CS 4115 Final Report: **JaTesté**

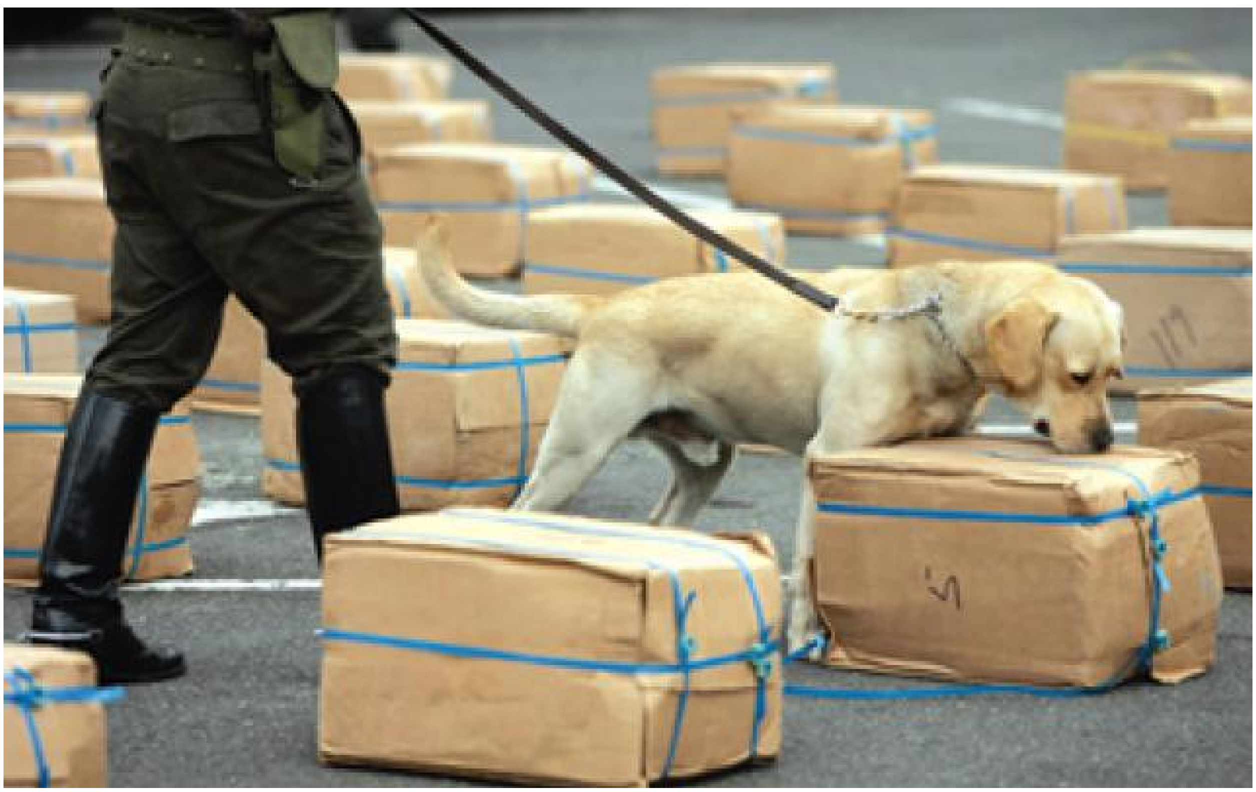
Andrew Grant
amg2215@columbia.edu

Jemma Losh
jal2285@columbia.edu

Jared Weiss
jbw2140@columbia.edu

Jake Weissman
jdw2159@columbia.edu

5/11/2016



JaTesté: build software so secure you may actually make America Great Again.

Contents

1 Introduction

1.1 Motivation

The goal of JaTesté is to design a language that promotes good coding practices - mainly as it relates to testing. JaTesté will allow the programmer to easily define test cases, for any function, directly into his or her source code. This will ensure that no code goes untested and will increase the overall quality of programmer code written in our language. By directly embedding test cases into source code, we remove the hassle associated with manually creating test files.

1.2 Language Description

JaTesté is an imperative, C-like language, with a few object oriented features added, that makes it easy to add test cases to ones code. The syntax is very similar to C, but with the added capability of associating functions with "structs", similarly to how methods are implemented in objects in Java. Test cases are easily appended to user-defined functions, by appending the keyword "with test" onto the end of a function. The compiler subsequently outputs two separate files: 1) a regular executable 2) an executable test file that runs all user defined tests.

1.3 Related Work

The JaTesté syntax is very much inspired by C and Java, two of the most popular programming languages in use today. Nonetheless, JaTesté's syntax is relatively simple, as so anyone with basic, imperative programming language experience should be able to pick it up quickly.

1.4 Source Code

We have open-sourced the repository under the MIT license and it is available at `https://github.com/jaredweiss/JaTeste`

2 Short Tutorial

2.1 Environment

The compiler was developed and tested on an Ubuntu 15.10 virtual machine. We ran the Linux image through VirtualBox, but any standard hypervisor should suffice.

The compiler translates JaTesté source code into LLVM, a portable assembly-like language. You need to download LLVM from `http://llvm.org/releases/download.html` in order to run LLVM code.

The compiler is written completely in OCaml. The OCaml compiler can be downloaded from `http://caml.inria.fr/download.en.html`

2.2 Using the JaTeste Compiler

From any given JaTesté source file, the compiler generates (1) an executable file, and if the "-t" command line argument is supplied, (2) an executable test file with all the relevant user-defined test cases. This relieves the programmer from having to manually create test files from scratch. All code is compiled into LLVM, a portable assembly-like language. To run the compiled LLVM code, we use 'lli", an LLVM interpreter.

For (1) the regular executable, the compiler completely disregards the tests and thus produces an executable as if the test cases had never been written. This enables the programmer to produce a regular executable without the overhead of the test cases when he or she desires. Thus, while a JaTesté program can be embedded with an unlimited number of test cases, the programmer can always generate a standard runnable program without the test case code.

For the test file, the compiler turns each test case into it's own function, and subsequently runs each of these functions from a brand new, compiler generated "main" function. "main" simply runs through each of these compiler-generated functions, each of which runs the user-defined tests. Furthermore, the compiler adds "print" calls to each test letting the user know whether a given test passed or failed.

When inside the src folder, type "make all" to generate the Jateste executable. To run type ./jateste.native [optional -options] <source_file.jt>

The possible arguments are:

- No arguments If run without arguments, the compiler ignores the test cases and creates a regular executable, source_file.ll, as if the test cases were never there to begin with.
- "-t" Compile with test This results in the compiler creating two LLVM files: 1) a regular executable named "source_file.ll" as above, and 2) a test file named "source_file-test.ll". Both of these are LLVM executables.
- "-l" Scan only This results in the compiler simply scanning the source code. This is mainly used for debugging purposes.
- "-p" Parse only This results in the compiler simply parsing the source code. Also mainly used for debugging purposes.
- "-se" SAST This results in the compiler running the semantic checker on the source code and then stopping. Also mainly used for debugging purposes.
- "-ast" AST This also results in the compiler running the semantic checker on the source code and then stopping. Also mainly used for debugging purposes.

A maximum of one command line argument at a time can be supplied when running the compiler.

2.3 JaTesté Program Structure

Any given JaTesté program can be broken down into four segments:

1. List of includes. JaTesté programs can include other JaTesté source code files. This list should go at the top of the source code file.

2. global variable declarations. Global variable declarations are exactly like in C and immediately follow included headers.

3. function definitions. Function definitions are similar to C, except the keyword "func" is needed before the return type. Furthermore, all variable declarations must be done at the beginning of each function. A "main" function is required for all JaTesté programs; this is where execution starts when a program is run. Included JaTesté headers shouldn't have a "main" function, however.

4. struct definitions. Structs are also similar to C, except the programmer can define methods within the struct. All struct fields must be declared before the struct's methods. The syntax for struct methods is exactly like any regular function, except the keyword "method" is used instead of "func".

Each of these segments must be used in the order given above.

2.4 Programming Language Paradigm

2.4.1 Imperative Paradigm

JaTesté is a pretty standard imperative programming language that has light object-oriented features. Since JaTesté is not functional, functions can have side-effects. Anyone familiar with C, C++, or Java should have any especially easy time understanding JaTesté.

2.4.2 Pass-by-value

JaTesté is a pass-by-value programming language. Nonetheless, there is strong support for pointers which gives the programmer the ability to pass by reference. & is used to get the address of a variable. *<type> is used to declare a variable as a pointer type. * can subsequently be used to deference a pointer.

2.4.3 Typing

All variables must be declared along with their respective type before they are used. JaTesté has relatively strict typing checking - values of different types cannot be cast to each other. Note, void pointers are not allowed; that is, pointers must define what data type they are pointing to.

2.4.4 Memory Layout

Global variables are stored in the data section, local variables are allocated on the stack, arrays and structs can be allocated on the stack with the "new" keyword, and code is stored in the text segment of the program. When external JaTesté headers are included, the respective code is simply appended to the source code file. Thus, the memory layout of a given JaTesté program is pretty standard.

2.5 Basics

2.5.1 Primitives

JaTesté supports the following primitives:

- int
- double
- char
- boolean
- string

2.5.2 Arrays

In JaTesté, an array of type "t" and size "n" is an allocated block of memory that holds n contiguous values all of type t. This exactly how arrays are implemented in C, C++, and Java. They can be allocated on the stack or heap.

2.5.3 Structs

Structs in JaTesté are just like in C, but with the added capability of giving them methods. This makes it easier to associate functions with the data they are meant to manipulate. Structs can be allocated on the stack or heap.

2.5.4 Operators

JaTesté supports the following operators:

- Arithmetic: $+, -, *, \hat{;}\ , \backslash$
- Logical: && , ||
- Relational: $==, <, <=, !=, >, >=$

2.5.5 Control Flow

JaTesté supports standard control flow constructs, such as for and while loops, and if-else statements. "return" is used to return control to the caller, as in almost any other programming language.

2.5.6 Test Cases

Test cases are used to test user-defined functions, and are at the heart of the JaTesté programming language. The best way to illustrate how to take advantage of JaTesté's built in testing functionality is through an example:

```
func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0)  == 10);
        assert(add(b,b)  == 10);
        assert(add(a,b)  == 15);
} using {
        int a;
        int b;
        a = 10;
        b = 5;
}
```

Here we've defined a function, "add", and appended a few test cases using the built-in "with test" and "using" keywords. It is within "with test { . . . }" where the programmer actually defines his or her tests. In this example, the programmer is verifying that the add() function returns the correct value for three specific inputs. Notice how each test uses variables "a" and/or "b"; these variables are defined inside "using { . . . }". Thus, "using { . . . }" is used to set up the environment for the test cases. This makes it easier for the programmer to write meaningful "assert" statements inside the "with test { . . . }" testing block.

2.6 Sample Programs

Here are a few example programs.

1. Here's the first example of a JaTesté program. As illustrated, the syntax is very similar to C.

```
#include_jtlib <math.jt>
int my_global;

func int main()
{
        int i;
        i = add(2,3);
        if (i == 5) {
                print("passed");
        }
        return 0;
}


func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0)  == 10);
} using {
        int a;
        int b;
        a = 10;
        b = 5;
}

struct house {
    int price;
    int zipcode;
};
```

Note the structure of the program. More specifically, include files are specified at the top, global variables are declared next, functions definitions are coded in the middle, and structs are defined at the end of the source file.

As can be seen the "add" function has a snippet of code directly following it. This is an example of a program that takes advantage of JaTesté's built-in testing framework. The code within the "with test" block defines the test cases for the add function, via an assert statement. In this case, the programmer has only specified one test. Furthermore, note the code following the test case that starts with "using { ... }". This block is used to set up the environment for the test cases. In this example, the single test case "assert(a == 10);" references the variable "a"; it is within the scope of the "using " block that "a" is defined.

2. Here's another JaTesté program:

```
func int main()
{
        int a;
        int b;
        int c;

        a = 10;
        b = 5;
        c = 0;

        a = b - c;
        if (a == 5) {
```

```
                print("passed");
        }
        return 0;
}


func int sub(int x, int y)
{
        return x - y;
} with test {
        assert(sub(10,5) == b - 5);
        assert(sub(b,d) == 1);
        assert(sub(c,d) == 4);
} using {
        int a;
        int b;
        int c;
        int d;
        a = 5;
        b = 10;
        c = 13;
        d = 9;
}
```

This example is similar to the previous one; however, note that there are now multiple "asserts". The programmer may define as many test cases as he or she wants. When compiled with the "-t" command line argument, the compiler creates a file "test-testcase2-test.ll" (the name of the source program being "test-testcase2.jt" in this case) in addition to a regular executable (which would be named testcase2.ll in this case). When "lli test-testcase2-test.ll" is run, the output is:
Tests:
subtest tests:
sub(10,5) == b - 5 passed
sub(b,d) == 1 passed
sub(c,d) == 4 passed
As illustrated, the test program will let you know which tests pass and which fail.

3. Here we introduce structs. The syntax is very similar to C:

```
int global_var;

func int main()
{
        int tmp;
        struct rectangle *rec_pt;
        rec_pt = new struct rectangle;
        update_rec(rec_pt, 6);
        tmp = rec_pt->width;

        print(tmp);

        return 0;
}

func void update_rec(struct rectangle *p, int x)
{
        p->width = x;
} with test {
```

```
        assert(t->width == 10);
} using {
        struct rectangle *t;
        t = new struct rectangle;
        update_rec(t, 10);
}

struct rectangle {
        int width;
        int height;
};
```

Again, note the syntax of the whole program here. More precisely, global variables are declared at the top, functions are defined in the middle, and structs are defined at the bottom. Note, this file does not use any header files; these would go above the global variable declaration. This is the required order for *all* JaTesté programs.

4. As previously explained, JaTesté is a pass-by-value programming language. For those familiar with C, this paradigm should be very familiar. For those not, this simply means every variable is passed around by value, not address. Pointers can be used to mimic pass-by-reference as the following example shows:

```
1  func int main()
2  {
3          int a;
4          int b;
5          int *c;
6
7
8          a = 10;
9          b = 500;
10
11         c = &b;
12
13         if (*c == 500) {
14                 print("passed");
15         } else {
16                 print("failed");
17         }
18
19         return 0;
20 }
```

& is used to return the address of a variable, as in done on line 11 of this program. * is used to declare a variable as a pointer, as is done with the variable “c” above on line 5. Thus, line 11 sets the variable “c” to the address of “b”. Since “b” contains value 500, and “c” contains the address of “b”, we can say that “c” points to “b’s ” value of 500. * can subsequently be used to deference pointers, as is done on line 13 inside “if (*c == 500)”. Here, we use * to access the value pointed to by “c”, which is “b’s ” value of 500 and so the expression inside the if-statement will evaluate to true.

5. All variables are allocated on the stack, unless the “new” keyword is used in conjunction with structs and/or arrays, as the following example illustrates.

```
func int main()
{

        struct house *my_house;
        int price;
        int vol;
```

```

        my_house = new struct house;

        my_house->set_price(100);
        my_house->set_height(88);
        my_house->set_width(60);
        my_house->set_length(348);

        price = my_house->get_price();
        vol = my_house->get_volumne();

        print(price);
        print(vol);
        return 0;
}

struct house {
        int price;
        int height;
        int width;
        int length;

        method void set_price(int x)
        {
                price = x;
        }

        method void set_height(int x)
        {
                height = x;
        }

        method void set_width(int x)
        {
                width = x;
        }

};
```

The line my_house = new struct house; is used to allocate memory on the heap for a struct object. Note “− >” is used to access the given structs methods. This syntax is required because my_house is a pointer to a struct. If my_house was a regular house struct variable, and not a pointer, a dot would suffice (e.g. my_house.set_price(100);) This example also illustrates the use of methods within structs. Unlike C, you can directly embed methods in structs. The functionality is very similar to how methods work in object-oriented languages.

3 Language Reference Manual

3.1 Lexical Conventions

This section will describe how input code will be processed and how tokens will be generated.

3.1.1 Identifiers

Identifiers are used to name variables as in most programming language. An identifier can include all letters, digits, and the underscore character. An identifier must start with either a letter or an underscore - it cannot start with a digit. Capital letters will be treated differently from lower case letters. The set of keyword, listed below, cannot be used as identifiers.
Here's the regular expression for an identifier:

```
['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm)}
```

3.1.2 Keywords

Keywords are a set of words that serve a specific purpose in our language and may not be used by the programmer for any other reason. The list of keywords the language recognizes and reserves is as follows:
`if, else, return, while, for, assert, void, struct, method, double, int, char, string, bool, true, false, func, new, free, NULL` Each keyword's meaning will be explained at some point later in this chapter.

3.1.3 Constants

Our language includes integer, character, real number, and string constants. They're defined in the following sections.

3.1.4 Integer Constants

Integer constants are a sequence of digits. An integer is taken to be decimal. The regular expression for an integer is as follows:

```
digit = ['0' - '9']
int = digit+
```

3.1.5 Double Constants

Real number constants represent a floating point number. They are composed of a sequence of digits, representing the whole number portion, followed by a decimal and another sequence of digits, representing the fractional part. Here are some examples.

```
let double = (digit+) ['.'] digit+
```

3.1.6 Character Constants

Character constants hold a single character and are enclosed in single quotes. They are stored in a variable of type "char". The regular expression for a character is as follows:

```
let my_char = '''['a' - 'z' 'A' - 'Z']'''
```

3.1.7 String Constants

Strings are a sequence of characters enclosed by double quotes. A String is treated like a character array. The regular expression for a string is as follows:

```
my_string = '"' (['a' - 'z'] | [' '] | ['A' - 'Z'] | ['_'] | '!' | ',' )+ '"'
```

Strings are immutable; once they have been defined, they cannot change.

3.1.8 Operators

Operators are special tokens such as multiply, equals, etc. that are applied to one or two operands. Their use will be explained further in section 3.2.

3.1.9 White Space

White space is considered to be a space, tab, or newline. It is used for token delimitation, but has no meaning otherwise. That is, when compiled, white space is thrown away.

```
WHITESPACE = "[' '  '\t' '\r' '\n']"
```

3.1.10 Comments

A comment is a sequence of characters beginning with a forward slash followed by an asterisk. It continues until it is ended with an asterisk followed by a forward slash. Comments are treated as white space.

```
COMMENT = "/\* [^ \*/]* \*/ "
```

3.1.11 Separators

Separators are used to separate tokens. Separators are single character tokens, except for white space which is a separator, but not a token.

```
'('         { LPAREN }
')'         { RPAREN }
'{'         { LBRACE }
'}'         { RBRACE }
';'         { SEMI }
','         { COMMA }
```

3.2 Data Types

The data types in JaTeste can be classified into three categories: primitive types, structures, and arrays.

3.2.1 Primitives

The primitives our language recognizes are int, double, bool, char, and string.

3.2.2 Integer Types

The integer data type is a 32 bit value that can hold whole numbers ranging from $-2,147,483,648$ to $2,147,483,647$. Keyword `int` is required to declare a variable with this type. A variable must be declared before it can be assigned a value; this cannot be done in one step.

```
int a;
a = 10;
a = 21 * 2;
```

The grammar that recognizes an integer deceleration is:

```
typ ID
```

The grammar that recognizes an integer initialization is:

```
ID ASSIGN expr
```

3.2.3 Boolean Types

The “bool” type is your standard Boolean data type that can take on one of two values: 1) true 2) false. Booleans get compiled into 1 bit integers.

```
bool my_bool;
my_bool = true;
```

3.2.4 Double Types

The double data type is a 64 bit value. Keyword `double` is required to declare a variable with this type. A variable must be declared before it can be assigned a value, this cannot be done in one step just like with ints.

```
double a;
a = 9.9;
a = 17 / 3;
```

The grammar that recognizes a double deceleration is:

```
typ ID
```

The grammar that recognizes a double initialization is:

```
ID ASSIGN expr
```

3.2.5 Character Type

The character type is an 8 bit value that is used to hold a single character. Like most programming languages, characters in Jateste get compiled into a 1 byte integer. The keyword `char` is used to declare a variable with this type. A variable must be declared before it can be assigned a value.

```
char a;
a = 'h';
```

The grammar that recognizes a char deceleration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

3.2.6 String Type

The string type is variable length and used to hold a string of chars. The keyword `string` is used to declare a variable with this type. A variable must be declared before it can be assigned a value, as with all variables.

```
string a;
a = "hello";
```

The grammar that recognizes a char deceleration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

3.2.7 Structures

The structure data type is a user-defined collection of primitive types, other structure data types and, optionally, methods. The keyword “struct” followed by the name of the struct is used to define structures. Curly braces are then used to define what the structure is actually made of. As an example, consider the following:

3.2.8 Defining Structures

```
struct square {
        int height;
        int width;

        method int get_area()
        {
                int temp_area;
                temp_area = height * width;
                return temp_area;
        }

        method void set_height(int h) {
                height = h;
        }

        method void set_width(int w) {
                width = w;
        }

};

struct manager = {
struct person name;
int salary;
};
```

Here we have defined two structs, the first being of type `struct square` and the second of type `struct manager`. Note square struct has methods associated with it, unlike the manage struct which is just like a regular C struct. The grammar that recognizes defining a structure is as follows:

```
STRUCT ID LBRACE vdecl_list struc_func_decls RBRACE SEMI
```

3.2.9 Initializing Structures

To create a structure on the heap, the "new" keyword is used:

```
struct manager *yahoo_manager;
struct person sam;

yahoo_manager = new struct manager;
sam = new struct person;
```

```
NEW STRUCT ID
```

Here, we create two variables yahoo_manager and sam on the heap. The first is of type "struct manager", and the second is of type "struct person". When using the "new" keyword, the memory is allocated on the heap for the given struct. "free(p)" is used to de-allocate heap memory pointed to by "p". Structs can also be allocated on the stack as follows:

```
struct manager yahoo_manager;
struct person sam;
```

3.2.10 Accessing Structure Members

To access structs allocated on the heap, and modify its variables, a right arrow C is used followed by the variable name:

```
yahoo_manager->name = sam;
yahoo_manager->age = 45;
yahoo_manager->salary = 65000;
```

If the struct is allocated on the stack, just use a dot as follows:

```
yahoo_manager.name = sam;
yahoo_manager.age = 45;
yahoo_manager.salary = 65000;
```

```
expr DOT expr
```

3.2.11 Using Structure Methods

Methods are accessed in the same way as fields: if the struct is allocated on the stack, use a dot, otherwise use a right arrow.

```
        struct square p;
        int area;
        p.height = 7;
        p.width = 9;
        area = p.get_area();
        p.set_height(55);
        p.set_width(3);
        area = p.get_area();
```

```
        struct square *p;
        int area;
        p = new struct square;
        p->height = 7;
```

```
        p->width = 9;
        area = p->get_area();
        p->set_height(55);
        p->set_width(3);
        area = p->get_area();
```

3.2.12 Arrays

An array is a data structure that allows for the storage of one or more elements of the same data type contiguously in memory. Each element is stored at an index, and array indexes begin at 0. This section will describe how to use Arrays.

3.2.13 Defining Arrays

An array is declared by specifying its data type, name, and size. The size must be positive. Here is an example of defining an integer array on the heap wth size 5:

```
arr = new int[5];
```

```
ID ASSIGN NEW prim_typ LBRACKET INT_LITERAL RBRACKET
```

You can also create arrays on the stack as follows:

```
int arr[10];
```

It is not required to initialize all of the elements. Elements that are not initialized will have a default value of zero.

3.2.14 Accessing Array Elements

To access an element in an array, use the array name followed by the element index surrounded by square brackets. Here is an example that assigns the value 1 to the first element (at index 0) in the array:

```
arr[0] = 1;
```

Accessing arrays is simply an expression:

```
expr LBRACKET INT_LITERAL RBRACKET
```

The syntax is the same for arrays allocated on the heap or stack. Also, JaTeste does not test for index out of bounds, so the following code would compile although it is incorrect; thus it is up to the programmer to make sure he or she does not write past the end of arrays.

```
arr = new int[2];
arr[5] = 1;
```

This will compile, but will of course will give unpredictable results.

3.3 Expressions and Operators

3.3.1 Expressions

An expression is a collection of one or more operands and zero or more operators that can be evaluated to produce a value. A function that returns a value can be an operand as part of an expression. Additionally, parenthesis can be used to group smaller expressions together as part of a larger expression. A semicolon terminates an expression. Some examples of expressions include:

```
35 - 6;
foo(42) * 10;
8 - (9 / (2 + 1) );
```

The grammar for expressions is:

```
expr:
          INT_LITERAL
        | STRING_LITERAL
        | CHAR_LITERAL
        | DOUBLE_LITERAL
        | TRUE
        | FALSE
        | ID
        | LPAREN expr RPAREN
        | expr PLUS expr
        | expr MINUS expr
        | expr STAR expr
        | expr DIVIDE expr
        | expr EQ  expr
        | expr EXPO  expr
        | expr MODULO  expr
        | expr NEQ  expr
        | expr LT expr
        | expr LEQ  expr
        | expr GT expr
        | expr GEQ expr
        | expr AND  expr
        | expr OR expr
        | NOT expr
        | AMPERSAND expr
        | expr ASSIGN expr
        | expr DOT expr
        | expr POINTER_ACCESS expr
        | STAR expr
        | expr LBRACKET INT_LITERAL RBRACKET
        | NEW prim_typ LBRACKET INT_LITERAL RBRACKET
        | NEW STRUCT ID
        | FREE LPAREN expr RPAREN
        | ID LPAREN actual_opts_list RPAREN
        | NULL LPAREN any_typ_not_void RPAREN
```

3.3.2 Assignment Operators

Assignment can be used to assign the value of an expression on the right side to a named variable on the left hand side of the equals operator. The left hand side can either be a named variable that has already been declared or a literal value:

```
int x;
int y;
x = 5;
y = x;
float y;
y = 9.9;
```

```
expr ASSIGN expr
```

All assignments are pass by value. Our language supports pointers and so pass by reference can be mimicked using addresses (explained below).

3.3.3 Arithmetic Operators

- `+` can be used for addition
- `-` can be used for subtraction (on two operands) and negation (on one operand)
- `*` can be used for multiplication
- `/` can be used for division
- ∧ can be used for exponents
- % can be used for modular division
- `&` can be used to get the address of an identifier

The grammar for the above operators, in order, is as follows:

```
        | expr PLUS expr
        | expr MINUS expr
        | expr TIMES expr
        | expr DIVIDE expr
        | expr EQ  expr
        | expr EXPO  expr
        | expr MODULO  expr
        | AMPERSAND expr
```

3.3.4 Comparison Operators

- `==` can be used to evaluate equality
- `!=` can be used to evaluate inequality
- `<` can be used to evaluate is the left less than the right
- `<=` can be used to evaluate is the left less than or equal to the right
- `>` can be used to evaluate is the left greater than the right
- `>=` can be used to evaluate is the left greater than or equal to the right

  The grammar for the above operators, in order, is as follows:

```
expr EQ    expr
expr NEQ   expr
expr LT    expr
expr LEQ   expr
expr GT    expr
expr GEQ   expr
```

3.3.5 Logical Operators

- `!` can be used to evaluate the negation of one expression
- && can be used to evaluate logical and
- || can be used to evaluate logical or

The grammar for the above operators, in order, is as follows:

```
NOT  expr
expr AND  expr
expr OR   expr
```

3.3.6 Operator Precedence

We adhere to standard operator precedence rules.

```
/*
   Precedence rules
*/
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left STAR DIVIDE MODULO
%right EXPO
%right NOT NEG AMPERSAND
%right RBRACKET
%left LBRACKET
%right DOT POINTER_ACCESS
```

3.3.7 Order of Evaluation

Order of evaluation is dependent on the operator. For example, assignment is right associative, while addition is left associative. Associativity is indicated in the table above.

3.4 Statements

Statements include: `if, while, for, return, assert`, as well all expressions, as explained in the following sections. That is, statements include all expressions, as well as snippets of code that are used solely for their side effects.

```
stmt:
            expr SEMI
          | LBRACE stmt_list RBRACE
          | RETURN SEMI
          | RETURN expr SEMI
          | IF LPAREN expr RPAREN stmt ELSE stmt
          | IF LPAREN expr RPAREN stmt %prec NOELSE
          | WHILE LPAREN expr RPAREN stmt
```

```
| FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
| ASSERT LPAREN expr RPAREN SEMI
```

3.4.1 If Statement

The if, else if, else construct will work as expected in other languages. Else clauses match with the closest corresponding if clause. Thus, their is no ambiguity when it comes to which if-else clauses match.

```
if (x == 42) {
  print("Gotcha");
}
else if (x > 42) {
  print("Sorry, too big");
}
else {
  print("I\'ll allow it");
}
```

The grammar that recognizes an if statement is as follows:

```
IF LPAREN expr RPAREN stmt ELSE stmt
| IF LPAREN expr RPAREN stmt %prec NOELSE
```

3.4.2 While Statement

The while statement will evaluate in a loop as long as the specified condition in the while statement is true.

```
/* Below code prints "Hey there" 10 times */
int x = 0;
while (x < 10) {
  print("Hey there");
  x = x + 1;
}
```

The grammar that recognizes a while statement is as follows:

```
WHILE LPAREN expr RPAREN stmt
```

3.4.3 For Statement

The for condition will also run in a loop so long as the condition specified in the for statement is true. The expectation for a for statement is as follows:

for (<initial state>; <test condition>; <step forward>)

Examples are as follows:

```
/*  This will run as long as i is less than 100
  i will be incremented on each iteration of the loop */
for (i = 0; i < 100; i = i + 1) {
  /* do something */
}
```

The grammar that recognizes a for statement is as follows:

```
FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN
```

Note, since all variables must be declared at the beginning of functions, you can't declared i inside the "initial state" part of the for loop.

3.4.4 Code Blocks

Blocks are code that is contained within a pair of brackets, { `code` }, that gets executed within a statement. For example, any code blocks that follow an `if` statement will get executed if the `if` condition is evaluated as true:

```
int x = 42;
if (x == 42) {
  /* the following three lines are executed */
  print("Hey");
  x = x + 1;
  print("Bye");
}
```

The grammar that recognizes a block of code is as follows:

```
LBRACE stmt RBRACE
```

Code blocks are used to define scope. Local variables are always given precedence over global variables.

3.4.5 Return Statement

The `return` statement is used to exit out of a function and return a value. The return value must be the same type that is specified by the function deceleration. Return can be used as follows:

```
/* The function trivially returns the input int value */
func int someValue(int x) {
  return x;
}
```

The grammar that recognizes a return statement is as follows:

```
RETURN SEMI
RETURN expr SEMI
```

Note that functions can be declared as returning void, and don't need to use the return statement at all subsequently. Also, there should not be any code after return statements as is usual convention.

3.4.6 Assert Statement

The assert statement is used only for test cases. Thus, using assert outside of a test case will throw an error. Asserts wrap all tests with a given test case as the following illustrates:

```
func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0) == 10);
          assert(add(5,1) == 6);
} using {
        int a;
        int b;
        a = 10;
        b = 5;
}
```

Asserts ultimately get compiled into if-else statements.

3.5 Functions

Functions allow you to group snippets of code together that can subsequently be called from other parts of your program. All functions are global. You don't declare functions before defining them. To use functions from other Jateste files, you need to include those files at the top of your program using "#include_jtlib <filename.jt>". If the file is your current directory, use quotations instead of carets.

3.5.1 Function Definitions

Function definitions contain the instructions to be performed when that function is called. The first part of the syntax is similar to how you define them in C, except the keyword "func" is additionally required. For example,

```
func int add(int x, int y) /* definition */
{
return x + y;
}
```

```
fdecl:
        FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE
```

A functions can accept any type of formal arguments, except for void. Thus, functions can accept pointers as arguments, enabling the programmer to mimic pass by reference functionality. Note, variables must be declared at the top of each function. For example, the following is not allowed:

```
func int do_something(int x, int y) /* definition */
{
    int c;
    c = x + y;
    int a;    /* This is illegal. a must be declared at the top of this function,
   above c = x + y; */
    return c;
}
```

The following is the correct implementation of the above example:

```
func int do_something(int x, int y) /* definition */
{
    int c;
    int a;
    c = x + y;
    return c;
}
```

3.5.2 Calling Functions

A function is called using the name of the function along with any parameters it requires. You *must* supply a function with the parameters it expects. For example, the following will not work:

```
func int main()
{
add(); /* this is wrong and will not compile because add expects two ints as
   parameters */
return 0;
}
func int add(int x, int y) /* definition */
{
```

```
return x + y;
}
```

Here's the grammar for a functional call:

```
        ID LPAREN actual_opts_list RPAREN { Call($1, $3)}
```

Note, calling functions is simply another expression. This means they are guaranteed to return a value (except for void functions) and so can be used as part of other expressions. Of course, a function's return type must be compatible with the context it's being used in. For example, a function that returns a char cannot be used as an actual parameter to a function that expects an int. Consider the following:

```
func int main()
{
int answer = subtract(add(10,10), 10); /* this is ok */
int answer2 = subtract(add_float(10.0,10.0), 10); /* this is NOT ok because subtract expects its first parameter to be an int while add_float returns a float */
return 0;
}

func int add_int(int x, int y) /* definition */
{
return x + y;
}

func float add_float(float x, float y)
{
  return x + y;
}

func int subtract(int x, int y)
{
  return x - y;
}
```

Structs can be defined with methods. The syntax for calling these functions is slightly different as the following illustrates:

```
func int main()
{

        struct house *my_house;
        int price;
        int vol;

        my_house = new struct house;

        my_house->set_price(100);
        my_house->set_height(88);
        my_house->set_width(60);
        my_house->set_length(348);

        price = my_house->get_price();
        vol = my_house->get_volumne();

        print(price);
        print(vol);
        return 0;
```

```
}

struct house {
        int price;
        int height;
        int width;
        int length;

        method void set_price(int x)
        {
                price = x;
        }

        method void set_height(int x)
        {
                height = x;
        }

        method void set_width(int x)
        {
                width = x;
        }
```

Thus, a variable of type “struct t” must be used with either “− >” (if the variable is stored on the heap) or “.” (if the variable is stored on the stack) to call the method associated with “struct t”.

3.5.3 Function Parameters

Formal parameters can be any data type including pointers, except “void”. Furthermore, they need not be of the same type. For example, the following is syntactically fine:

```
func void speak(int age, string name)
{
  print_string ("My name is" + name + " and I am "  + age);
}
```

```
formal_opts_list:
          /* nothing */
        | formal_opt

formal_opt:
             any_typ_not_void ID
           | formal_opt COMMA any_typ_not_void ID
```

While functions may be defined with multiple formal parameters, that number must be fixed. That is, functions cannot accept a variable number of arguments. As mentioned above, our language is pass by value. However, there is explicit support for passing pointers and addresses using * and &.

```
int* int_pt;
int a = 10;
int_pt  = &a;
```

3.5.4 Recursive Functions

Functions can be used recursively. Each recursive call results in the creation of a new stack frame and new set of local variables. It is up to the programmer to prevent infinite loops.

3.5.5 Main Function

Each Jateste program must have a main function that serves as the entry point for execution.

3.5.6 Function Test Cases

Functions can be appended with test cases directly in the source code. Most importantly, the test cases will be compiled into a separate (executable) file as previously explained. The keyword "with test" is used to define a test case as illustrated here:

```
func int add(int a, int b); /* declaration */

func int add(int x, int y) /* definition */
{
return x + y;
}
with test {
  assert(add(1,2) == 3);
  assert(add(-1, 1) == 0);
    assert(add(a, 2) == 4);
} using {
    int a;
    a = 2;
}
```

```
FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE testdecl

testdecl:
        WTEST LBRACE stmt_list RBRACE usingdecl
```

Test cases contain a sets of boolean expressions, wrapped in assert statements. Multiple boolean expressions can be defined, they just must be separated with semi-colons. As shown above, the programmer may define as many tests within a given test case as he or she wants. Snippets of code can also be used to set up a given test case's environment via the "using" keyword. That is, "using" is used to define code that is executed right before the test case is run. Consider the following:

```
func void changeAge(struct person *temp_person, int age)
{
*temp_person.age = age;
}
with test {
  assert(sam.age == 11);
}
using {
struct person sam;
sam.age = 10;
changeAge(&sam, 11);
}
```

"using" is used to create a struct and then call function changeAge; thus it is setting up the environment for it's corresponding test case. Variables defined in the "using" section of code can safely be referenced in the corresponding test case as shown. Basically, the code in the "using" section is executed right before the boolean expressions are evaluated and tested.
The "using" section is required, but can be left empty if desired

```

func int add(int x, int y) /* definition */
```

```
{
return x + y;
}
with test { /*  variables a, b defined below are NOT in this test case's scope*/
  add(10,2) == 12;
  add(-1, 1) == 0;
}
using {

}
```

Test cases are compiled into a separate program which can subsequently be run. The program will run all test cases and output appropriate information. Here's an example of what the test executable could output:

```
Tests:
addtest tests:
add(a,0) == 10 passed
add(a,b) == 15 passed
```

Of course, it's possible tests fail. Consider the following source code:

```
func int add(int x, int y)
{
        return x;
} with test {
        assert(add(a,1) == 11);
        assert(add(a,b) == 15);
} using {
        int a;
        int b;
        a = 10;
        b = 5;
}
```

The add function implementation is clearly wrong (it returns x, instead of x + y). After compiling and running the test executable we get:

```
Tests:
addtest tests:
add(a,1) == 11 failed
add(a,b) == 15 failed
```

4 Project Plan

4.1 Team Roles

From the onset of the project, we assigned roles among the team as was recommended. Andy came up with the idea for the language, so it seemed natural that he would be the Language Guru. All of us had input on the design of the language but we always consulted with Andy to ensure continuity with his vision for the project. Jake helped form the team, had good organization skills, and was on top of things from the start, so it seemed like he would be a good fit as the team Manager. Jake worked throughout the term to make sure that team meetings took place and deadlines were met. Jared had extensive experience with group projects and version control software, so he fell nicely into the role of System Architect. Jared drew up a work flow, based on pull requests, for our group to adhere to in order to ensure things went smoothly. Jemma had significant prior experience with testing and agreed to take the lead as the Tester for the team. Jemma worked to ensure that tests were created alongside of feature implementation to ensure that code was fully tested. As the project progressed, roles became more fluid as work was required in varying areas and everyone pitched in where things needed to get done. However, final say in any given area always remained with the assigned team member for that role.

4.2 Planning and Development

As a team, we made a commitment to meet weekly with David to make sure we were on the right track and to help answer any question we had about how to move forward. On weeks that we did not meet with David, we were conscious to meet as a team to discuss our progress over that week. Each week we identified tasks that needed to get done and assigned work for the week. We also utilized team meeting time to do research when necessary, and implement some feature together to make sure everyone was on the same page. We communicated throughout the week on our progress when it affected the work of another team member. Additionally, for tasks that could be picked up and implemented by anyone when they had a chance, we used a system of creating "issues" on GitHub that described portions of work that needed to get done. We also made some "milestones" on GitHub to motivate each other to get large segments of work done.

4.3 Testing Procedure

Throughout the writing of our compiler, we wrote tests to verify the functionality we were implementing. This served the twofold purpose of ensuring that we were generating the proper code output when we implemented new functionality, and also that we didn't break previously functioning parts of our compiler. Tests were written as canned recipes in a separate Makefile specifically written for our *tests* folder. These recipes compiled example programs from (.jt) source and checked the output of executing the compiled .ll code against a precomputed output that we paired with each source file. For compilation errors, we had a separate canned recipe to verify that the JaTeste compiler failed to finish compiling the bad source files.

4.3.1 Continuous Integration

As our testing suite became more complex, we decided to implement a continuous-integration build using Travis-CI that ensure that all pull requests to our master branch passed all existing tests before they could be merged. This helped reduce the need for a reviewer to actually download and compile all updates to make sure that no tests broke, which in turn increased the productivity of our team. In addition, all pushes to our master branch are built and tested in order to ensure that our master branch is always working.

Implementing continuous integration came with it's own challenges, as Travis-CI uses a containerized work flow to provide virtual testing environments with very little boot time. In order to run our build, we needed to find a way to install various dependencies, including OCaml and LLVM which were rather tricky to install on a Linux 12.04 Docker container. Once we were able to install all dependencies however, the continuous-integration system made testing our code a much simpler procedure.

Figure 1: Example commit list showing continuous integration build status next to commit SHA

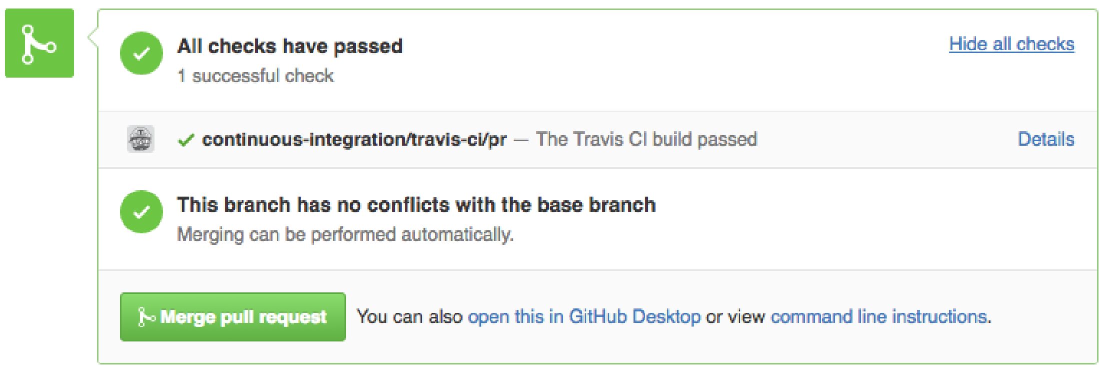


Figure 2: Merging on Github with automatic continuous integration checks

4.4 Programming Style Guide

4.4.1 Comments

Comments used are to be associated with the code directly below the comment. Multi-line comments are allowed when necessary but discouraged. Keep comments concise and to one line when possible.

4.4.2 Naming Conventions

When possible, use names that are meaningful and relate to the use of the code. Function names are to be all lower case with underscores to separate words as_such. Types are to be started with a capital and the rest of the deceleration will be lower case, with underscores to separate words As_such. Variable names are to be all lower case with underscores separating words the same way functions are.

4.4.3 Indentation

Indent using tabs and set tabbing to 4 spaces for consistency. A new block of code should start on a new, indented line. A very long line can be broken into two lines, and the second line should be indented.

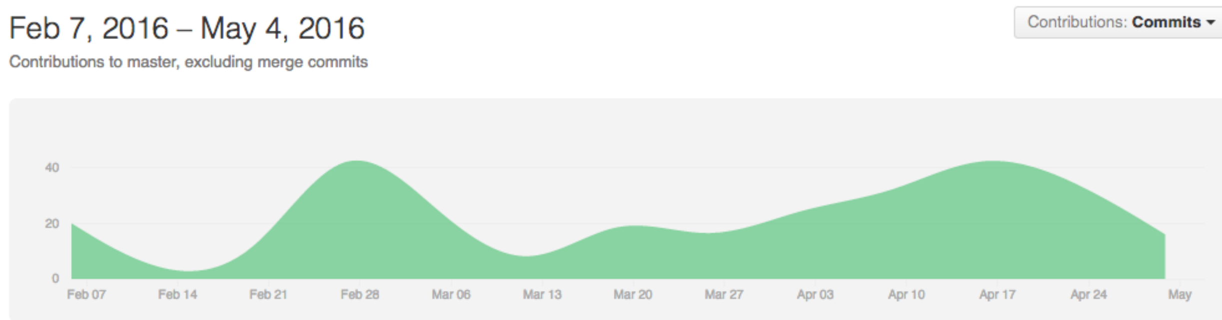
4.4.4 Parenthesis

Use parenthesis for chunks of code when necessary but avoid unnecessary parenthesis that clutters up the code.

4.5 Project Timeline

Date	Goal
1/29/16	Set group meeting, TA meeting, Come up with idea
2/5/16	Finish language proposal
2/12/16	Hash out specs of language, start LRM
2/19/16	Build scanner for the language
2/26/16	Build parser, finish LRM
3/4/16	Start working on AST
3/11/16	Spring Break
3/18/16	Continue work on AST, discuss code gen plan
3/25/16	Get up to speed on LLVM, work on AST
4/1/16	Finish AST, start SAST, code gen for ”Hello, World”
4/8/16	Work on SAST, code gen, incremental testing
4/15/16	Implement code gen to two files, one for testing
4/22/16	Continue code gen / testing, automatic continuous integration
4/29/16	Finish automatic continuous integration, clean up code
5/6/16	Work on final report and presentation

4.6 GitHub Progression



As you can see from our chart, we were slow to start as we had to hash out the details of our language and did not involve a ton of code. The first major bump is at the time of the LRM deadline as a lot of code was written leading up to that deadline to get everything up and running. From that point on, we worked at a slow and steady pace, through the "Hello, World" deadline, and leading into the final deadline.

4.7 Software Development Environment

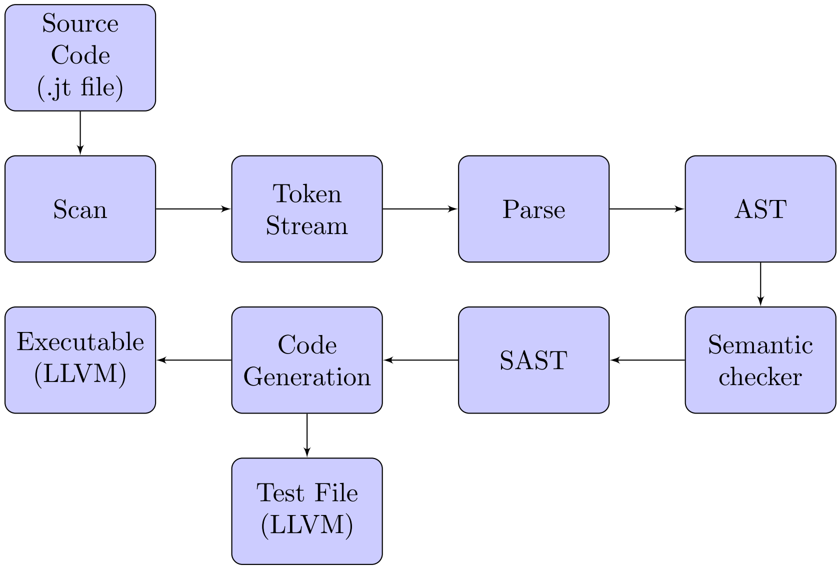
We used Git and a private GitHub repository for version control. Each team-member had their own private fork of the main repository for doing their own development. This allowed us to maintain a central master branch that was always working and passing tests. In order to merge into the master branch, we devised a work flow early based on reviewing pull requests from feature branches on each team-member's fork. Each pull request was reviewed by another team-member, and later on in our development, we even added continuous integration to our build using Travis-CI to ensure that all tests were passing. We have since open-sourced this repository under the MIT license and it is available at `https://github.com/jaredweiss/JaTeste`

All of our compiler was written in OCaml, compiling .jt source code to LLVM. This was made easy by the fact that we were provided a VirtualBox image with OCaml and LLVM pre-installed (installing these dependencies on our Travis-CI builds was actually a fairly difficult task). For our CI builds, we installed OCaml (and ocamlfind via opam), make, and LLVM 3.8 as dependencies.

All submissions and reports were written in LaTeX.

5 Architecture Design

5.1 Block Diagram



5.2 The Compiler

The entry point of the compiler for a given source.jt file is jateste.ml. This is where the various phases of the compilation process are coordinated. At a high level, the compiler reads characters from source.jt, builds up an AST in the parser, performs a walk of the AST to create the SAST, passes the SAST on to codegen.ml, which finally generates the LLVM code.

As described in the introduction section the compiler is capable of producing two executables:

1. regular executable: source.ll
2. test executable: source-test.ll

Both can be run using the LLVM interpreter "lli".

jateste.ml is also where include files are handled. More specifically, if a given source file wants to include an external .jt file, jateste.ml is where the given file is searched for.

5.3 The Scanner

The scanner reads characters from source.jt according to the regular expressions in scanner.ml and outputs a stream of tokens to parser.mly. The regular expressions for each token are in scanner.ml.

5.4 The Parser

The parser receives tokens from the scanner and creates an AST from the given context free grammar. The CFG is defined in parser.mly. At a high level, the AST is made up of a 4-element record:

```
type program = header list * bind list * func_decl list * struct_decl list
```

As illustrated, the AST consists of a list of header files, a list of global variables, a list of function definitions, and a list of struct definitions. If the parser is not able to build up an AST, it will throw a parsing error.

5.5 The Semantic Checker

The semantic checker receives the AST from the parser, walks the tree, and creates an SAST. The SAST carries additional information that helps the codegen phase of the compiler. For example, each array access is represented by a node in the AST; the SAST adds the array type information to such a node, which the AST does not.

An important part of the semantic checker is converting test cases into functions. More specifically, after checking the test case for a given function is semantically valid, semant.ml turns the test cases into standalone functions, where the using clause is copied and pasted to the top of the new function. Codegen is subsequently responsible for turning the new test case functions into standalone snippets of code.

If the semantic checker finds an error, it will immediately abort and print a relevant error message to the console.

5.6 The Code Generator

codegen.ml takes an SAST as input and creates LLVM code. We take advantage of OCaml's built in support for LLVM to help build the assembly code.

One of the most important jobs of the Code Generator is to create the test file. If instructed to, codegen.ml creates code for the test functions that were constructed as nodes in the SAST in the semantic checking phase. Importantly, codegen.ml ignores the user-defined main function, and calls the test functions from a brand new main. For example, consider the following snippet of code:

```

func int main()
{
    Do_insightful_stuff;
    return 0;
}

func int add(int x, int y)
{
        return x + y;
} with test {
        assert(add(a,0) == 10);
} using {
        int a;
        a = 10;
}
```

codegen.ml would compile this into the following pseudo-code test file:

```

func int main()
{
    printResultOf: addtest();
    return 0;
}

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

func void addtest()
{
        int a;
        a = 10;
        assert(add(a,0) == 10);
```

```
}
```

For the regular file, codegen.ml would compile the snippet of code into something like the following pseudo code:

```

func int main()
{
     Do_insightful_stuff;
    return 0;
}

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

5.7 Supplementary Code

There is a Jateste standard library located in the lib folder. To include other jateste files in a given source code file, source.jt, the programmer has two options. If the file to include is in the current directory, the following syntax is used to include a file called file.jt:

```
#include_jtlib "file.jt"
```

If the file to include is in the standard library, use:

```
#include_jtlib <file.jt>
```

6 Test Plan

6.1 Test Suite Log

We wrote tests for every feature in the compiler. There are several small tests that we used to test individual elements such as structs, function calls, loops, ethnicc. We included tests that were expected to pass, as well as tests that were expected to fail

Test Suite Log:
========= Running All Tests! ==========
make[1]: Entering directory ’/home/plt/JaTeste/test’
Testing ’global-scope.jt’
—-> Test passed!
Testing ’global-scope.jt’
—-> Test passed!
Testing ’test-func1.jt’
—-> Test passed!
Testing ’test-func2.jt’
—-> Test passed!
Testing ’test-func3.jt’
—-> Test passed!
Testing ’test-pointer1.jt’
—-> Test passed!
Testing ’test-while1.jt’
—-> Test passed!
Testing ’test-for1.jt’
—-> Test passed!
Testing ’test-malloc1.jt’
—-> Test passed!
Testing ’test-free1.jt’
—-> Test passed!
Testing ’test-testcase1.jt’
—-> Test passed!
Testing ’test-testcase1.jt’
—-> Test passed!
Testing ’test-testcase2.jt’
—-> Test passed!
Testing ’test-testcase2.jt’
—-> Test passed!
Testing ’test-testcase3.jt’
—-> Test passed!
Testing ’test-testcase3.jt’
—-> Test passed!
Testing ’test-array1.jt’
—-> Test passed!
Testing ’test-lib1.jt’
—-> Test passed!
Testing ’test-gcd1.jt’
—-> Test passed!
Testing ’test-struct-access1.jt’
—-> Test passed!
Testing ’test-bool1.jt’
—-> Test passed!
Testing ’test-bool2.jt’

—-> Test passed!
Testing 'test-bool3.jt'
—-> Test passed!
Testing 'test-arraypt1.jt'
—-> Test passed!
Testing 'test-linkedlist1.jt'
—-> Test passed!
Testing 'test-linkedlist2.jt'
—-> Test passed!
Testing 'test-linkedlist-delete1.jt'
—-> Test passed!
Testing 'test-linkedlist-free1.jt'
—-> Test passed!
Testing 'test-class1.jt'
—-> Test passed!
Testing 'test-class2.jt'
—-> Test passed!
Testing 'test-class3.jt'
—-> Test passed!
Testing 'test-testcase4.jt'
—-> Test passed!
Testing 'test-testcase4.jt'
—-> Test passed!
Testing 'test-struct-malloc1.jt'
—-> Test passed!
Testing 'test-negative1.jt'
—-> Test passed!
Testing 'test-double1.jt'
—-> Test passed!
Testing 'test-double2.jt'
—-> Test passed!
Testing 'test-mod1.jt'
—-> Test passed!
Testing 'test-nested-loop1.jt'
—-> Test passed!
========= Runtime Tests Passed! ==========
Testing 'local-var-fail.jt', should fail to compile...
—-> Test passed!
Testing 'no-main-fail.jt', should fail to compile...
—-> Test passed!
Testing 'return-fail1.jt', should fail to compile...
—-> Test passed!
Testing 'return-fail2.jt', should fail to compile...
—-> Test passed!
Testing 'return-fail3.jt', should fail to compile...
—-> Test passed!
Testing 'return-fail4.jt', should fail to compile...
—-> Test passed!
Testing 'struct-access-fail1.jt', should fail to compile...
—-> Test passed!
Testing 'invalid-assignment-fail1.jt', should fail to compile...
—-> Test passed!
Testing 'class1-var-fail1.jt', should fail to compile...
—-> Test passed!

Testing ’class2-method-args-fail.jt’, should fail to compile...
—-> Test passed!
Testing ’class-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’class-fail2.jt’, should fail to compile...
—-> Test passed!
Testing ’header-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’add-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’struct-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’struct-fail2.jt’, should fail to compile...
—-> Test passed!
Testing ’dereference-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’method-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’var-fail1.jt’, should fail to compile...
—-> Test passed!
Testing ’var-fail2.jt’, should fail to compile...
—-> Test passed!
======= Compilation Tests Passed! ========
=========== All Tests Passed! ============
make[1]: Leaving directory ’/home/plt/JaTeste/test’

6.2 Test Automation

We had 126 tests in our test suite. In order to run all of the tests and see if they pass, run **make all** or **make test** in the src directory. This diffs the outputs of the tests with the files that we created that include expected outputs. If there are differences, it marks the test as a failure, otherwise it prints "Test passed!" as can be seen in the Test Suite Log

6.3 Tests

add-fail1.jt

```
func int main()
{
  int a;
  string s;

  a = 10;
  s =  "cool";

  a = a + s;

  return 0;
}
```

add-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidExpr("Illegal binary op")
```

class-fail1.jt

```
func int main()
{

  struct house *my_house;
  int price;
  int vol;

  my_house = new struct house;

  my_house->set_price(100);
  my_house->set_height(88);
  my_house->set_width(60);
  my_house->set_length(348);

  price = my_house->get_price();
  vol = my_house->get_volumne(10);

  print(price);
  print(vol);
  return 0;
}

func void update_price(struct house *h, int a)
{
  h->set_price(a);
} with test {
  assert(my_house->price == 100);
} using {
  struct house *my_house;
  my_house = new struct house;
  update_price(my_house, 100);
}

struct house {
  int price;
  int height;
  int width;
  int length;

  method void set_price(int x)
  {
    price = x;
  }

  method void set_height(int x)
  {
    height = x;
  }

  method void set_width(int x)
  {
    width = x;
  }

  method void set_length(int x)
  {
    length = x;
```

```
  }

  method int get_price()
  {
    return price;
  }

  method int get_volumne()
  {
    int temp;
    temp = height * width * length;
    return temp;
  }


};
```

class-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidArgumentsToFunction("houseget_volumne is
    supplied with wrong args")
```

class-fail2.jt

```
func int main()
{

  struct house *my_house;
  struct condo *my_condo;
  int a;
  int b;
  int c;

  my_house = new struct house;
  my_condo = new struct condo;

  my_house->set_price(100);
  my_condo->set_price(59);

  a = my_house->get_price();
  b = my_condo->geat_price();

  c = a - b;

  print(c);



  return 0;
}


struct house {
  int price;

  method void set_price(int x)
  {
    price = x;
  }

  method int get_price()
  {
    return price;
  }


};

struct condo {
  int price;

  method void set_price(int x)
  {
    price = x;
  }

  method int get_price()
  {
    return price;
  }
```

```

};
```

class-fail2.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidStructField
```

class1-var-fail1.jt

```
func int main()
{

  struct house *my_house;
  int price;
  int vol;

  my_house->set_price(100);
  my_house->set_height(88);
  my_house->set_width(60);
  my_house->set_length(348);


  return 0;
}

struct house {
  int price;
  int height;
  int width;
  int length;

  method void set_price(int x)
  {
    pricee = x;
  }

  method void set_height(int x)
  {
    height = x;
  }

  method void set_width(int x)
  {
    width = x;
  }

  method void set_length(int x)
  {
    length = x;
  }

  method int get_price()
  {
    return price;
  }

  method int get_volumne()
  {
    int temp;
    temp = height * width * length;
    return temp;
  }


};
```

class1-var-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.UndeclaredVariable("pricee")
```

class2-method-args-fail.jt

```
func int main()
{
  struct circle *my_circle;
  int diameter;
  int i;
  my_circle = new struct circle;
  my_circle->set_radius(10);

  diameter = my_circle->get_diameter();
  print(diameter);

  my_circle->set_radius(10,1);

  return 0;
}

struct circle {
  int radius;
  int diameter;

  method void set_radius(int c)
  {
    radius = c;
    diameter = radius * 2;
  }

  method int get_radius()
  {
    return radius;
  }

  method int get_diameter()
  {
    return diameter;
  }


};
```

class2-method-args-fail.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidArgumentsToFunction("circleset_radius is
    supplied with wrong args")
```

dereference-fail1.jt

```
func int main()
{
  int a;
  *a = 10;
  return 0;
}
```

dereference-fail1.out

```

Scanned
Parsed
Fatal error: exception Exceptions.InvalidDereference
```

global-scope.jt

```
int global_var;

func int main()
{
  int temp;
  global_var = 10;
  temp = 20;
  my_print();
  return 0;
}

func void my_print()
{
  int temp;
  if (global_var == 10) {
    print("passed");
  } else {
    print("failed");
  }

  if (temp == 20) {
    print("failed");
  } else {
    print("passed");
  }

}
```

global-scope.out

```
passed
passed
```

header-fail1.jt

```
#include_jtlib "nvlkj"
func int main()
{
  return 0;
}
```

header-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidHeaderFile("./nvlkj")
```

hello-world.jt

```
func int main()
{
  print("hello world!");

  return 0;
}
```

hello-world.out

```
hello world!
```

invalid-assignment-fail1.jt

```
func int main()
{
  int a;
  char b;
  a = b;
}
```

invalid-assignment-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.IllegalAssignment
```

local-var-fail.jt

```
func int main()
{
  int main_var;
  main_var = 10;
  return 0;
}
func void do_something_sick()
{
  int my_var;
  main_var;
}
```

local-var-fail.out

```
Scanned
Parsed
Fatal error: exception Exceptions.UndeclaredVariable("main_var")
```

method-fail1.jt

```

func int main()
{
  struct car *my_car;

  my_car->0;

  return 0;
}


struct car {
  int price;
  int year;
  string model;

  method void set_model(string s)
  {
    model = s;
  }
};
```

method-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.MissingMainFunction
```

no-main-fail.jt

```
func int my_main()
{
  return 0;
}
```

no-main-fail.out

```
Scanned
Parsed
Fatal error: exception Exceptions.MissingMainFunction
```

pointer-fail1.jt

```
func int main()
{
  struct house my_house;
  int a;
  a = my_house->price;

  return 0;
}

struct house {
  int price;
  int zipcode;
};
```

pointer-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidPointerAccess
```

pointer-fail2.jt

```
func int main()
{
  void *p;

  return 0;
}
```

pointer-fail2.out

```

Scanned
Fatal error: exception Parsing.Parse_error
```

return-fail1.jt

```
func int main()
{
  int a;
  int b;
  int c;
  int d;

  a = 1;
  b = 2;
  c = 3;

  d = do_something(a,b,c);

  return 0;
  d = 10;
}

func int do_something(int x, int y, int z)
{
  return x + y + z;
}
```

return-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidReturnType("Can't have any code after
    return statement")
```

return-fail2.jt

```
func int main()
{

  struct house *my_house;
  struct condo *my_condo;
  int a;
  int b;
  int c;

  my_house = new struct house;
  my_condo = new struct condo;

  my_house->set_price(100);
  my_condo->set_price(59);

  a = my_house->get_price();
  b = my_condo->get_price();

  c = a - b;

  print(c);

  return 0;
}


struct house {
  int price;
  char c;

  method void set_price(int x)
  {
    price = x;
  }

  method int get_price()
  {
    return c;
  }


};

struct condo {
  int price;

  method void set_price(int x)
  {
    price = x;
    return 0;
  }

  method int get_price()
  {
    return price;
  }

```

```
};
```

return-fail2.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidReturnType("return type doesnt match with
    function definition")
```

return-fail3.jt

```
func int main()
{

  string s;
  s = add(1,1);

  return 0;
}

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

return-fail3.out

```
Scanned
Parsed
Fatal error: exception Exceptions.IllegalAssignment
```

return-fail4.jt

```
func int main()
{

  return 0;
}

func int do_something(int a, int b, int c, int d)
{
  int i;
  return a + b + c + d;
  i = i + 1;
}
```

return-fail4.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidReturnType("Can't have any code after
    return statement")
```

struct-access-fail1.jt

```
func int main()
{
  struct car *toyota;

  toyota = new struct car;

  toyota->priice;

  return 0;
}

struct car {
  int price;
  int year;
  int weight;
};
```

struct-access-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.InvalidStructField
```

struct-fail1.jt

```
func int main()
{


}

struct ahouse {
  int price;

  method int get_price()
  {
    return price;
  }

  int zipcode;

};
```

struct-fail1.out

```

Scanned
Fatal error: exception Parsing.Parse_error
```

struct-fail2.jt

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

struct garden {
  int trees;
  int plants;

  func int set_trees(int a)
  {
    tree = a;
  }

};
```

struct-fail2.out

```
Scanned
Fatal error: exception Parsing.Parse_error
```

test-array1.jt

```
func int main()
{
  int[10] arr;
  int a;
  int b;

  a = 10;

  arr[2] = 10;

  b = arr[2];

  if (b == 10) {
    print("passed");
  }

  return 0;
}
```

test-array1.out

```
passed
```

test-arraypt1.jt

```
func int main()
{
  int[10] *arr;
  int a;
  int b;
  int c;

  arr = new int[10];

  arr[8] = 9;
  arr[3] = 7;

  c = arr[3];
  b = arr[8];

  if (c == 7) {
    print("passed");
    if (b == 9) {
      print("passed");
    }
  }

  return 0;
}
```

test-arraypt1.out

```
passed
passed
```

test-bool1.jt

```
func int main()
{
  bool my_bool;
  bool my_bool2;

  my_bool = true;
  my_bool2 = false;

  if (my_bool || my_bool2) {
    print("or passed");
  }

  if (my_bool && my_bool2) {
  } else {
    print("and passed");
  }

  return 0;
}
```

test-bool1.out

```
or passed
and passed
```

test-bool2.jt

```
func int main()
{
  bool my_bool;

  my_bool = false;

  if (!my_bool) {
    print("passed");
  }

  return 0;
}
```

test-bool2.out

```
passed
```

test-bool3.jt

```
func int main()
{
  int a;
  int b;
  int c;
  int d;

  double d1;
  double d2;
  double d3;
  double d4;

  bool my_bool;

  a = 10;
  b = 11;
  c = 10;
  d = 20;

  d1 = 19.18;
  d2 = 0.7;
  d3 = 0.7;
  d4 = 19.19;

  my_bool = true;

  if ((a == c) && (d1 < d4)) {
    print("passed");
  }

  if ((a < c) || (d2 == d3)) {
    print("passed");
  }

  if ((d1 != d2) && (a <= c) && (d1 < d4) && (my_bool == true)) {
    print("passed");
  }

  b = 10;
  d = 10;

  if (((a != b) && (c == d)) || (d2 == d3)) {
    print("passed");
  }

  return 0;
```

test-bool3.out

```
passed
passed
passed
passed
```

test-class1.jt

```
func int main()
{

  struct square *p;
  int area;
  p = new struct square;
  p->height = 7;
  p->width = 9;
  area = p->get_area();
  print(area);
  p->set_height(55);
  p->set_width(3);
  area = p->get_area();
  print(area);


  return 0;
}


struct square {
  int height;
  int width;

  method int get_area()
  {
    int temp_area;
    temp_area = height * width;
    return temp_area;
  }

  method void set_height(int h) {
    height = h;
  }

  method void set_width(int w) {
    width = w;
  }

};
```

test-class1.out

```
63
165
```

test-class2.jt

```
func int main()
{

  struct house *my_house;
  int price;
  int vol;

  my_house->set_price(100);
  my_house->set_height(88);
  my_house->set_width(60);
  my_house->set_length(348);

  price = my_house->get_price();
  vol = my_house->get_volumne();

  print(price);
  print(vol);
  return 0;
}

struct house {
  int price;
  int height;
  int width;
  int length;

  method void set_price(int x)
  {
    price = x;
  }

  method void set_height(int x)
  {
    height = x;
  }

  method void set_width(int x)
  {
    width = x;
  }

  method void set_length(int x)
  {
    length = x;
  }

  method int get_price()
  {
    return price;
  }

  method int get_volumne()
  {
    int temp;
    temp = height * width * length;
    return temp;
  }
```

```


};
```

test-class2.out

```
100
1837440
```

test-class3.jt

```
func int main()
{

  struct house *my_house;
  struct condo *my_condo;
  int a;
  int b;
  int c;

  my_house = new struct house;
  my_condo = new struct condo;

  my_house->set_price(100);
  my_condo->set_price(59);

  a = my_house->get_price();
  b = my_condo->get_price();

  c = a - b;

  print(c);



  return 0;
}


struct house {
  int price;

  method void set_price(int x)
  {
    price = x;
  }

  method int get_price()
  {
    return price;
  }


};

struct condo {
  int price;

  method void set_price(int x)
  {
    price = x;
  }

  method int get_price()
  {
    return price;
  }

```

```
};
```

test-class3.out

```
41
```

test-double1.jt

```

func int main()
{
  double d1;
  double d2;
  double d3;

  d1 = 10.1;
  d2 = 7.33;
  d3 = d1 + d2;

  if (d1 == 10.1 ) {
    print("passed");
  }

  if (d3 == 17.43) {
    print("passed");
  } else {
    print("failed");
  }

  d1 = 7.33;

  if (d1 == d2) {
    print(d1);
  }

  return 0;
}
```

test-double1.out

```
passed
passed
7.330000
```

test-double2.jt

```
func int main()
{
  double d1;
  double d2;
  double d3;

  d1 = -10.1;
  d2 = 7.89;
  d3 = d1 + d2;

  if (d1 == -10.1 ) {
    print("passed");
  }

  if (d3 == -2.21) {
    print("passed");
  } else {
    print("failed");
  }

  d1 = -9.14;
  d2 = -9.14;

  if (d1 == d2) {
    print(d1);
  }

  return 0;
}
```

test-double2.out

```
passed
passed
-9.140000
```

test-for1.jt

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

test-for1.out

```
0
1
2
3
4
```

test-free1.jt

```
func int main()
{
  struct person *sam;

  sam = new struct person;

  sam->age = 100;
  sam->height = 100;
  sam->gender = 100;

  free(sam);

  print("freed");


  return 0;
}

struct person {
  int age;
  int height;
  int gender;
};
```

test-free1.out

```
freed
```

test-func1.jt

```
func int main()
{
  int sum;
  sum = add(10,10);
  if (sum == 20) {
    print("passed");
  } else {
    print("failed");
  }
  return 0;
}

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

test-func1.out

```
passed
```

test-func2.jt

```
int global_var;

func int main()
{
  global_var = 0;
  add_to_global();
  if (global_var == 1) {
    print("passed");
  } else {
    print("failed");
  }

}

func void add_to_global()
{
  global_var = global_var + 1;
}
```

test-func2.out

```
passed
```

test-func3.jt

```
func int main()
{
  int a;
  struct person *sam;
  sam = new struct person;
  update_age(sam);

  a = sam->age;

  if (a == 10) {
    print("passed");
  }

  return 0;
}

func void update_age(struct person *p)
{
  p->age = 10;
}

struct person {
  int age;
  int height;
};
```

test-func3.out

```
passed
```

test-gcd1.jt

```
func int main()
{
  int a;
  int b;
  int c;

  c = gcd(15,27);

  if (c == 3) {
    print("passed");
  }

  return 0;

}

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

test-gcd1.out

```
passed
```

test-lib1.jt

```

#include_jtlib <math.jt>

func int main()
{
  int a;
  int b;
  int c;
  a = 10;
  b = 3;

  c = add(a,b);
  if (c == 13) {
    print("passed");
  }
}
```

test-lib1.out

```
passed
```

test-linkedlist-delete1.jt

```

#include_jtlib <int_list.jt>

func int main()
{

  struct int_list *header;
  header = int_list_initialize();
  int_list_insert(header, 0);
  int_list_insert(header, 9);
  int_list_insert(header, 9);
  int_list_insert(header, 13);
  int_list_insert(header, 19);
  int_list_delete(header, 13);
  int_list_insert(header, 8);
  int_list_delete(header, 19);

  int_list_print(header);


  return 0;
}
```

test-linkedlist-delete1.out

```
0
9
9
8
```

test-linkedlist-free1.jt

```
#include_jtlib <int_list.jt>

func int main()
{

  struct int_list *header;
  int len;

  header = int_list_initialize();

  int_list_insert(header,5);
  int_list_insert(header,9);
  int_list_insert(header,1);
  int_list_insert(header,18);
  int_list_insert(header,4738);
  int_list_insert(header,17);
  int_list_insert(header,5);

  len = int_list_length(header);
  print(len);
  int_list_free_list(header);
  len = int_list_length(header);
  print(len);

  return 0;
}
```

test-linkedlist-free1.out

```
7
0
```

test-linkedlist1.jt

```
#include_jtlib <int_list.jt>

func int main()
{

  struct int_list *my_list;
  my_list = int_list_initialize();
  int_list_insert(my_list,9);
  int_list_insert(my_list,5);
  int_list_insert(my_list,8);
  int_list_insert(my_list,10);
  int_list_insert(my_list,40);
  int_list_insert(my_list,11);
  int_list_insert(my_list,0);
  int_list_insert(my_list,9);
  int_list_insert(my_list,478);
  int_list_print(my_list);

  return 0;
}
```

test-linkedlist1.out

```
9
5
8
10
40
11
0
9
478
```

test-linkedlist2.jt

```
#include_jtlib <int_list.jt>

func int main()
{
  struct int_list *header;
  header = int_list_initialize();
  int_list_insert(header,2);
  int_list_insert(header,2);
  int_list_insert(header,3);
  int_list_insert(header,9);
  int_list_insert(header,100);
  int_list_insert(header,61);

  if (int_list_contains(header,100) == true) {
    print("passed contains test");
  }

  return 0;
}
```

test-linkedlist2.out

```
passed contains test
```

test-malloc1.jt

```
func int main()
{

  struct person *andy;
  int *a;
  int b;
  int zipcode;

  andy = new struct person;

  b = 25;

  a = &b;

  andy->age = *a;
  andy->height = 100;
  andy->zipcode = 10027;


  zipcode = andy->zipcode;

  if (zipcode == 10027) {
    print("passed");
  }

  *a = andy->age;

  if (*a == 25) {
    print("word up");
  }

  return 0;

}


struct person {
  int age;
  int zipcode;
  int height;
};
```

test-malloc1.out

```
passed
word up
```

test-mod1.jt

```
func int main()
{
  int a;
  int b;
  int c;
  int d;
  int e;
  int mod;


  a = 15;
  b = 7;
  c = 23;
  d = 5;
  e = 100;

  mod = a % b;
  print(mod);
  mod = c % d;
  print(mod);
  mod = e % 10;
  print(mod);
  mod = d % b;
  print(mod);
  mod = b % d;
  print(mod);

  return 0;
}
```

test-mod1.out

```
1
3
0
5
2
```

test-negative1.jt

```
func int main()
{
  int a;
  int b;
  int c;
  int d;
  int e;
  int sum;

  a = -23;
  b = 15;
  c = -3;
  d = -9;
  e = 8;

  sum = a + b + c + d + e;

  print(sum);

  return 0;
}
```

test-negative1.out

```
-12
```

test-pointer1.jt

```
func int main()
{
  int a;
  int b;
  int *c;


  a = 10;
  b = 500;

  c = &b;

  if (*c == 500) {
    print("passed");
  } else {
    print("failed");
  }

  return 0;
}
```

test-pointer1.out

```
passed
```

test-struct-access1.jt

```
func int main()
{
  struct house my_house;
  int a;
  int b;
  int c;

  a = 99;
  my_house.price = a;
  c = my_house.price;
  my_house.age = 10;
  b = my_house.age;

  print(c);
  print(b);

  return 0;
}

struct house {
  int price;
  int age;
};
```

test-struct-access1.out

```
99
10
```

test-struct-malloc1.jt

```
func int main()
{

  struct rectangle *my_rec;
  struct house *my_house;
  struct house my_house2;
  int a;
  int i;
  char my_char;
  char my_char2;
  my_rec = new struct rectangle;
  my_house = new struct house;

  update_width(my_rec, 19);

  my_house2.set_a('r');

  my_char2 = my_house2.a;

  if (my_char2 == 'r') {
    print("is r");
  }

  a = my_rec->width;
  print(a);
  i = 0;
  while (i < 10) {
    update_width(my_rec, i);
    a = my_rec->width;
    print(a);
    update_height(my_rec, (i+5));
    a = my_rec->height;
    print(a);
    i = i + 1;
  }

    update_num(&a);
  print(a);

  if (a <= 9) {
    print("noo");
  } else if (a >= 11) {
    print("nooo");
  } else {
    print("coool");
  }

  update_house_a(my_house);

  if (my_house->a == 'y') {
    print("nice");
  } else {
    print("not nice");
  }


  my_house2.a = 'e';
```

```

  my_char = my_house2.a;

  if (my_house2.a != 'f') {
    print("hey");
  }

  free(my_rec);
  free(my_house);

  return 0;
}

func void update_num(int *i)
{
  *i = 10;
}

func void update_house_a(struct house *h)
{
  h->a = 'y';
}

func void update_width(struct rectangle *r, int w)
{

  r->set_width(w);
}

func void update_height(struct rectangle *r, int w)
{

  r->set_height(w);
} with test {
  assert(my_square->height == d);
} using {
  int a;
  int b;
  int c;
  int d;
  int e;
  int f;
  int g;
  struct rectangle *my_square;

  my_square = new struct rectangle;

  d = 10;
  update_height(my_square, d);

  while ( a < 10 ) {
    update_height(my_square, a);
    a = a + 1;
  }

}
struct rectangle {

  int width;
```

```
  int height;

  method void set_height(int x)
  {
    height = x;
  }

  method void set_width(int x)
  {
    width = x;
  }

  method int get_area()
  {
    int a;
    int b;
    int c;
    a = width;
    b = height;
    c = a * b;
    return c;
  }
};

struct house {
  char a;
  char b;

  method void set_a(char c)
  {
    a = c;
  }
};
```

test-struct-malloc1.out

```
is r
19
0
5
1
6
2
7
3
8
4
9
5
10
6
11
7
12
8
13
9
14
10
```

```
coool
nice
hey
```

test-testcase1.jt

```
func int main()
{
  int i;
  i = add(2,3);
  if (i == 5) {
    print("passed");
  }
  return 0;
}


func int add(int x, int y)
{
  return x + y;
} with test {
  assert(a == a);
} using {
  int a;
  int b;
  a = 10;
  b = 5;
}
```

test-testcase1.out

```
passed
```

test-testcase2.jt

```
func int main()
{
  int a;
  int b;
  int c;

  a = 10;
  b = 5;
  c = 0;

  a = b - c;
  if (a == 5) {
    print("passed");
  }
  return 0;
}


func int sub(int x, int y)
{
  return x - y;
} with test {
  assert(a == b - 5);
} using {
  int a;
  int b;
  a = 5;
  b = 10;
}
```

test-testcase2.out

```
passed
```

test-testcase3.jt

```
func int main()
{
  int a;
  int b;
  int c;

  a = 10;
  b = 23;

  c = max(a, b);

  if (c == 23) {
    print("passed");
  }

  return 0;
}

func int max(int x, int y)
{
  if (x > y) {
    return x;
  }
  return y;
} with test {
   assert((max(a,b) == 10));
} using {
  int a;
  int b;
  a = 10;
  b = 9;
}
```

test-testcase3.out

```
passed
```

test-testcase4.jt

```

int global_var;

func int main()
{
  int tmp;
  struct rectangle *rec_pt;
  rec_pt = new struct rectangle;
  update_rec(rec_pt, 6);
  tmp = rec_pt->width;

  print(tmp);

  return 0;
}

func void update_rec(struct rectangle *p, int x)
{
  p->width = x;
} with test {
  assert(t->width == 30);
  assert(t->height == 4239);
} using {
  struct rectangle *t;
  t = new struct rectangle;
  update_rec(t, 10);
  t->multiply_width(3);
  t->height = 471;
  t->multiply_height(9);
}

struct rectangle {
  int width;
  int height;

  method void multiply_width(int a)
  {
    width = width * a;
  }

  method void multiply_height(int a)
  {
    height = height * a;
  }

};
```

test-testcase4.out

```
6
```

test-while1.jt

```
func int main()
{
  int i;
  int sum;
  i = 0;
  while (i < 10) {
    print("looping");
    i = i + 1;
  }

  return 0;
}
```

test-while1.out

```
looping
looping
looping
looping
looping
looping
looping
looping
looping
looping
```

var-fail1.jt

```
func int main()
{

  return 0;
}

struct phone {
  int price;
  int model;
  int year;
  bool iphone;

  method void set_iphone(bool b)
  {
    phone = b;
  }
};
```

var-fail1.out

```
Scanned
Parsed
Fatal error: exception Exceptions.UndeclaredVariable("phone")
```

7 Lessons Learned

7.1 Andrew Grant

One of the main things I learned was the importance of providing clean and well defined interfaces between the different parts of a large software systems project. There were a few times where we tried to work around the interfaces we had in place, but that only ended up costing us time. For example, at first our SAST was almost exactly the same as our AST. During code generation we often needed access to the types of certain variables. We created code in codegen.ml to do this for us, but it made a lot more sense to add that information to the SAST *while* performing semantic checking. Eventually we added this functionality, but having a clearer understanding of the interface between semantic checking and code generation would have served us well.

One thing I think we did well was start early. There wasn't really a time during the semester that we felt rushed, which I think ended up letting us think clearer and more rationally about how to tackle the next problem. For example, we were able to implement our testing functionality about three weeks before the project was due; this enabled us to focus on the presentation of the testing output, as well as add some simple object-oriented features to our language.

I also think we did a good job communicating with each other and working as a team. We were pretty much all present at most meetings. We met with David just about every week too which was very helpful. There was very little conflict which enabled us to focus on writing the compiler as opposed to wasting time arguing about unimportant things.

Overall I'm very proud of the project we were able to pull off given none of us have any compiler experience before.

7.2 Jemma Losh

I learned the importance of communication when tackling a large-scale group project. It can be difficult to make sure everyone is on the same page and up-to-date with the information they need to complete their portion. Weekly meetings helped our team with this aspect, but between these each group member had to be proactive in reaching out to others and coordinating ideas. The weekly meetings also allowed our team to work on the major key components together, so that everyone was able to understand the course of the project, and not just the parts they put work into. Overall, I was very lucky to have worked with a talented and well rounded team that had little problem with collaboration. For future teams I would suggest defining roles at the beginning, but realize that roles will become more fluid throughout the project, so be able to be flexible and put in work where it's needed.

7.3 Jared Weiss

I think this project helped us learn about working well as a team of software developers. Since we couldn't always work in the same room together, it was important to communicate well and make sure that 2 team members weren't working on the same thing. We needed to manually resolve a few merge conflicts when we started working on this project, but as we developed our git workflow and began using issues to track the work we were doing, there became a lot fewer conflicts. Furthermore, when we all worked together as a group in a library, I feel like we were able to get more work done more quickly since we were able to just ask our teammates simple questions and not need to wait for a response on a github issue tracker.

More regular 'hackathon'-style meetings would have likely made this project even easier, but overall I think we did a very good job of working together. Our roles were fairly well-defined and it was easy to know what each of us was supposed to be working on.

The one area we definitely could have done a little better was in our system of reviewing pull requests. At the start of this project, it was easy to get multiple sets of eyes on a pull request before merging, but as we progressed and the scope of each PR became more complicated, we stopped being as diligent with our reviews. This is why we ultimately ended up moving to a continuous-integration build system: to make sure that if a PR was merged, it didn't break any of the existing functionality. While having this extra layer of security in place is ultimately helpful, it still can't beat having teammates doing code review on every line.

7.4 Jake Weissman

This was a very rewarding experience for me as I got to work on a meaningful project with a team of people that I got along really well with. I'm happy to say that our team worked really well together and got along with little to no conflict. Beyond the joy of completing a really cool project, it was an added bonus to become such good friends with my teammates. As tough as the project was at times, we had some good laughs along the way and made the most out of it. My main takeaway from the project is that teamwork should be a project in itself. We are all smart and motivated students, but working together to such a great extent was a new experience for most of us and it took us some time to get into a groove of working together. There were some early struggles when we couldn't agree on our language, or decide on what we were going to compile down to, but we got through them and made decisions as a team with minimal personality clashes. I think it helped that we were all pretty easy going for the most part. As the project continued, we divided up work nicely and all tried to do our part, with time were people had to pick up the slack for others being inevitable. One comment I would make is that it might have been better to have more structure and more deadlines for our work - setting our own deadlines were often not enough motivation and things might have gotten done more smoothly and continuously if there were more regular, intermediate, deadlines to meet. Thankfully our group was motivated enough to get work done and we didn't have a ton to do at the last minute. All in all the project was a blast and I'm glad I had the opportunity to work with my awesome team!

8 Source Code

This section contains the JaTesté compiler source code. It includes the following files:

- jateste.ml
- scanner.ml
- parser.mly
- ast.ml
- semant.ml
- sast.ml
- codegen.ml
- exceptions.ml

All of the code is open source and available at `https://github.com/jaredweiss/JaTeste`

8.1 jateste.ml

```
open Printf
module A = Ast
module S = Sast


(* Location of Jateste's standard library *)
let standard_library_path = "/home/plt/JaTeste/lib/"
let current_dir_path = "./"

type action = Scan | Parse |  Ast | Sast | Compile | Compile_with_test

(* Determines what action compiler should take based on command line args *)
let determine_action args =
  let num_args = Array.length args in
  (match num_args with
    1 -> raise Exceptions.IllegalInputFormat
  | 2 -> Compile
  | 3 -> let arg = Array.get args 1 in
    (match arg with
      "-t" -> Compile_with_test
    | "-l" -> Scan
    | "-p" -> Parse
    | "-se" ->Sast
    | "-ast" -> Ast
    | _ -> raise (Exceptions.IllegalArgument arg)
    )

  | _ -> raise (Exceptions.IllegalArgument "Can't recognize arguments")
  )

(* Create executable filename *)
let executable_filename filename =
  let len = String.length filename in
  let str = String.sub filename 0 (len - 3) in
  let exec = String.concat "" [str ; ".ll"] in
  exec

(* Create test executable filename *)
let test_executable_filename filename =
  let len = String.length filename in
  let str = String.sub filename 0 (len - 3) in
  let exec = String.concat "" [str ; "-test.ll"] in
  exec

(* Just scan input *)
let scan input_raw =
  let lexbuf = Lexing.from_channel input_raw in (print_string "Scanned\n"); lexbuf

(* Scan, then parse input *)
let parse input_raw =
  let input_tokens = scan input_raw in
  let ast:(A.program) = Parser.program Scanner.token input_tokens in (print_string
     "Parsed\n"); ast

(* Process include statements. Input is ast, and output is a new ast *)
let process_headers ast:(A.program) =
  let (includes,_,_,_) = ast in
```

```
  let gen_header_code (incl,globals, current_func_list, structs) (path, str) =
    let tmp_path = (match path with A.Curr -> current_dir_path | A.Standard -> standard_library_path) in
    let file = tmp_path ^ str in
    let ic =
    try open_in file with _ -> raise (Exceptions.InvalidHeaderFile file) in
    let (_,_,funcs,strs) = parse ic in
    let tmp_funcs = List.map (fun n -> let tmp = {A.typ = n.A.typ ; A.fname = n.A.fname ; A.formals = n.A.formals ; A.vdecls = n.A.vdecls ; A.body = n.A.body ; A.tests = n.A.tests ; A.struc_method = false ; A.includes_func = true } in tmp) funcs in
    let new_ast:(A.program) = (incl, globals, current_func_list @ tmp_funcs, structs @ strs) in
    new_ast
  in
  let modified_ast:(A.program) = List.fold_left gen_header_code ast includes in
  modified_ast


(* Scan, parse, and run semantic checking. Returns Sast *)
let semant input_raw =
  let tmp_ast = parse input_raw in
  let input_ast = process_headers tmp_ast in
  let sast:(S.sprogram) = Semant.check input_ast in (print_string "Semantic check passed\n"); sast

(* Generate code given file. @bool_tests determines whether to create a test file *)
let code_gen input_raw exec_name bool_tests =
  let input_sast = semant input_raw in
  let file = exec_name in
  let oc = open_out file in
  let m = Codegen.gen_llvm input_sast bool_tests in
  Llvm_analysis.assert_valid_module m;
  fprintf oc "%s\n" (Llvm.string_of_llmodule m);
  close_out oc;
  ()

let get_ast input_raw =
  let ast = parse input_raw in
  ast

(*****************************************************************************)

(* Entry pointer for compiler. Input is a .jt text file, output is LLVM code in a .ll file. *)
(*
file.jt text file ->
scanner.mll: convert raw text to tokens according to regexes ->
parser.mly: creates Ast according to CFG defined in parser.mly ->
semant.ml: checks the semantics of the program (e.g. type checking), and converts the Ast into an Sast ->
codege.ml: takes Sast as input and creates LLVM code in a .ll file ->
file.ll file
*)
(*****************************************************************************)

let _ =
  (* Read in command line args *)
```

```
  let arguments = Sys.argv in
  (* Determine what the compiler should do based on command line args *)
  let action = determine_action arguments in
  let source_file = open_in arguments.((Array.length Sys.argv - 1)) in
  (* Create a file to put executable in *)
  let exec_name = executable_filename arguments.((Array.length Sys.argv -1)) in
  (* Create a file to put test executable in *)
  let test_exec_name = test_executable_filename arguments.((Array.length Sys.argv
    -1)) in

  (* Determine what the compiler should do, then do it *)
  let _ = (match action with
    Scan -> let _  = scan source_file in ()
  | Parse -> let _ = parse source_file in ()
  | Ast ->  let _ = parse source_file in ()
  | Sast ->  let _ = semant source_file in ()
  | Compile ->  let _ = code_gen source_file exec_name false in ()
  | Compile_with_test -> let _ = code_gen source_file exec_name false in
      let source_test_file = open_in arguments.((Array.length Sys.argv - 1)) in
           let _ = code_gen source_test_file test_exec_name true in ()
  ) in
  close_in source_file
```

8.2 scanner.mll

```
{ open Parser }

(* Regex shorthands *)
let digit = ['0' - '9']
let my_int = digit+
let double = (digit+) ['.'] digit+
let my_char = '''['a' - 'z' 'A' - 'Z']'''
let newline = '\n'
let my_string = '"' (['a' - 'z'] | [' '] | ['A' - 'Z'] | ['_'] | '!' | ',' )+ '"'

rule token = parse
     [' ' '\t' '\r' '\n' ] { token lexbuf } (* White space *)
  | "/*"       { comment lexbuf }
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE}
  | '}'      { RBRACE}
  | ','      { COMMA }
  | ';'      { SEMI }
  | '#'      { POUND }

  (*Header files *)
  | "include_jtlib"    { INCLUDE }

  (* Operators *)
  | "+"      { PLUS }
  | "-"      { MINUS }
  | "*"      { STAR }
  | "/"      { DIVIDE }
  | "%"      { MODULO }
  | "^"      { EXPO }
  | "="      { ASSIGN }
  | "=="       { EQ }
  | "!="       { NEQ }
  | "!"      { NOT }
  | "&&"       { AND }
  | "&"      { AMPERSAND }
  | "||"       { OR }
  | "<"      { LT }
  | ">"      { GT }
  | "<="       { LEQ }
  | ">="       { GEQ }
  | "["      { LBRACKET }
  | "]"      { RBRACKET }
  | "."      { DOT }
  | "->"       { POINTER_ACCESS }

  (* Control flow *)
  | "if"       { IF }
  | "else"     { ELSE }
  | "return"     { RETURN }
  | "while"      { WHILE }
  | "for"      { FOR }
  | "assert"     { ASSERT }

  (* Datatypes *)
  | "void"     { VOID }
```

```
  | "struct"    { STRUCT }
  | "method"    { METHOD }
  | "double"    { DOUBLE }
  | "int"     { INT }
  | "char"    { CHAR }
  | "string"    { STRING }
  | "bool"    { BOOL }
  | "true"    { TRUE }
  | "false"   { FALSE }
  | "func"    { FUNC }
  | "new"     { NEW }
  | "free"    { FREE }
  | "NULL"    { NULL }
  | "DUBS"    { DUBS }

  (* Testing keywords *)
  | "with test"     { WTEST }
  | "using"   { USING }

  | ['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm)}
  | ['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* ".jt" as lxm { INCLUDE_FILE(lxm) }
  | my_int as lxm       { INT_LITERAL(int_of_string lxm)}
  | double as lxm      { DOUBLE_LITERAL((float_of_string lxm)) }
  | my_char as lxm     { CHAR_LITERAL(String.get lxm 1) }
  | '"' {let buffer = Buffer.create 1 in STRING_LITERAL(string_find buffer lexbuf) }

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


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

and string_find buffer = parse
    '"' {Buffer.contents buffer }
  | _ as chr { Buffer.add_char buffer chr; string_find buffer lexbuf }
```

8.3 parser.mly

```
%{ open Ast %}

/*
   Tokens/terminal symbols
*/
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA SEMI POUND INCLUDE
%token PLUS MINUS STAR DIVIDE ASSIGN NOT MODULO EXPO AMPERSAND
%token FUNC
%token WTEST USING STRUCT DOT POINTER_ACCESS METHOD
%token EQ NEQ LT LEQ GT GEQ AND OR TRUE FALSE
%token INT DOUBLE VOID CHAR STRING BOOL NULL
%token INT_PT DOUBLE_PT CHAR_PT STRUCT_PT
%token ARRAY
%token NEW FREE DUBS
%token RETURN IF ELSE WHILE FOR ASSERT

/*
   Tokens with associated values
*/
%token <int> INT_LITERAL
%token <float> DOUBLE_LITERAL
%token <char> CHAR_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token <string> INCLUDE_FILE
%token EOF

/*
   Precedence rules
*/
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left STAR DIVIDE MODULO
%right EXPO
%right NOT NEG AMPERSAND
%right RBRACKET
%left LBRACKET
%right DOT POINTER_ACCESS

/*
   Start symbol
*/

%start program

/*
   Returns AST of type program
*/

%type<Ast.program> program

```

```
%%

/*
   Use List.rev on any rule that builds up a list in reverse. Lists are built in reverse
   for efficiency reasons
 */

program: includes var_decls func_decls struc_decls  EOF { ($1, List.rev $2, List.rev $3, List.rev $4) }

includes:
    /* noting */ { [] }
  | includes include_file { $2 :: $1 }

include_file:
    POUND INCLUDE STRING_LITERAL { (Curr, $3) }
  | POUND INCLUDE LT INCLUDE_FILE GT        { (Standard,$4) }

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

func_decls:
   fdecl {[$1]}
  | func_decls fdecl  {$2::$1}

mthd:
    METHOD any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body RBRACE {{
    typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
    $9; tests = None ; struc_method = false ; includes_func = false }}

struc_func_decls:
    /* nothing */ { [] }
  | struc_func_decls mthd { $2::$1 }

struc_decls:
    /*nothing*/ { [] }
  | struc_decls sdecl {$2::$1}

prim_typ:
  | STRING  { String }
  | DOUBLE  { Double }
  | INT     { Int }
  | CHAR    { Char }
  | BOOL    { Bool }

void_typ:
  | VOID    { Void }

struct_typ:
  | STRUCT ID { $2 }

array_typ:
      prim_typ LBRACKET INT_LITERAL RBRACKET    { ($1, $3) }
    | prim_typ LBRACKET RBRACKET      { ($1, 0) }

pointer_typ:
```

```
  | prim_typ STAR     { Primitive($1) }
  | struct_typ STAR     { Struct_typ($1) }
  | array_typ STAR    { Array_typ(fst $1, snd $1) }

double_pointer_typ:
  | pointer_typ STAR    { Pointer_typ($1)  }



any_typ:
    prim_typ    { Primitive($1) }
  | struct_typ    { Struct_typ($1) }
  | pointer_typ     { Pointer_typ($1) }
  | double_pointer_typ  { Pointer_typ($1) }
  | void_typ    { Primitive($1) }
  | array_typ   { Array_typ(fst $1, snd $1) }


any_typ_not_void:
        prim_typ    { Primitive($1) }
    | struct_typ    { Struct_typ($1) }
    | pointer_typ     { Pointer_typ($1) }
    | double_pointer_typ  { Pointer_typ($1) }
    | array_typ   { Array_typ(fst $1, snd $1) }

/*
Rules for function syntax
*/
fdecl:
    FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
    RBRACE {{
    typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
    $9; tests = None ; struc_method = false ; includes_func = false}}
  | FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
    RBRACE testdecl {{
    typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
    $9; tests = Some({asserts = $11;  using = { uvdecls = []; stmts = [] }}) ;
    struc_method = false ; includes_func = false  }}
  | FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
    RBRACE testdecl usingdecl {{
    typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
    $9; tests = Some({asserts = $11;  using = { uvdecls = (fst $12); stmts = (snd
    $12)}}) ; struc_method = false ; includes_func = false }}

/*
"with test" rule
*/
testdecl:
  WTEST LBRACE stmt_list RBRACE { $3 }

/*
"using" rule
*/
usingdecl:
  USING LBRACE vdecl_list stmt_list RBRACE { (List.rev $3, List.rev $4) }


/*
Formal parameter rules
```

```
*/
formal_opts_list:
    /* nothing */    { [] }
  | formal_opt { $1 }

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

actual_opts_list:
    /* nothing */ { [] }
  | actual_opt  { $1 }

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

/*
Rule for declaring a list of variables, including variables of type struct x
*/
vdecl_list:
    /* nothing */ { [] }
  | vdecl_list vdecl { $2::$1 }

/*
Includes declaring a struct
*/

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

/*
Rule for defining a struct
*/
sdecl:
  STRUCT ID LBRACE vdecl_list struc_func_decls RBRACE SEMI {{
    sname = $2; attributes = List.rev $4; methods = List.rev $5 }}


func_body:
  stmt_list   {[Block(List.rev $1)]}

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

/*
Rule for statements. Statments include expressions
*/
stmt:
      expr SEMI                 { Expr $1 }
    | LBRACE stmt_list RBRACE           { Block(List.rev $2) }
    | RETURN SEMI                     { Return Noexpr}
    | RETURN expr SEMI                    { Return $2 }
    | IF LPAREN expr RPAREN stmt ELSE stmt              { If($3, $5, $7) }
    | IF LPAREN expr RPAREN stmt %prec NOELSE               { If($3, $5, Block([])
    ) }
    | WHILE LPAREN expr RPAREN stmt                  { While($3, $5) }
```

```
      | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9)}
    | ASSERT LPAREN expr RPAREN SEMI           { Assert($3) }

/*
Rule for building expressions
*/
expr:
    INT_LITERAL     { Lit($1)}
  | STRING_LITERAL  { String_lit($1) }
  | CHAR_LITERAL    { Char_lit($1) }
  | DOUBLE_LITERAL        { Double_lit($1) }
  | TRUE      { BoolLit(true) }
  | FALSE     { BoolLit(false) }
  | ID      { Id($1) }
  | LPAREN expr RPAREN  { $2 }
  | expr PLUS expr  { Binop($1, Add, $3) }
  | expr MINUS expr   { Binop($1, Sub, $3) }
  | expr STAR expr  { Binop($1, Mult, $3)}
  | expr DIVIDE expr  { Binop($1, Div, $3)}
  | expr EQ  expr   { Binop($1, Equal, $3)}
  | expr EXPO  expr   { Binop($1, Exp, $3)}
  | expr MODULO  expr   { Binop($1, Mod, $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) }
  | AMPERSAND expr  { Unop(Addr, $2) }
  | MINUS expr    { Unop(Neg, $2) }
  | expr ASSIGN expr  { Assign($1, $3) }
  | expr DOT expr   { Struct_access($1, $3)}
  | expr POINTER_ACCESS expr  { Pt_access($1, $3)}
  | STAR expr       { Dereference($2) }
  | expr LBRACKET INT_LITERAL RBRACKET       { Array_access($1, $3)}
  | NEW prim_typ LBRACKET INT_LITERAL RBRACKET { Array_create($4, $2) }
  | NEW STRUCT ID             { Struct_create($3)}
  | FREE LPAREN expr RPAREN         { Free($3) }
  | ID LPAREN actual_opts_list RPAREN          { Call($1, $3)}
  | NULL LPAREN any_typ_not_void RPAREN        { Null($3) }
  | DUBS                 { Dubs }
expr_opt:
    /* nothing */ { Noexpr }
  | expr    { $1 }
```

8.4 ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And
    | Or | Mod | Exp
type uop = Neg | Not | Addr
type prim = Int | Double | String | Char | Void | Bool
type typ = Primitive of prim | Struct_typ of string | Func_typ of string |
    Pointer_typ of typ | Array_typ of prim * int | Any
type bind = typ * string

type dir_location = Curr | Standard

(* include files node *)
type header = dir_location * string

(* Jateste expressions *)
type expr =
    Lit      of int
  | String_lit of string
  | Char_lit of char
  | Double_lit of float
  | Binop    of expr * op * expr
  | Unop     of uop * expr
  | Assign   of expr * expr
  | Noexpr
  | Id of string
  | Struct_create of string
  | Struct_access of expr * expr
  | Pt_access of expr * expr
  | Dereference of expr
  | Array_create of int * prim
  | Array_access of expr * int
  | Free of expr
  | Call of string * expr list
  | BoolLit of bool
  | Null of typ
  | Dubs

(* Jateste statements *)
type stmt =
    Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | For of expr * expr * expr * stmt
  | Return of expr
  | Assert of expr

(* Node that describes the envoirnment for with_test_decl node *)
type with_using_decl = {
  uvdecls : bind list;
  stmts : stmt list;
}

(* Node the describes test cases *)
type with_test_decl = {
  asserts : stmt list;
  using : with_using_decl;
}
```

```
(* Node that describes a function *)
type func_decl = {
  typ : typ;
  fname : string;
  formals : bind list;
  vdecls  : bind list;
  body  :   stmt list;
  tests   :   with_test_decl option;
  struc_method : bool;
  includes_func : bool;
}

(* Node that describes a given struct *)
type struct_decl = {
  sname   : string;
  attributes  : bind list;
  methods  : func_decl list;
}

(* Root of tree. Our program is made up four things 1) list of header/include
    files 2) list of global variables 3) list of function definitions 4) list of
    struct definitions *)
type program = header list * bind list * func_decl list * struct_decl list
```

8.5 semant.ml

```
(* Semantic checker code. Takes Ast as input and returns a Sast *)

module A = Ast
module S = Sast
module StringMap = Map.Make(String)

type variable_decls = A.bind;;

(* Hashtable of valid structs. This is filled out when we iterate through the user
     defined structs *)
let struct_types:(string, A.struct_decl) Hashtbl.t = Hashtbl.create 10
let func_names:(string, A.func_decl) Hashtbl.t = Hashtbl.create 10

let built_in_print_string:(A.func_decl) = {A.typ = A.Primitive(A.Void) ; A.fname =
     "print"; A.formals = [A.Any, "arg1"]; A.vdecls = []; A.body = []; A.tests =
    None ; A.struc_method = false ; includes_func = false }

(* Symbol table used for checking scope *)
type symbol_table = {
  parent : symbol_table option;
  variables : (string, A.typ) Hashtbl.t;
}

(* Environment*)
type environment = {
  scope : symbol_table;
  return_type : A.typ option;
  func_name : string option;
  in_test_func : bool;
  in_struct_method : bool;
  struct_name : string option
}

(* For debugging *)
let rec string_of_typ t =
  match t with
    A.Primitive(A.Int) -> "Int"
  | A.Primitive(A.Double) -> "Double"
  | A.Primitive(A.String) -> "String"
  | A.Primitive(A.Char) -> "Char"
  | A.Primitive(A.Void) -> "Void"
  | A.Struct_typ(s) -> "struct " ^ s
  | A.Pointer_typ(t) -> "pointer " ^ (string_of_typ t)
  | A.Array_typ(p,_) -> "Array type " ^ (string_of_typ (A.Primitive(p)))
  | _ -> "not sure"

(* Search symbol tables to see if the given var exists somewhere *)
let rec find_var (scope : symbol_table) var =
  try Hashtbl.find scope.variables var
  with Not_found ->
  match scope.parent with
    Some(parent) -> find_var parent var
  | _ -> raise (Exceptions.UndeclaredVariable var)

(* Helper function to reeturn an identifers type *)
let type_of_identifier var env =
  find_var env.scope var
```

```
(* left side of Binop. Returns an expression *)
let left_side_of_binop e =
  (match e with
    A.Binop(ls,_,_) -> ls
  | _ -> raise (Exceptions.BugCatch "left side of binop")
  )

(* left side of Binop. Returns an expression *)
let right_side_of_binop e =
  (match e with
    A.Binop(_,_,rs) -> rs
  | _ -> raise (Exceptions.BugCatch "left side of binop")
  )

(* Returns the type of the arrays elements. E.g. int[10] arr... type_of_array arr
    would return A.Int *)
let type_of_array arr _ =
  match arr with
    A.Array_typ(p,_) -> A.Primitive(p)
  | A.Pointer_typ(A.Array_typ(p,_)) -> A.Primitive(p)
  | _ -> raise (Exceptions.InvalidArrayVariable)

(* Function is done for creating sast after semantic checking. Should only be
    called on struct or array access *)
let rec string_identifier_of_expr expr =
  match expr with
    A.Id(s) -> s
  | A.Struct_access(e1, _) -> string_identifier_of_expr e1
  | A.Pt_access(e1, _) -> string_identifier_of_expr e1
  | A.Array_access(e1, _) -> string_identifier_of_expr e1
  | A.Call(s,_) -> s
  | _ -> raise (Exceptions.BugCatch "string_identifier_of_expr")

(* Used for generating test prints *)
let rec string_of_expr e env =
  match e with
    A.Lit(i) -> string_of_int i
    | A.String_lit(s) -> s
    | A.Char_lit(c) -> String.make 1 c
    | A.Double_lit(_) -> ""
    | A.Binop(e1,op,e2) -> let str1 = string_of_expr e1 env in
    let str2 = string_of_expr e2 env in
    let str_op =
    (match op with
      A.Add-> "+"
    | A.Sub -> "-"
    | A.Mult -> "*"
    | A.Div -> "/"
    | A.Equal -> "=="
    | A.Neq -> "!="
    | A.Less -> "<"
    | A.Leq -> "=<"
    | A.Greater -> ">"
    | A.Geq -> ">="
    | A.And -> "&&"
    | A.Or -> "||"
    | A.Mod  -> "%"
    | A.Exp -> "^"
```

```
    ) in (String.concat " " [str1;str_op;str2])
    | A.Unop(u,e) -> let str_expr = string_of_expr e env in
       let str_uop =
      (match u with
        A.Neg -> "-"
      | A.Not -> "!"
      | A.Addr -> "&"
      ) in
      let str1 = String.concat "" [str_uop; str_expr] in str1
    | A.Assign (_,_) -> ""
    | A.Noexpr -> ""
    | A.Id(s) -> s
    | A.Struct_create(_) -> ""
    | A.Struct_access(e1,e2) -> let str1 = string_of_expr e1 env in
        let str2 = string_of_expr e2 env in
        let str_acc = String.concat "." [str1; str2] in str_acc
    | A.Pt_access(e1,e2) -> let str1 = string_of_expr e1 env in
        let str2 = string_of_expr e2 env in
        let str_acc = String.concat "->" [str1; str2] in str_acc

    | A.Dereference(e) -> let str1 = string_of_expr e env in (String.concat "" ["*"; str1])
    | A.Array_create(i,p) -> let str_int = string_of_int i in
      let rb = "]" in
      let lb = "[" in
      let new_ = "new" in
      let str_prim =
      (match p with
        A.Int -> "int"
      | A.Double ->"double"
      | A.Char -> "char"
      | _ -> raise (Exceptions.InvalidArrayType)
      ) in let str_ar_ac = String.concat "" [new_; " "; str_prim; lb; str_int; rb] in str_ar_ac
    | A.Array_access(e,i) -> let lb = "[" in
      let rb = "]" in
      let str_int = string_of_int i in
      let str_expr = string_of_expr e env in
      let str_acc = String.concat "" [str_expr; lb; str_int; rb] in str_acc
    | A.Free(_) -> ""
    | A.Call(s,le) -> let str1 = s ^"(" in
    let str_exprs_rev = List.map (fun n -> string_of_expr n env) le in
    let str_exprs = List.rev str_exprs_rev in
    let str_exprs_commas = (String.concat "," str_exprs) in
    let str2 = (String.concat "" (str1::str_exprs_commas::[")"])) in str2
    | A.BoolLit (b) ->
    (match b with
      true -> "true"
    | false -> "false"
    )
    | A.Null(_) -> "NULL"
    | A.Dubs -> ""

(* Function is done for creating sast after semantic checking. Should only be called on struct fields *)
let string_of_struct_expr expr =
  match expr with
    A.Id(s) -> s
  | _ -> raise (Exceptions.BugCatch "string_of_struct_expr")
```

```

(* Helper function to check for dups in a list *)
let report_duplicate exceptf list =
    let rec helper = function
        n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)

(* Used to check include statments *)
let check_ends_in_jt str =
  let len = String.length str in
  if len < 4 then raise (Exceptions.InvalidHeaderFile str);
  let subs = String.sub str (len - 3) 3 in
  (match subs with
    ".jt" -> ()
  | _ -> raise (Exceptions.InvalidHeaderFile str)
  )

let check_in_test e = if e.in_test_func = true then () else raise (Exceptions.InvalidAssert "assert can only be used in tests")

(* Helper function to check a typ is not void *)
let check_not_void exceptf = function
      (A.Primitive(A.Void), n) -> raise (Failure (exceptf n))
    | _ -> ()

(* Helper function to check two types match up *)
let check_assign lvaluet rvaluet err =
  (match lvaluet with
    A.Pointer_typ(A.Array_typ(p,0)) ->
          (match rvaluet with
          A.Pointer_typ(A.Array_typ(p2,_)) -> if p = p2 then lvaluet else raise err
          | _ -> raise err
          )
  | A.Primitive(A.String) -> (match rvaluet with A.Primitive(A.String) -> lvaluet | A.Array_typ(A.Char,_) -> lvaluet | _ -> raise err)
  | A.Array_typ(A.Char,_) -> (match rvaluet with A.Array_typ((A.Char),_) -> lvaluet | A.Primitive(A.String) -> lvaluet | _ -> raise err)
  | _ -> if lvaluet = rvaluet then lvaluet else raise err
  )


(* Search hash table to see if the struct is valid *)
let check_valid_struct s =
  try Hashtbl.find struct_types s
  with | Not_found -> raise (Exceptions.InvalidStruct s)

(* Checks the hash table to see if the function exists *)
let check_valid_func_call s =
  try Hashtbl.find func_names s
  with | Not_found -> raise (Exceptions.InvalidFunctionCall (s ^ " does not exist. Unfortunately you can't just expect functions to magically exist"))


(* Helper function that finds index of first matching element in list *)
let rec index_of_list x l =
  match l with
```

```
    [] -> raise (Exceptions.BugCatch "index_of_list")
  | hd::tl -> let (_,y) = hd in if x = y then 0 else 1 + index_of_list x tl

let index_helper s field env =
    let struct_var = find_var env.scope s in
    match struct_var with
      A.Struct_typ(struc_name) ->
    (let stru:(A.struct_decl) = check_valid_struct struc_name in
    try let index = index_of_list field stru.A.attributes in index with |
    Not_found -> raise (Exceptions.BugCatch "index_helper"))
    | A.Pointer_typ(A.Struct_typ(struc_name)) ->
    (let stru:(A.struct_decl) = check_valid_struct struc_name in
    try let index = index_of_list field stru.A.attributes in index with |
    Not_found -> raise (Exceptions.BugCatch "index_helper"))
    | _ -> raise (Exceptions.BugCatch "struct_contains_field")


(* Function that returns index of the field in a struct. E.g. given: stuct person
   {int age; int height;};.... index_of_struct_field *str "height" env will return
    1 *)
let index_of_struct_field stru expr env =
    match stru with
        A.Id(s) -> (match expr with A.Id(s1) -> index_helper s s1 env | _ -> raise
     (Exceptions.BugCatch "index_of_struct"))
      | _ -> raise (Exceptions.InvalidStructField)



(* Checks the relevant struct actually has a given field *)
let struct_contains_field s field env =
    let struct_var = find_var env.scope s in
    match struct_var with
      A.Struct_typ(struc_name) ->
    (let stru:(A.struct_decl) = check_valid_struct struc_name in
    try let (my_typ,_) = (List.find (fun (_,nm) -> if nm = field then true else
    false) stru.A.attributes) in my_typ with
        | Not_found -> raise (Exceptions.InvalidStructField))
    | A.Pointer_typ(A.Struct_typ(struc_name)) ->
    (let stru:(A.struct_decl) = check_valid_struct struc_name in
    try let (my_typ,_) = (List.find (fun (_,nm) -> if nm = field then true else
    false) stru.A.attributes) in my_typ with
    | Not_found ->  try let tmp_fun = (List.find (fun f -> if f.A.fname = field
    then true else false) stru.A.methods) in  tmp_fun.A.typ with
        | Not_found -> raise (Exceptions.InvalidStructField))

    | _ -> raise (Exceptions.BugCatch "struct_contains_field")

let struct_contains_method s methd env =
    let struct_var = find_var env.scope s in
    match struct_var with
     A.Pointer_typ(A.Struct_typ(struc_name)) | A.Struct_typ(struc_name) ->
    (let stru:(A.struct_decl) = check_valid_struct struc_name in
     try let tmp_fun = (List.find (fun f -> if f.A.fname = methd then true else
    false) stru.A.methods) in tmp_fun.A.typ with | Not_found -> raise (Exceptions.
    InvalidStructField))

    | _ -> raise (Exceptions.BugCatch "struct_contains_field")

```

```
(* Checks that struct contains expr *)
let struct_contains_expr stru expr env =
  match stru with
    A.Id(s) -> (match expr with
          A.Id(s1) -> struct_contains_field s s1 env
       |  A.Call(s1, _) -> struct_contains_method s s1 env
       | _ -> raise (Exceptions.InvalidStructField))
  | _ -> raise (Exceptions.InvalidStructField)

let struct_field_is_local str fiel env =
  try (let _ = struct_contains_field str fiel env in false)
  with | Exceptions.InvalidStructField -> true

(* Returns type of expression - used for checking for type mismatches *)
let rec type_of_expr env e =
  match e with
    A.Lit(_) -> A.Primitive(A.Int)
  | A.String_lit(_) -> A.Primitive(A.String)
    | A.Char_lit (_) -> A.Primitive(A.Char)
    | A.Double_lit(_) -> A.Primitive(A.Double)
    | A.Binop(e1,_,_) -> type_of_expr env e1
    | A.Unop (_,e1) -> type_of_expr env e1
    | A.Assign(e1,_) -> type_of_expr env e1
    | A.Id(s) -> find_var env.scope s
  | A.Struct_create(s) -> A.Pointer_typ(A.Struct_typ(s))
  | A.Struct_access(e1,e2) -> struct_contains_expr e1 e2 env
  | A.Pt_access(e1,e2) -> let tmp_type = type_of_expr env e1 in
        (match tmp_type with
        A.Pointer_typ(A.Struct_typ(_)) ->
          (match e2 with
              A.Call(_,_) -> struct_contains_expr e1 e2 env
            | A.Id(_) -> struct_contains_expr e1 e2 env
          | _ -> raise (Exceptions.BugCatch "type_of_expr")
          )
        | _ -> raise (Exceptions.BugCatch "type_of_expr")
        )
  | A.Dereference(e1) -> let tmp_e = type_of_expr env e1 in
    (
    match tmp_e with
      A.Pointer_typ(p) -> p
    | _ -> raise (Exceptions.BugCatch "type_of_expr")
    )
  | A.Array_create(i,p) -> A.Pointer_typ(A.Array_typ(p,i))
  | A.Array_access(e,_) -> type_of_array (type_of_expr env e) env
  | A.Call(s,_) -> let func_info = (check_valid_func_call s) in func_info.A.typ
    | A.BoolLit (_) -> A.Primitive(A.Bool)
    | A.Null(t) -> t
  | _ -> raise (Exceptions.BugCatch "type_of_expr")

(* convert expr to sast expr *)
let rec expr_sast expr env =
  match expr with
    A.Lit a -> S.SLit a
  | A.String_lit s -> S.SString_lit s
  | A.Char_lit c -> S.SChar_lit c
  | A.Double_lit d -> S.SDouble_lit d
  | A.Binop (e1, op, e2) -> let tmp_type = type_of_expr env e1 in
      S.SBinop (expr_sast e1 env, op, expr_sast e2 env, tmp_type)
```

```
    | A.Unop (u, e) -> let tmp_type = type_of_expr env e in S.SUnop(u, expr_sast e env, tmp_type)
    | A.Assign (s, e) -> S.SAssign (expr_sast s env, expr_sast e env)
    | A.Noexpr -> S.SNoexpr
    | A.Id s ->  (match env.in_struct_method with
        true ->
        (match env.struct_name with
          Some(nm) -> let local_struct_field = struct_field_is_local nm s env in
        (match local_struct_field with
          true -> S.SId (s)
        | false -> let tmp_id = A.Id(nm) in
        let tmp_pt_access = A.Pt_access(tmp_id, A.Id(s)) in
        (expr_sast tmp_pt_access env)
        )
        | None -> raise (Exceptions.BugCatch "expr_sast")
        )
      | false -> S.SId (s)
         )
    | A.Struct_create s -> S.SStruct_create s
    | A.Free e -> let st = (string_identifier_of_expr e) in S.SFree(st)
    | A.Struct_access (e1, e2) ->
      (match e2 with
        A.Id(_) ->  let index = index_of_struct_field e1 e2 env in
          let tmp_type = (type_of_expr env (A.Struct_access(e1,e2))) in
            S.SStruct_access (string_identifier_of_expr e1, string_of_struct_expr e2, index, tmp_type)
      | A.Call(ec, le) -> let string_of_ec = string_identifier_of_expr e1 in let struct_decl = find_var env.scope string_of_ec in
        (match struct_decl with
        A.Struct_typ(struct_type_string) -> let tmp_unop = A.Unop(A.Addr, e1) in S.SCall (struct_type_string ^ ec, (List.map (fun n -> expr_sast n env) ([tmp_unop]@le)))
        | _ -> raise (Exceptions.BugCatch "expr_sast")
        )
      | _ -> raise (Exceptions.BugCatch "expr_sast")
      )
    | A.Pt_access (e1, e2) ->
     (match e2 with
       A.Id(_) ->let tmp_type =  (type_of_expr env (A.Pt_access(e1,e2))) in let index = index_of_struct_field e1 e2 env in let t =  S.SPt_access (string_identifier_of_expr e1, string_identifier_of_expr e2, index, tmp_type) in t
     | A.Call(ec,le) -> let string_of_ec = string_identifier_of_expr e1 in let struct_decl = find_var env.scope string_of_ec in
       (match struct_decl with
       A.Pointer_typ(A.Struct_typ(struct_type_string)) -> S.SCall (struct_type_string ^ ec, (List.map (fun n -> expr_sast n env) ([e1]@le)))
       | _ -> raise (Exceptions.BugCatch "expr_sast")
       )
     | _ -> raise (Exceptions.BugCatch "expr_sast")
     )
    | A.Array_create (i, p) -> S.SArray_create (i, p)
    | A.Array_access (e, i) -> let tmp_string = (string_identifier_of_expr e) in
     let tmp_type = find_var env.scope tmp_string in S.SArray_access (tmp_string, i, tmp_type)
    | A.Dereference(e) -> let tmp_type = (type_of_expr env (A.Dereference(e))) in S.SDereference(string_identifier_of_expr e, tmp_type)
    | A.Call (s, e) -> S.SCall (s, (List.map (fun n -> expr_sast n env) e))
    | A.BoolLit(b) -> S.SBoolLit((match b with true -> 1 | false -> 0))
```

```
  | A.Null(t) -> S.SNull t
  | A.Dubs -> S.SDubs


(* Convert ast struct to sast struct *)
let struct_sast r =
  let tmp:(S.sstruct_decl) = {S.ssname = r.A.sname ; S.sattributes = r.A.
    attributes} in
  tmp


(* function that adds struct pointer to formal arg *)
let add_pt_to_arg s f =
  let tmp_formals = f.A.formals in
  let tmp_type = A.Pointer_typ(A.Struct_typ(s.A.sname)) in
  let tmp_string = "pt_hack" in
  let new_formal:(A.bind) = (tmp_type, tmp_string) in
  let formals_with_pt = new_formal :: tmp_formals in
  let new_func = {A.typ = f.A.typ ; A.fname = s.A.sname ^ f.A.fname ; A.formals =
    formals_with_pt ; A.vdecls = f.A.vdecls; A.body = f.A.body; A.tests = f.A.tests
     ; A.struc_method = true ; A.includes_func = f.A.includes_func} in
  new_func

(* Creates new functions whose first paramters is a pointer to the struct type
    that the method is associated with *)
let add_pts_to_args s fl =
  let list_of_struct_funcs = List.map (fun n -> add_pt_to_arg s n) fl in
  list_of_struct_funcs


(* Struct semantic checker *)
let check_structs structs =
  (report_duplicate(fun n -> "duplicate struct " ^ n) (List.map (fun n -> n.A.
    sname) structs));

  ignore (List.map (fun n -> (report_duplicate(fun n -> "duplicate struct field "
    ^ n) (List.map (fun n -> snd n) n.A.attributes))) structs);

  ignore (List.map (fun n -> (List.iter (check_not_void (fun n -> "Illegal void
    field" ^ n)) n.A.attributes)) structs);
  ignore(List.iter (fun n -> Hashtbl.add struct_types n.A.sname n) structs);
  let tmp_funcs = List.map (fun n -> (n, n.A.methods)) structs in
  let tmp_funcs_with_formals = List.fold_left (fun l s  -> let tmp_l = (
    add_pts_to_args (fst s) (snd s)) in l @ tmp_l) [] tmp_funcs in
  (structs, tmp_funcs_with_formals)

(* Globa variables semantic checker *)
let check_globals globals env =
  ignore(env);
  ignore (report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd
    globals));
  List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;
  (* Check that any global structs are actually valid structs that have been
    defined *)
  List.iter (fun (t,_) -> match t with
      A.Struct_typ(nm) -> ignore(check_valid_struct nm); ()
    | _ -> ()
  ) globals;
  (* Add global variables to top level symbol table. Side effects *)
```

```
  List.iter (fun (t,s) -> (Hashtbl.add env.scope.variables s t)) globals;
  globals

(* Main entry pointer for checking the semantics of an expression *)
let rec check_expr expr env =
  match expr with
    A.Lit(_) -> A.Primitive(A.Int)
  | A.String_lit(_) -> A.Primitive(A.String)
  | A.Char_lit(_) -> A.Primitive(A.Char)
  | A.Double_lit(_) -> A.Primitive(A.Double)
  | A.Binop(e1,op,e2) -> let e1' = (check_expr e1 env) in
    let e2' = (check_expr e2 env) in
    (match e1' with
      A.Primitive(A.Int) | A.Primitive(A.Double) | A.Primitive(A.Char)  ->
    (match op with
      A.Add | A.Sub | A.Mult | A.Div | A.Exp | A.Mod  when e1' = e2' && (e1' = A.Primitive(A.Int) || e1' = A.Primitive(A.Double))-> e1'
    | A.Equal | A.Neq when e1' = e2' -> A.Primitive(A.Bool)
    | A.Less | A.Leq | A.Greater | A.Geq when e1' = e2' && (e1' = A.Primitive(A.Int) || e1' = A.Primitive(A.Double))-> A.Primitive(A.Bool)
    | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
)
    | A.Primitive(A.Bool) ->
      (match op with
      | A.And | A.Or when e1' = e2' && (e1' = A.Primitive(A.Bool)) -> e1'
      | A.Equal | A.Neq when e1' = e2' -> A.Primitive(A.Bool)
      | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
    )
    | A.Pointer_typ(_) -> let e1' = (check_expr e1 env) in
      let e2' = (check_expr e1 env)  in
    (match op with
      A.Equal | A.Neq when e1' = e2' && (e1 = A.Null(e2') || e2 = A.Null(e1') ) -> A.Primitive(A.Bool)
    | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
    )
    | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
    )
  | A.Unop(uop,e) -> let expr_type = check_expr e env in
      (match uop with
          A.Not -> (match expr_type with
            A.Primitive(A.Bool) -> expr_type
            | _ -> raise Exceptions.NotBoolExpr
             )
        | A.Neg -> (match expr_type with
               A.Primitive(_) -> expr_type
             | _ -> raise Exceptions.InvalidNegativeType
             )
        | A.Addr -> (match e with
               A.Id(_) -> A.Pointer_typ(expr_type)
             | _ -> raise Exceptions.InvalidNegativeType
             )      )
  | A.Assign(var,e) -> (let right_side_type = check_expr e env in
      let left_side_type  = check_expr var env in
        check_assign left_side_type right_side_type Exceptions.IllegalAssignment)
  | A.Noexpr -> A.Primitive(A.Void)
  | A.Id(s) -> type_of_identifier s env
  | A.Struct_create(s) -> (try let tmp_struct = check_valid_struct s in (A.Pointer_typ(A.Struct_typ(tmp_struct.A.sname))) with
      | Not_found -> raise (Exceptions.InvalidStruct s))
```

```
  | A.Struct_access(e1,e2) -> let e1' = check_expr e1 env in
        (match e1' with
          A.Struct_typ(st) ->
        (match e2 with
            A.Call(sc,args) -> ignore(struct_contains_expr e1 e2 env);
               let tmp_expr = A.Unop(A.Addr, e1) in
               let tmp_formals = [tmp_expr] @ args in
               let tmp_struc_string = st in
               let tmp_func_name = tmp_struc_string ^ sc in
               let tmp_call = A.Call(tmp_func_name, tmp_formals) in
               check_expr tmp_call env
          | A.Id(_) ->  struct_contains_expr e1 e2 env
          | _ ->  raise (Exceptions.BugCatch "check_expr")
        )
        | _ -> raise (Exceptions.BugCatch "check_expr")
        )

  | A.Pt_access(e1,e2) -> let e1' = check_expr e1 env in
      (match e1' with
        A.Pointer_typ(A.Struct_typ(_)) ->
      (match e2 with
        A.Call(sc,args) -> ignore(struct_contains_expr e1 e2 env);
               let tmp_string2 = string_identifier_of_expr e1 in
               let tmp_formals = [e1] @ args in
               let tmp_struc = find_var env.scope tmp_string2 in
               let tmp_struc_string =
               (match tmp_struc with
                    A.Pointer_typ(A.Struct_typ(sst)) -> sst
            |   _ -> raise (Exceptions.InvalidStructMethodCall)
               ) in
               let tmp_func_name = tmp_struc_string ^ sc in
               let tmp_call = A.Call(tmp_func_name, tmp_formals) in
               check_expr tmp_call env
      | A.Id(_) ->  struct_contains_expr e1 e2 env
      | _ ->  raise (Exceptions.InvalidPointerAccess)
      )
      | A.Pointer_typ(A.Primitive(p)) -> (let e2' = check_expr e2 env in ( check_assign (A.Primitive(p)) e2') (Exceptions.InvalidPointerDereference))
      | _ -> raise (Exceptions.InvalidPointerAccess)
      )
  | A.Dereference(i) ->  let pointer_type = (check_expr i env)  in
      (
       match pointer_type with
         A.Pointer_typ(pt) -> pt
       | _ -> raise (Exceptions.InvalidDereference)
      )

  | A.Array_create(size,prim_type) -> A.Pointer_typ(A.Array_typ(prim_type, size))
  | A.Array_access(e, _) -> type_of_array (check_expr e env) env
  | A.Free(p) -> let pt = string_identifier_of_expr p in
           let pt_typ = find_var env.scope pt in (match pt_typ with A.Pointer_typ(_) -> pt_typ | _ -> raise (Exceptions.InvalidFree "not a pointer"))
  | A.Call("print", el) ->  if List.length el != 1 then raise Exceptions.InvalidPrintCall
        else
        List.iter (fun n -> ignore(check_expr n env); ()) el; A.Primitive(A.Int)
  | A.Call(s,el) -> let func_info = (check_valid_func_call s) in
        let func_info_formals = func_info.A.formals in
        if List.length func_info_formals != List.length el then
```

```
        raise (Exceptions.InvalidArgumentsToFunction (s ^ " is supplied with wrong args"))
  else
    List.iter2 (fun (ft,_) e -> let e = check_expr e env in ignore(check_assign ft e (Exceptions.InvalidArgumentsToFunction ("Args to functions " ^ s ^ " don't match up with it's definition")))) func_info_formals el;
  func_info.A.typ
  | A.BoolLit(_) -> A.Primitive(A.Bool)
  | A.Null(t) -> t
  | A.Dubs -> A.Primitive(A.Void)

(* Checks if expr is a boolean expr. Used for checking the predicate of things like if, while statements *)
let check_is_bool expr env =
  ignore(check_expr expr env);
  match expr with
   A.Binop(_,A.Equal,_) | A.Binop(_,A.Neq,_) | A.Binop(_,A.Less,_) | A.Binop(_,A.Leq,_) | A.Binop(_,A.Greater,_) | A.Binop(_,A.Geq,_) | A.Binop(_,A.And,_) | A.Binop(_,A.Or,_) | A.Unop(A.Not,_) -> ()

  | _ ->  raise (Exceptions.InvalidBooleanExpression)

(* Checks that return value is the same type as the return type in the function definition*)
let check_return_expr expr env =
  match env.return_type with
    Some(rt) -> if rt = check_expr expr env then () else raise (Exceptions.InvalidReturnType "return type doesnt match with function definition")
  | _ -> raise (Exceptions.BugCatch "Should not be checking return type outside a function")

(* Main entry point for checking semantics of statements *)
let rec check_stmt stmt env =
  match stmt with
    A.Block(l) -> (let rec check_block b env2=
      (match b with
        [A.Return _ as s] -> let tmp_block = check_stmt s env2 in ([tmp_block])
      | A.Return _ :: _ -> raise (Exceptions.InvalidReturnType "Can't have any code after return statement")
      | A.Block l :: ss -> check_block (l @ ss) env2
      | l :: ss -> let tmp_block = (check_stmt l env2) in
        let tmp_block2 = (check_block ss env2) in ([tmp_block] @ tmp_block2)
      | [] -> ([]))
      in
      let checked_block = check_block l env in S.SBlock(checked_block)
      )
  (*| A.Block(b) -> S.SBlock (List.map (fun n -> check_stmt n env) b) *)
  | A.Expr(e) -> ignore(check_expr e env); S.SExpr(expr_sast e env)
  | A.If(e1,s1,s2) ->ignore(check_expr e1 env); ignore(check_is_bool e1 env); S.SIf (expr_sast e1 env, check_stmt s1 env, check_stmt s2 env)
  | A.While(e,s) -> ignore(check_is_bool e env); S.SWhile (expr_sast e env, check_stmt s env)
  | A.For(e1,e2,e3,s) -> ignore(e1);ignore(e2);ignore(e3);ignore(s); S.SFor(expr_sast e1 env, expr_sast e2 env, expr_sast e3 env, check_stmt s env)
  | A.Return(e) -> ignore(check_return_expr e env);S.SReturn (expr_sast e env)
  | A.Assert(e) -> ignore(check_in_test env); ignore(check_is_bool e env);
      let str_expr = string_of_expr e env in
      let lhs = (expr_sast (left_side_of_binop e) env) in
      let rhs = (expr_sast (right_side_of_binop e) env) in
```

```
      let then_stmt = S.SExpr(S.SCall("print", [S.SString_lit(str_expr ^ " passed!")])) in
      let else_stmt = S.SBlock([S.SExpr(S.SCall("print", [S.SString_lit(str_expr ^ " failed!")]))]
      @[S.SExpr(S.SCall("print", [S.SString_lit("LHS evaluated to: ")]))]
      @[S.SExpr(S.SCall("print", [lhs]))]
      @[S.SExpr(S.SCall("print", [S.SString_lit("RHS evaluated to: ")]))]         @[S.SExpr(S.SCall("print", [rhs]))]) in S.SIf (expr_sast e env, then_stmt, else_stmt)

(* Converts 'using' code from ast to sast *)
let with_using_sast r env =
  let tmp:(S.swith_using_decl) = {S.suvdecls = r.A.uvdecls; S.sstmts = (List.map (fun n -> check_stmt n env) r.A.stmts)} in
   tmp

(* Converts 'test' code from ast to sast *)
let with_test_sast r env =
  let tmp:(S.swith_test_decl) = {S.sasserts = (List.map (fun n -> check_stmt n env) r.A.asserts) ; S.susing = (with_using_sast r.A.using env)} in
  tmp

(* Here we convert the user defined test cases to functions which can subseqeuntly be called by main in the test file *)
let convert_test_to_func using_decl test_decl env =
  List.iter (fun n -> (match n with A.Assert(_) -> () | _ -> raise Exceptions.InvalidTestAsserts)) test_decl.A.asserts;
  let test_asserts = List.rev test_decl.A.asserts in
  let concat_stmts = using_decl.A.stmts @ test_asserts  in
  (match env.func_name with
    Some(fn) ->let new_func_name = fn ^ "test" in
    let new_func:(A.func_decl) = {A.typ = A.Primitive(A.Void); A.fname = (new_func_name); A.formals = []; A.vdecls =  using_decl.A.uvdecls; A.body = concat_stmts ; A.tests = None ; A.struc_method = false ; includes_func = false } in new_func

  |  None -> raise (Exceptions.BugCatch "convert_test_to_func")
)

(* Function names (aka can't have two functions with same name) semantic checker *)
let check_function_names functions =
  ignore(report_duplicate (fun n -> "duplicate function names " ^ n) (List.map (fun n -> n.A.fname) functions));
  (* Add the built in function(s) here. There shouldnt be too many of these *)
  ignore(Hashtbl.add func_names built_in_print_string.A.fname built_in_print_string);
  (* Go through the functions and add their names to a global hashtable that stores the whole function as its value -> (key, value) = (func_decl.fname, func_decl) *)
  ignore(List.iter (fun n -> Hashtbl.add func_names n.A.fname n) functions); ()

let check_prog_contains_main funcs =
  let contains_main = List.exists (fun n -> if n.A.fname = "main" then true else false) funcs in
  (match contains_main with
    true -> ()
  | false -> raise Exceptions.MissingMainFunction
  )
```

```

(* Checks programmer hasn't defined function print as it's reserved *)
let check_function_not_print names =
  ignore(if List.mem "print" (List.map (fun n -> n.A.fname) names ) then raise ( Failure ("function print may not be defined")) else ()); ()

(* Check the body of the function here *)
let rec check_function_body funct env =
  let curr_func_name = funct.A.fname in
  report_duplicate (fun n -> "duplicate formal arg " ^ n) (List.map snd funct.A.formals);
  report_duplicate (fun n -> "duplicate local " ^ n) (List.map snd funct.A.vdecls);
  (* Check no duplicates *)

  let in_struc = env.in_struct_method in
  let formals_and_locals =
    (match in_struc with
       true ->
      let (struct_arg_typ, _) = List.hd funct.A.formals in
                          (match struct_arg_typ with
                            A.Pointer_typ(A.Struct_typ(s)) -> let struc_arg = check_valid_struct s in
      List.append (List.append funct.A.formals funct.A.vdecls) struc_arg.A.attributes
                          | _ -> raise (Exceptions.BugCatch "check function body")
                          )
                  | false -> List.append funct.A.formals funct.A.vdecls
                  )
         in

  report_duplicate (fun n -> "same name for formal and local var " ^ n) (List.map snd formals_and_locals);
  (* Check structs are valid *)
  List.iter (fun (t,_) -> match t with
      A.Struct_typ(nm) -> ignore(check_valid_struct nm); ()
    | _ -> ()
  ) formals_and_locals;
  (* Create new enviornment -> symbol table parent is set to previous scope's symbol table *)
  let new_env = {scope = {parent = Some(env.scope) ; variables = Hashtbl.create 10}; return_type = Some(funct.A.typ) ; func_name = Some(curr_func_name); in_test_func = env.in_test_func ; in_struct_method = env.in_struct_method ; struct_name = env.struct_name} in
  (* Add formals + locals to this scope symbol table *)
  List.iter (fun (t,s) -> (Hashtbl.add new_env.scope.variables s t)) formals_and_locals;
  let body_with_env = List.map (fun n -> check_stmt n new_env) funct.A.body in
  (* Compile code for test case iff a function has defined a with test clause *)
  let sast_func_with_test =
    (match funct.A.tests with
    Some(t) ->  let func_with_test = convert_test_to_func t.A.using t new_env in let new_env2 = {scope = {parent = None; variables = Hashtbl.create 10}; return_type = Some(A.Primitive(A.Void)) ; func_name = Some(curr_func_name ^ "test") ; in_test_func = true ; in_struct_method = false ; struct_name = None } in
  Some(check_function_body func_with_test new_env2)
    | None -> None
    )
```

```
  in

  let tmp:(S.sfunc_decl) = {S.styp = funct.A.typ; S.sfname = funct.A.fname; S.
    sformals = funct.A.formals; S.svdecls = funct.A.vdecls ; S.sbody =
    body_with_env; S.stests = (sast_func_with_test) ; S.sstruc_method = funct.A.
    struc_method ; S.sincludes_func = funct.A.includes_func } in
  tmp

(* Entry point to check functions *)
let check_functions functions_with_env includes globals_add structs_add =
  let function_names = List.map (fun n -> fst n) functions_with_env in

  (check_function_names function_names);
  (check_function_not_print function_names);
  (check_prog_contains_main function_names);
  let sast_funcs = (List.map (fun n -> check_function_body (fst n) (snd n))
    functions_with_env) in
  (*let sprogram:(S.sprogram) = program_sast (globals_add, functions, structs_add)
     in *)
  let sast = (includes, globals_add, sast_funcs, (List.map struct_sast structs_add
     )) in
  sast
  (* Need to check function test + using code here *)

let check_includes includes =
  let headers = List.map (fun n -> snd n) includes in
  report_duplicate (fun n -> "duplicate header file " ^ n) headers;
  List.iter check_ends_in_jt headers;
  ()


(*****************************************************************)
(* Entry point for semantic checking. Input is Ast, output is Sast *)
(*****************************************************************)
let check (includes, globals, functions, structs) =
  let prog_env:environment = {scope = {parent = None ; variables = Hashtbl.create
    10 }; return_type = None; func_name = None ; in_test_func = false ;
    in_struct_method = false ; struct_name = None } in
  let _ = check_includes includes in
  let (structs_added, struct_methods) = check_structs structs in
  let globals_added = check_globals globals prog_env in
  let functions_with_env = List.map (fun n -> (n, prog_env)) functions in
  let methods_with_env = List.map (fun n -> let prog_env_in_struct:environment = {
    scope = {parent = None ; variables = Hashtbl.create 10 }; return_type = None;
    func_name = None ; in_test_func = false ; in_struct_method = true ; struct_name
     = Some(snd (List.hd n.A.formals)) }  in  (n, prog_env_in_struct))
    struct_methods in
  let sast = check_functions (functions_with_env @ methods_with_env) includes
    globals_added structs_added in
  sast
```

8.6 sast.ml

```
open Ast

type var_info = (string * typ)

type sexpr =
    SLit     of int
  | SString_lit of string
  | SChar_lit of char
  | SDouble_lit of float
  | SBinop   of sexpr * op * sexpr * typ
  | SUnop    of uop * sexpr * typ
  | SAssign  of sexpr * sexpr
  | SNoexpr
  | SId of string
  | SStruct_create of string
  | SStruct_access of string * string * int * typ
  | SPt_access of string * string * int * typ
  | SArray_create of int * prim
  | SArray_access of string * int * typ
  | SDereference of string * typ
  | SFree of string
  | SCall of string * sexpr list
  | SBoolLit of int
  | SNull of typ
  | SDubs

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SReturn of sexpr

type swith_using_decl = {
  suvdecls : bind list;
  sstmts : sstmt list;
}

type swith_test_decl = {
  sasserts : sstmt list;
  susing : swith_using_decl;
}

(* Node that describes a function *)
type sfunc_decl = {
  styp : typ;
  sfname : string;
  sformals : bind list;
  svdecls  : bind list;
  sbody  :   sstmt list;
  stests   :   sfunc_decl option;
  sstruc_method : bool;
  sincludes_func : bool;
}

(* Node that describes a given struct *)
```

```
type sstruct_decl = {
  ssname    : string;
  sattributes  : bind list;
}

(* Root of tree. Our program is made up three things 1) list of global variables
    2) list of functions 3) list of struct definition *)
type sprogram = header list * bind list * sfunc_decl list * sstruct_decl list
```

8.7 codegen.ml

```
(* Code generation code. Converts a Sast into LLVM code*)

module L = Llvm
module A = Ast
module S = Sast
module C = Char
module StringMap = Map.Make(String)

let context = L.global_context ()
(* module is what is returned from this file aka the LLVM code *)
let main_module = L.create_module context "Jateste"
let test_module = L.create_module context "Jateste-test"

(* Defined so we don't have to type out L.i32_type ... every time *)
let i32_t = L.i32_type context
let i64_t = L.i64_type context
let i8_t = L.i8_type context
let i1_t = L.i1_type context
let d_t = L.double_type context
let void_t = L.void_type context
let str_t = L.pointer_type i8_t

(* Hash table of the user defined structs *)
let struct_types:(string, L.lltype) Hashtbl.t = Hashtbl.create 10
(* Hash table of global variables *)
let global_variables:(string, L.llvalue) Hashtbl.t = Hashtbl.create 50

(* Helper function that returns L.lltype for a struct. This should never fail as
    semantic checker should catch invalid structs *)
let find_struct_name name =
  try Hashtbl.find struct_types name
  with | Not_found -> raise(Exceptions.InvalidStruct name)

let rec index_of_list x l =
         match l with
              [] -> raise (Exceptions.InvalidStructField)
    | hd::tl -> let (_,y) = hd in if x = y then 0 else 1 + index_of_list x tl


let cut_string s l = let len = String.length s in
    if l >= len then raise (Exceptions.BugCatch "cut_string")
          else let string_len = len - l in String.sub s 0 string_len

(* Code to declare struct *)
let declare_struct s =
  let struct_t = L.named_struct_type context s.S.ssname in
  Hashtbl.add struct_types s.S.ssname struct_t


let prim_ltype_of_typ = function
    A.Int -> i32_t
  | A.Double -> d_t
  | A.Char -> i8_t
  | A.Void -> void_t
  | A.String -> str_t
  | A.Bool -> i1_t

```

```
let rec ltype_of_typ = function
  | A.Primitive(s) -> prim_ltype_of_typ s
  | A.Struct_typ(s) ->  find_struct_name s
  | A.Pointer_typ(s) -> L.pointer_type (ltype_of_typ s)
  | A.Array_typ(t,n) -> L.array_type (prim_ltype_of_typ t) n
      | _ -> void_t

let type_of_llvalue v = L.type_of v

let string_of_expr e =
  match e with
    S.SId(s) -> s
  | _  -> raise (Exceptions.BugCatch "string_of_expr")

(* Function that builds LLVM struct *)
let define_struct_body s =
  let struct_t = try Hashtbl.find struct_types s.S.ssname with | Not_found -> raise (Exceptions.BugCatch "defin_struct") in
  let attribute_types = List.map (fun (t, _) -> t) s.S.sattributes in
  let attributes = List.map ltype_of_typ attribute_types in
  let attributes_array = Array.of_list attributes in
  L.struct_set_body struct_t attributes_array false

(* Helper function to create an array of size i fille with l values *)
let array_of_zeros i l =
  Array.make i l

let default_value_for_prim_type t =
  match t with
      A.Int -> L.const_int (prim_ltype_of_typ t) 0
    | A.Double ->L.const_float (prim_ltype_of_typ t) 0.0
    | A.String ->L.const_string context ""
    | A.Char ->L.const_int (prim_ltype_of_typ t) 0
    | A.Void ->L.const_int (prim_ltype_of_typ t) 0
    | A.Bool ->L.const_int (prim_ltype_of_typ t) 0

(* Here we define and initailize global vars *)
let define_global_with_value (t, n) =
    match t with
      A.Primitive(p) ->
      (match p with
        A.Int -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n init main_module)
      | A.Double -> let init = L.const_float (ltype_of_typ t) 0.0 in (L.define_global n init main_module)
      | A.String -> let init = L.const_pointer_null (ltype_of_typ t) in (L.define_global n init main_module)
      | A.Void -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n init main_module)
      | A.Char -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n init main_module)
      | A.Bool -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n init main_module)
     )
    | A.Struct_typ(s) -> let init = L.const_named_struct (find_struct_name s) [||] in (L.define_global n init main_module)

```

```
    | A.Pointer_typ(_) ->let init = L.const_pointer_null (ltype_of_typ t) in (L.define_global n init main_module)

    | A.Array_typ(p,i) ->let init = L.const_array (prim_ltype_of_typ p) (array_of_zeros i (default_value_for_prim_type ((p)))) in (L.define_global n init main_module)

    | A.Func_typ(_) ->let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n init main_module)
    | A.Any -> raise (Exceptions.BugCatch "define_global_with_value")


(* Where we add global variabes to global data section *)
let define_global_var (t, n) =
    match t with
      A.Primitive(_) -> Hashtbl.add global_variables n (define_global_with_value (t,n))
    | A.Struct_typ(_) -> Hashtbl.add  global_variables n (define_global_with_value (t,n))
    | A.Pointer_typ(_) -> Hashtbl.add  global_variables n (define_global_with_value (t,n))
    | A.Array_typ(_,_) -> Hashtbl.add global_variables n (define_global_with_value (t,n))
    | A.Func_typ(_) -> Hashtbl.add global_variables n (L.declare_global (ltype_of_typ t) n main_module)
    | A.Any -> raise (Exceptions.BugCatch "define_global_with_value")


(* Translations functions to LLVM code in text section  *)
let translate_function functions the_module =

(* Here we define the built in print function *)
let printf_t = L.var_arg_function_type i32_t [||] in
let printf_func = L.declare_function "printf" printf_t the_module in


(* Here we iterate through Ast.functions and add all the function names to a HashMap *)
  let function_decls =
    let function_decl m fdecl =
    let name = fdecl.S.sfname
          and formal_types =
              Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.S.sformals)
              in let ftype = L.function_type (ltype_of_typ fdecl.S.styp) formal_types in
              StringMap.add name (L.define_function name ftype the_module, fdecl) m in
        List.fold_left function_decl StringMap.empty functions in

    (* Create format strings for printing *)
    let (main_function,_) = try StringMap.find "main" function_decls with | Not_found -> raise (Exceptions.BugCatch "function decls") in
    let builder = L.builder_at_end context (L.entry_block main_function) in
    (*let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in *)
    let str_format_str = L.build_global_stringptr "%s\n" "fmt_string" builder in
    let int_format_str = L.build_global_stringptr "%d\n" "fmt_int" builder in
    let float_format_str = L.build_global_stringptr "%f\n" "fmt_float" builder in

```

```
(* Method to build body of function *)
  let build_function_body fdecl =
  let (the_function, _) = try StringMap.find fdecl.S.sfname function_decls with |
    Not_found -> raise (Exceptions.BugCatch "build function body") in
  (* builder is the LLVM instruction builder *)
  let builder = L.builder_at_end context (L.entry_block the_function) in


  (* This is where we push local variables onto the stack and add them to a local
    HashMap*)
  let local_vars =
    let add_formal m(t, n) p = L.set_value_name n p;
    let local = L.build_alloca (ltype_of_typ t) n builder in
    ignore (L.build_store p local builder);
    StringMap.add n local m in

    let add_local m (t, n) =
          let local_var = L.build_alloca (ltype_of_typ t) n builder
          in StringMap.add n local_var m in

  (* This is where we push formal arguments onto the stack *)
  let formals = List.fold_left2 add_formal StringMap.empty fdecl.S.sformals
          (Array.to_list (L.params the_function)) in
          List.fold_left add_local formals fdecl.S.svdecls in


  (* Two places to look for a variable 1) local HashMap 2) global HashMap *)
  let find_var n = try StringMap.find n local_vars
    with Not_found -> try Hashtbl.find global_variables n
    with Not_found -> raise (Failure ("undeclared variable " ^ n))
    in

  let print_format_typ t =
      (match t with
        A.Primitive(A.Int) -> int_format_str
       | A.Primitive(A.Double) -> float_format_str
       | A.Primitive(A.String) -> str_format_str
       | A.Primitive(A.Char) -> int_format_str
       | A.Primitive(A.Bool) -> int_format_str
       | _ -> raise (Exceptions.BugCatch "print format")
      )
      in

  (* Format to print given arguments in print(...) *)
  let rec print_format e =
    (match e with
      (S.SString_lit(_)) -> str_format_str
    | (S.SLit(_)) -> int_format_str
    | (S.SDouble_lit(_)) -> float_format_str
    | S.SBinop(l,_,_,_) -> print_format l
    | S.SUnop(op,e,_) ->
      (match op with
        A.Neg -> print_format e
        | _ -> raise (Exceptions.BugCatch "print format")
      )
    | S.SAssign(_,_) -> raise (Exceptions.InvalidPrintFormat)
    | S.SNoexpr -> raise (Exceptions.InvalidPrintFormat)
    | (S.SId(i)) -> let i_value = find_var i in
      let i_type = L.type_of i_value in
```

```
      let string_i_type = L.string_of_lltype i_type in
      (match string_i_type with
        "i32*" -> int_format_str
      | "i1*" -> int_format_str
      | "i8**" -> str_format_str
      | "float*" -> float_format_str
      | "double*" -> float_format_str
      | _ -> raise (Exceptions.InvalidPrintFormat)
      )
    | S.SStruct_access(_,_,_,t) -> print_format_typ t
    | S.SPt_access(_,_,_,t) -> print_format_typ t
    | S.SArray_create(_,_) -> raise (Exceptions.InvalidPrintFormat)
    | S.SArray_access(_,_,t) -> print_format_typ t
    | S.SDereference(_,t) -> print_format_typ t
    | S.SFree(_) -> raise (Exceptions.InvalidPrintFormat)
    | S.SCall(f,_) ->let (_, fdecl) = try StringMap.find f function_decls with | Not_found -> raise (Exceptions.BugCatch "print format") in
      let tmp_typ = fdecl.S.styp in print_format_typ tmp_typ
    | S.SBoolLit(_) -> int_format_str
    | S.SNull(_) -> raise (Exceptions.InvalidPrintFormat)
    | _ -> raise (Exceptions.InvalidPrintFormat)
    )
    in

  (* Returns address of i. Used for lhs of assignments *)
  let rec addr_of_expr i builder=
  match i with
    S.SLit(_) -> raise Exceptions.InvalidLhsOfExpr
  | S.SString_lit (_) -> raise Exceptions.InvalidLhsOfExpr
  | S.SChar_lit (_) -> raise Exceptions.InvalidLhsOfExpr
  | S.SId(s) -> find_var s
  | S.SBinop(_,_,_,_) ->raise (Exceptions.UndeclaredVariable("Unimplemented addr_of_expr"))
  | S.SUnop(_,e,_) -> addr_of_expr e builder
  | S.SStruct_access(s,_,index,_) -> let tmp_value = find_var s in
      let deref = L.build_struct_gep tmp_value index "tmp" builder in deref
  | S.SPt_access(s,_,index,_) -> let tmp_value = find_var s in
      let load_tmp = L.build_load tmp_value "tmp" builder in
      let deref = L.build_struct_gep load_tmp index "tmp" builder in deref
  | S.SDereference(s,_) -> let tmp_value = find_var s in
      let deref = L.build_gep tmp_value [|L.const_int i32_t 0|] "tmp" builder in L.build_load deref "tmp" builder

  | S.SArray_access(ar,index, t) -> let tmp_value = find_var ar in
    (match t with
      A.Array_typ(_) -> let deref = L.build_gep tmp_value [|L.const_int i32_t 0 ; L.const_int i32_t index|] "arrayvalueaddr" builder in deref
    | A.Pointer_typ(_) -> let loaded_value = L.build_load tmp_value "tmp" builder in
      let deref = L.build_gep loaded_value [|L.const_int i32_t 0 ; L.const_int i32_t index|] "arrayvalueaddr" builder in deref
    | _ -> raise Exceptions.InvalidArrayAccess)
  | _ -> raise (Exceptions.UndeclaredVariable("Invalid LHS of assignment"))

  in
  let add_terminal builder f =
          match L.block_terminator (L.insertion_block builder) with
            Some _ -> ()
          | None -> ignore (f builder) in
```

```
(* This is where we build LLVM expressions *)
let rec expr builder = function
  S.SLit l -> L.const_int i32_t l
| S.SString_lit s -> let temp_string = L.build_global_stringptr s "str" builder
  in temp_string
| S.SChar_lit c -> L.const_int i8_t (C.code c)
| S.SDouble_lit d -> L.const_float d_t d
| S.SBinop (e1, op, e2,t) ->
  let e1' = expr builder e1
  and e2' = expr builder e2 in
  (match t with
    A.Primitive(A.Int) | A.Primitive(A.Char) -> (match op with
    A.Add -> L.build_add
  | A.Sub -> L.build_sub
  | A.Mult -> L.build_mul
  | A.Div -> L.build_sdiv
  | A.Mod -> L.build_srem
  | A.Equal -> L.build_icmp L.Icmp.Eq
  | A.Neq -> L.build_icmp L.Icmp.Ne
  | A.Less -> L.build_icmp L.Icmp.Slt
  | A.Leq -> L.build_icmp L.Icmp.Sle
  | A.Greater -> L.build_icmp L.Icmp.Sgt
  | A.Geq -> L.build_icmp L.Icmp.Sge
  | A.And -> L.build_and
  | A.Or -> L.build_or
  | _ -> raise (Exceptions.BugCatch "Prim Binop")
  )e1' e2' "add" builder
  | A.Primitive(A.Double) ->
  (match op with
    A.Add -> L.build_fadd
  | A.Sub -> L.build_fsub
  | A.Mult -> L.build_fmul
  | A.Div -> L.build_fdiv
  | A.Mod -> L.build_frem
  | A.Equal -> L.build_fcmp L.Fcmp.Oeq
  | A.Neq -> L.build_fcmp L.Fcmp.One
  | A.Less -> L.build_fcmp L.Fcmp.Olt
  | A.Leq -> L.build_fcmp L.Fcmp.Ole
  | A.Greater -> L.build_fcmp L.Fcmp.Ogt
  | A.Geq -> L.build_fcmp L.Fcmp.Oge
  | A.And -> L.build_and
  | A.Or -> L.build_or
  | _ -> raise (Exceptions.BugCatch "Double Binop")
  ) e1' e2' "addfloat" builder
  | A.Primitive(A.Bool) ->
  (
  match op with
    A.And -> L.build_and
  | A.Or -> L.build_or
  | A.Equal -> L.build_icmp L.Icmp.Eq
  | _ -> raise (Exceptions.BugCatch "Binop")
  ) e1' e2' "add" builder
  | A.Pointer_typ(_) ->
    (match op with
      A.Equal -> L.build_is_null
    | A.Neq -> L.build_is_not_null
    | _ -> raise (Exceptions.BugCatch "Binop")
    )e1' "add" builder
```

```
    | _ -> raise (Exceptions.BugCatch "Binop"))

  | S.SUnop(u,e, t) ->
      (match u with
          A.Neg -> let e1 = expr builder e in
        (match t with
          A.Primitive(A.Int) ->  L.build_neg e1 "neg" builder
        | A.Primitive(A.Double) -> L.build_fneg e1 "neg" builder
        | _ -> raise (Exceptions.BugCatch "expr builder")
        )
        | A.Not -> let e1 = expr builder e in L.build_not e1 "not" builder
        | A.Addr ->let iden = string_of_expr e in
              let lvalue = find_var iden in lvalue
      )
  | S.SAssign (l, e) -> let e_temp = expr builder e in
    ignore(let l_val = (addr_of_expr l builder) in  (L.build_store e_temp l_val
   builder)); e_temp
  | S.SNoexpr -> L.const_int i32_t 0
  | S.SId (s) -> L.build_load (find_var s) s builder
  | S.SStruct_create(s) -> L.build_malloc (find_struct_name s) "tmp" builder
  | S.SStruct_access(s,_,index,_) -> let tmp_value = find_var s in
      let deref = L.build_struct_gep tmp_value index "tmp" builder in
      let loaded_value = L.build_load deref "dd" builder in loaded_value
  | S.SPt_access(s,_,index,_) -> let tmp_value = find_var s in
      let load_tmp = L.build_load tmp_value "tmp" builder in
      let deref = L.build_struct_gep load_tmp index "tmp" builder in
      let tmp_value = L.build_load deref "dd" builder in tmp_value
  | S.SArray_create(i,p) -> let ar_type = L.array_type (prim_ltype_of_typ p) i in
   L.build_malloc ar_type "ar_create" builder
  | S.SArray_access(ar,index,t) -> let tmp_value = find_var ar in
    (match t with
      A.Pointer_typ(_) -> let loaded_value = L.build_load tmp_value "loaded"
   builder in
      let deref = L.build_gep loaded_value [|L.const_int i32_t 0 ; L.const_int
   i32_t index|] "arrayvalueaddr" builder in
      let final_value = L.build_load deref "arrayvalue" builder in final_value
    | A.Array_typ(_) -> let deref = L.build_gep tmp_value [|L.const_int i32_t 0 ;
   L.const_int i32_t index|] "arrayvalueaddr" builder in
      let final_value = L.build_load deref "arrayvalue" builder in final_value
    | _ -> raise Exceptions.InvalidArrayAccess)
  | S.SDereference(s,_) -> let tmp_value = find_var s in
      let load_tmp = L.build_load tmp_value "tmp" builder in
      let deref = L.build_gep load_tmp [|L.const_int i32_t 0|] "tmp" builder in
          let tmp_value2 = L.build_load deref "dd" builder in tmp_value2

  | S.SFree(s) -> let tmp_value = L.build_load (find_var s) "tmp" builder in L.
   build_free (tmp_value) builder
  | S.SCall("print", [e]) | S.SCall("print_int", [e])-> L.build_call printf_func
   [|(print_format e); (expr builder e) |] "printresult" builder
  | S.SCall(f, args) -> let (def_f, fdecl) = try StringMap.find f function_decls
   with | Not_found -> raise (Exceptions.BugCatch f) in
           let actuals = List.rev (List.map (expr builder) (List.rev args)) in
          let result = (match fdecl.S.styp with A.Primitive(A.Void) -> "" | _ -> f
   ^ "_result") in L.build_call def_f (Array.of_list actuals) result builder
  | S.SBoolLit(b) -> L.const_int i1_t b
  | S.SNull(t) -> L.const_null (ltype_of_typ t)
  | S.SDubs -> let tmp_call = S.SCall("print", [(S.SString_lit("dubs!"))]) in expr
     builder tmp_call
  in
```

```
  (* This is where we build the LLVM statements *)
  let rec stmt builder = function
    S.SBlock b -> List.fold_left stmt builder b
  | S.SExpr e -> ignore (expr builder e); builder


  | S.SIf(pred, then_stmt, else_stmt) ->
    (*let curr_block = L.insertion_block builder in *)
    (* the function (of type llvalue that we are currently in *)
    let bool_val = expr builder pred in
    let merge_bb = L.append_block context "merge" the_function in
    (* then block *)
    let then_bb = L.append_block context "then" the_function in

    add_terminal (stmt (L.builder_at_end context then_bb) then_stmt) (L.build_br
   merge_bb);
    (* else block*)
    let else_bb = L.append_block context "else" the_function in
    add_terminal (stmt (L.builder_at_end context else_bb) else_stmt) (L.build_br
   merge_bb);
    ignore (L.build_cond_br bool_val then_bb else_bb builder);
    L.builder_at_end context merge_bb
  | S.SWhile(pred,body_stmt) ->
    let pred_bb = L.append_block context "while" the_function in
    ignore (L.build_br pred_bb builder);
    let body_bb = L.append_block context "while_body" the_function in
    add_terminal (stmt (L.builder_at_end context body_bb) body_stmt) (L.build_br
   pred_bb);
    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = expr pred_builder pred in
    let merge_bb = L.append_block context "merge" the_function in
    ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
    L.builder_at_end context merge_bb

  | S.SFor(e1,e2,e3,s) -> ignore(expr builder e1); let tmp_stmt = S.SExpr(e3) in
      let tmp_block = S.SBlock([s] @ [tmp_stmt]) in
      let tmp_while = S.SWhile(e2, tmp_block) in stmt builder tmp_while
  | S.SReturn r -> ignore (match fdecl.S.styp with
              A.Primitive(A.Void) -> L.build_ret_void builder
            | _ -> L.build_ret (expr builder r) builder); builder
  in

  (* Build the body for this function *)
  let builder = stmt builder (S.SBlock fdecl.S.sbody) in

  add_terminal builder (match fdecl.S.styp with
          A.Primitive(A.Void) -> L.build_ret_void
        | _ -> L.build_ret (L.const_int i32_t 0) )
  in

(* Here we go through each function and build the body of the function *)
List.iter build_function_body functions;
the_module

(* Create a main function in test file - main then calls the respective tests *)
let test_main functions =
```

```
  let tests = List.fold_left (fun l n -> (match n.S.stests with Some(t) -> l @ [t]  | None -> l)) [] functions in
  let names_of_test_calls = List.fold_left (fun l n -> l @ [(n.S.sfname)]) [] tests in
  let print_stars = S.SExpr(S.SCall("print", [S.SString_lit("************")])) in
  let sast_calls = List.fold_left (fun l n -> l @ [S.SExpr(S.SCall("print",[S.SString_lit((cut_string n 4) ^ " results:")]))] @ [S.SExpr(S.SCall(n,[]))]@ [print_stars] ) [] names_of_test_calls in
  let print_stmt = [S.SExpr(S.SCall("print",[S.SString_lit("TEST RESULTS!")]))]@[print_stars] in
  let tmp_main:(S.sfunc_decl) = { S.styp = A.Primitive(A.Void); S.sfname = "main"; S.sformals = []; S.svdecls = []; S.sbody = print_stmt@sast_calls; S.stests = None; S.sstruc_method = false ; S.sincludes_func = false } in tmp_main


let func_builder f b =
  (match b with
    true -> let tests = List.fold_left (fun l n -> (match n.S.stests with Some(t) -> l @ [n] @ [t]  | None -> if (n.S.sstruc_method = false && n.S.sincludes_func = false) then (l) else (l@[n]))) [] f in (tests @ [(test_main f)])
  | false -> f
  )

(*********************************************************)
(* Entry point for translating Sast.program to LLVM module *)
(*********************************************************)
let gen_llvm (_, input_globals, input_functions, input_structs) gen_tests_bool =
  let _ = List.iter declare_struct input_structs in
  let _ = List.iter define_struct_body input_structs in
  let _ = List.iter define_global_var input_globals in
  let the_module = (match gen_tests_bool with true -> test_module | false -> main_module) in
  let _ = translate_function (func_builder input_functions gen_tests_bool) the_module in
  the_module
```

8.8 exceptions.ml

```
(* Program structure exceptions *)
exception MissingMainFunction

exception InvalidHeaderFile of string

(* Struct exceptions*)
exception InvalidStruct of string
exception InvalidStructField
exception InvalidStructMethodCall

(* Array exceptions*)
exception InvalidArrayVariable
exception InvalidArrayAccess
exception InvalidArrayType

(* Variable exceptions*)
exception UndeclaredVariable of string

(* Expression exceptions *)
exception InvalidExpr of string
exception InvalidBooleanExpression
exception IllegalAssignment
exception InvalidFunctionCall of string
exception InvalidArgumentsToFunction of string
exception InvalidFree of string
exception InvalidPointerDereference
exception InvalidDereference
exception InvalidPointerAccess
exception NotBoolExpr
exception InvalidLhsOfExpr
exception InvalidNegativeType

(* Print exceptions *)
exception InvalidPrintCall
exception InvalidPrintFormat

(* Statement exceptions*)
exception InvalidReturnType of string

(* Bug catcher *)
exception BugCatch of string

(* Input *)
exception IllegalInputFormat
exception IllegalArgument of string

(* Test cases *)
exception InvalidTestAsserts
exception InvalidAssert of string
```