l-> (function
  ↪  A.Formal(ty, _) -> get_llvm_type ty :: l )) []
  ↪  sfunc_decl.sformals) in

let func_type = L.function_type (get_llvm_type sfunc_decl.styp)
↪  (Array.of_list params_types) in
(* raise(Failure("reached here!")) *)
  L.define_function sfunc_decl.sfname func_type the_module

(* function body is generated in llvm *)
let func_body_gen sfunc_decl =
  Hash.clear values;
  Hash.clear params;
  let func = func_lookup sfunc_decl.sfname in

  (* this generates the entry point *)
  let llbuilder = L.builder_at_end context (L.entry_block func) in
  let _ = init_params func sfunc_decl.sformals in
```

```
  (* initialize this pointer *)
  let this_type = A.Datatype(A.ObjTyp(sfunc_decl.scontext_class))
   ↪  in
  let this_name = (sfunc_decl.scontext_class ^ "_this" ) in

  let init_this = [SLocal(this_type, this_name)] in

  (* initialize Lambda object *)
  let lambda_obj_type = A.Datatype(A.ObjTyp("Lambda")) in
  let lambda_obj_name = "lambda_obj" in
  let init_lambda_obj = [SLocal(lambda_obj_type, lambda_obj_name)]
   ↪  in

  (* setup this pointer on the heap if it doesn't exist for this
   ↪  class *)
  (*let _ =
      try Hash.find class_this sfunc_decl.scontext_class
        with | Not_found -> setup_this_pointer llbuilder
↪  sfunc_decl.scontext_class
  in*)

  (* create a pointer to the this object stored on the heap *)
  (*let generated_lhs = L.build_alloca (get_llvm_type this_type)
   ↪  this_name llbuilder in
  let this_val = Hash.find class_this sfunc_decl.scontext_class in
  ignore(Hash.add values this_name this_val);
  ignore(L.build_store this_val generated_lhs llbuilder);*)

  (*let generated_lhs = L.build_alloca (get_llvm_type this_type)
   ↪  this_name llbuilder in
  let this_val = Hash.find class_this sfunc_decl.scontext_class in
  ignore(Hash.add values this_name this_val);
  ignore(L.build_store this_val generated_lhs llbuilder);
 *)

  let _ = stmt_gen llbuilder (SBlock(init_lambda_obj @
   ↪  sfunc_decl.sbody)) in
  if sfunc_decl.styp = Datatype(Void)
                then ignore(L.build_ret_void llbuilder);
       ()


(*Class stubs and class gen created here*)
let class_stub_gen s =
    let class_type = L.named_struct_type context s.scname in
    Hash.add class_types s.scname class_type

let class_gen s =

  let class_type = Hash.find class_types s.scname in
  let type_list = List.map  (function A.Vdecl(d,_) ->
   ↪  get_llvm_type d) s.scbody.sfields in
  let name_list = List.map (function A.Vdecl(_,s) -> s)
   ↪  s.scbody.sfields in

  (*Addition of a key field to all structs/classes, assuming
   ↪  serialized*)
  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 name ->
    let n = s.scname ^ "." ^ name in
    Hash.add class_field_indexes n i;
  ) name_list;

  (*ignore(setup_this_pointer s.scname);*)
  L.struct_set_body class_type type_array true


(*Code generation for the main function of program*)
let main_gen main classes=
  Hash.clear values;
  Hash.clear params;
  let ftype = L.function_type i32_t [||] in
  let func = L.define_function "main" ftype the_module in
  let llbuilder = L.builder_at_end context (L.entry_block func) in

  (*let _ = List.map (fun s-> setup_this_pointer llbuilder s)
  ↪  classes in*)
  (* malloc the this pointer for the corresponding class if it is
  ↪  not already present *)
  (*let _ =
      try Hash.find class_this main.scontext_class
        with | Not_found -> setup_this_pointer llbuilder
↪  main.scontext_class
  in*)

  (* initialize this pointer *)
  let this_type = A.Datatype(A.ObjTyp(main.scontext_class)) in
  let this_name = ("this" ) in
  let init_this = [SLocal(this_type, this_name)] in

  (* initialize Lambda object *)
  let lambda_obj_type = A.Datatype(A.ObjTyp("Lambda")) in
  let lambda_obj_name = "lambda_obj" in
  let init_lambda_obj = [SLocal(lambda_obj_type, lambda_obj_name)]
  ↪  in

  (*let generated_lhs = L.build_alloca (get_llvm_type this_type)
  ↪  this_name llbuilder in
  let this_val = Hash.find class_this main.scontext_class in
  ignore(Hash.add values this_name this_val);
  ignore(L.build_store this_val generated_lhs llbuilder);*)

  let _ = stmt_gen llbuilder (SBlock(init_this @ init_lambda_obj @
  ↪  main.sbody)) in
  L.build_ret (L.const_int i32_t 0) llbuilder

(* declare library functions *)
let construct_library_functions =
  let printf_type = L.var_arg_function_type i32_t [|
  ↪  L.pointer_type i8_t |] in
      let _ = L.declare_function "printf" printf_type the_module
      ↪  in
  ()
```

```
let translate sprogram =
  (*match sprogram with *)

  (*(raise (Failure("In codegen")))*)
  let _ = construct_library_functions in
  let _ = List.map (fun s -> class_stub_gen s) sprogram.classes in
  let _ = List.map(fun s -> class_gen s) sprogram.classes in


  (* generate llvm code for function prototypes *)
  let _ = List.map (fun f -> func_stub_gen f) sprogram.functions
   ↪  in
  (* generate llvm code for the function body *)
  let _ = List.map (fun f -> func_body_gen f) sprogram.functions
   ↪  in

  let _ = main_gen sprogram.main sprogram.classes in

  the_module
```

8.1.7 espresso.ml

```
(* Top-level of the Espresso compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module
 ↪  *)
module L = Llvm

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 Ast 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 (Ast.string_of_program ast)
  | LLVM_IR -> print_string (L.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)
```