Java but better (but also worse)

JAVA+-

Zeynep Ejder, Ashley Daguanno, Amel Abid,
Anna Wen, Tin Nilar Hlaing

Table of Contents

Introduction

Java+- is a general purpose, object-oriented programming language. As its name suggests Java+- possesses many of Java's most useful features, such as classes, class objects, and scoping, as well as some that Java lacks, namely, tuples. We decided to integrate tuples into a Java-like language in order to merge the functionality of tuples that exist in languages like Python with the familiarity of an object-oriented programming language that most programmers are already comfortable and familiar with. It compiles to LLVM IR and thus is runnable on any platform that supports LLVM. Our essential rewriting of a language that already exists in its full capacity was born out of a desire to understand the internals of implementing an object-oriented programming language.

Language Tutorial

Environment

The compiler was built and tested on OSX and Ubuntu 16.04, and can be found in the following GitHub repository: https://github.com/aw2802/PLT

Download the repository to your local desktop via the command:

```
$ git clone https://github.com/aw2802/PLT.git
```

If you do not have git installed on your computer, you may get started with it here:
https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Java+- was written using the OCaml programming language and LLVM compiler infrastructure languages, and as such, requires both to be installed to compile and run.

See the OCaml and LLVM websites for installation instructions.
http://ocaml.org/docs/install.html
http://llvm.org/docs/GettingStarted.html

Running and Testing

To run the test suite, type in the following commands:

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

To write your own program and run it, write your program and give it the extension **".javapm"**

First the compiler is compiled and the output is generated to LLVM IR code to a **".ll"** file. Run your program by running this **".ll"** file using the command `lli <yourFileName>.ll`.

Steps to test your program:

```
$ make
$ ./javapm <nameOfYourProgram>.javapm > <nameOfYourProgram>.ll
$ lli <nameOfYourProgram>.ll
```

Example:

printTest.javapm

```
public class PrintTest{
      public void main(){
            print(“Hello There!”);
      }
}
```

Steps to run the above program:

```
$ make
$ ./javapm printTest.javapm > printTest.ll
$ lli printTest.ll
$ Hello There!
```

Language Manual

1 Introduction

Java+- is an object-oriented programming language largely based on Java that features the added functionality of tuples and compiles into LLVM.

2 Lexical Elements

2.1 Identifiers

Identifiers are sequences of characters used to name classes, variables, and functions.
Syntax for valid Java+- identifiers:

1. Each identifier must have at least one character.
2. Identifiers may contain capital letters, lowercase letters, digits, and the underscore symbol.
3. Identifiers may not begin with a digit
4. Identifiers are case sensitive.
5. Keywords may not be used as identifiers.

2.2 Keywords

```
boolean     for         and       float
int         while       or        return
char        if          public    new
true        elseif      private   void
false       else        null
```

2.3 Operators

More information on operators can be found in the section titled *Expressions and Operators*.
Supported operators include:

Arithmetic	**Comparison**	**Logical**
+	==	and
-	!=	or
*	<=	!
/	>=	
	<	
	>	

2.4 White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character, and the vertical tab character. White space is ignored (outside of string and character constants), and is therefore optional, except when it is used to separate tokens. This means that the two following functions are functionally the same.

```
public void getFive() {
      number five = 5;
      print(five);
}

public void getFive() {number five = 5; print(five);}
```

No white space is required between operators and operands.

```
/* All of these are valid. */

x=y+z;
x = y + z ;
x=array[2];
x = array [ 2 ] ;
```

Furthermore, wherever one space is allowed, any amount of white space is allowed.

```
/* These two statements are functionally identical. */
x= x+1;

x
      =x+1        ;
```

In string constants spaces and tabs are not ignored, but rather are part of the string. Therefore, the following two strings are not the same.

```
      "mac and cheese"
      "mac              and               cheese"
```

2.5 Comments

Java+- supports two types of comments

1. `/* multiline comments */`
   The compiler will ignore everything between /* and */
2. `// single line comments`
   The compiler will ignore everything from // to the end of the line.

3 Data Types

Java+- is a strongly typed language. One must specify a variable's type at the time of its declaration.

3.1 Primitive Types

Primitive Type	Description
`float`	a single-precision 32-bit IEEE 754 floating point
`int`	a 32-bit signed two's complement integer
`char`	a single 16-bit Unicode character
`boolean`	a logical entity that can have one of two values: true or false

3.2 Composite Types

Composite Type	Description
`Tuple`	a sequence of immutable objects separated by commas and enclosed within parentheses
`array`	a fixed-size sequential collection of elements of the same type separated by commas and enclosed within square brackets
`String`	a sequence of characters surrounded by double quotes. each character in a string has an index, and indexing begins at 0. a string's length corresponds to the number of characters it contains.

3.3 Tuples

3.3.1 Creating Tuples

Upon creation, one must use the *new* operator and indicate the types of the objects that the tuple will contain.

```
Tuple<int, boolean> myTup = new Tuple<int, boolean>(49, true);
```

3.3.2 Accessing Tuple Elements

Use square brackets to access an element at a particular index. Indexing begins at 0.

```
print(myTup[0]); // prints “A string.”
```

3.4 Arrays

3.4.1 Declaring Arrays

An array declaration has two components: the type of the array, and the name of the array. The type is indicated as *dataType[]*. The array’s name can be anything so long as it is not a Java+- keyword.

```
number[] myArr;
```

3.4.2 Creating and Initializing Arrays

The size of the array must be indicated at the time of its creation and is immutable.
One way to create an array is with the *new* operator. The following statement will allocate an array with enough memory for five number elements and assign it to the variable *myArr*.

```
int[] myArr = new int[5]; // OK
int[] myArr = new int[];  // NOT OK, must indicate array size
```

All values are initialized to 0 upon creation.
To add values to the array, you must set each value one at a time.

```
myArr[0] = 0;
myArr[1] = 2;
```

3.4.3 Accessing Array Elements

Elements are accessed by their numerical index.

```
int[] myArr = new int[2];
myArr[0] = 5;
print(“Elem at index 1: “ + myArr[1]); // prints “Elem at index 1: 5”
```

To assign or modify array values the syntax is as follows:

```
myArr[0] = 1;
myArr[1] = 45;
```

3.5 Classes and Objects

3.5.1 Classes

A *class* is a blueprint that describes the behavior that an object of its type supports. A class may contain fields and methods to describe the behavior of these objects.

A class's general syntax is as follows:

```
<class scope> class <class name> {
       <field declarations>
       <constructor declaration>
       <method declarations>
}
```

A class may contain either of the following variable types:

1. Local Variables: Variables defined within methods and constructors. These variables will be declared and initialized inside methods and will be inaccessible outside of them.
2. Instance Variables: Variables defined within a class but outside any method. These variables may be accessed by any method or constructor in the class.

Classes may have constructors, which must be invoked to create an object of that class type. A class constructor may take zero or more arguments, and must have the same name as the class it belongs to. There is no need to declare a scope as all constructors are public by default.

The following is an example of a constructor:

```
public class Car {
      int cost;

      Car(int c) {
          cost = c;
      }
}
```

3.5.2 Objects

Objects are defined by classes and have a set of behaviors and attributes. In Java+-, the *new* keyword is used to create new objects. The new operator returns a reference to a new instance of a class.

The syntax for creating an object is as follows:

```
Car myCar = new Car(5);
```

One may access instance variables and methods through these objects. To access an instance variable, the general syntax is:

```
<classname>.<attributeName>

print(myCar.cost);                  // prints 5

//setting public instance variables
```

// no permissions to set private instance variables???

4 Expressions and Operators

4.1 Assignment Operator

Operator	Type	Associativity	Example
=	Assignment	Right to left	number x = 5

Java +- implements the standard assignment operator, = , to store the value of the right operand in the variable of the left operand. Left and right operands must be of the same type. The left operand cannot be a literal value.

```
String sayHi = "hello world" // good
5 = 8;                       // bad
```

4.2 Arithmetic Operators:

Java +- has standard arithmetic operators that are applied to operands of type number. Left and right operands must be of the type number.

Assume variable X = 20, variable Y = 5:

Operator	Type	Associativity	Example
+	Addition	Left to right	X + Y will give 25
-	Subtraction	Left to right	X – Y will give 15
*	Multiplication	Left to right	X * Y will give 100
/	Division	Left to right	X / Y will give 4
+	Unary plus	Right to left	print(+X) will print 20
-	Unary minus	Right to left	print(-X) will print 5

4.3 Comparison Operators:

Comparison operators operate on operands of the same type.

Assume variable X = 20, variable Y = 5:

Operator	Type	Associativity	Example
==	Equal to	Left to right	(X == Y) will return false

!=	Not equal to	Left to right	(X != Y) will return true
>	Greater than	Left to right	(X > Y) will return true
>=	Greater than or equal to	Left to right	(X >= Y) will return true
<	Less than	Left to right	(X < Y) will return false
<=	Less than or equal to	Left to right	(X <= Y) will return false

4.4 Logical Operators

Operator	Type	Associativity	Example
and	Logical AND	Left to right	(true and false) will give false
or	Logical OR	Left to right	(true or false) will give true
!	Logical NOT	Right to left	(!true) will give false

4.5 Operator Precedence

For an expression that contains multiple operators, the operators are grouped based on rules of precedence.

The following list is presented in order of highest to lowest precedence; operators are applied from left to right:

1. Method calls and member access
2. Logical Not, Unary Plus and Minus,
3. Multiplication and division expressions
4. Addition and subtraction expressions
5. Greater than, less than, greater than or equal to, and less than or equal to expressions
6. Equal to and not equal to expressions
7. Logical AND expressions
8. Logical OR expressions
9. All assignment expressions

4.6 Order of Evaluation

Subexpressions will be evaluated from left to right.

```
( A() + B() ) + ( C() * D() )
```

According to this example, A() will be called first followed by B(), C(), and D() regardless of operator precedence.

5 Statements

In Java+-, a statement is used to create actions and to control flow throughout the program.

5.1 Expression Statements

A statement forms a complete unit of execution. Assignment expressions, method invocations, and object creation expressions may all be turned into statements by adding a semicolon (;) at the end.

```
x = 2.4;                              //assignment statement
print(“Hello World!”);                //method invocation statement
Car myCar = new Car();                //object creation statement
```

In addition to expression statements, there are two other types of statements: *declaration statements* and *control flow statements*.

5.2 Declaration Statements

A declaration statement is used to declare a variable by indicating its data type and name. One may also initialize it with a value. One may declare more than one variable of the same data type in the same declaration statement.

```
int x;
float y = 9.99;
```

5.3 Control Flow Statements

Statements are generally executed from top to bottom. However, control flow statements break up the execution flow by performing decision making, looping and branching specific blocks of code.

5.3.1 Decision-Making Statements (if-then, if-then else)

These statements tell the program to execute a particular section of code *only if* a specified condition evaluates to true. The opening and closing braces are mandatory to avoid the dangling else problem. One must use the keywords *if* and *else*.

An *else* must always follow an if.

```
if (x == 10) {      // “if” clause
 print("x is 10");  // “then” clause, will only execute if (x == 10) is true
}
else {
 print();          // the equivalent of a noop
}

if (x == 10) {
```

```
 print("x is 10");
} else {
 print("x is not 10"); // executed if (x==10) is false
}
```

5.3.2 Looping Statements (while, for)

While

The while statement repeatedly executes a block of code while a specific condition is true. Its syntax is as follows:

```
while(expression) {
 statement(s)
}
```

The *expression* above must return a boolean value. If it evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false.

An example of printing ints 1 through 5 using a while statement:

```
int count = 1;
while (count < 6) {
 println(“Count is: “, count);
 count = count + 1;
}
```

For

The for statement repeatedly loops over a code block until a particular condition is satisfied. Its syntax is as follows:

```
for (initialization; termination; increment or decrement) {
  statement(s)
}
```

These rules apply to the for statement:

1. The *initialization* expression is executed once as the loop begins.
2. The *termination* expression must return a boolean value. The loop stops executing once it evaluates to false.
3. The *increment* expression is invoked after each iteration of the for loop.

Here’s how to print ints 1 through 5 using a for statement:

```
int count;
for (count = 1; count < 6; count=count+1) {
```

```
 println(“Count is: “, count);
}
```

6 Functions

Functions are useful to separate your program into smaller logical pieces. Each function must be declared before it can be used.

6.1 Function

The required elements of a method declaration are a modifier (**public** or **private**), its return type, a name, a pair of parentheses (), and a body between braces {}.

```
public/private return-type function-name (list of parameters) {function-body}
```

- *public/private* indicates whether or not other classes can access this method. When public the function is globally accessible; when private the method is only accessible within its declared class.
- *return-type* indicates the type of the object being returned from the function. A function may return void, indicating that it does not return anything.
- *function-name* may be any valid identifier.
- *list of parameters* refers to a comma separated list of input parameters and is optional, but empty parentheses must be present if there is no list.
- Each function declaration must be followed by an opening brace { after which the body of the function should begin. The *function body* should be followed by a closing brace }.

Every function declaration within a class should have a unique combination of *name* and *list of parameters*.

Here is an example of a function declaration with no parameters:

```
public void myFunction() {
 statement(s);
}
```

A function declaration with two parameters that returns a number:

```
public number myFunction(number x, number y) {
 return 5;
}
```

6.2 The Return Statement

The return statement ends the execution of the function and returns the program control to the function that initially called this function.

```
return return-value;
```

The **return** keyword is reserved for denoting a return statement. It should be followed by the object being returned. The type of the returned value should match the return-type defined in the function declaration.

An example of a function with a return statement:

```
public number addOne(number x) {
 number y = x + 1;
 return y;
}
```

6.3 Calling Functions

You can call a function by using its name and its required parameters. Using the above addOne function:

```
addOne(1);
```

A function call can make up an entire statement (example above) or it can be a sub-statement as in the following example:

```
number result = addOne(1);
```

6.4 The Main Function

This is the entry point to your program. Every program in Java+- should have a main function. The **main** keyword is reserved for this function. The function returns void.

Here is how your main function should be structured at all times:

```
public void main() {
 body statement(s)
}
```

7 Program Structure and Scope

7.1 Program Structure

A Java++/-- program can exist in one file, but more commonly an elaborate program will be broken into Objects and each object will have its own source file.
Each source file should have the extension .javapm.

7.2 Scope

Scope refers to what Object can see what variable or function. Anything that is declared as **public** can be seen by anything. Anything declared as **private** can only be seen within the Class it is declared in.

Variables that are declared inside functions are only visible inside that function and their lifespan is only that function.

```
public void example(){
      number x = 0;
}
```

x is not accessible by anything outside the scope of function example().

8 Standard Library Functionality

`print()`	A function that prints the arguments passed to it to stdout.
`Dictionary`	A mapping type which maps keys (of any type) to values (of any type). Keys must be unique within the dictionary.

print() & println()

Raw string arguments must be enclosed in quotes when passed to print() and println().
Separate arguments using the character ‘,’ as with all function calls with multiple arguments.

```
boolean printMe = true;
print(printMe);                          // prints “true”;
print(“Hello World”)                     // prints “Hello World”
print(“My number is “ + 42 + “.”);       // prints “My number is 42.”
```

Project Plan

Project Plan

Our group met, at a minimum, twice weekly throughout the semester to discuss and work on the various components that make up our compiler – once with our TA Jacob Graff, and once as a full team during a designated meeting time. We met in subgroups on an as-needed basis during the latter half of the semester. Throughout our process we referred to our initial language reference manual, DICE (2015), and input and suggestions we received from our TA to grow and improve our compiler. Our test suite was modeled after microC's and was built on as we added to the functionality of our compiler. More information on the test suite can be found in the Test Plan section.

Programming Style Guide

General Principles

Above all, it was important to us that code be readable. Team members were advised to use their best judgement when it came to whitespace, indentation, and variable naming. All tests written in our language were expected to adhere to Java coding conventions.

Naming

Variable and function names should be camel cased in general. Underscores should be used only for function names that are very long and difficult to read if camel cased.

Whitespace, Line Length, & Readability

Tab length should be set at four spaces and indentation should be used to clearly mark and delineate code blocks. That is, indentation should tell the reader what code falls under what sets of parentheses.

Comments

Comments are encouraged, though not required. Any particularly dense code block should be commented, especially when a function name does not encapsulate all that the function does.

Project Timeline

Date	Tasks & Activities
09/20	First group meeting
09/28	Proposal
10/05	Receive proposal feedback from TA, revise our language definition
10/18	*Team away for Grace Hopper Conference*
10/24	LRM Outline and division of responsibilities
10/27	Language Reference Manual Due
11/30	Hello World Demo, Testing Suite is born
12/16	Final meeting with Jacob
12/16 - 12/20	Work towards finishing code generation, semantic analysis
12/20	Project Report submitted, Demo done

Roles and Responsibilities

Our official roles are as follows:

Student	Role
Zeynep Ejder	*Language Guru*
Amal Abid	*Systems Architect*
Tin Nilar Hlaing	*Systems Architect*
Ashley Daguanno	*Manager*
Anna Wen	*Tester*

We spent the first half of our time together this semester working as a group to complete the scanner, parser, and ast. It wasn't until code generation came into play that we split into subgroups to complete the compiler.

Subgroup	Responsibilities
Zeynep & Ashley	Code Generation, Scanning, Parsing, Testing, Final Report
Anna, Tin, & Amal	Semantic Analysis, Scanning, Parsing, Testing, Final Report

In reality, things weren't so black and white and roles were quite fluid. We often found ourselves adding components, removing features, and editing throughout the entire compiler regardless of the specific tasks we were assigned.

Ashley and Zeynep pair programmed whilst completing code generation, in part because Zeynep was unable to install the Virtual Machine and test locally. As a result, local changes were made on Zeynep's computer because editing outside of Vim ended up being much faster, and these were tested in the Ubuntu Virtual Machine on Ashley's computer.

Software Development Environment Used

Our development environment made use of OCaml, Bash, and LLVM. OCaml was used to write the compiler components, Bash was used to script and automate testing, and LLVM was used to compile and test the code that we were generating.

With regard to the coding environment used, the majority of our team developed both in the Ubuntu 16.04 virtual machine that was provided, and locally. We typically used Vim and the Sublime Text Editor. In general Sublime was used for files that were worked on locally and did not require LLVM to compile or test (parser, scanner, ast). Vim was used within the virtual machine to work on and test code generation.

We made use of a Github repository hosted on Github for version control and created and worked on all team documents (LRM, Final Report) using Google Docs. Communication was primarily done via Facebook Messenger and shared Google Documents that highlighted key features that should be prioritized by each subgroup.

Project Log

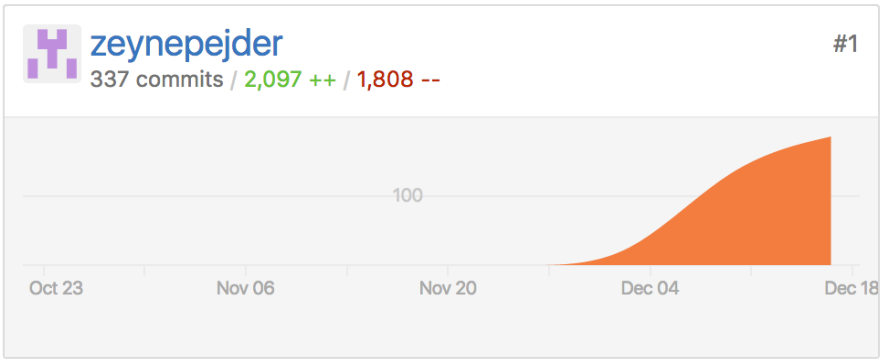
Contributors

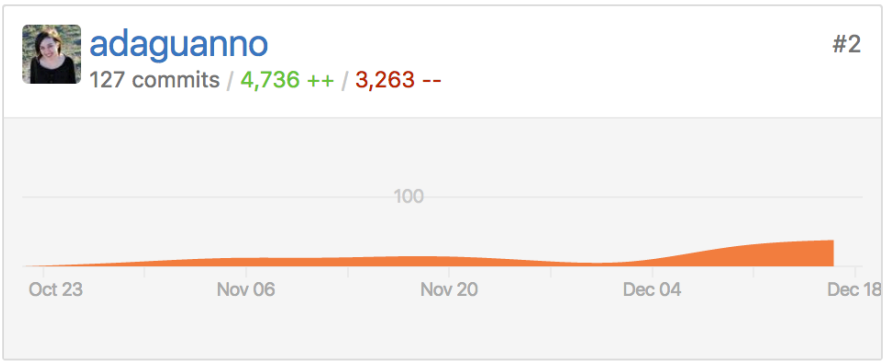
ast.ml: Zeynep, Amal, Ashley, Anna, Tin
scanner.mll: Zeynep, Amal, Ashley, Anna, Tin
parser.mly: Zeynep, Amal, Ashley, Anna, Tin
sast.ml: Zeynep, Amal, Ashley, Anna
semant.ml: Zeynep, **Amal**, Ashley, Anna, Tin
utils.ml: **Amal**, Ashley, Zeynep
codegen.ml: **Zeynep, Ashley**
Makefile: **Zeynep**
test.sh: **Ashley**
tests/: **Ashley, Zeynep**, Anna, Tin, Amal
javapm.ml: Zeynep

Statistical Breakdown of Git Commits

Ashley & Zeynep

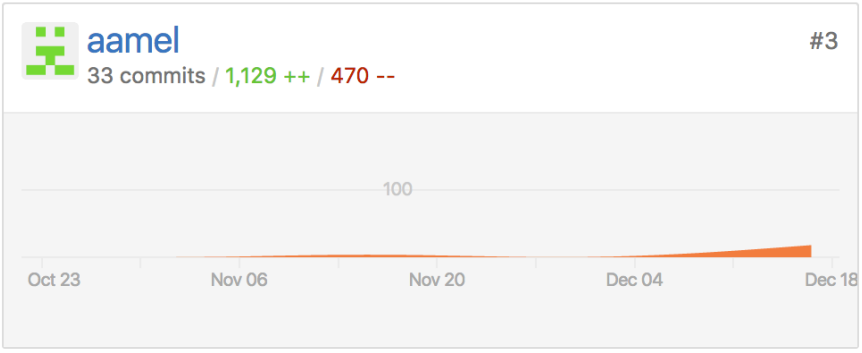
Ashley and Zeynep pair programmed throughout the semester – commits were made from Zeynep's local machine and tested on Ashley's virtual machine.



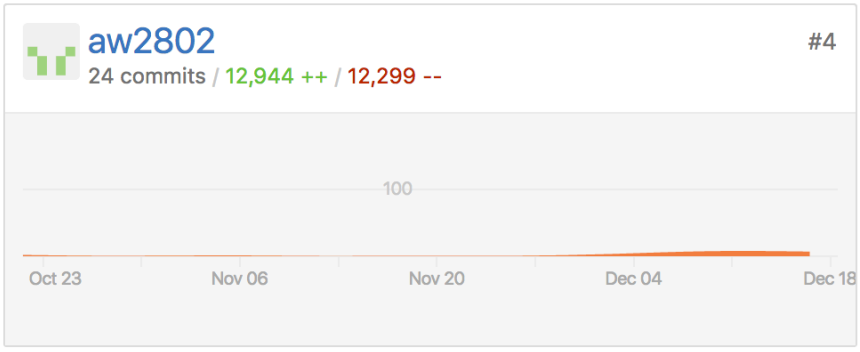


Note: Not pictured in this graph are commits Zeynep made before she set up her Github username on her local machine.

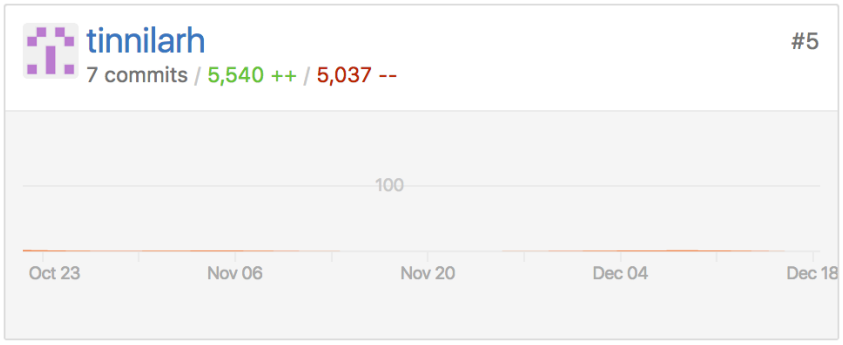
Amal



Anna

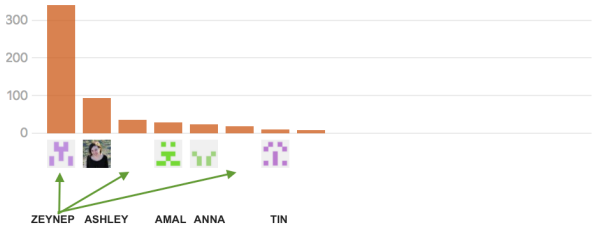


Tin



Overall Stats

Excluding merges, **8 authors** have pushed **545 commits** to master and **559 commits** to all branches. On master, **199 files** have changed and there have been **3,338 additions** and **311 deletions**.



Architectural Design

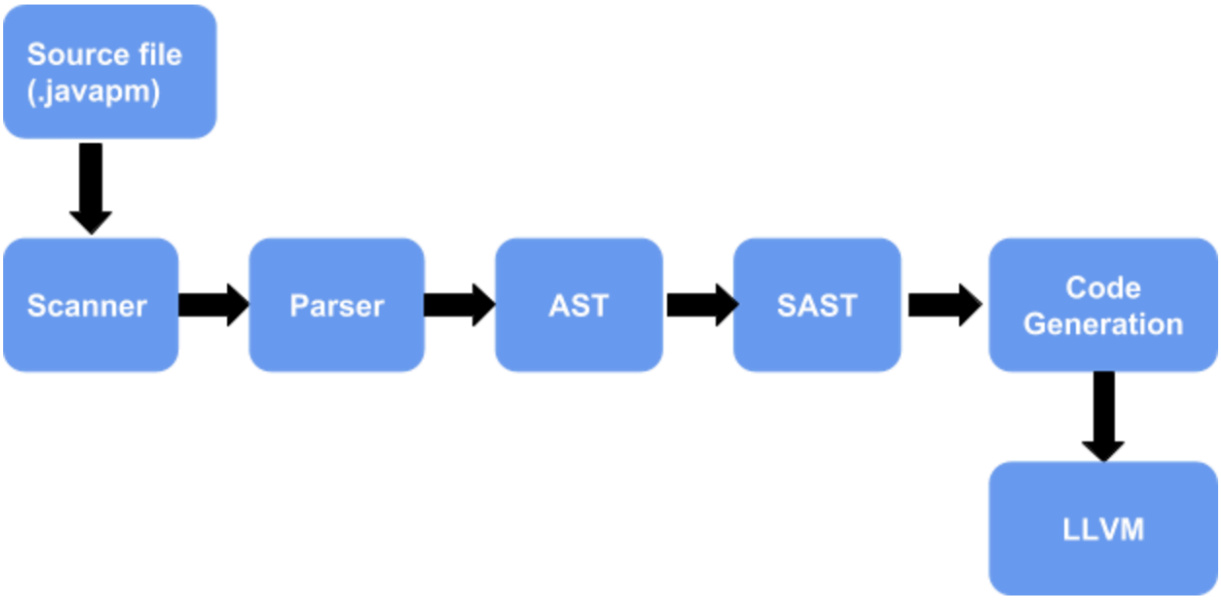


Figure 5.1 Overview of Compiler Architectural Design

Java+- follows the traditional compiler architectural design with the front end analysis (lexical analysis, parsing, abstract syntax tree (AST) generating, static semantic checking and the back end synthesis (translation of the abstract syntax tree into LLVM intermediate code).

We have a total of **5 modules**:

- Scanner.mll: reads the source file and assigns it into valid tokens
- Parser.mly: parses through tokens from the scanner and builds an abstract syntax tree
- Semant.ml: takes in AST and walks through each AST node for type and semantic checking
- Utils.ml: includes functions to print string representation of the program
- Codegen.ml: converts the semantically checked AST into LLVM IR

and **2 interfaces**:

- Ast.ml: program representation after parsing
- Sast.ml: semantically checked program representation

Scanner

The scanner reads the source file and does lexical analysis by assigning valid input into tokens. It raises the illegal character error if invalid input is found. The tokens are then passed to the parser.

Parser

The parser takes in tokens from the scanner and constructs the abstract syntax tree following the rules and definitions provided. The top level of the AST includes classes which then divides into variables, constructors and methods. Inside methods, there are statements which include some expressions containing literals, operations, variable assignment, access and etc.

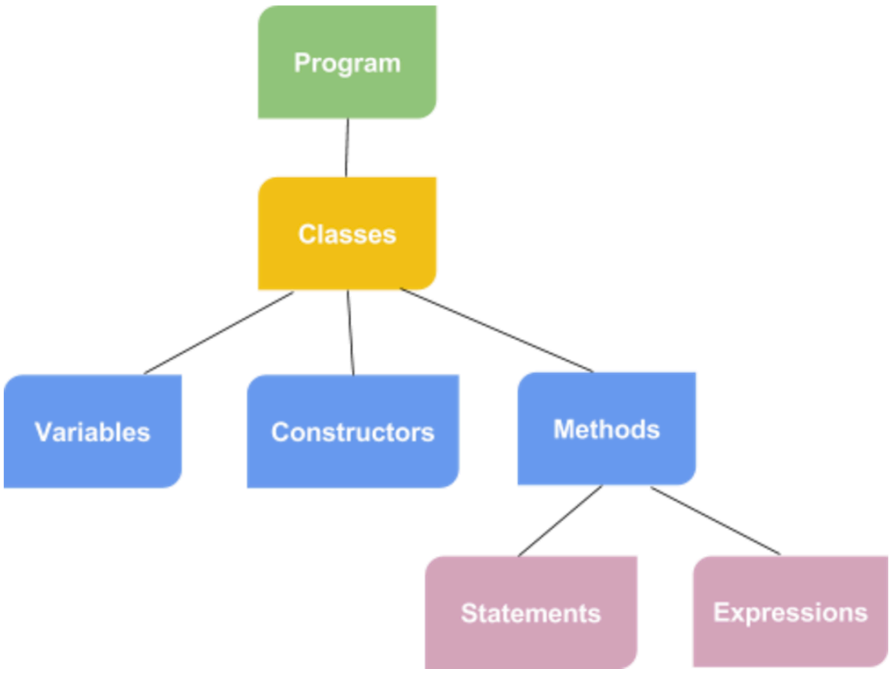


Figure 5.2 AST program representation

Semantic Checker

The semantic checker goes through AST nodes and checks if they are semantically correct such as whether there are duplicate variables inside one method or class scope, duplicate methods in one class scope. It also checks types of variables in formal, actual arguments of methods, assignment, operators and so on. In addition, it makes sure to consistently keep track of scoping using scope environments and checks if names or identifiers are well defined before they are used. This semantic checking then outputs the well-checked AST to be used in the code generator.

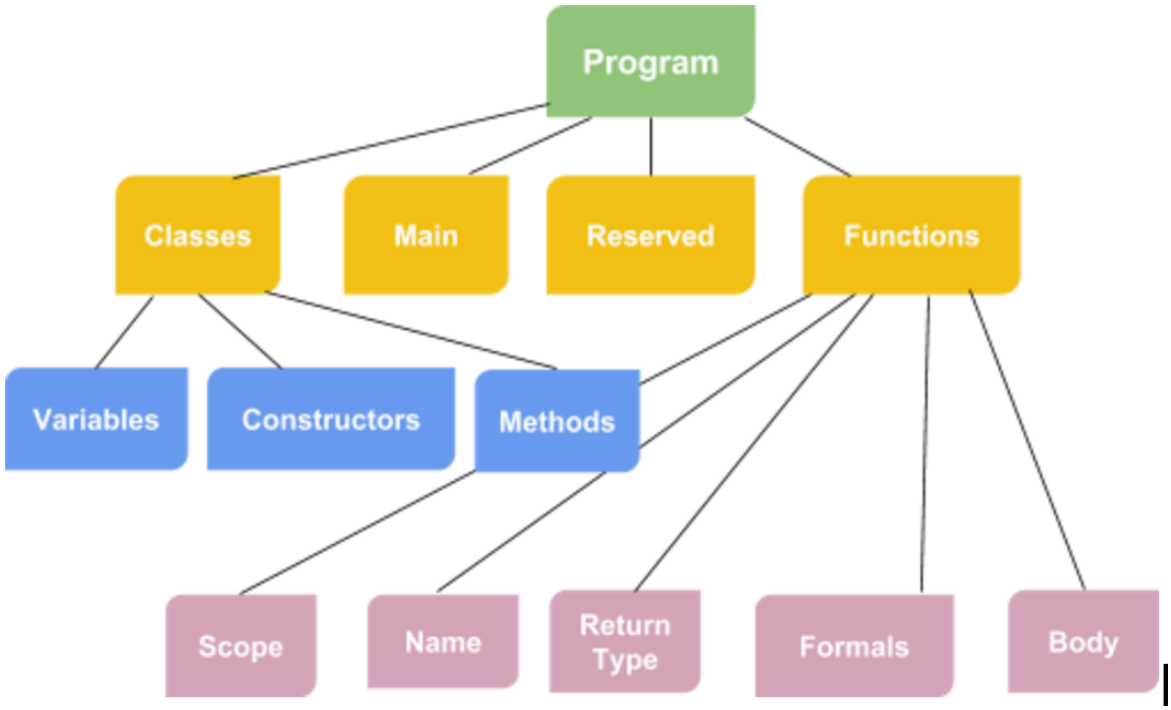


Figure 5.3 SAST program representation

Code Generator

Codegen.ml takes in a SAST and generates LLVM code by traversing the tree. We then extrac all of the program's Classes, the main method and its Functions from the root node, Program. The first step at code generation is to define each class of the program as a struct type and store the generated type in a hash table that keeps track of all classes, which are now converted to struct types.

Next each instance variable is added to hash tables that keep track of public or private variables and finally function definitions are generated for future use. Once we have class and functions definitions set up we start generating code for functions. Since our semantic checker couldn't return the proper type of expressions we do not internally name functions ClassName.FunctionName. This means that the current implementation of our language does not allow duplicate functions name across the board.

We separated constructors from general function generation because the constructor needs to allocate memory for the object that is being constructed and return this object at the end. Since statements like *return* are not in the constructor body that the user provides we add these at compilation time. Finally, the main method is built.

What works and what doesn't

Working

Statement and Expression Generation
Statements and expressions are matched to their respective subtypes and code is generated accordingly. We were able to write code for every possible match for a statement and expression in the Sast except for *SNull*.

Object creation
We are able to use the keyword *new* to crate new objects of defined classes. At first I had trouble conceptualizing how the instance variables will be initialized and stored so that the constructor and the other functions in the class can use them later on. I am currently storing these member variables in the struct that is created once an object is initialized. Due to set backs caused by variable type coming from Semant being stubbed to int I couldn't test my object creation code thoroughly.

Loops and If statements
Our compiler supports *for loops*, *while loops and if statements*. A for loop is converted to a while loop at compilation time. Loops and conditional branching were particularly fun to implement.

Binary and Unary Operators
We are able to perform Add, Subtract, Multiply, Divide, And, Or,
Not, and Comparison operations on integers, floats, chars, and booleans.

Variables
Our compiler supports variable declaration with scope, variable access and variable assignment.

Arrays
Arrays can be created, elements of the array can be accessed and changed.

Functions, Function Calls and Return Statements
Multiple functions can be defined and called. The return statement of functions could be any primitive data type supported or and Class defined by the user. If the *return* keyword is not used and the function has a return type of void we add a void return at compile time.

Tuples
Our compiler has support for the *Tuple* built in type of our language. In the background all Tuples are structs. Once a Tuple is created its elements can change value, but not type. Below are two diagrams that demonstrate the Tuple creation and access process:

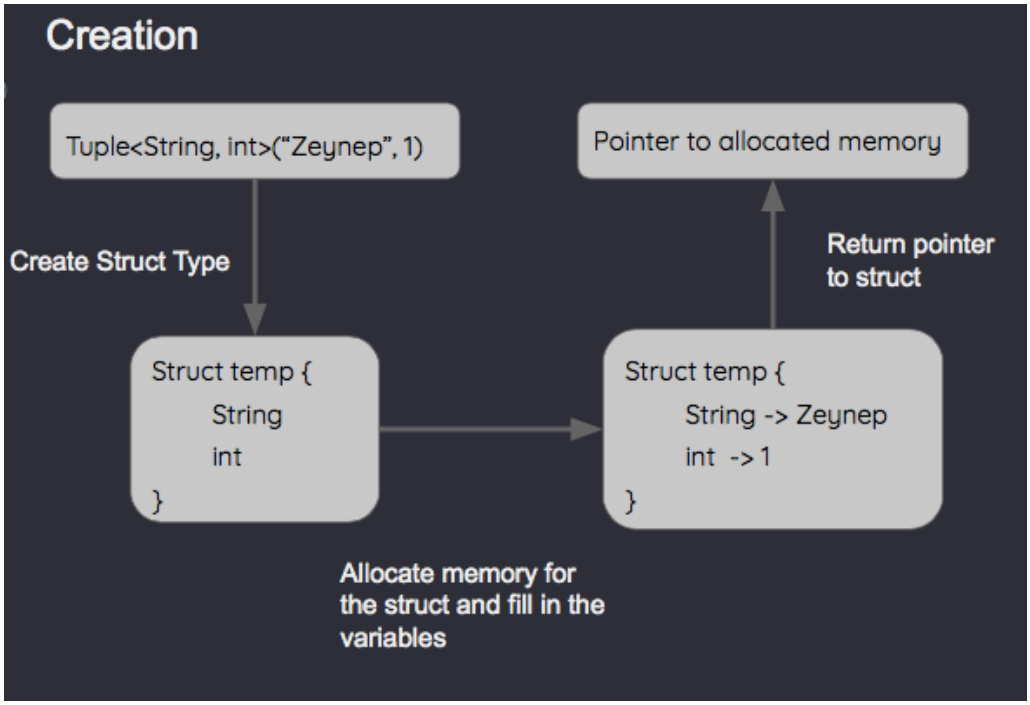


Figure 5.4 Tuple Creation Process

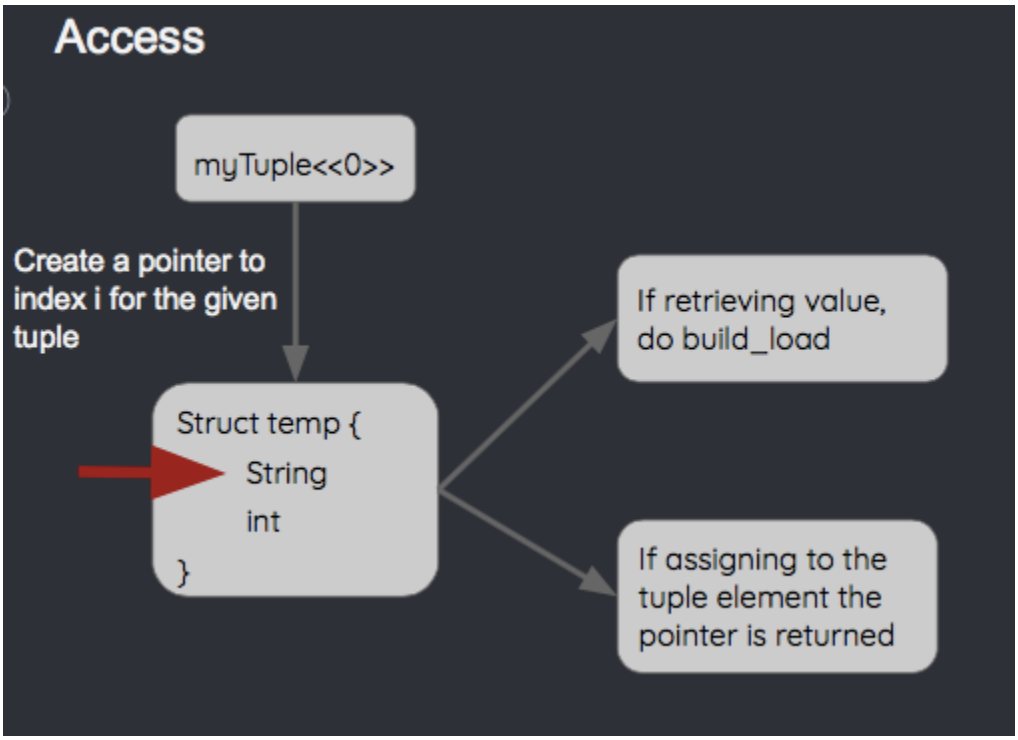


Figure 5.5 Tuple Access Process

Not working at present

Object access: We currently only have support for instance variable access and this had to be hardcoded to one Class name (Car) for testing purposes. This is again due to not knowing the type of the variable that represents the object.

Printing: The printing of literals is working, but due to variable types not being available at code generation, only integer variable are being printed. This is a result of each variable's (SId's) datatype being stubbed to int (JInt) in the semantic checker's conversion from AST to SAST.

Who Implemented What

All team members contributed substantial amount to scanner.mll, parser.mly, ast.ml, sast.ml and utils.ml together. Later, Amel, Anna and Tin Nilar helped implement semant.ml while Ashley and Zeynep mainly worked on codegen.ml.

Test Plan

All test cases in our test suite are under the folder “tests”. The test cases were first written to cover the basic features of Java +- and later expanded to cover more complexities as Java+- became able to handle new features.

Test Suite

Representative Programs

represent_code1.javapm

```
public class Tuples{
      public void main(){
            Tuple<int,int,int> tup = new Tuple<int, int, int>(2,2,2);
            tup<<2>> = 1;
            boolean test = true;
            while(test == true){
                  println("Is it winter break yet?");
                  if( tup<<0>> == 1 ){
                        test=false;
                        println("Finally, it's break!");
                  }
                  else{
                        println("No.");
                  }
                  println("Is it winter break yet?");
                  if(tup<<1>> == 1){
                        test=false;
                        println("Finally, it's break!");
                              }
                  else{
                        println("No.");
                  }
                  println("Ist it winter break yet?");
                  if(tup<<2>> == 1){
                        test=false;
                        println("Yes, it's finally break!");
                  }
                  else{
                        println("No.");
                  }
            }
      }
}
```

generated llvm code

```
LLVM generated code:
; ModuleID = 'javapm'

@tmp = private unnamed_addr constant [24 x i8] c"Is it winter break yet?\00"
@printf.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.2 = private unnamed_addr constant [21 x i8] c"Finally, it's break!\00"
@printf.3 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.4 = private unnamed_addr constant [4 x i8] c"No.\00"
@printf.5 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.6 = private unnamed_addr constant [24 x i8] c"Is it winter break
yet?\00"
@printf.7 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.8 = private unnamed_addr constant [21 x i8] c"Finally, it's break!\00"
@printf.9 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.10 = private unnamed_addr constant [4 x i8] c"No.\00"
@printf.11 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.12 = private unnamed_addr constant [25 x i8] c"Ist it winter break
yet?\00"
@printf.13 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.14 = private unnamed_addr constant [25 x i8] c"Yes, it's finally
break!\00"
@printf.15 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.16 = private unnamed_addr constant [4 x i8] c"No.\00"
@printf.17 = private unnamed_addr constant [4 x i8] c"%s\0A\00"

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

declare i8* @malloc(i32)

define i64* @lookup(i32 %c_index, i32 %f_index) {
entry:
 %tmp = alloca i64**
 %tmp1 = alloca i64*
 %tmp2 = getelementptr i64*, i64** %tmp1, i32 0
 store i64* bitcast (i32 ()* @main to i64*), i64** %tmp2
 %tmp3 = getelementptr i64**, i64*** %tmp, i32 0
 store i64** %tmp1, i64*** %tmp3
 %tmp4 = getelementptr i64**, i64*** %tmp, i32 %c_index
 %tmp5 = load i64**, i64*** %tmp4
 %tmp6 = getelementptr i64*, i64** %tmp5, i32 %f_index
 %tmp7 = load i64*, i64** %tmp6
 ret i64* %tmp7
}

define i32 @main() {
entry:
 %tup = alloca <{ i32, i32, i32 }>*
 %dummy = alloca <{ i32, i32, i32 }>
 %temp = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
```

```
%dummy, i32 0, i32 0
 store i32 2, i32* %temp
 %temp1 = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
%dummy, i32 0, i32 1
 store i32 2, i32* %temp1
 %temp2 = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
%dummy, i32 0, i32 2
 store i32 2, i32* %temp2
 store <{ i32, i32, i32 }>* %dummy, <{ i32, i32, i32 }>** %tup
 %tup3 = load <{ i32, i32, i32 }>*, <{ i32, i32, i32 }>** %tup
 %dummy4 = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
%tup3, i32 0, i32 2
 store i32 1, i32* %dummy4
 %test = alloca i1
 store i1 true, i1* %test
 br label %while

while:                                            ; preds = %merge25, %entry
 %test30 = load i1, i1* %test
 %binop_int31 = icmp eq i1 %test30, true
 br i1 %binop_int31, label %while_body, label %merge32

while_body:                                       ; preds = %while
 %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.1, i32 0, i32 0), i8* getelementptr inbounds ([24 x i8],
[24 x i8]* @tmp, i32 0, i32 0))
 %tup5 = load <{ i32, i32, i32 }>*, <{ i32, i32, i32 }>** %tup
 %dummy6 = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
%tup5, i32 0, i32 0
 %dummy7 = load i32, i32* %dummy6
 %binop_int = icmp eq i32 %dummy7, 1
 br i1 %binop_int, label %then, label %else

merge:                                            ; preds = %else, %then
 %printf10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @printf.7, i32 0, i32 0), i8* getelementptr inbounds ([24 x
i8], [24 x i8]* @tmp.6, i32 0, i32 0))
 %tup11 = load <{ i32, i32, i32 }>*, <{ i32, i32, i32 }>** %tup
 %dummy12 = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
%tup11, i32 0, i32 1
 %dummy13 = load i32, i32* %dummy12
 %binop_int14 = icmp eq i32 %dummy13, 1
 br i1 %binop_int14, label %then16, label %else18

then:                                             ; preds = %while_body
 store i1 false, i1* %test
 %printf8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.3, i32 0, i32 0), i8* getelementptr inbounds ([21 x i8],
[21 x i8]* @tmp.2, i32 0, i32 0))
 br label %merge
```

```
else:                                             ; preds = %while_body
 %printf9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.5, i32 0, i32 0), i8* getelementptr inbounds ([4 x i8], [4
x i8]* @tmp.4, i32 0, i32 0))
 br label %merge

merge15:                                          ; preds = %else18, %then16
 %printf20 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @printf.13, i32 0, i32 0), i8* getelementptr inbounds ([25 x
i8], [25 x i8]* @tmp.12, i32 0, i32 0))
 %tup21 = load <{ i32, i32, i32 }>*, <{ i32, i32, i32 }>** %tup
 %dummy22 = getelementptr inbounds <{ i32, i32, i32 }>, <{ i32, i32, i32 }>*
%tup21, i32 0, i32 2
 %dummy23 = load i32, i32* %dummy22
 %binop_int24 = icmp eq i32 %dummy23, 1
 br i1 %binop_int24, label %then26, label %else28

then16:                                           ; preds = %merge
 store i1 false, i1* %test
 %printf17 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @printf.9, i32 0, i32 0), i8* getelementptr inbounds ([21 x
i8], [21 x i8]* @tmp.8, i32 0, i32 0))
 br label %merge15

else18:                                           ; preds = %merge
 %printf19 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @printf.11, i32 0, i32 0), i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @tmp.10, i32 0, i32 0))
 br label %merge15

merge25:                                          ; preds = %else28, %then26
 br label %while

then26:                                           ; preds = %merge15
 store i1 false, i1* %test
 %printf27 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @printf.15, i32 0, i32 0), i8* getelementptr inbounds ([25 x
i8], [25 x i8]* @tmp.14, i32 0, i32 0))
 br label %merge25

else28:                                           ; preds = %merge15
 %printf29 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @printf.17, i32 0, i32 0), i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @tmp.16, i32 0, i32 0))
 br label %merge25

merge32:                                          ; preds = %while
 ret i32 0
}
```

Result

```
Is it winter break yet?
No.
Is it winter break yet?
No.
Is it winter break yet?
Yes, it’s finally break!
```

represent_code2.javapm

```
public class PrintList{
      public void main(){
            int [] list = new int[3];
            list[0] = 9;
            list[1] = 5;
            list[2] = 8;

            boolean print = true;
            while (print == true){
                  int i;
                  for (i = 0; i < 3; i = i + 1){
                        println(list[i]);
                  }
            print = false;
            }
      }
}
```

generated llvm code

```
; ModuleID = 'javapm'

@printf.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

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

declare noalias i8* @malloc(i32)

define i64* @lookup(i32 %c_index, i32 %f_index) {
entry:
 %tmp = alloca i64**
 %tmp1 = alloca i64*
 %tmp2 = getelementptr i64*, i64** %tmp1, i32 0
 store i64* bitcast (i32 ()* @main to i64*), i64** %tmp2
 %tmp3 = getelementptr i64**, i64*** %tmp, i32 0
 store i64** %tmp1, i64*** %tmp3
```

```
 %tmp4 = getelementptr i64**, i64*** %tmp, i32 %c_index
 %tmp5 = load i64**, i64*** %tmp4
 %tmp6 = getelementptr i64*, i64** %tmp5, i32 %f_index
 %tmp7 = load i64*, i64** %tmp6
 ret i64* %tmp7
}

define i32 @main() {
entry:
 %list = alloca i32*
 %malloccall = tail call i8* @malloc(i32 mul (i32 add (i32 mul (i32 ptrtoint
(i32* getelementptr (i32, i32* null, i32 1) to i32), i32 3), i32 1), i32
ptrtoint (i32* getelementptr (i32, i32* null, i32 1) to i32)))
 %"6tmp" = bitcast i8* %malloccall to i32*
 store i32 add (i32 mul (i32 ptrtoint (i32* getelementptr (i32, i32* null,
i32 1) to i32), i32 3), i32 1), i32* %"6tmp"
 br label %array.cond

array.cond:                                       ; preds = %array.init,
%entry
 %counter = phi i32 [ 0, %entry ], [ %tmp, %array.init ]
 %tmp = add i32 %counter, 1
 %tmp1 = icmp slt i32 %counter, add (i32 mul (i32 ptrtoint (i32*
getelementptr (i32, i32* null, i32 1) to i32), i32 3), i32 1)
 br i1 %tmp1, label %array.init, label %array.done

array.init:                                       ; preds = %array.cond
 %tmp2 = getelementptr i32, i32* %"6tmp", i32 %counter
 store i32 0, i32* %tmp2
 br label %array.cond

array.done:                                       ; preds = %array.cond
 store i32* %"6tmp", i32** %list
 %list3 = load i32*, i32** %list
 %"2tmp" = getelementptr i32, i32* %list3, i32 1
 store i32 9, i32* %"2tmp"
 %list4 = load i32*, i32** %list
 %"2tmp5" = getelementptr i32, i32* %list4, i32 2
 store i32 5, i32* %"2tmp5"
 %list6 = load i32*, i32** %list
 %"2tmp7" = getelementptr i32, i32* %list6, i32 3
 store i32 8, i32* %"2tmp7"
 %print = alloca i1
 store i1 true, i1* %print
 br label %while

while:                                            ; preds = %merge,
%array.done
 %print16 = load i1, i1* %print
 %binop_int17 = icmp eq i1 %print16, true
```

```
 br i1 %binop_int17, label %while_body, label %merge18

while_body:                                       ; preds = %while
 %i = alloca i32
 store i32 0, i32* %i
 br label %while8

while8:                                           ; preds = %while_body9,
%while_body
 %i14 = load i32, i32* %i
 %binop_int15 = icmp slt i32 %i14, 3
 br i1 %binop_int15, label %while_body9, label %merge

while_body9:                                      ; preds = %while8
 %i10 = load i32, i32* %i
 %"1tmp" = add i32 %i10, 1
 %list11 = load i32*, i32** %list
 %"2tmp12" = getelementptr i32, i32* %list11, i32 %"1tmp"
 %"3tmp" = load i32, i32* %"2tmp12"
 %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.1, i32 0, i32 0), i32 %"3tmp")
 %i13 = load i32, i32* %i
 %binop_int = add i32 %i13, 1
 store i32 %binop_int, i32* %i
 br label %while8

merge:                                            ; preds = %while8
 store i1 false, i1* %print
 br label %while

merge18:                                          ; preds = %while
 ret i32 0
}
```

Result

```
9
5
8
```

represent_code3.javapm

```
public class Animal{
      Animal(int x){
            println(x);
      }
}

public class HowManyAnimals{
      public void main(){
            println("How many cats do you have?");
            Animal cat = new Animal(6);
            println("You're a cat lady.");
      }
}
```

Generated llvm code

```
; ModuleID = 'javapm'

%Animal = type <{ i32 }>

@printf.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@tmp = private unnamed_addr constant [27 x i8] c"How many cats do you
have?\00"
@printf.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@tmp.3 = private unnamed_addr constant [19 x i8] c"You're a cat lady.\00"
@printf.4 = private unnamed_addr constant [4 x i8] c"%s\0A\00"

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

declare i8* @malloc(i32)

define i64* @lookup(i32 %c_index, i32 %f_index) {
entry:
 %tmp = alloca i64**, i32 2
 %tmp1 = alloca i64*, i32 0
 %tmp2 = getelementptr i64**, i64*** %tmp, i32 0
 store i64** %tmp1, i64*** %tmp2
 %tmp3 = alloca i64*
 %tmp4 = getelementptr i64*, i64** %tmp3, i32 0
 store i64* bitcast (i32 ()* @main to i64*), i64** %tmp4
 %tmp5 = getelementptr i64**, i64*** %tmp, i32 1
 store i64** %tmp3, i64*** %tmp5
 %tmp6 = getelementptr i64**, i64*** %tmp, i32 %c_index
 %tmp7 = load i64**, i64*** %tmp6
 %tmp8 = getelementptr i64*, i64** %tmp7, i32 %f_index
 %tmp9 = load i64*, i64** %tmp8
```

```
 ret i64* %tmp9
}

define %Animal* @Animal(i32 %x1) {
entry:
 %object = alloca %Animal
 %x = alloca i32
 store i32 %x1, i32* %x
 %x2 = load i32, i32* %x
 %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.1, i32 0, i32 0), i32 %x2)
 ret %Animal* %object
}

define i32 @main() {
entry:
 %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.2, i32 0, i32 0), i8* getelementptr inbounds ([27 x i8],
[27 x i8]* @tmp, i32 0, i32 0))
 %cat = alloca %Animal*
 %tmp = call %Animal* @Animal(i32 6)
 store %Animal* %tmp, %Animal** %cat
 %printf1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
[4 x i8]* @printf.4, i32 0, i32 0), i8* getelementptr inbounds ([19 x i8],
[19 x i8]* @tmp.3, i32 0, i32 0))
 ret i32 0
}
```

Result

```
How many cats do you have?
6
You’re a cat lady
```

Test Suite

test-add.javapm

```
public class TestAdd{
 public void main(){
   int i = 1 + 1;
   println(i);

   int j = i + 1;
   println(j);
 }
}
```

test-addAssign.javapm

```
public class TestAdd{
 public void main(){
   int i;
   i = 1 + 1;
   println(i);
   int j;
   j = 2 + 1;
   println(j);
 }
}
```

test-and.javapm

```
public class TestAnd {
 public void main(){
   int i = 1;
   int j = 3;
   if (true and false){
      println("hi");
   }
   else {
     println(i);
     println(j);
   }
 }
}
```

test-andOp.javampm

```
public class TestAndOp {
      public void main() {
            println(1 and 0);
            println(1 and 1);
      }
}
```

test-arithmetic.javapm

```
public class TestArith {
 public void main()
 {
   int i = 1 + 3 * 4 / 2 + 10;
   print(i);
 }
}
```

test-array.javapm

```
public class TestArray {
 public void main() {
   int[] b = new int[10];
   int i;
   b[0] = 0;
   for(i = 1; i < 4; i = i+ 1) {
      b[i] = b[0] + 1;
      println(b[i]);
   }
 }
}
```

test-array_create.javapm

```
public class TestArray {
 public void main() {
   int[] b = new int[10];
   println("Created Array!");
 }
}
```

test-array_object.javapm

```
public class Car{
 public string color;
 getColor(){
   color = "green";
   return color;
 }
}

public class TestArrayObject{
 public void main(){
   Car myCar = new Car();
   Car[] carArray = new Car[5];
   carArray[0] = myCar;
   print(carArray[0].getColor());
 }
}
```

test-assign.javapm

```
public class TestAssign {
      public void main() {
            int x = 5;
            println(x);
      }
}
```

test-bool.javapm

```
public class TestBool {

	public void main() {
		boolean trueBool = true;
		boolean falseBool = false;

		if(trueBool) {
			println("Inside truebool");
			println(trueBool);
		}
		else {
			println("True not processed correctly.");
		}

		if(falseBool) {

			println("False not processed correctly.");
		}
		else {

			println("Inside falsebool");
			println(falseBool);
		}

		if(true) {
			println(true);
		}
		else {
			println("Should never go here.");
		}

		println(false);
	}
}
```

test-char_decl.javapm

```
public class TestCharDecl {
	public void main() {
		char c = 'c';
		println(c);
	}
}
```

test-check_variable.javapm

```
public class DummyClass {
      public int x = 3;
      public void main() {
            print(3);
      }
}
```

test-comments.javapm

```
public class TestComments{
 public void main(){
   int i = 10;
   /* blah blah*/
   // comments..
   print(i);
 }
}
```

test-div.javapm

```
public class TestDivision{
 public void main(){
   int i = 10 / 5;
   print(i);
 }
}
```

test-equal.javapm

```
public class TestEqual{
 public void main(){
   int i= 2;
   if (i == 2){
     print(100);
   }
   else{
     print(200);
   }
 }
}
```

test-fibonacci.javapm

```
public class TestFibonacci{
 public int fib(int x){
   int z;
   if (x < 2){
     z = 1;
   }
   else{
     z = fib(x-1) + fib(x-2); //this. needed?
   }
   return z;
 }

 public void main(){
   int x;
   int y;
   int z;
   int m;

   x = 5;
   y = 6;
   z = fib(x); //this. needed?

   print(z);
 }
}
```

test-float.javapm

```
public class TestFloat{
 public void main(){
   float i;
   i = 2.25-0.25;
   print(i);
 }
}
```

test-for.javapm

```
public class TestFor{
 public void main(){
   int i;
   for (i = 0; i < 5; i = i +1){
     println(i);
   }
 }
}
```

test-for_nest.javapm

```
public class TestFor_Nest{
 public void main(){
   int i;
   int j;
   for (i = 0; i < 3; i = i+1){
     for(j = 0; j < 2; j = j+1){
       println("Inside");
     }
   }
 }
}
```

test-func_in_object.javapm

```
public class Calculator{
 public int add(int x, int y){
   int z;
   z = x + y;
   return z;
 }
}

public class TestFunction{
 public void main(){
   int x;
   int y;
   int z;
   x = 10;
   y = 5;

   Calculator c = new Calculator();
   z = c.add(x,y);

   print(z);
 }
}
```

test-gcd.javapm

```
public class TestGcd{
 public void main(){
   int x;
   int y;
   int z;
   x = 66;
   y = 98;

   while(x != y){
     if(x > y){
       x = x - y;
     }
     else{
       y = y - x;
     }
   }
   print(x);
 }
}
```

test-geq.javapm

```
public class TestGeq{
 public void main(){
   int i = 1;
   int j = 1;
   if (i >= j){
     print("yes");
   }
   else{
     print("no");
   }
 }
}
```

test-gt.javapm

```
public class TestGt{
 public void main(){
   int i = 4;
   int j = 1;
   if (i > j){
     print(1);
   } else{
     print(0);
   }
 }
}
```

test-hello.javapm

```
public class HelloWorld {
      public void main() {
            print("Hello World");
      }
}
```

test-if.javapm

```
public class TestIf{
 public void main(){
   println(111);

   if (true){
     println(1);
   }
   else{
     println(0);
   }

   println("Wahoo");
 }
}
```

test-if_nest.javapm

```
public class TestIfNested{
 public void main(){
   int i = 2;
   int j = 4;
   if (true){
     if (i == 2){
       if (i < j){
         print(j);
       }
       else{
         print("In else 1");
       }
     }
     else{
       print("In else 2");
     }
   }
   else{
     print("In else 3");
   }
 }
}
```

test-initialize.javapm

```
public class TestInitialize {
      public void main() {
            int x;
            print("It's okay not to initialize.");
      }
}
```

test-integer.javapm

```
public class TestInteger{
 public void main(){
   int i;
   i = 6/3;
   print(i);
 }
}
```

test-leq.javapm

```
public class TestLeq{
 public void main(){
   int i = 1;
   int j = 1;
   if (i <= j){
     print(1);
   }
   else{
     print(0);
   }
 }
}
```

test-lt.javapm

```
public class TestLt{
 public void main(){
   int i = 3;
   int j = 1;
   if (i < j){
     print(1);
   }
   else{
     print(0);
   }
 }
}
```

test-mult.javapm

```
public class TestMult{
 public void main(){
   int i = 2 * 5;
   println(i);
   println(4 * 2);
 }
}
```

test-mult_classes.javapm

```
public class Car {

}

public class TestMultClasses {
      public void main() {
            println("Success");
      }
}
```

test-mult_func_param.javapm

```
public class TwoFunctionsWithParameters {
      public void main() {
            println("Hi");
            int printMe = say5(5);
            println(printMe);
      }

      public int say5(int x) {
            println(x);
            return 5;
      }
}
```

test-mult_functions.javapm

```
public class TwoFunctions {
      public void main() {
            println("Hi");
            int printMe = sayBye();
            println(printMe);
      }
      public int sayBye() {
            println("Bye");
            return 1;
      }
}
```

test-negate.javapm

```
public class TestNegate {
      public void main() {
            int x = -1;
            println(x+1);
            println(x);
      }
}
```

test-not.javapm

```
public class TestNot{
 public void main(){
   int i = 1;
   int j = 3;
   if (i != 3){
     println(0);
   }
   else{
      println(1);
   }

   if(true) {
      println("True");
   }
   else {
      println("False");
   }
 }
}
```

test-notEqual.javapm

```
public class TestNotEqual{
 public void main(){
   int i = 3;
   if (i != 1){
     print(1);
   }
   else{
     print(0);
   }
 }
}
```

test-object.javapm

```
public class Car{
 Car(int x) {
      println(x);
 }
}

public class TestObject{
 public void main(){
      Car myCar = new Car(5);
      println("Created!");
 }
}
```

test-or.javapm

```
public class TestOr{
 public void main(){
   int i = 1;
   int j = 3;
   if ((i != 1) or (j == 3)){
     println("J was 3!");
   }
   else {
     println("Dummy print");
   }
 }
}
```

test-print_mult-args.javapm

```
public class TestPrintMultArgs {
      public void main() {
            print("hello ", "world");
            println();
            println("hello ", "world");
      }
}
```

test-printAdd.javapm

```
public class TestAdd{
 public void main(){

      int x = 4;
   print(x + 1);
 }
}
```

test-subtract.javapm

```
public class TestSubtract{
 public void main(){
   int i = 5;
   int j = 10 - i;
   print(j);
 }
}
```

test-tuple_access.javapm

```
public class TestTupleAccess {
      public void main() {
            int x = 5;
            Tuple<int, int> myTuple = new Tuple<int, int>(x, 10);

            println(myTuple<<0>>);
            myTuple<<0>> = 1;

            println(myTuple<<0>>);
            println(myTuple<<1>>);
      }
}
```

test-tuple_creation.javapm

```
public class TestTupleCreation {
      public void main() {
            Tuple<int> myTuple = new Tuple<int>(5);
      }
}
```

test-while.javapm

```
public class TestWhile{
      public void main(){
            int i;
            i = 1;
            while(i < 5){
                  println(i);
                  i = i +1;
            }
      }
}
```

test-while_for_nested.javapm

```
public class TestWhileForNested {
 public void main(){
   int i = 0;
   int j;
   while(i < 2){
     for(j = 0; j < 2; j = j + 1){
      println("Inside first test");
     }
     i = i + 1;
   }
   println();

   for(i = 0; i < 2; i = i + 1){
     j = 0;
     while(j < 2){
       println("Inside second test");
      j = j + 1;
     }
   }
 }
}
```

test-while_nested.javapm

```
public class TestWhileNested{
 public void main(){
   int i = 0;
   int j;
   while (i < 1){
     j = 0;
     while (j < 3){
       println("Inside");
       j = j+1;
     }
     i = i+1;
   }
 }
}
```

test-global_var.javapm

```
public class TestGlobalVar {
	public void main() {
		OtherClass x = new OtherClass();
	}
}

public class OtherClass {
	public int y;
	OtherClass() {
		y = 5;
	}
}
```

test-string2.javapm

```
public class String{
	private char[] string;

	puclic String(char[] s){
		string = s;
	}

	public String toString(){
		return new String(string);
	}

	public void printString(){
		println(string);
	}
}

public class Tester{
	public void main(){
		String s = new String("hello");
		s.printString();
	}
}
```

test-initialize.javapm

```
public class TestInitialize {
	public void main() {
		int x;
		print("It's okay not to initialize.");
	}
}
```

test-object_instance_var.javapm

```
public class Car{
	public int y;
	Car(int x) {
		int f;
		y = x;
		println("in constructor");
		//printMe();
	}
/*
	public void printMe(){
		println(y);
	}
	*/
}

public class TestObject{
 public void main(){
	Car myCar = new Car(5);
	println("Created!");
	int z = myCar.y;
	println(z);
 }
}
```

test-array_func_param.javapm

```
public class TestArrayParam {

	public void main() {
		int[] myArr = new int[2];
		printArr(myArr);
	}

	public int printArr(int[] a) {
		int x = a[0];
		println(x);
		return 1;
	}
}
```

fail-assign1.javapm

```
public class FailAssign1{
      public void main(){
            int i;
            boolean boo;
            i = 2;
            boo = true;
            i = false; /* Fail: assigning a boolean to an integer */
      }
}
```

fail-assign2.javapm

```
public class FailAssign2{
      public void main(){
            int i;
            boolean boo;
            boo = 2; /* Fail: assigning an integer to a boolean */
      }
}
```

test-assign3.javapm

```
public class FailAssign3{
      public void myvoid()
      {
       return;
      }

      public void main()
      {
       int i;

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

fail-check-class.javapm

```
public class dummy {

}
```

fail-check-class2.javapm

```
public class Hello {
}
public class Hello {
}
```

fail-check-constructor1.javapm

```
public class Dummy {
	Dummy(int a) {
	}
	Dummy(int x) {
	}
	public void main() {
		print(3);
	}
}
```

fail-check-constructor2.javapm

```
public class Dummy {
	Dummy(int x, float x) {
	}

	public void main() {
		print(3);
	}
}
```

fail-check-localVarDecl1.javapm

```
public class Dummy {
	public boolean x = true;
	public int a = 3+4;
	public void main() {
		3;
		int x = 2;
		float a;
		boolean a;
	}
}
```

fail-check-localVarDecl2.javapm

```
public class Dummy {
	public void main() {
		int x = true;
	}
}
```

fail-check-method1.javapm

```
public class Dummy {
      public void hello() {
      }

      public void hello() {
      }
}
```

fail-check-method2.javapm

```
public class Dummy {
      public void hello(int a, float a) {
      }
}
```

fail-check-variable1.javapm

```
public class Dummy {

      public int x;
      public float x;
}
```

fail-check-variable2.javapm

```
public class Dummy {
      public int x = true;
}
```

fail-dead1.javapm

```
public class FailDead1{
      public int main()
      {
       int i;

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

fail-dead2.javapm

```
public class FailDead2{
      public int main()
      {
       int i;

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

fail-expr1.javapm

```
public class FailExpr1{
      int a;
      boolean b;

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

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

fail-expr2.javapm

```
public class FailExpr2{
      int a;
      boolean b;

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

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

fail-for1.javapm

```
public class FailFor1{
      public int main()
      {
       int i;
       for (true) {} /* OK: Forever */

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

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

       return 0;
      }
}
```

fail-for2.javapm

```
public class FailFor2{
      public int main()
      {
       int i;
       for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */

       return 0;
      }
}
```

fail-for3.javapm

```
public class FailFor3{
      public int main()
      {
       int i;

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

       return 0;
      }
}
```

fail-for4.javapm

```
public class FailFor4{
      public int main()
      {
       int i;

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

       return 0;
      }
}
```

fail-for5.javapm

```
public class FailFor5{
      public int main()
      {
       int i;

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

       return 0;
      }
}
```

fail-func1.javapm

```
public class FailFunc1{
      public int foo() {}

      public int bar() {}

      public int baz() {}

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

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

fail-func2.javapm

```
public class FailFunc2{
      public int foo(int a, boolean b, int c) { }

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

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

fail-func3.javapm

```
public class FailFunc3{
      public int foo(int a, boolean b, int c) { }

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

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

fail-func4.javapm

```
public class FailFunc4{
      public int foo() {}

      public void bar() {}

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

      public void baz() {}

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

fail-func5.javapm

```
public FailFunc5{
      public int foo() {}

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

       return 0;
      }

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

fail-func6.javapm

```
public class FailFunc6{
      public void foo(int a, boolean b)
      {
      }

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

fail-func7.javapm

```
public class FailFunc7{
      public void foo(int a, boolean b)
      {
      }

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

fail-func8.javapm

```
public class FailFunc8{
      public void foo(int a, boolean b)
      {
      }

      public void bar()
      {
      }

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

fail-func9.javapm

```
public class FailFunc9{
      public void foo(int a, boolean b)
      {
      }

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

fail-global1.javapm

```
public class FailGlobal1{
      int c;
      boolean b;
      void a; /* global variables should not be void */

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

fail-global2.javapm

```
public class FailGlobal2{
      int b;
      boolean c;
      int a;
      int b; /* Duplicate global variable */
      public int main()
      {
       return 0;
      }
}
```

fail-if1.javapm

```
public class FailIf1{
      public int main()
      {
       if (true) {}
       if (false) {} else {}
       if (42) {} /* Error: non-boolean predicate */
      }
}
```

fail-if2.javapm

```
public class FailIf2{
      public int main()
      {
       if (true) {
         foo; /* Error: undeclared variable */
       }
      }
}
```

fail-if3.javapm

```
public class FailIf3{
      public int main()
      {
       if (true) {
         42;
       } else {
         bar; /* Error: undeclared variable */
       }
      }
}
```

fail-nomain.javapm

```
public class noMain{
      int i;
}
```

fail-return1.javapm

```
public class FailReturn1{
      public int main()
      {
       return true; /* Should return int */
      }
}
```

fail-return2.javapm

```
public class FailReturn2{
      public void foo()
      {
       if (true){
            return 42; /* Should return void */
       }
       else return;
      }

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

fail-while1.javapm

```
public class FailWhile1{
      public void main()
      {
       int i = 0;

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

       while (42) { /* boolean expected */
         i = i + 1;
       }

      }
}
```

fail-while2.javapm

```
public class FailWhile2{
      public void main()
      {
       int i;

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

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

      }
}
```

How Test Cases Were Selected

Our first set of test cases were unit test cases to outline the basic features the language should have. We had two kinds of test cases: tests that are supposed to pass and tests that are suppose to fail. By splitting the test cases as such, we are able to ensure that Java+- works as expected by producing the expected outputs and error messages.

Automation

We used the automated test suite from the MicroC compiler in order to quickly test new features and ensure they did not interfere with previously written test cases. Test cases meant to pass start with “test-” and the expected outputs were written into “.out” files. On the other hand, test cases meant to fail start with “fail-” and expected error messages are written in “.err” files. run the test

script, the command is “./testall.sh”. Results of whether the test cases produced the expected outputs will appear.

Division of Tasks

Tin Nilar and Anna produced the initial set of unit test cases. These test cases tested only the basic functionalities. As codegen grew more complex, Zeynep and Ashley modified and added test cases to cover additional features. Similarly, Anna and Amal modified and added cases as they worked on Semantic analysis.

Lessons Learned

Ashley

All things considered, just know that every warning and word of advice you receive about this project is absolutely, 100% sincere. You must, and I mean *must,* start this early and try to meet every deadline beyond its bare minimum. For example, have code generation started and outlined before the week "Hello World" is due. This project has taught me that setting flexible deadlines and goals is not a thing that's going to work. In particular, if you're the manager, you have to learn to be objective, direct, and firm early on, and call it out when you notice that work isn't getting done or is getting done incorrectly. Also, weeks without deadlines are not grounds for a code stop – you must do this project incrementally.

More technically, code generation is pretty hard with regard to logical complexity as well as LLVM syntax and error messages being pretty obtuse. As Edwards always says, when in doubt the answer is almost always either pattern matching or another level of indirection! Try one before giving up completely.

With regard to working in a team, establish subgroups and divide tasks fairly early into the game. I would also suggest that you use a mode of communication other than Facebook Messenger for the simple fact that important messages get lost very easily in Facebook chat thread. (Just use a platform that allows you to pin messages so no one can ignore you.)

Lastly, read these project reports at the beginning of the semester so you get a good sense of what your role entails, what you'll need for your report deadline, and most importantly, learn what we have learned.

Zeynep

Just because the theory is straightforward doesn't necessarily mean that the implementation is going to be easy. This was especially an issue with understanding how the memory is managed and the stack is kept. I had trouble understanding how a pre generated function code was going to access the right variables. Everything made more sense after I implemented loops and conditional branching. The idea of basic blocks, even though too late for the final exam, is now a concept I think I understand well.

Be realistic. Java wasn't implemented overnight.

Anna

Just like the others in this group and in other previous groups - start early! You won't realize how much work needs to be done until too late if you don't keep up with the deadlines. Meeting frequently is key, once a week will probably not be enough. Dividing into smaller teams and dividing the workload will make much more progress.

Tin Nilar

Definitely start the project early and take the scanner, parser deadline seriously. It might be hard being motivated to meet those deadlines because it is not like turning in code on Canvas. But checking in with your TA mentor early for help and working on them early will save your life during the reading and finals week. For bigger teams, it will work better if subgroups are formed within the team and different tasks are divided early on.

Amal

Most important lesson in my opinion is to set up milestones for the project early on. Having a clear understanding of what needs to be done saves you from freaking out at the end when you realize how demanding some parts of the project can be. Also, copying and pasting is practical, but make sure you allow yourself to implement solution of your own. It's tough in the beginning, but once you get a hang of the OCaml syntax and the general logic in the compiler, things get a little bit easier.

Appendix

scanner.mml Zeynep, Ashley, Amel, Anna, Tin

```
{
  open Parser
  let unescape s = Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}

let whitespace = [' ' '\t' '\r' '\n']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let int = digit+
let float = (digit+) '.' (digit+)

let id = (alpha | '_') (alpha | digit | '_')*
let char = ''' (ascii) '''

let string_lit = '"'((ascii|escape)* as lxm)'"'

rule token = parse
       whitespace { token lexbuf }
       | "/*" { comment lexbuf }
       | "//"    { singleComment lexbuf }

       (* Operators and Separators *)
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE }
  | '}'      { RBRACE }
  | '['      { LBRACKET }
  | ']'      { RBRACKET }
  | '.'     { DOT }
  | ';'      { SEMI }
  | ','      { COMMA }
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { TIMES }
  | '/'      { DIVIDE }
  | '='      { ASSIGN }
  | "=="     { EQ }
  | "!="     { NEQ }
  | '<'      { LT }
  | "<="     { LEQ }
  | ">"      { GT }
  | ">="     { GEQ }
  | "and"    { AND }
  | "or"     { OR }
  | "!"      { NOT }

       (* Branching Control *)
  | "if"       { IF }
  | "else"     { ELSE }
  | "elseif"   { ELSEIF }
  | "for"      { FOR }
  | "while"    { WHILE }
  | "return"   { RETURN }
```

```
  | "break"    { BREAK }
  | "continue"        { CONTINUE }

  (* Primitive Data Types and Return Types *)
  | "char"      { JCHAR }
  | "int"       { JINT }
  | "float"     { JFLOAT }
  | "void"      { JVOID }
  | "boolean"   { JBOOLEAN }
  | "true"      { TRUE }
  | "false"     { FALSE }
  | "null"      { NULL }
  | "Tuple"    { TUPLE }

 (* | "tuple" { JTuple($3, $5) }*)

  (* Classes *)
  | "class"   { CLASS }
  | "new"       { NEW       }
  | "public"  { PUBLIC }
  | "private" { PRIVATE }

  | int as lxm    { INT_LITERAL(int_of_string lxm) }
  | float as lxm  { FLOAT_LITERAL(float_of_string lxm) }
  | char as lxm   { CHAR_LITERAL(String.get lxm 1) }
  | string_lit    { STRING_LITERAL(lxm) }
  | id as lxm     { ID(lxm) }
  | eof           { EOF }
  | _ as illegal  { raise (Failure("illegal character " ^ Char.escaped illegal )) }

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

and singleComment = parse
  '\n' { token lexbuf }
| _    { singleComment lexbuf }
```

parser.mly Zeynep, Ashley, Amel, Anna, Tin

```
%{ open Ast %}

%token CLASS PUBLIC PRIVATE
%token JBOOLEAN JCHAR JINT JFLOAT JVOID TRUE FALSE NULL TUPLE
%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT AND OR
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSEIF ELSE FOR WHILE NEW BREAK CONTINUE

%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <char> CHAR_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSEIF
%nonassoc ELSE
%right ASSIGN
```

```
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT
%right RBRACKET
%left LBRACKET
%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 }

scope:
         PRIVATE { Private }
       | PUBLIC  { Public }

cdecl:
              scope CLASS ID LBRACE cbody RBRACE { {
                     cscope = $1;
                     cname = $3;
                     cbody = $5
              } }


cbody: /* make sure defined in ast, rename to variables */
       /* nothing */ { {
              variables = [];
              constructors = [];
              methods = [];
       } }
       |      cbody vdecl { {
                     variables = $2 :: $1.variables;
                     constructors = $1.constructors;
                     methods = $1.methods;
              } }
       |      cbody constructor { {
                     variables = $1.variables;
                     constructors = $2 :: $1.constructors;
                     methods = $1.methods;
              } }
       |      cbody fdecl { {
                     variables = $1.variables;
                     constructors = $1.constructors;
                     methods = $2 :: $1.methods;
              } }

vdecl:
       scope datatype ID SEMI{{
```

```
                vscope = $1;
                vtype = $2;
                vname = $3;
                vexpr = Noexpr;
        }}
        | scope datatype ID ASSIGN expr SEMI {{
                vscope = $1;
                vtype = $2;
                vname = $3;
                vexpr = $5
        }}



/* constructors */
constructor:
        ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE {
                {
                        fscope = Public;
                        fname = $1;
                        freturn = Object($1);
                        fformals = $3;
                        fbody = List.rev $6;

                }
        }


/* methods */
fdecl:
        scope datatype ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
                { { fscope = $1;
                        freturn = $2;
                        fname = $3;
                        fformals = $5;
                        fbody = List.rev $8 } }

tdatatype_args:
                 datatype                          {[$1]}
         |       tdatatype_args COMMA datatype     {$3 :: $1}

tuple_type :
         TUPLE LT tdatatype_args GT { Tuple($3) }

/* datatypes + formal & actual params */
primitive:
          JCHAR                                    { JChar }
        | JINT                                     { JInt }
        | JFLOAT                                   { JFloat }
        | JBOOLEAN                                 { JBoolean }
        | JVOID                                    { JVoid }

po:
                primitive   { $1 }
        | ID                  {Object($1)}

array_type:
        primitive LBRACKET brackets RBRACKET { Arraytype($1, $3) }

datatype:
                po {$1}
```

```
        | array_type { $1 }
        | tuple_type { $1 }

brackets:
          /* nothing */       { 1 }
        | brackets RBRACKET LBRACKET { $1 + 1 }

formal:
        datatype ID
         {{
               fvtype = $1;
               fvname = $2;
               }}

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

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

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

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

/* statements */

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

stmt:
          expr SEMI { Expr ($1) }
        | vdecl { VarDecl ($1) }
        | datatype ID SEMI {LocalVarDecl($1, $2, Noexpr)}
        | datatype ID ASSIGN expr SEMI {LocalVarDecl($1, $2, $4)}
        | 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) }

/* expressions */

expr:
          literals    { $1 }
        | expr PLUS expr { Binop($1, Add, $3) }
        | expr MINUS expr { Binop($1, Sub, $3) }
        | expr TIMES expr { Binop($1, Mult, $3) }
        | expr DIVIDE expr { Binop($1, Div, $3) }
        | expr EQ expr { Binop($1, Equal, $3) }
        | expr NEQ expr { Binop($1, Neq, $3) }
        | expr LT expr { Binop($1, Less, $3) }
        | expr LEQ expr { Binop($1, Leq, $3) }
        | expr GT expr { Binop($1, Greater, $3) }
```

```
        | expr GEQ expr { Binop($1, Geq, $3) }
        | expr AND expr { Binop($1, And, $3) }
        | expr OR expr { Binop($1, Or, $3) }
        | NOT expr { Unop(Not, $2) }
        | MINUS expr { Unop(Sub, $2) }
        | expr ASSIGN expr { Assign($1, $3) }
        | LPAREN expr RPAREN { $2 }
        | ID LPAREN actuals_opt RPAREN { FuncCall($1, $3) }
        | NEW ID LPAREN actuals_opt RPAREN { CreateObject($2, $4)}
        | expr DOT expr { ObjAccess($1, $3)}
        | NEW TUPLE LT tdatatype_args GT LPAREN actuals_opt RPAREN  { TupleCreate($4, $7) }
        | expr LT LT expr GT GT {TupleAccess($1, $4)}
        | NEW po brackets_args RBRACKET { ArrayCreate($2, List.rev $3) }
        | expr brackets_args RBRACKET { ArrayAccess($1, List.rev $2) }

brackets_args:
          LBRACKET expr { [$2] }
        | brackets_args RBRACKET LBRACKET expr { $4 :: $1 }

expr_opt:
        /* nothing */ { Noexpr }
        | expr { $1 }

literals:
          INT_LITERAL         { Int_Lit($1) }
        | FLOAT_LITERAL       { Float_Lit($1) }
        | TRUE                        { Bool_Lit(true) }
        | FALSE                       { Bool_Lit(false) }
        | STRING_LITERAL      { String_Lit($1) }
        | CHAR_LITERAL        { Char_Lit($1) }
        | ID                          { Id($1) }
        | NULL                        { Null }
```

ast.ml Zeynep, Ashley, Amel, Anna, Tin

```
(* Possible data types *)
type data_type =
  | JChar
  | JVoid
  | JBoolean
  | JFloat
  | JInt
  | Object of string
  | Arraytype of data_type * int
  | Tuple of data_type list

(* Operators *)
type op = Add | Sub | Div | Mult | Equal | Neq | Less | Leq | Greater | Geq |
Or | And | Not
(* removed assign from op list, may need to add back *)

type scope = Private | Public

type formal = {
  fvtype: data_type;
  fvname: string;
}

(* Expressions *)
type expr =
    Id of string
  | Int_Lit of int
  | Float_Lit of float
  | Char_Lit of char
  | Bool_Lit of bool
  | String_Lit of string
  | Noexpr
  | Binop of expr * op * expr
  | Unop of op * expr
  | Assign of expr * expr
  | FuncCall of string * expr list
  | Null
  | CreateObject of string * expr list
  | ObjAccess of expr * expr
  | TupleCreate of data_type list * expr list
  | TupleAccess of expr * expr
  | ArrayCreate of data_type * expr list
  | ArrayAccess of expr * expr list

  (* Variable Declarations *)
type vdecl = {
  vscope: scope;
  vtype: data_type;
  vname: string;
  vexpr: expr;
}

(* Statements *)
type stmt =
  Block of stmt list
| Expr of expr
| VarDecl of vdecl
| LocalVarDecl of data_type * string * expr
| Return of expr
| If of expr * stmt * stmt
```

```
| For of expr * expr * expr * stmt
| While of expr * stmt

(* Functions *)
type func_decl = {
  fscope : scope;
  fname : string; (* name of the function *)
  fformals : formal list; (* formal params *)
  freturn : data_type; (* return type *)
  fbody : stmt list; (* statements, including local variable declarations *)
}

type cbody = {
  variables : vdecl list;
  constructors : func_decl list;
  methods : func_decl list;
}

type class_decl =  {
    cscope : scope;
    cname : string;
    cbody : cbody;
}

type program = Program of class_decl list
```

sast.ml Zeynep, Ashley, Amel, Anna

```
open Ast

type sformal = {
       sformal_type: data_type;
       sformal_name: string;
}

type sexpr =
         SInt_Lit of int
       | SBoolean_Lit of bool
       | SFloat_Lit of float
       | SString_Lit of string
       | SChar_Lit of char
       | SNull
       | SId of string * data_type
       | SBinop of sexpr * op * sexpr * data_type
       | SAssign of sexpr * sexpr * data_type
       | SNoexpr
       | SFuncCall of string * sexpr list * data_type * int
       | SUnop of op * sexpr * data_type
       | SCreateObject of string * sexpr list * data_type
       | SArrayCreate of data_type * sexpr list * data_type
       | SArrayAccess of sexpr * sexpr list * data_type
       | SArrayElements of sexpr list * data_type
       | SObjAccess of sexpr * sexpr * data_type
       | STupleCreate of data_type list * sexpr list * data_type
       | STupleAccess of sexpr * sexpr * data_type
```

```
type svdecl = {
       svscope: scope;
       svtype: data_type;
       svname: string;
       svexpr: sexpr;
}

type sstmt =
         SBlock of sstmt list
       | SExpr of sexpr * data_type
       | SVarDecl of svdecl
       | SLocalVarDecl of data_type * string * sexpr
       | SReturn of sexpr * data_type
       | SIf of sexpr * sstmt * sstmt
       | SFor of sexpr * sexpr * sexpr * sstmt
       | SWhile of sexpr * sstmt

type func_type = User | Reserved

type sfunc_decl = {
       sfscope: scope;
       sfname: string;
       sfformals: sformal list;
       sfreturn: data_type;
       sfbody: sstmt list;
(** @TODO causing issues in semant   functype: func_type; **)
}

type scbody = {
       svariables: svdecl list;
       sconstructors: sfunc_decl list; (**@TODO**)
       smethods: sfunc_decl list;
}

type sclass_decl = {
       scscope: scope;
       scname: string;
       scbody: scbody;
}

type sprogram = {
       classes: sclass_decl list;
       functions: sfunc_decl list;
       main: sfunc_decl;
       reserved: sfunc_decl list;
}
```

```
open Ast
open Sast
open Utils

module StringMap = Map.Make(String)

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

let createClassIndices cdecls=
       let classHandler index cdecl=
       Hashtbl.add classIndices cdecl.cname index in (*scope handling is missing*)
       List.iteri classHandler cdecls

let builtinMethods = ["print"; "println"]

let isMain f = f.sfname = "main"

let get_methods l classy =  List.concat [classy.scbody.smethods;l]

let get_main m = try (List.hd (List.filter isMain (List.fold_left get_methods [] m))) with
Failure("hd") -> raise(Failure("No main method was defined"))

let get_methods_minus_main m = snd (List.partition isMain (List.fold_left get_methods [] m))

let newEnv env = {
       envClassName = env.envClassName;
       envClassMaps = env.envClassMaps;
       envClassMap  = env.envClassMap;
       envLocals    = env.envLocals;
       envParams    = env.envParams;
       envReturnType = env.envReturnType;
       envBuiltinMethods = env.envBuiltinMethods;
}

let typOFSexpr = function
              SInt_Lit(i)                  -> JInt
       |      SBoolean_Lit(b)                      -> JBoolean
       |      SFloat_Lit(f)                -> JFloat
       |      SString_Lit(s)                      -> Arraytype(JChar, 1)
       |      SChar_Lit(c)                 -> JChar
       |      SId(_, d)                    -> d
       |      SBinop(_, _, _, d)           -> d
       |   SAssign(_, _, d)         -> d
       |      SArrayCreate(_, _, d) -> d
       |      SArrayAccess(_, _, d)        -> d
       |      SFuncCall(_, _, d, _) -> d
       |      SUnop(_, _, d)                      -> d
       |      SCreateObject(_,_,d)  -> d
       |      SObjAccess(_,_,d) -> d
       |      STupleAccess(_,_, d)  -> d

let convertToSast classes =

              let convertFormalToSast formal env =
              {
                     sformal_type = formal.fvtype;
                     sformal_name = formal.fvname;
              }
              in
              let checkBinop e1 op e2 =
```

```
                        "placeholder"
                in
                let checkUnop op e env =
                        "placeholder"
                in
                let checkFuncCall s el env =
                        "placeholder"
                in
                let checkAssign e1 e2 env =
                        "placeholder"
(*                      let typ1  = (getType e1 env)
                        and typ2  = (getType e2 env)
                        in
                        if typ1 <> typ2
                                then raise(Failure("Expected " ^ (str_of_type typ1) ^ "
expression in " ^ (str_of_expr e1)))
*)              in
                let rec convertExprToSast expr env = match expr with
                          Int_Lit(i)   -> SInt_Lit(i)
                        | Bool_Lit(b)  -> SBoolean_Lit(b)
                        | Float_Lit(f) -> SFloat_Lit(f)
                        | String_Lit(s)       -> SString_Lit(s)
                        | Char_Lit(c)  -> SChar_Lit(c)
                        | Null         -> SNull
                        | Noexpr       -> SNoexpr
                        | Id(id)       -> SId(id, JInt (*getIdType id env*)) (** @TODO Sast has
SId(string, datatype) **)
                        | Binop(expr1, op, expr2)    -> (checkBinop expr1 op
expr2;SBinop(convertExprToSast expr1 env, op, convertExprToSast expr2 env, JInt (*getType
expr1 env)*)))
                        | ArrayCreate(d, el)  -> SArrayCreate(d, (List.map (fun e ->
convertExprToSast e env) el),JInt (*Arraytype(d, List.length el)*))
                        | ArrayAccess(e, el)  -> SArrayAccess(convertExprToSast e env, (List.map
(fun e -> convertExprToSast e env) el), JInt) (* @TODO *)
                        | Assign(e1, e2)              -> (checkAssign e1 e2 env;
SAssign(convertExprToSast e1 env, convertExprToSast e2 env,JInt(* getType e1 env*)))
                        | FuncCall(s, el)             -> (checkFuncCall s el env; SFuncCall(s,
(List.map (fun e -> convertExprToSast e env) el), JInt(*getType (FuncCall(s, el)) env*), 0))
                        | Unop(op, expr)              -> (checkUnop op expr env; SUnop(op,
convertExprToSast expr env, JInt (*getType expr env*)))
                        | CreateObject(s,el)  -> SCreateObject(s, (List.map (fun e ->
convertExprToSast e env) el), JInt (*Object(s)*))
                        | ObjAccess(e1,e2) -> SObjAccess(convertExprToSast e1 env,
convertExprToSast e2 env, JInt (*getType e1 env*)) (* @TODO Double check type *)
                        | TupleCreate(dl, el) -> STupleCreate(dl, (List.map (fun e ->
convertExprToSast e env) el), Tuple(dl))
                        | TupleAccess(e1, e2) -> STupleAccess(convertExprToSast e1 env,
convertExprToSast e2 env, JInt (*getType e1 env*)) (* @TODO Double Check type*)
                in
                let convertVdeclToSast vdecl env = {
                        svscope = vdecl.vscope;
                        svtype = vdecl.vtype;
                        svname = vdecl.vname;
                        svexpr = convertExprToSast vdecl.vexpr env;
                } in
                let checkLocalVarDecl dt id e env =
(*                      "placeholder"
                        if StringMap.mem id env.envParams || StringMap.mem id env.envLocals ||
StringMap.mem id env.envClassMap.variableMap
                                then raise(Failure("Variable name already used"))
                        else if ((e <> Ast.Noexpr) && ((getType e env) <> dt))
```

```
                            then raise(Failure(str_of_expr e ^ " is of type " ^ str_of_type
(getType e env) ^ " but expression of type " ^ str_of_type dt ^ " was expected"));
*)                      env.envLocals <- StringMap.add id dt env.envLocals
               in
               let checkReturn e env =
                       "placeholder"
(*                     if getType e env <> env.envReturnType
                            then raise(Failure("Return type doesn't match expected return
type"))
*)             in
               let checkIf e s1 s2 env =
                       "placeholder"
(*                     if (getType e env) <> JBoolean
                            then raise(Failure("Expected boolean expression in " ^
(str_of_expr e)))
*)             in
               let checkFor e1 e2 e3 s env =
                       "placeholder"
(*                     if (getType e2 env) <> JBoolean
                            then raise(Failure("Expected boolean expression in " ^
(str_of_expr e2)))
*)             in
               let checkWhile e s env =
                       "placeholder"
(*                     if (getType e env) <> JBoolean
                               then raise(Failure("Expected boolean expression in " ^
(str_of_expr e)))
*)             in
               let rec convertStmtToSast stmt env = match stmt with
                         Block(sl)                   -> SBlock(List.map (fun s ->
convertStmtToSast s env) sl)
                       | Expr(expr)                  -> SExpr(convertExprToSast expr env,
getType expr env)
(*                     | VarDecl(vdecl)              -> SVarDecl(convertVdeclToSast vdecl env)
*)                     | LocalVarDecl(dt, id, expr) -> (checkLocalVarDecl dt id expr env;
SLocalVarDecl(dt, id, convertExprToSast expr env))
                       | Return(expr)                -> checkReturn expr env;
SReturn(convertExprToSast expr env, getType expr env)
                       | If(expr, stmt1, stmt2)      -> checkIf expr stmt1 stmt2 env;
SIf(convertExprToSast expr env, convertStmtToSast stmt1 (newEnv env), convertStmtToSast stmt2
(newEnv env))
                       | For(expr1, expr2, expr3, stmt)-> checkFor expr1 expr2 expr3 stmt env;
SFor(convertExprToSast expr1 env, convertExprToSast expr2 env, convertExprToSast expr3 env,
convertStmtToSast stmt (newEnv env))
                       | While(expr, stmt)           -> checkWhile expr stmt env;
SWhile(convertExprToSast expr env, convertStmtToSast stmt (newEnv env))
       in

       (* Semantic Checking for class methods *)
       let rec strOfFormals fl = match fl with
              |  [] -> ""
              | h::t -> str_of_type h ^ (strOfFormals t)
       in
let getListOfFormalTypes c = List.map (fun f -> f.fvtype) c.fformals
       in
       let hasDuplicateFormalNames l =
              let result = ref false
              in let names = List.map (fun f -> f.fvname) l
              in List.iter (fun e -> result := !result || e) (List.map (fun n -> List.length
(List.filter (fun s -> s = n) names) > 1) names);!result
       in
```

```
	let checkMethod func_decl classEnv =
(*
		let formalTypes = getListOfFormalTypes func_decl
		in
		let checkSignature _ v =
			v.mformalTypes=formalTypes
		in
		let compareName k _ =
			k = func_decl.fname
		in
		let mp = StringMap.filter compareName classEnv.classMap.methodMap
		in

		let _ = StringMap.iter (fun k v -> if checkSignature k v then
raise(Failure("Duplicate Method Declaration"))) mp
		in StringMap.iter (fun kc c -> ((StringMap.iter ((fun k v -> if
k=func_decl.fname && checkSignature k v then raise(Failure("Duplicate Method Declaration")))
c.methodMap));
			if StringMap.mem (strOfFormals formalTypes) c.constructorMap then
raise(Failure("Duplicate Method Declaration")))) classEnv.classMaps;
		if hasDuplicateFormalNames func_decl.fformals
			then raise(Failure("Formal names must be unique"));
		let signature = {
			mscope = func_decl.fscope;
			mname = func_decl.fname;
			mformalTypes = List.map (fun fl -> fl.fvtype) func_decl.fformals;
			mReturn = func_decl.freturn;
		} in signature
	in
*)	{
		mscope = Public;
		mname = "";
		mformalTypes = [];
		mReturn = JInt
	}

	in
	let setEnvParams formals env =
			List.map (fun f -> env.envParams <- StringMap.add f.fvname f.fvtype
env.envParams) formals

	in
	let convertMethodToSast func_decl classEnv =

		let methodSignature = checkMethod func_decl classEnv
		in
		let _ = classEnv.classMap.methodMap <- StringMap.add func_decl.fname
methodSignature classEnv.classMap.methodMap
		in
		let env = {
			envClassName = classEnv.className;
			envClassMaps = classEnv.classMaps;
			envClassMap  = classEnv.classMap;
			envLocals    = StringMap.empty;
			envParams    = StringMap.empty;
			envReturnType= func_decl.freturn;
			envBuiltinMethods = classEnv.builtinMethods;
		} in
		let _ = setEnvParams func_decl.fformals env
		in
		{
```

```
                        sfscope = func_decl.fscope;
                        sfname = func_decl.fname;
                        sfformals = List.map (fun f -> convertFormalToSast f env)
func_decl.fformals;
                        sfreturn = func_decl.freturn;
                        sfbody = List.map (fun s -> convertStmtToSast s env) func_decl.fbody;
                }

        in
        (* Semantic checking for class constructor *)
        let checkConstructor constructor classEnv =
(*              let formalTypes = getListOfFormalTypes constructor
                in
                let checking =

                        if(StringMap.mem (strOfFormals formalTypes)
classEnv.classMap.constructorMap)
                                then raise (Failure("Duplicate Constructor Definition"))
                        else if ((List.length formalTypes <> 0) && (hasDuplicateFormalNames
constructor.fformals = true))
                                then raise(Failure("Formal names must be unique"))
                in checking;
                 {
                        mscope = constructor.fscope;
                          mname  = constructor.fname;
                          mformalTypes = formalTypes;
                        mReturn = JVoid;
                  }
*)
                {
                  mscope = Public;
                  mname = "";
                  mformalTypes = [];
                  mReturn = JInt
                }
        in
        let convertConstructorToSast constructor classEnv =
                let constructorSignature = checkConstructor constructor classEnv
                in
                let _ = classEnv.classMap.constructorMap <- StringMap.add (strOfFormals
constructorSignature.mformalTypes) constructorSignature classEnv.classMap.constructorMap
                in
                let env = {
                        envClassName = classEnv.className;
                        envClassMaps = classEnv.classMaps;
                        envClassMap  = classEnv.classMap;
                        envLocals    = StringMap.empty;
                        envParams    = StringMap.empty;
                        envReturnType= constructor.freturn;
                        envBuiltinMethods = classEnv.builtinMethods;
                } in
                let _ = setEnvParams constructor.fformals env
                in
                {
                        sfscope = constructor.fscope;
                        sfname = constructor.fname;
                        sfformals = List.map (fun f -> convertFormalToSast f env)
constructor.fformals;
                        sfreturn = constructor.freturn;
                        sfbody = List.map (fun s -> convertStmtToSast s env) constructor.fbody;
                }
```

```
	in
	(* Sematic checking for class variable *)
	let checkVdecl vdecl env =
		"placeholder"
(*		let check =
			if StringMap.mem vdecl.vname env.envClassMap.variableMap
				then raise (Failure("Variable name already used"))
			else if vdecl.vexpr <> Ast.Noexpr && getType vdecl.vexpr env <>
vdecl.vtype
				then raise (Failure(str_of_expr vdecl.vexpr ^ " is of type " ^
str_of_type (getType vdecl.vexpr env) ^ " but type " ^ str_of_type vdecl.vtype ^ " is
expected"))
		in check
*)	in

	let convertVariableToSast vdecl classEnv =
		let env = {
			  envClassName = classEnv.className;
			  envClassMaps = classEnv.classMaps;
			  envClassMap  = classEnv.classMap;
			  envLocals    = StringMap.empty;
			  envParams    = StringMap.empty;
			  envReturnType= JVoid; (*@TODO make sure return is not possible here*)
			envBuiltinMethods = classEnv.builtinMethods;
		 } in
		let _ = checkVdecl vdecl env
		in
		let _ =  classEnv.classMap.variableMap <- StringMap.add vdecl.vname vdecl
classEnv.classMap.variableMap
	in {
			svscope = vdecl.vscope;
			svtype  = vdecl.vtype;
			svname  = vdecl.vname;
			svexpr  = convertExprToSast vdecl.vexpr env;
		 }
	in
	let convertCbodyToSast cbody classEnv =
	{
		svariables = List.map (fun v -> convertVariableToSast v classEnv) (List.rev
cbody.variables);
		 sconstructors = List.map (fun cst -> convertConstructorToSast cst classEnv) (List.rev
cbody.constructors);
		smethods = List.map (fun m -> convertMethodToSast m classEnv) (List.rev
cbody.methods);
    }

	in
	let checkClass class_decl classEnv =
		let firstChar = String.get class_decl.cname 0
		in
		let lowerChar = Char.lowercase firstChar
		in
		let checking =
			if lowerChar = firstChar
				then raise (Failure ("Class name not capitalized: " ^
class_decl.cname))
			else if StringMap.mem class_decl.cname classEnv.classMaps
				then raise (Failure ("Duplicate Class Name: " ^
class_decl.cname))
			in checking
```

```
        in
        let convertClassToSast class_decl classEnv =
                classEnv.className <- class_decl.cname;
                checkClass class_decl classEnv;
                let classMap = {
                        variableMap = StringMap.empty;
                        constructorMap = StringMap.empty;
                        methodMap = StringMap.empty;
                } in
                classEnv.classMap <- classMap;
                let result =
                {
                        scscope = class_decl.cscope;
                        scname  = class_decl.cname;
                        scbody  = convertCbodyToSast class_decl.cbody classEnv;
                } in
                classEnv.classMaps <- StringMap.add classEnv.className classEnv.classMap
classEnv.classMaps;
                result

        in
        let classEnv = {
                className = "";
                classMaps = StringMap.empty;
                classMap  = {
                variableMap = StringMap.empty;
                constructorMap = StringMap.empty;
                methodMap = StringMap.empty;
                };
                builtinMethods = builtinMethods;
        }
        in
        let get_classes =
                List.map (fun c -> convertClassToSast c classEnv) classes
        in
        let sprogram =
        {
                classes = get_classes;
                functions = get_methods_minus_main get_classes;
                main = get_main get_classes;
                reserved = [];
        }
        in
        sprogram

 (* Translates Ast to Sast *)
let check program = match program with
          Program (classes) -> ignore (createClassIndices classes); convertToSast classes
```

javapm.ml Zeynep

```
open Ast
open Sast

let _ =
        let filename = Sys.argv.(1) ^ ".javapm" in

        let in_channel = open_in Sys.argv.(1) in

        let lexbuf = Lexing.from_channel in_channel in
```

```
       let ast  = Parser.program Scanner.token lexbuf in

       let sast = Semant.check ast in
       let outprog = Codegen.translate sast in
       Llvm_analysis.assert_valid_module outprog;
       print_string (Llvm.string_of_llmodule outprog);;
```

Makefile Zeynep

```
OBJS = ast.cmx sast.cmx parser.cmx scanner.cmx utils.cmx semant.cmx codegen.cmx javapm.cmx

YACC = ocamlyacc

javapm: $(OBJS)
       ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis -o javapm $(OBJS)

scanner.ml: scanner.mll
       ocamllex scanner.mll

parser.ml parser.mli: parser.mly
       $(YACC) -v parser.mly

%.cmo: %.ml
       ocamlc -c $<

%.cmi: %.mli
       ocamlc -c $<

%.cmx: %.ml
       ocamlfind ocamlopt -c -package llvm $<

.PHONY: clean
clean:
       rm -f javapm parser.ml parser.mli scanner.ml \
           *.cmo *.cmi *.cmx *.o *.out *.diff *.output javapm *.dSYM *.err *.ll

.PHONY: all
all: clean javapm

ast.cmo :
ast.cmx :
sast.cmo :
sast.cmx :
utils.cmo : ast.cmo
utils.cmx : ast.cmx
codegen.cmo : ast.cmo sast.cmo utils.cmo
codegen.cmx : ast.cmx sast.cmx utils.cmx
parser.cmo : ast.cmo sast.cmo parser.cmi
parser.cmx : ast.cmx sast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : sast.cmi ast.cmo
semant.cmx : sast.cmi ast.cmx
javapm.cmo : scanner.cmo parser.cmo codegen.cmo ast.cmo sast.cmo utils.cmo
javapm.cmx : scanner.cmx parser.cmx codegen.cmx ast.cmx sast.cmx utils.cmo
parser.cmi : ast.cmo
javapm.cmo : scanner.cmo parser.cmo codegen.cmo ast.cmo sast.cmo semant.cmo utils.cmo
javapm.cmx : scanner.cmx parser.cmx codegen.cmx ast.cmx sast.cmx semant.cmx utils.cmo
parser.cmi : ast.cmo
```

```
semant.cmo: ast.cmo sast.cmo
semant.cmx: ast.cmx sast.cmx
semant.cmx: ast.cmx sast.cmx
```

utils.ml

```
(* Pretty Printer *)

open Ast
open Sast

module StringMap = Map.Make(String)

type methodSignature = {
        mscope: scope;
        mname : string;
        mformalTypes: data_type list;
       mReturn: data_type;
}

type classMap = {
        mutable variableMap: Ast.vdecl StringMap.t;
        mutable constructorMap: methodSignature StringMap.t;
        mutable methodMap:  methodSignature StringMap.t;
}

type classEnv = {
       mutable className: string;
       mutable classMaps: classMap StringMap.t;
       mutable classMap: classMap;
       builtinMethods: string list;
}

type env = {
        mutable envClassName: string;
        mutable envClassMaps: classMap StringMap.t;
        mutable envClassMap: classMap;
        mutable envLocals: data_type StringMap.t;
        mutable envParams: data_type StringMap.t;
        mutable envReturnType: data_type;
       envBuiltinMethods: string list;
}

let getListOfFormalTypes c =
       if List.length c.fformals == 0 then []
       else List.map (fun f -> f.fvtype) c.fformals

let getIdType s env =
       let checking =
       if StringMap.mem s env.envLocals
              then begin
                     try (StringMap.find s env.envLocals)
                     with Not_found -> raise(Failure("Unfound local"))
              end
       else if StringMap.mem s env.envParams
              then try (StringMap.find s env.envParams)
              with Not_found -> raise(Failure("Unfound param"))
       else if StringMap.mem s env.envClassMap.variableMap
                then begin
```

```
                        try (StringMap.find s env.envClassMap.variableMap).vtype
                        with Not_found -> raise(Failure("Unfound class variable"))
                end
        else if StringMap.mem s env.envClassMaps
                then Object(s)
        else raise(Failure("Undeclared identifier " ^ s))
        in checking
(*
let getObjElmType s e env =
        let objType = getIdType Id(s)
        in try (  List.find e(List.find objType env.envClassMaps)
*)
let getFuncType s fl env =
        if List.mem s env.envBuiltinMethods
                then JVoid
        else begin
                let updated = ref false
                and
                t = ref JVoid
                in
                StringMap.iter (fun _ v -> if v.mformalTypes=fl
                        then (t:=v.mReturn; updated:=true))
                        (StringMap.filter (fun k _-> k=s) env.envClassMap.methodMap);
                if !updated = true
                        then !t
                else raise(Failure("Undeclared method " ^ s))
        end

let rec getType expr env = match expr with
          Id(s)               -> JInt(*getIdType s env*)
        | Int_Lit(s)          -> JInt
        | Float_Lit(f)        -> JFloat
        | Char_Lit(c)         -> JChar
        | String_Lit(s)               -> Arraytype(JChar, 1)
        | Bool_Lit(b)         -> JBoolean
        | Noexpr              -> JVoid
        | Null                -> JVoid
        | Binop(e1, op, e2)   -> (match op with
                                        Equal|Neq|Less|Leq|Greater|Geq|Or|And|Not -> JBoolean
                                      | Add|Sub|Mult|Div -> if ((getType e1 env)=JFloat) ||
((getType e2 env)=JFloat)
                                                                then JFloat
                                                                else JInt)
        | Unop(op, e)         -> getType e env
        | Assign(s, e)        -> getType e env
        | TupleCreate(dl, el)         -> Tuple(dl)
        | TupleAccess(e1, e2)         -> JInt (*(match e1 with
                                          Id(s) -> getTupleElmType s e2 env)*)
        | ObjAccess(e1, e2)   -> JInt(*(match e1 with Id(s) -> getObjElmType s e2 env*)
        | CreateObject(s, el)         -> Object(s)
        | ArrayCreate(d, el)  -> Arraytype(d, List.length el)
        | ArrayAccess(e, el)  -> JInt(*(match e with Id(s) -> getType Id(s))*)
        | FuncCall(s, el)     -> JInt(*getFuncType s (List.map (fun e -> getType e env) el) env
*)

let addComma l =
        if l = [] then ""
        else begin
                let s = ref ""
                in
                let _ = List.iter (fun d -> s := !s ^ d ^ ",") l
```

```
               in
               String.sub !s 0 ((String.length !s) -2)
       end

let to_string l =
       let s = ref ""
       in let _ = List.iter (fun d -> s:= !s^d)
       in !s

(* Print data types *)

let rec str_of_type = function
               JChar     -> "char"
       |       JVoid     -> "void"
       |       JBoolean  -> "boolean"
       |       JFloat    -> "float"
       |       JInt     -> "int"
       |       Object(s) -> "class " ^ s
       |       Arraytype(d, l) -> str_of_type d ^ "[]"
       |       Tuple(dl)      -> "tuple<" ^ addComma (List.map str_of_type dl) ^ ">"

let str_of_scope = function
               Public  -> "public"
       |       Private -> "private"

let str_of_op = function
               Add     -> "+"
       |       Sub     -> "-"
        |      Mult     -> "*"
        |              Div      -> "/"
        |              Equal    -> "=="
        |              Neq      -> "!="
        |              Less     -> "<"
        |              Leq      -> "<="
        |              Greater -> ">"
        |              Geq     -> ">="
        |              And      -> "and"
        |              Not      -> "!"
        |              Or       -> "or"

(* Pretty Printing for Ast *)

let rec str_of_expr expr = match expr with
         Int_Lit(i)          -> string_of_int i
       | Bool_Lit(b)         -> if b then "true" else "false"
       | Float_Lit(f)        -> string_of_float f
       | String_Lit(s)              -> s
       | Char_Lit(c)         -> Char.escaped c
       | Null                -> "null"
       | Id(s)               -> s
       | Binop(e1, op, e2)   -> "" ^ str_of_expr e1 ^ " " ^ str_of_op op ^ " " ^ str_of_expr
e2
       | Assign(e1, e2)             -> "" ^ str_of_expr e1 ^ " = " ^ str_of_expr e2
       | Noexpr              -> ""
       | FuncCall(s, el)     -> "" ^ s ^ "(" ^ addComma (List.map str_of_expr el) ^ ")"
       | Unop(op, e)         -> "" ^ str_of_op op ^ " " ^ str_of_expr e
       | CreateObject(s, el)        -> "new " ^ s ^ "(" ^ addComma (List.map str_of_expr el) ^
")"
       | ObjAccess(e1, e2)   -> "(" ^ str_of_expr e1 ^ ").(" ^ str_of_expr e2 ^")"
       | TupleCreate(dl, el) -> "new Tuple<" ^ addComma (List.map str_of_type dl) ^ "> (" ^
addComma (List.map str_of_expr el)
```

```
       | TupleAccess(e1, e2) -> "(" ^ str_of_expr e1 ^ ")<<" ^ str_of_expr e2 ^ ">>"
       | ArrayCreate(dt, el) -> "new " ^ str_of_type dt ^ to_string (List.map (fun e -> "[" ^
str_of_expr e ^ "]") el)
       | ArrayAccess(e, el)  -> "(" ^ str_of_expr e ^ ")" ^ to_string (List.map (fun e -> "["
^ str_of_expr e ^ "]") el)
(* Pretty Printing for Sast *)

let appendList h t = match t with
         []   -> ""
       | _    -> "" ^ (List.fold_left (fun hd tl -> hd^tl) h t)

let get_str l f =
       if List.length l = 0 then ""
       else if List.length l = 1 then f (List.hd l)
       else begin
              print_endline("run");
              let strList = List.map f l in
              let h = List.hd strList
              and t = List.tl strList
              in
              appendList h t
       end


let str_of_svdecl svdecl =
       "" ^ str_of_scope svdecl.svscope ^ " " ^ str_of_type svdecl.svtype ^ " " ^
svdecl.svname ^ ";\n"

let str_of_svariables variables =
       get_str variables str_of_svdecl

let rec str_of_sexpr expr = match expr with
         SInt_Lit(i)         -> string_of_int i
       | SBoolean_Lit(b)     -> if b then "true" else "false"
       | SFloat_Lit(f)              -> string_of_float f
       | SString_Lit(s)      -> s
       | SChar_Lit(c)        -> Char.escaped c
       | SNull               -> "null"
       | SId(s,_)            -> s
       | SBinop(e1, op, e2, _)      -> "" ^ str_of_sexpr e1 ^ " " ^ str_of_op op ^ " " ^
str_of_sexpr e2
       | SAssign(e1, e2, _)  -> "" ^ str_of_sexpr e1 ^ " = " ^ str_of_sexpr e2
       | SNoexpr             -> ""
       | SFuncCall(s, el, _, _)->  "" ^ s ^ "(" ^ addComma (List.map str_of_sexpr el) ^ ")"
       | SUnop(op, e, _)     -> "" ^ str_of_op op ^ " " ^ str_of_sexpr e

let rec str_of_sstmt stmt = match stmt with
         SBlock(sl)   -> "" ^ (get_str sl str_of_sstmt)
       | SExpr(e,_)   -> "" ^ str_of_sexpr e ^ ";\n"
       | SVarDecl(svdecl)    -> "" ^ str_of_svdecl svdecl
       | SReturn(e, _)       -> "return " ^ str_of_sexpr e ^ ";\n"
       | SIf(e, s1, s2)      -> "if(" ^ str_of_sexpr e ^ ") {\n" ^ str_of_sstmt s1 ^ "} else
{\n" ^ str_of_sstmt s2 ^ "}\n"
       | SFor(e1, e2, e3, s) -> "for(" ^ str_of_sexpr e1 ^ "; " ^ str_of_sexpr e2 ^ "; " ^
str_of_sexpr e3 ^ ") {\n" ^ str_of_sstmt s ^ "}\n"
       | SWhile(e, s)        -> "while(" ^ str_of_sexpr e ^ ") {\n" ^ str_of_sstmt s ^ "}\n"

let str_of_sfbody fbody = get_str fbody str_of_sstmt

let str_of_sformal formal =
       "" ^ str_of_type formal.sformal_type ^ " " ^ formal.sformal_name
```

```
let str_of_sformals formals = get_str formals str_of_sformal


let str_of_sfunction fdecl =
       "" ^ str_of_scope fdecl.sfscope ^ " "^ str_of_type fdecl.sfreturn ^ " " ^ fdecl.sfname
^ " (" ^
               str_of_sformals fdecl.sfformals ^ ") {\n" ^ str_of_sfbody fdecl.sfbody ^ "}"

let str_of_sconstructor constructor =
       "" ^ str_of_sfunction constructor

let str_of_sconstructors constructors = get_str constructors str_of_sconstructor

let str_of_smethods methods = get_str methods str_of_sfunction

let str_of_scbody cbody =
       "" ^ str_of_svariables cbody.svariables ^ str_of_sconstructors cbody.sconstructors ^
str_of_smethods cbody.smethods

let str_of_sclass c =
       "" ^ str_of_scope c.scscope ^ " " ^ c.scname ^ " {\n" ^ str_of_scbody c.scbody ^
"\n}\n"

let str_of_sclasses scdecls = get_str scdecls str_of_sclass

let print_str_of_sast classes = print_string (str_of_sclasses classes)
```

codegen.ml Zeynep, Ashley

```
open Utils
open Llvm
open Hashtbl
open Ast
open Sast
open Semant

module L = Llvm
module A = Ast

let context = global_context ()
let the_module = create_module context "javapm"
let builder = builder context

let i32_t = L.i32_type context;; (* integer *)
let i8_t = L.i8_type context;; (* printf format string *)
let i1_t = L.i1_type context;; (* boolean *)
let i64_t = L.i64_type context ;; (* idk *)
let f_t = L.double_type context;; (* float *)

let boolean_True =  L.const_int i1_t 1;;
let boolean_False = L.const_int i1_t 0;;

let str_t = L.pointer_type i8_t;;
let void_t = L.void_type context;; (* void *)

let global_var_table:(string, llvalue) Hashtbl.t = Hashtbl.create 100
let class_private_vars:(string, llvalue) Hashtbl.t = Hashtbl.create 100 (*Must be cleared
after class build*)
let local_var_table:(string, llvalue) Hashtbl.t = Hashtbl.create 100 (*Must be cleared
```

```
evertime after a function is built*)
let struct_typ_table:(string, lltype) Hashtbl.t = Hashtbl.create 100
let struct_field_idx_table:(string, int) Hashtbl.t = Hashtbl.create 100

let rec get_ptr_type datatype = match datatype with
               A.Arraytype(t, 0) -> get_llvm_type t
       |       A.Arraytype(t, 1) -> L.pointer_type (get_llvm_type t)
       |       A.Arraytype(t, i) -> L.pointer_type (get_ptr_type (A.Arraytype(t, (i-1))))
       |       _ -> raise(Failure("InvalidStructType Array Pointer Type"))

and get_llvm_type datatype = match datatype with (* LLVM type for AST type *)
         A.JChar -> i8_t
       | A.JVoid -> void_t
       | A.JBoolean -> i1_t
       | A.JFloat -> f_t
       | A.JInt -> i32_t
       | A.Object(s) -> L.pointer_type(find_llvm_struct_type s)
       | A.Arraytype(data_type, i) ->  get_ptr_type (A.Arraytype(data_type, (i)))
       | A.Tuple(dt_list) -> L.pointer_type(find_llvm_tuple_type dt_list)
       | _ -> raise(Failure("Invalid Data Type"))

and find_llvm_tuple_type dt_list =
       let type_list = List.map (function dt -> get_llvm_type dt) dt_list in
       let type_array = (Array.of_list type_list) in
       L.packed_struct_type context type_array

and find_llvm_struct_type name =
       try Hashtbl.find struct_typ_table name
       with | Not_found -> raise(Failure ("undeclared struct "^ name))

let find_func_in_module fname =
       match (L.lookup_function fname the_module) with
                 None -> raise(Failure("Function: " ^ fname ^ " not found in module."))
               | Some f -> f

(** code gen top level begins here **)

let translate sast =
       let classes = sast.classes in
       let main = sast.main in
       let functions = sast.functions in

       let util_func () =
              let printf_t = L.var_arg_function_type i32_t [| pointer_type i8_t |] in
              let malloc_t = L.function_type (str_t) [| i32_t |] in
              let lookup_t = L.function_type (pointer_type i64_t) [| i32_t; i32_t |] in

              let _ = L.declare_function "printf" printf_t the_module in
              let _ = L.declare_function "malloc" malloc_t the_module in
              let _ = L.define_function "lookup" lookup_t the_module in
              ()
       in
       let _ = util_func () in

       let zero = const_int i32_t 0 in

       (*Define Classes*)

       let add_classes_to_hashTable c =
              let struct_typ = L.named_struct_type context c.scname in
              Hashtbl.add struct_typ_table c.scname struct_typ
```

```
        in
        let _ = List.map add_classes_to_hashTable classes in

        let define_vardecl c =
                List.iteri (
                        fun i v ->
                        Hashtbl.add global_var_table v.svname boolean_True;
                )
            c.scbody.svariables;
        in
        let _ = List.map define_vardecl classes in

        let define_classes c =
                let struct_t = Hashtbl.find struct_typ_table c.scname in
                let type_list = List.map (function sv -> get_llvm_type sv.svtype)
c.scbody.svariables in
                let name_list = List.map (function sv -> sv.svname) c.scbody.svariables in
                let type_list = i32_t :: type_list in
                let name_list = ".key" :: name_list in
                let type_array = (Array.of_list type_list) in
                List.iteri (
                        fun i f ->
                 let n = c.scname ^ "." ^ f in
                 Hashtbl.add struct_field_idx_table n i;
                )
            name_list;
                L.struct_set_body struct_t type_array true
        in
        let _ = List.map define_classes classes in

        (*Define Functions*)
        let define_functions f =
                let fname = f.sfname in
                let is_var_arg = ref false in
                let types_of_parameters = List.rev ( List.fold_left
                (fun l ->
                        (function
                          sformal-> get_llvm_type sformal.sformal_type::l
                        | _ -> ignore(is_var_arg = ref true); l
                        )
                )
                []  f.sfformals)
                in

                let fty =
                        if !is_var_arg
                                then L.var_arg_function_type (get_llvm_type f.sfreturn)
                                (Array.of_list types_of_parameters)
                        else L.function_type (get_llvm_type f.sfreturn)
                                (Array.of_list types_of_parameters)
                in
                L.define_function fname fty the_module (*The function name should be
Class.fname?*)
        in
        let _ =  List.map define_functions functions in

        let define_constructors c =
                List.map define_functions c.scbody.sconstructors
        in
        let _ = List.map define_constructors classes in
```

```
       (*Stmt and expr handling*)

       let rec stmt_gen llbuilder = function
                 SBlock sl        -> generate_block sl llbuilder
               | SExpr (se, _)    ->   expr_gen llbuilder se
               | SVarDecl sv           ->  generate_vardecl sv.svscope sv.svtype sv.svname
sv.svexpr llbuilder
               | SLocalVarDecl (dt, vname, vexpr)         -> generate_local_vardecl dt vname
vexpr llbuilder
               | SIf(e, s1, s2) -> generate_if e s1 s2 llbuilder
               | SWhile(e, s) -> generate_while e s llbuilder
               | SFor(e1, e2, e3, s) -> generate_for e1 e2 e3 s llbuilder
               | SReturn(e, d)              -> generate_return e d llbuilder

       and generate_block sl llbuilder =
              try List.hd (List.map (stmt_gen llbuilder) sl) with
              | Failure("hd") -> raise(Failure("No body"));

       and generate_return e d llbuilder =
              match e with
              | SNoexpr -> L.build_ret_void llbuilder
              | _ -> L.build_ret (expr_gen llbuilder e) llbuilder

       and generate_vardecl scope datatype vname expr llbuilder =
              let allocatedMemory = L.build_alloca (get_llvm_type datatype) vname llbuilder in
              Hashtbl.add
                     (match scope with
                     | A.Public -> global_var_table
                     | A.Private -> class_private_vars) vname allocatedMemory;

              let variable_value = expr_gen llbuilder expr in
                     match expr with
                     | SNoexpr -> allocatedMemory
                     | _ -> L.build_store variable_value allocatedMemory llbuilder;
variable_value

       and generate_local_vardecl datatype vname expr llbuilder =
              let allocatedMemory = L.build_alloca (get_llvm_type datatype) vname llbuilder in
              Hashtbl.add local_var_table vname allocatedMemory;
              let variable_value = expr_gen llbuilder expr in
                     match expr with
                     | SNoexpr -> allocatedMemory
                     | _ -> ignore (L.build_store variable_value allocatedMemory llbuilder);
variable_value

       and generate_while e s llbuilder =
              let start_block = L.insertion_block llbuilder in
              let parent_function = L.block_parent start_block in

              let pred_block = L.append_block context "while" parent_function in
              L.build_br pred_block llbuilder;

              let body_block = L.append_block context "while_body" parent_function in
              let body_builder = L.builder_at_end context body_block in

              let stmt = stmt_gen body_builder s in
              L.build_br pred_block body_builder;

              let pred_builder = L.builder_at_end context pred_block in
```

```
               let boolean_condition = expr_gen pred_builder e in
               let merge_block = L.append_block context "merge" parent_function in

               let whileStatement = L.build_cond_br boolean_condition body_block merge_block
pred_builder in

               L.position_at_end merge_block llbuilder;

               whileStatement

       and generate_for e1 e2 e3 s llbuilder =
               expr_gen llbuilder e1;
               let whileBody = SBlock [s; SExpr(e3, JInt)] in  (* JInt hardcoded*)
               generate_while e2 whileBody llbuilder

       and generate_if e s1 s2 llbuilder =
               let boolean_condition =
               match e with
                       | SId (n, dt)  -> get_value true n llbuilder
                       | _ -> expr_gen llbuilder e
               in

               let start_block = L.insertion_block llbuilder in
               let parent_function = L.block_parent start_block in

               let merge_block = L.append_block context "merge" parent_function in

               let then_block = L.append_block context "then" parent_function in
               L.position_at_end then_block llbuilder;

               let stmt1 = stmt_gen llbuilder s1 in
               L.build_br merge_block llbuilder;

               let else_block = L.append_block context "else" parent_function in
               L.position_at_end else_block llbuilder;
               let stmt2 = stmt_gen llbuilder s2 in
               L.build_br merge_block llbuilder;

               L.position_at_end start_block llbuilder;
               let ifStatement = L.build_cond_br boolean_condition then_block else_block
llbuilder in
               L.position_at_end merge_block llbuilder;

               ifStatement

       and expr_gen llbuilder = function
                 SInt_Lit (i)     -> L.const_int i32_t i
               | SBoolean_Lit (b) -> if b then L.const_int i1_t 1 else L.const_int i1_t 0
               | SFloat_Lit (f)   -> L.const_float f_t  f
               | SChar_Lit (c)    -> L.const_int i8_t (Char.code c)
               | SString_Lit (s)  -> build_global_stringptr s "tmp" llbuilder
               (*SNull*)
               | SId (n, dt)         -> get_value false n llbuilder
               | SBinop(e1, op, e2, dt) -> binop_gen e1 op e2 llbuilder
               | SUnop(op, e, dt)      -> unop_gen op e llbuilder
               | SAssign (e1, e2, dt)       -> assign_to_variable e1 e2 llbuilder
               | SCreateObject(id, el, d) -> generate_object_create id el llbuilder
               | SObjAccess(e1, e2, dt) -> generate_object_access e1 e2 llbuilder
               | SFuncCall (fname, expr_list, d, _) -> generate_function_call fname expr_list d
llbuilder
               | SNoexpr -> L.build_add (L.const_int i32_t 0) (L.const_int i32_t 0) "nop"
```

```
llbuilder
               | SArrayCreate (datatype, el, d)     -> generate_array datatype el llbuilder
               | SArrayAccess(e, el, d) -> generate_array_access true e el llbuilder
               | STupleCreate(dt_list, el, d) -> generate_create_tuples dt_list el llbuilder
               | STupleAccess(e1, e2, d) -> generate_tuple_access false e1 e2 llbuilder
               | _ -> raise(Failure("No match for expression"))

       and generate_object_access e1 e2 llbuilder =
               (*
               let classname = match e1 with
                      | SId(id, dt) -> (match dt with
                                                | Object(s) -> s)
               in*)
               let objectMemory = match e1 with
                      | SId(id, dt) -> get_value true id llbuilder
                      | _ -> raise(Failure("Not an id of object"))
               in
               let get_variable n llbuilder =
                (*Hard coded to demonstrate code working*)
                      let index = Hashtbl.find struct_field_idx_table ("Car."^n) in
                      let var = L.build_struct_gep objectMemory index "temp" llbuilder in
                      L.build_load var n llbuilder
               in
               let rhs = match e2 with
                      | SId(id, dt) -> get_variable id llbuilder
                      | _ -> raise(Failure("Function acces not yet supported"))
               in
               rhs

       and generate_create_tuples dt_list expr_list llbuilder =
               let type_list = List.map (function dt -> get_llvm_type dt) dt_list in
               (*let type_list = i32_t :: type_list in *)
               let type_array = (Array.of_list type_list) in
               let struct_type = L.packed_struct_type context type_array in
               let vname = "dummy" in
               let allocatedMemory = L.build_alloca struct_type vname llbuilder in
               List.iteri (
                      fun i f ->
                let tuple_value = L.build_struct_gep allocatedMemory i "temp" llbuilder in
                      ignore(L.build_store (match f with
                                     | SId(id, d) -> get_value true id llbuilder
                                                | SArrayAccess(e, el, d) -> generate_array_access
true e el llbuilder
                                                | STupleAccess(e1, e2, d) -> generate_tuple_access
true e1 e2 llbuilder
                                                | _ -> expr_gen llbuilder f) tuple_value
llbuilder);
           ) expr_list;

               L.build_pointercast allocatedMemory (L.pointer_type struct_type) "tupleMemAlloc"
llbuilder

       and generate_tuple_access deref e1 e2 llbuilder =
               let vname = "dummy" in
               let index =  match e2 with
                                | SInt_Lit(i) -> i
                                | _ -> raise(Failure("Not an int"))
                      in
               let tuple = match e1 with
                      | SId(id, d) -> get_value true id llbuilder
                      | _ -> raise(Failure("Not an id"))
```

```
               in
               let tuple_value = L.build_struct_gep tuple index vname llbuilder in
               if deref
                      then L.build_load tuple_value vname llbuilder
                      else tuple_value

       and generate_array_access deref e el llbuilder =
               match el with
               | [h] -> let index = match h with
                                    | SId(id, d) -> get_value true id llbuilder
                                    | _ -> expr_gen llbuilder h
                             in
                             let index = L.build_add index (const_int i32_t 1) "1tmp"
llbuilder in
                      let arr = match e with
                             | SId(n, dt) -> get_value true n llbuilder
                             | _ -> raise(Failure("Can't access array!"))
                      in
                      let _val = L.build_gep arr [| index |] "2tmp" llbuilder in
                      if deref
                             then build_load _val "3tmp" llbuilder
                             else _val
               | _ ->  raise(Failure("Two dimentional array not supported"))

       and generate_array datatype expr_list llbuilder =
               match expr_list with
               | [h] -> generate_one_d_array datatype (expr_gen llbuilder h) llbuilder
               | _ ->  raise(Failure("Two dimentional array not supported"))
               (*| [h;s] -> generate_one_d_array datatype (expr_gen llbuilder h) (expr_gen
llbuilder s) llbuilder *)

       and generate_one_d_array datatype size llbuilder =
               let t = get_llvm_type datatype in

               let size_t = L.build_intcast (L.size_of t) i32_t "4tmp" llbuilder in
               let size = L.build_mul size_t size "5tmp" 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 "6tmp" llbuilder in
               let arr = L.build_pointercast arr (pointer_type t) "7tmp" llbuilder in

               let arr_len_ptr = L.build_pointercast arr (pointer_type i32_t) "8tmp" llbuilder
in

               ignore(L.build_store size_real arr_len_ptr llbuilder);
               initialise_array arr_len_ptr size_real (const_int i32_t 0) 0 llbuilder;
               arr

       and initialise_array arr arr_len init_val start_pos llbuilder =
               let new_block label =
                      let f = L.block_parent (L.insertion_block llbuilder) in
                      L.append_block (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;

               (* Counter into the length of the array *)
```

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

                (* Assign array position to init_val *)
                let arr_ptr = build_gep arr [| counter |] "tmp" llbuilder in
                ignore (build_store init_val arr_ptr llbuilder);
                ignore (build_br bbcond llbuilder);
                position_at_end bbdone llbuilder

        and generate_function_call fname expr_list d llbuilder =
                match fname with
                        | "print" -> print_func_gen "" expr_list llbuilder
                        | "println" -> print_func_gen "\n" expr_list llbuilder
                        | _ -> let f = find_func_in_module fname in
                                        let map_param_to_llvalue llbuilder e = match e with
                                                | SId(id, d) -> get_value true id llbuilder
                                                | SArrayAccess(e, el, d) -> generate_array_access
true e el llbuilder
                                                | STupleAccess(e1, e2, d) -> generate_tuple_access
true e1 e2 llbuilder
                                                | _ -> expr_gen llbuilder e
                                        in
                                        let params = List.map (map_param_to_llvalue llbuilder)
expr_list in (*Fix passing variable to function*)
                                        L.build_call f (Array.of_list params) (fname^"_result")
llbuilder

        and generate_object_create id el llbuilder =
                let f = find_func_in_module id in
                let params = List.map (expr_gen llbuilder) el in
                let obj = L.build_call f (Array.of_list params) "tmp" llbuilder in
                obj

        and binop_gen e1 op e2 llbuilder =
                let value1 =  match e1 with
                        | SId(id, d) -> get_value true id llbuilder
                        | SArrayAccess(e, el, d) -> generate_array_access true e el llbuilder
                        | STupleAccess(e1, e2, d) -> generate_tuple_access true e1 e2 llbuilder
                        | _ -> expr_gen llbuilder e1
                in
                let value2 = match e2 with
                        | SId(id, d) -> get_value true id llbuilder
                        | SArrayAccess(e, el, d) -> generate_array_access true e el llbuilder
                        | STupleAccess(e1, e2, d) -> generate_tuple_access true e1 e2 llbuilder
                        | _ -> expr_gen llbuilder e2
                in

                let int_binop value1 value2 llbuilder = (match op with
                          Add          -> L.build_add
                        | Sub          -> L.build_sub
                        | Mult         -> L.build_mul
                        | Div          -> L.build_sdiv
                        | Equal        -> L.build_icmp L.Icmp.Eq
                        | Neq          -> L.build_icmp L.Icmp.Ne
                        | Less         -> L.build_icmp L.Icmp.Slt
                        | Leq          -> L.build_icmp L.Icmp.Sle
```

```
                | Greater      -> L.build_icmp L.Icmp.Sgt
                | Geq          -> L.build_icmp L.Icmp.Sge
                | And          -> L.build_and
                | Or           -> L.build_or
                | _            -> raise(Failure("Invalid operator for ints"))
        ) value1 value2 "binop_int" llbuilder
        in

        let float_binop value1 value2 llbuilder = (match op with
                  Add          -> L.build_fadd
                | Sub          -> L.build_fsub
                | Mult         -> L.build_fmul
                | Div          -> L.build_fdiv
                | Equal        -> L.build_fcmp L.Fcmp.Oeq
                | Neq          -> L.build_fcmp L.Fcmp.One
                | Less         -> L.build_fcmp L.Fcmp.Ult
                | Leq          -> L.build_fcmp L.Fcmp.Ole
                | Greater      -> L.build_fcmp L.Fcmp.Ogt
                | Geq          -> L.build_fcmp L.Fcmp.Oge
                | And          -> L.build_and
                | Or           -> L.build_or
                | _            -> raise(Failure("Invalid operator for ints"))
        ) value1 value2 "binop_float" llbuilder

        in

        let decide_on_type e1 e2 llbuilder =
               match ((Semant.typOFSexpr e1), (Semant.typOFSexpr e2)) with
                | (JInt,JInt) -> int_binop value1 value2 llbuilder
                | (JInt, JFloat) -> float_binop value1 value2 llbuilder
                | (JFloat, JInt) -> float_binop value1 value2 llbuilder
                | (JFloat, JFloat) -> float_binop value1 value2 llbuilder
                | (JInt, _) -> int_binop value1 value2 llbuilder
                | (_, JInt) -> int_binop value1 value2 llbuilder
                | (_, _) -> int_binop value1 value2 llbuilder

        in

        decide_on_type e1 e2 llbuilder


and unop_gen op e llbuilder =
        let exp_type = Semant.typOFSexpr e in
        let value = expr_gen llbuilder e in
        (match (op, exp_type) with
                  (Not, JBoolean) -> L.build_not
                | (Sub, JInt) -> L.build_neg
                | (Sub, JFloat) -> L.build_fneg
                | _ -> raise(Failure("Invalid unop usage"))
        ) value "tmp" llbuilder

and get_value deref vname llbuilder =
        if deref then
        let var = try Hashtbl.find global_var_table vname with
        | Not_found -> try Hashtbl.find local_var_table vname with
                | Not_found -> raise (Failure("unknown variable name " ^ vname))
        in
        L.build_load var vname llbuilder

else
        let var = try Hashtbl.find global_var_table vname with
```

```
                | Not_found -> try Hashtbl.find local_var_table vname with
                        | Not_found -> raise (Failure("unknown variable name " ^ vname))
                in
                var

        and assign_to_variable e1 e2 llbuilder =
                let vmemory = match e1 with
                        | SId(s, d) -> get_value false s llbuilder
                        | SArrayAccess(e, el, d) -> generate_array_access false e el llbuilder
                        | STupleAccess(e1, e2, d) -> generate_tuple_access false e1 e2 llbuilder
                in
                let value = match e2 with
                | SId(id, d) -> get_value true id llbuilder
                | _ -> expr_gen llbuilder e2
                in
                L.build_store value vmemory llbuilder

        and print_func_gen newLine expr_list llbuilder =
                let printf = find_func_in_module "printf" in
                let tmp_count = ref 0 in
                let incr_tmp = fun x -> incr tmp_count in
                let map_expr_to_printfexpr expr = match expr with
                        | SId(id, d) -> get_value true id llbuilder (*(match d with
                                                | A.JBoolean -> incr_tmp ();
                                                                let tmp_var =
"print_bool" in
                                                                let trueStr =
SString_Lit("true") in
                                                                let falseStr =
SString_Lit("false") in
                                                                let id =
SId(tmp_var, Arraytype(JChar, 1)) in

        ignore(stmt_gen llbuilder (SLocalVarDecl(Arraytype(JChar, 1), tmp_var, SNoexpr)));

        ignore(stmt_gen llbuilder (SIf(expr,

        SExpr(SAssign(id, trueStr, Arraytype(JChar, 1)), Arraytype(JChar, 1)),

        SExpr(SAssign(id, falseStr, Arraytype(JChar, 1)), Arraytype(JChar, 1)))));
                                                                expr_gen
llbuilder id
                                                | A.Arraytype(JChar, _) -> let llvalue =
get_value true id llbuilder in

        L.build_load llvalue "string" llbuilder

                                                | _ -> get_value true id llbuilder)*)
                        | SArrayAccess(e, el, d) -> generate_array_access true e el llbuilder
                        | STupleAccess(e1, e2, d) -> generate_tuple_access true e1 e2 llbuilder
                        | SBoolean_Lit (b) -> if b then (expr_gen llbuilder
(SString_Lit("true"))) else (expr_gen llbuilder (SString_Lit("false")))
                        | _ -> expr_gen llbuilder expr
                in
                let params = List.map map_expr_to_printfexpr expr_list in
                let expr_types = List.map (Semant.typOFSexpr) expr_list in

                let map_expr_to_type e = match e with
                        JInt     ->    "%d"
                | JBoolean      ->    "%s" (*needs to be implemented*)
                | JFloat        ->    "%f"
```

```
                | JChar         ->    "%c"
                | Arraytype(JChar, _)        -> "%s"
                | _           -> raise (Failure("Print invalid type"))

                in
                let print_types = List.fold_left (fun s t -> s ^ map_expr_to_type t) ""
expr_types in
                let s = build_global_stringptr (print_types ^ newLine) "printf" llbuilder in

                (**    let zero = const_int i32_t 0 in**)
                let s = build_in_bounds_gep s [| zero |] "printf" llbuilder in

                L.build_call printf (Array.of_list (s :: params)) "printf" llbuilder
        in

        (*Function generation*)

        let build_function sfunc_decl =
                Hashtbl.clear local_var_table;

                let f = find_func_in_module sfunc_decl.sfname in
                let llbuilder = L.builder_at_end context (L.entry_block f) in

                (*L.position_at_end (L.entry_block f) llbuilder; *)

                let init_formals f sfformals =
                        let sfformals = Array.of_list (sfformals) in
                        Array.iteri (
                                fun i a ->
                                let formal = sfformals.(i) in
                                let allocatedMemory = stmt_gen llbuilder
(SLocalVarDecl(formal.sformal_type, formal.sformal_name, SNoexpr)) in
                                        let n = formal.sformal_name in
                                set_value_name n a;
                                ignore (L.build_store a allocatedMemory llbuilder);
                    )
                    (params f)
                in
                let _ = init_formals f sfunc_decl.sfformals in
                let _ = stmt_gen llbuilder (SBlock (sfunc_decl.sfbody)) in
                if sfunc_decl.sfreturn = JVoid
                then ignore (L.build_ret_void llbuilder);
                ()
        in
        let _ = List.map build_function functions in

        let build_constructors class_name =

                (*If a class has multiple constructors it will get overwritten at the moment*)
                let build_constructor constructor =
                        Hashtbl.clear local_var_table;

                        let f = find_func_in_module class_name.scname in
                        let llbuilder = L.builder_at_end context (L.entry_block f) in

                        let struct_type = find_llvm_struct_type class_name.scname in
                        let allocatedMemory = L.build_alloca struct_type "object" llbuilder in
                        List.iteri (
                                fun i f ->
                                let expr = f.svexpr in
                        let tuple_value = L.build_struct_gep allocatedMemory i "temp" llbuilder
```

```
in
                    ignore(L.build_store (match expr with
                           | SId(id, d) -> get_value true id llbuilder
                                  | SArrayAccess(e, el, d) -> generate_array_access true e
el llbuilder
                                  | STupleAccess(e1, e2, d) -> generate_tuple_access true e1
e2 llbuilder
                                  | _ -> expr_gen llbuilder expr) tuple_value llbuilder);

                    set_value_name f.svname tuple_value;

                    let scope = f.svscope in
                    Hashtbl.add (match scope with
                                         | A.Public -> global_var_table
                                         | A.Private -> class_private_vars) f.svname
tuple_value;
             ) class_name.scbody.svariables;

                    let pointer_to_class = L.build_pointercast allocatedMemory
(L.pointer_type struct_type) "tupleMemAlloc" llbuilder in

                    let init_formals f sfformals =
                           let sfformals = Array.of_list (sfformals) in
                           Array.iteri (
                                  fun i a ->
                                  let formal = sfformals.(i) in
                                  let varMem = stmt_gen llbuilder
(SLocalVarDecl(formal.sformal_type, formal.sformal_name, SNoexpr)) in
                                  let n = formal.sformal_name in
                                  set_value_name n a;
                                  ignore (L.build_store a varMem llbuilder);
                           )
                           (params f)
                    in
                    let _ = init_formals f constructor.sfformals in
                    let _ = stmt_gen llbuilder (SBlock (constructor.sfbody)) in

                    L.build_ret pointer_to_class llbuilder
             in
             List.map build_constructor class_name.scbody.sconstructors in
      let _ =  List.map build_constructors classes in


      (*Main method generation*)
      let build_main main =
             let fty = L.function_type i32_t[||] in
             let f = L.define_function "main" fty the_module in
             let llbuilder = L.builder_at_end context (L.entry_block f) in

             let _ = stmt_gen llbuilder (SBlock (main.sfbody)) in

             L.build_ret (L.const_int i32_t 0) llbuilder
      in
      let _ = build_main main in

      (*Class generation *)

      let build_classes sclass_decl =
             let rt = L.pointer_type i64_t in
             let void_pt = L.pointer_type i64_t in
             let void_ppt = L.pointer_type void_pt in
```

```
		let f = find_func_in_module "lookup" in
		let llbuilder = L.builder_at_end context (entry_block f) in

		let len = List.length sclass_decl in

		let total_len = ref 0 in

		let scdecl_llvm_arr = L.build_array_alloca void_ppt (const_int i32_t len) "tmp"
llbuilder in

		let handle_scdecl scdecl =
			let index = try Hashtbl.find Semant.classIndices scdecl.scname with
			| Not_found -> raise (Failure("can't find classname" ^ scdecl.scname))
in
			let len = List.length scdecl.scbody.smethods in
			let sfdecl_llvm_arr = L.build_array_alloca void_pt (const_int i32_t len)
"tmp" llbuilder in

			let handle_fdecl i sfdecl =
				let fptr = find_func_in_module sfdecl.sfname in
				let fptr = L.build_pointercast fptr void_pt "tmp" llbuilder in

				let ep = L.build_gep sfdecl_llvm_arr [| (const_int i32_t i) |]
"tmp" llbuilder in
				ignore(L.build_store fptr ep llbuilder);
			in
			List.iteri handle_fdecl scdecl.scbody.smethods;
			total_len := !total_len + len;

			let ep = L.build_gep scdecl_llvm_arr [| (const_int i32_t index) |] "tmp"
llbuilder in
			ignore(build_store sfdecl_llvm_arr ep llbuilder);
		in
		List.iter handle_scdecl sclass_decl;

		let c_index = param f 0 in (*breaks*)
		let f_index = param f 1 in
		L.set_value_name "c_index" c_index;
		L.set_value_name "f_index" f_index;

		if !total_len == 0 then
			L.build_ret (const_null rt) llbuilder
		else
			let vtbl = L.build_gep scdecl_llvm_arr [| c_index |] "tmp" llbuilder in
			let vtbl = L.build_load vtbl "tmp" llbuilder in
			let fptr = L.build_gep vtbl [| f_index |] "tmp" llbuilder in
			let fptr = L.build_load fptr "tmp" llbuilder in

			L.build_ret fptr llbuilder
	in
	let _ = build_classes classes in

	the_module;
```

testall.sh Ashley

```
#!/bin/sh

# Regression testing script for MicroC
# 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 microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
JAVAPM="./javapm"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.javapm 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/.javapm//'`
    reffile=`echo $1 | sed 's/.javapm$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
    Run "$JAVAPM" $1 ">" "${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
       if [ $keep -eq 0 ] ; then
           rm -f $generatedfiles
       fi
       echo "OK"
       echo "###### SUCCESS" 1>&2
    else
       echo "###### FAILED" 1>&2
       globalerror=$error
    fi
}

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$JAVAPM" $1 "2>" "${basename}.err" ">>" $globallog &&
```

```
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
       if [ $keep -eq 0 ] ; then
           rm -f $generatedfiles
       fi
       echo "OK"
       echo "###### SUCCESS" 1>&2
    else
       echo "###### FAILED" 1>&2
       globalerror=$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

# temporarily remove tests/fail-*.javapm from else files=...
if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.javapm tests/fail-*.javapm"
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
```