

PLT 4115 Final Report: **JaTesté**

Andrew Grant
amg2215@columbia.edu

Jemma Losh
jal2285@columbia.edu

Jared Weiss
jbw2140@columbia.edu

Jake Weissman
jdw2159@columbia.edu

May 4, 2016

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 4 |
| 1.1 | Motivation | 4 |
| 1.2 | Language Description | 4 |
| 1.3 | Running the JaTeste Compiler | 4 |
| 2 | Sample Programs | 5 |
| 3 | Lexical Conventions | 6 |
| 3.1 | Identifiers | 6 |
| 3.2 | Keywords | 6 |
| 3.3 | Constants | 6 |
| 3.3.1 | Integer Constants | 6 |
| 3.3.2 | Double Constants | 6 |
| 3.3.3 | Character Constants | 6 |
| 3.3.4 | String Constants | 7 |
| 3.4 | Operators | 7 |
| 3.5 | White Space | 7 |
| 3.6 | Comments | 7 |
| 3.7 | Separators | 7 |
| 4 | Data Types | 7 |
| 4.1 | Primitives | 7 |
| 4.1.1 | Integer Types | 8 |
| 4.1.2 | Double Types | 8 |
| 4.1.3 | Character Type | 8 |
| 4.1.4 | String Type | 9 |
| 4.2 | Structures | 9 |
| 4.2.1 | Defining Structures | 9 |
| 4.2.2 | Initializing Structures | 9 |
| 4.2.3 | Accessing Structure Members | 10 |
| 4.3 | Arrays | 10 |
| 4.3.1 | Defining Arrays | 10 |
| 4.3.2 | Initializing Arrays | 10 |
| 4.3.3 | Accessing Array Elements | 10 |
| 5 | Expressions and Operators | 11 |
| 5.1 | Expressions | 11 |
| 5.2 | Assignment Operators | 11 |
| 5.3 | Incrementing and Decrementing | 12 |
| 5.4 | Arithmetic Operators | 12 |
| 5.5 | Comparison Operators | 13 |
| 5.6 | Logical Operators | 13 |
| 5.7 | Operator Precedence | 13 |
| 5.8 | Order of Evaluation | 14 |
| 6 | Statements | 14 |
| 6.1 | If Statement | 14 |
| 6.2 | While Statement | 14 |
| 6.3 | For Statement | 15 |
| 6.4 | Code Blocks | 15 |
| 6.5 | Return Statement | 16 |

| | | |
|-----------|--|-----------|
| 7 | Functions | 16 |
| 7.1 | Function Declarations | 16 |
| 7.2 | Function Definitions | 16 |
| 7.3 | Calling Functions | 17 |
| 7.4 | Function Parameters | 18 |
| 7.5 | Recursive Functions | 18 |
| 7.6 | Function Test Cases | 18 |
| 8 | Project Plan | 21 |
| 8.1 | Project Timeline | 21 |
| 8.2 | Workflow | 21 |
| 8.3 | Github Stats | 21 |
| 8.4 | Team Responsibilities | 21 |
| 8.5 | Software Development Environment | 21 |
| 9 | Architecture | 22 |
| 9.1 | Block Diagram | 22 |
| 9.2 | The Compiler | 22 |
| 9.3 | The Scanner | 22 |
| 9.4 | The Parser | 22 |
| 9.5 | The Semantic Checker | 22 |
| 9.6 | The Code Generator | 22 |
| 9.7 | Supplementary Code | 22 |
| 10 | Testing | 23 |
| 10.1 | Test Plan | 23 |
| 10.2 | Test Suite Log | 24 |
| 10.3 | Test Automation | 26 |
| 10.4 | Tests | 27 |
| 11 | Conclusion | 62 |
| 11.1 | Lessons Learned | 62 |
| 11.1.1 | Andrew | 62 |
| 11.1.2 | Jemma | 62 |
| 11.1.3 | Jared | 62 |
| 11.1.4 | Jake | 62 |
| 12 | Code | 63 |
| 12.1 | scanner.mll | 64 |
| 12.2 | ast.ml | 71 |
| 12.3 | semant.ml | 73 |
| 12.4 | sast.ml | 87 |
| 12.5 | codegen.ml | 89 |
| 12.6 | myprinter.ml | 98 |
| 12.7 | exceptions.ml | 99 |
| 12.8 | jatest.ml | 100 |

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 require the user to explicitly define test cases for any function that is written in order to compile and execute 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 new test files.

1.2 Language Description

JaTesté is mostly a functional language with a syntax quite similar to C. The details of our language usage is provided in the rest of the document.

1.3 Running the JaTeste Compiler

The JaTesté compiler will generate a test file based on the input program. At compile time, two files will be generated: (1) an executable file, and (2) an executable test file with all the relevant test cases. This allows the user to continue with his or her normal work flow and minimize interference from the compiler, while at the same time providing a robust test file to fully test one's program.

When inside the src folder, type `make all` to generate the jateste executable. To run type `./jatests.native [optional -options] <source file>`

The optional -options are:

- "-t" Compile with test
- "-l" Scan only
- "-p" Parse only
- "-se" SAST
- "-ast" AST

2 Sample Programs

Language Reference Manual

3 Lexical Conventions

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

3.1 Identifiers

Identifiers are used to name a variable, a function, or other types of data. 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.

```
ID = "(['a'-'z' 'A'-'Z'] | '_' ) (['a'-'z' 'A'-'Z'] | ['0'-'9'] | '_' ) *"
```

3.2 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:

```
int, char, double, struct, if, else, for, while, with test, using, func, return, string,  
int*, char*, struct*, double*, new, int[], char[], double[]
```

3.3 Constants

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

3.3.1 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+
```

3.3.2 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. The whole part or the fractional part may be omitted, but not both. The regular expression for a double is as follows:

```
double = int | digit*['.']digit+ | digit+['.']digit*
```

3.3.3 Character Constants

Character constants hold a single character and are enclosed in single quotes. They are stored in a variable of type char. Character constants that are preceded with a backslash have special meaning. The regex for a character is as follows:

```
char = ['a' - 'z' 'A' - 'Z']
```

3.3.4 String Constants

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

```
string = char*
```

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

3.4 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 chapter 4.

3.5 White Space

Whitespace 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' ]"
```

3.6 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 whitespace.

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

3.7 Separators

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

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

4 Data Types

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

4.1 Primitives

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

4.1.1 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.

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

The grammar that recognizes an integer declaration is:

```
typ ID
```

The grammar that recognizes an integer initialization is:

```
ID ASSIGN expr
```

4.1.2 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.

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

The grammar that recognizes a double declaration is:

```
typ ID
```

The grammar that recognizes a double initialization is:

```
ID ASSIGN expr
```

4.1.3 Character Type

The character type is an 8 bit value that is used to hold a single character. The keyword `char` is used 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.

```
1 char a;  
2 a = 'h';
```

The grammar that recognizes a char declaration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```


4.1.4 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, this cannot be done in one step.

```
1 string a;  
2 a = "hello";
```

The grammar that recognizes a char declaration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

4.2 Structures

The structure data type is a user-defined collection of primitive types and other structure data types. 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:

4.2.1 Defining Structures

```
1 struct person = {  
2   string name;  
3   int age;  
4   int height;  
5 };  
6  
7 struct manager = {  
8   struct person name;  
9   int salary;  
10 };
```

Here we have defined two structs, the first being of type `struct person` and the second of type `struct manager`. The grammar that recognizes defining a structure is as follows:

```
STRUCT STRING_LITERAL ASSIGN LBRACE vdecl_list RBRACE
```

4.2.2 Initializing Structures

To create a structure, the new keyword is used as follow:

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

```
NEW STRUCT ID
```

Here, we create two variables `yahoo_manager` and `sam`. The first is of type “struct manager”, and the second is of type “struct person”.

4.2.3 Accessing Structure Members

To access structs and modify its variables, a period following by the variable name is used:

```
1 yahoo_manager.name = sam;  
2 yahoo_manager.age = 45;  
3 yahoo_manager.salary = 65000;
```

Ultimately, all structures are backed by some collection of primitives. For example, the first structure, “struct manager”, is made up of another struct and an int. Since “struct person” is made up of two ints, “struct manager” is really just made up of three ints. Note accessing members of structs is actually just another expression and thus always returns some value.

```
expr DOT expr
```

4.3 Arrays

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

4.3.1 Defining Arrays

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

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

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

4.3.2 Initializing Arrays

An array can be initialized by listing the element values separated by commas and surrounded by brackets. Here is an example:

```
1 arr = { 0, 1, 2, 3, 4 };
```

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

4.3.3 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:

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

Accessing arrays is simply an expression:

```
expr LBRACKET INT_LITERAL RBRACKET
```

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.

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

5 Expressions and Operators

5.1 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:

```
1 35 - 6;  
2 foo(42) * 10;  
3 8 - (9 / (2 + 1) );
```

The grammar for expressions is:

```
expr:  
expr:  
    INT_LITERAL  
    | ID  
    | expr PLUS expr  
    | expr MINUS expr  
    | expr TIMES 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 LBRACKET INT_LITERAL RBRACKET  
    | NEW prim_typ LBRACKET INT_LITERAL RBRACKET  
    | NEW STRUCT ID  
    | ID LPAREN actual_opts_list RPAREN
```

5.2 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 named variable that is being declared and initialized in this assignment. Examples include:

```
1 int x;  
2 x = 5;  
3 float y;  
4 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).

5.3 Incrementing and Decrementing

The following operators can also be used for variations of assignment:

- `+=` increments the left hand side by the result of the right hand side
- `-=` decrements the left hand side by the result of the right hand side

The `++` operator is used to increment and the `--` operator is used to decrement a value. If the operator is placed before a value it will be incremented / decremented first, then it will be evaluated. If the operator is placed following a value, it will be evaluated with its original value and then incremented / decremented.

5.4 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
```

5.5 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 |

5.6 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
```

5.7 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 TIMES DIVIDE MODULO
%right EXPO
%right NOT NEG AMPERSAND
%right RBRACKET
%left LBRACKET
%right DOT
```

5.8 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.

6 Statements

Statements include: **if**, **while**, **for**, **return**, 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

```

6.1 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, there is no ambiguity when it comes to which if-else clauses match.

```

1  if (x == 42) {
2      print("Gotcha");
3  }
4  else if (x > 42) {
5      print("Sorry, too big");
6  }
7  else {
8      print("I'll allow it");
9  }

```

The grammar that recognizes an if statement is as follows:

```

IF LPAREN expr RPAREN stmt ELSE stmt
IF LPAREN expr RPAREN stmt \%prec NOELSE

```

6.2 While Statement

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

```

1  /* Below code prints "Hey there" 10 times */
2  int x = 0;
3  while (x < 10) {
4      print("Hey there");
5      x++;
6  }

```

The grammar that recognizes a while statement is as follows:

```

WHILE LPAREN expr RPAREN stmt

```

6.3 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:

```

1  /* This will run as long as i is less than 100
2  i will be incremented on each iteration of the loop */

```

```

3  for (int i = 0; i < 100; i++) {
4      /* do something */
5  }
6
7  /* i can also be declared or initialized outside of the for loop */
8  int i;
9  for (i = 0; i < 100; i += 2) {
10     /* code block */
11 }

```

The grammar that recognizes a for statement is as follows:

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

6.4 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:

```

1  int x = 42;
2  if (x == 42) {
3      /* the following three lines are executed */
4      print("Hey");
5      x++;
6      print("Bye");
7  }

```

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.

6.5 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 declaration. Return can be used as follows:

```

1  /* The function trivially returns the input int value */
2  func int someValue(int x) {
3      return x;
4  }

```

The grammar that recognizes a return statement is as follows:

```
RETURN SEMI
RETURN expr SEMI
```

Note that functions can be declared as returning void; this is done as follows:

```
1  return ;
```

This adheres to the expectation that all functions return something.

7 Functions

Functions allow you to group snippets of code together that can subsequently be called from other parts of your program, depending on scope. Functions are global, unless they are prepended with the keyword “private”. While not necessary, it is encouraged that you declare functions before defining them. Functions are usually declared at the top of the file they’re defined in. Functions that aren’t declared can only be called after they have been defined.

7.1 Function Declarations

The keyword “func” is used to declare a function. A return type is also required using keyword “return”; if your function doesn’t return anything then use keyword “void” instead. Functions are declared with or without parameters; if parameters are used, their types must be specified. A function can be defined with multiple, different parameters. Though a function can only have one return type, it can also be any data type, including void.

```
1 func int add(int a, int b); /* this functions has two int parameters as input and
   returns an int */
2 func void say_hi(); /* this function doesn't return anything nor takes any
   parameters */
3 func int isSam(string name, int age); /* this functions has two input parameters,
   one of type string and one of type int */
```

7.2 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 declare functions; but curly brackets are used to define what the function actually does. For example,

```
1 func int add(int a, int b); /* declaration */
2
3 func int add(int x, int y) /* definition */
4 {
5     return x + y;
6 }
```

fdecl:

FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE

This snippet of code first declares add, and then defines it. Declaring before defining is best practice. Importantly, functions can *not* reference global variables; that is, the only variables they can act on are formal parameters and local variables. For example:

```
1 func int add_to_a(int x); /* declaration */
2 int a = 10;
3 func int add_to_a(int x) /* definition */
4 {
5     return x + a; /* this is NOT allowed */
6 }
```

This code is no good because it relies on global variable “a”. Functions can only reference formal parameters and/or local variables.

7.3 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:

```
1 func int add(int a, int b); /* declaration */
2
3 func int add(int x, int y) /* definition */
4 {
5     return x + y;
6 }
7
8 add(); /* this is wrong and will not compile because add expects two ints as
        parameters */
```

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 and so can be used as part of other expressions. Functions are first class objects and so can be used anywhere a normal data type can be used. 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:

```
1 func int add_int(int a, int b); /* declaration */
2
3 func int add_int(int x, int y) /* definition */
4 {
5     return x + y;
6 }
7
8 func float add_float(float x, float y)
9 {
10     return x + y;
11 }
12
13 func int subtract(int x, int y)
14 {
15     return x - y;
16 }
17
18 int answer = subtract(add(10,10), 10); /* this is ok */
19 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 */
```

7.4 Function Parameters

Formal parameters can be any data type. Furthermore, they need not be of the same type. For example, the following is syntactically fine:

```
1 func void speak(int age, string name)
2 {
3     print_string ("My name is" + name + " and I am " + age);
4 }
```

```

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 `&`.

```

1 int* int_pt;
2 int a = 10;
3 int_pt = &a;

```

7.5 Recursive Functions

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

7.6 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. The keyword “with test” is used to define a test case as illustrated here:

```

1 func int add(int a, int b); /* declaration */
2
3 func int add(int x, int y) /* definition */
4 {
5     return x + y;
6 }
7 with test {
8     add(1,2) == 3;
9     add(-1, 1) == 0;
10 }
11 with test {
12     add(0,0) <= 0;
13     add(0,0) >= 0;
14 }

```

```

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 set of boolean expressions. Multiple boolean expressions can be defined, they just must be separated with semi-colons. As shown above, you can define separate test cases one after another too.

Snippets of code can also be used to set up a given test case’s environment using the “using” keyword. That is, “using” is used to define code that is executed right before the test case is run. Consider the following:

```

1 func void changeAge(struct person temp_person, int age)
2 {
3     temp_person.age = age;
4 }
5 with test {
6     sam.age == 11;
7 }
8 using {
9     struct person sam;
10    sam.age = 10;
11    changeAge(sam, 11);
12 }

```

```

FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE testdecl usingdecl
usingdecl:
    USING LBRACE vdecl_list stmt_list RBRACE

```

“using” is used to create a struct and then call function changeAge; it is setting up the environment for its corresponding test. Variables defined in the “using” section of code can safely be referenced in its 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 optional. As a result some test cases may contain “using” sections and others might not. As per convention, each “using” section will match up with its closest test case. For, example:

```

1
2 func int add(int x, int y) /* definition */
3 {
4     return x + y;
5 }
6 with test { /* variables a, b defined below are NOT in this test case's scope*/
7     add(1,2) == 3;
8     add(-1, 1) == 0;
9 }
10 with test { /* variables a and b ARE in this test case's scope */
11     add(a, b) == 20;
12 }
13 using {
14     int a = 10;
15     int b = 10;
16 }

```

As explained in the comments, the “using” section is matched up with the second test case. Test cases are compiled into a separate program which can subsequently be run. The program will run all test cases and output appropriate information.

8 Project Plan

We arranged weekly meetings with our designated TA, David Watkins, to discuss progress and ask questions about issues we encountered. David's feedback helped us to make crucial decisions along the way, and kept us heading in the right direction. After meeting with David, we would work together as a group each week. During these meetings we would split up the work, and often have two people working together doing paired programming.

8.1 Project Timeline

****not sure about this....**

- Proposal submitted
- LRM submitted, scanner and parser working
- AST
- 'Hello, World' working
- Semantic analyzer and SAST
- Code generator
- With Test working
- Libraries and additional features

8.2 Workflow

8.3 Github Stats

8.4 Team Responsibilities

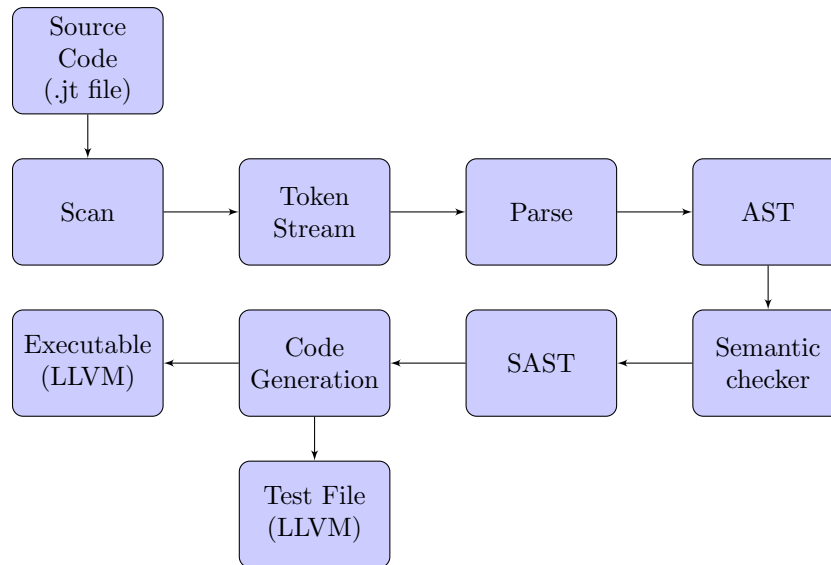
Our team didn't use rigid roles. The responsibilities became more fluid as the project progressed. In the beginning phases of the project, all members discussed and gave input on the components of the language and what features to include/omit. Throughout the project we exercised paired programming to write the code. In the end stages....

8.5 Software Development Environment

- Version Control
 - Git
- Languages
 - OCaml for parser, scanner, ast, semantic checker, sast, code generation
 - LaTeX for reports and documentation
- Text Editors
 - Vim

9 Architecture

9.1 Block Diagram



9.2 The Compiler

9.3 The Scanner

9.4 The Parser

9.5 The Semantic Checker

9.6 The Code Generator

9.7 Supplementary Code

10 Testing

10.1 Test Plan

10.2 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, etc. 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'
Makefile:23: warning: overriding recipe for target 'all-tests'
Makefile:15: warning: ignoring old recipe for target 'all-tests'
Testing 'hello-world.jt'
--> Test passed!
Testing 'global-scope.jt'
--> Test passed!
Testing 'test-func1.jt'
--> Test passed!
Testing 'test-func2.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 '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! ===== Compilation Tests 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-testcase2.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-arraypt1.jt'
--> Test passed!
Testing 'test-linkedlist1.jt'
--> Test passed!
Testing 'test-linkedlist2.jt'
--> Test passed!
Testing 'test-class1.jt'
--> Test passed!
Testing 'test-class2.jt'
--> Test passed!
Testing 'test-class3.jt'
--> Test passed!
===== All Tests Passed! =====
```


10.3 Test Automation

We had x tests in our test suite. In order to run all of the tests and see if they pass, type make all 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 Suit Log

10.4 Tests

class1-var-fail.jt

```
1 func int main()
2 {
3
4     struct house *my_house;
5     int price;
6     int vol;
7
8     my_house->set_price(100);
9     my_house->set_height(88);
10    my_house->set_width(60);
11    my_house->set_length(348);
12
13
14    return 0;
15 }
16
17 struct house {
18     int price;
19     int height;
20     int width;
21     int length;
22
23     method void set_price(int x)
24     {
25         pricee = x;
26     }
27
28     method void set_height(int x)
29     {
30         height = x;
31     }
32
33     method void set_width(int x)
34     {
35         width = x;
36     }
37
38     method void set_length(int x)
39     {
40         length = x;
41     }
42
43     method int get_price()
44     {
45         return price;
46     }
47
48     method int get_volumne()
49     {
50         int temp;
51         temp = height * width * length;
52         return temp;
53     }
54
55
56 };
```

class1-var-fail1.out

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

global-scope.jt

```
1 int global_var;
2
3 func int main()
4 {
5     int temp;
6     global_var = 10;
7     temp = 20;
8     my_print();
9     return 0;
10 }
11
12 func void my_print()
13 {
14     int temp;
15     if (global_var == 10) {
16         print("passed");
17     } else {
18         print("failed");
19     }
20
21     if (temp == 20) {
22         print("failed");
23     } else {
24         print("passed");
25     }
26
27 }
```

global-scope.out

```
1 passed
2 passed
```

hello-world.jt

```
1 func int main()  
2 {  
3   print("hello world!");  
4  
5   return 0;  
6 }
```

hello-world.out

```
1 hello world!
```

invalid-assignment-fail1.jt

```
1 func int main()  
2 {  
3     int a;  
4     char b;  
5     a = b;  
6 }
```

invalid-assignment-fail1.out

```
1 Scanned  
2 Parsed  
3 Fatal error: exception Exceptions.IllegalAssignment
```

local-var-fail.jt

```
1 func int main()
2 {
3     int main_var;
4     main_var = 10;
5     return 0;
6 }
7 func void do_something_sick()
8 {
9     int my_var;
10    main_var;
11 }
```

local-var-fail.out

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

no-main-fail.jt

```
1 func int my_main()  
2 {  
3     return 0;  
4 }
```

no-main-fail.out

```
1 Scanned  
2 Parsed  
3 Fatal error: exception Exceptions.MissingMainFunction
```


return-fail1.jt

```
1 func int main()
2 {
3     int a;
4     int b;
5     int c;
6     int d;
7
8     a = 1;
9     b = 2;
10    c = 3;
11
12    d = do_something(a,b,c);
13
14    return 0;
15    d = 10;
16 }
17
18 func int do_something(int x, int y, int z)
19 {
20     return x + y + z;
21 }
```

return-fail1.out

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

struct-access-fail1.jt

```
1 func int main()
2 {
3     struct car *toyota;
4
5     toyota = new struct car;
6
7     toyota->priice;
8
9     return 0;
10 }
11
12 struct car {
13     int price;
14     int year;
15     int weight;
16 };
```

struct-access-fail1.out

```
1 Scanned
2 Parsed
3 Fatal error: exception Exceptions.InvalidStructField
```

test-array1.jt

```
1 func int main()
2 {
3     int[10] arr;
4     int a;
5     int b;
6
7     a = 10;
8
9     arr[2] = 10;
10
11    b = arr[2];
12
13    if (b == 10) {
14        print("passed");
15    }
16
17    return 0;
18 }
```

test-array1.out

```
1 passed
```

test-arraypt1.jt

```
1 func int main()
2 {
3     int[10] *arr;
4     int a;
5     int b;
6     int c;
7
8     arr = new int[10];
9
10    arr[8] = 9;
11    arr[3] = 7;
12
13    c = arr[3];
14    b = arr[8];
15
16    if (c == 7) {
17        print("passed");
18        if (b == 9) {
19            print("passed");
20        }
21    }
22
23    return 0;
24 }
```

test-arraypt1.out

```
1 passed
2 passed
```

test-bool1.jt

```
1 func int main()
2 {
3     bool my_bool;
4     bool my_bool2;
5
6     my_bool = true;
7     my_bool2 = false;
8
9     if (my_bool || my_bool2) {
10         print("or passed");
11     }
12
13     if (my_bool && my_bool2) {
14     } else {
15         print("and passed");
16     }
17
18     return 0;
19 }
```

test-bool1.out

```
1 or passed
2 and passed
```

test-bool2.jt

```
1 func int main()
2 {
3     bool my_bool;
4
5     my_bool = false;
6
7     if (!my_bool) {
8         print("passed");
9     }
10
11     return 0;
12 }
```

test-bool2.out

```
1 passed
```

test-class1.jt

```
1 func int main()
2 {
3
4     struct square *p;
5     int area;
6     p = new struct square;
7     p->height = 7;
8     p->width = 9;
9     area = p->get_area();
10    print(area);
11    p->set_height(55);
12    p->set_width(3);
13    area = p->get_area();
14    print(area);
15
16
17    return 0;
18 }
19
20
21 struct square {
22     int height;
23     int width;
24
25     method int get_area()
26     {
27         int temp_area;
28         temp_area = height * width;
29         return temp_area;
30     }
31
32     method void set_height(int h) {
33         height = h;
34     }
35
36     method void set_width(int w) {
37         width = w;
38     }
39
40 };
```

test-class1.out

```
1 63
2 165
```

test-class2.jt

```
1 func int main()
2 {
3
4     struct house *my_house;
5     int price;
6     int vol;
7
8     my_house->set_price(100);
9     my_house->set_height(88);
10    my_house->set_width(60);
11    my_house->set_length(348);
12
13    price = my_house->get_price();
14    vol = my_house->get_volumne();
15
16    print(price);
17    print(vol);
18    return 0;
19 }
20
21 struct house {
22     int price;
23     int height;
24     int width;
25     int length;
26
27     method void set_price(int x)
28     {
29         price = x;
30     }
31
32     method void set_height(int x)
33     {
34         height = x;
35     }
36
37     method void set_width(int x)
38     {
39         width = x;
40     }
41
42     method void set_length(int x)
43     {
44         length = x;
45     }
46
47     method int get_price()
48     {
49         return price;
50     }
51
52     method int get_volumne()
53     {
54         int temp;
55         temp = height * width * length;
56         return temp;
57     }
```


58
59
60

```
};
```

test-class2.out

1
2

```
100  
1837440
```

test-class3.jt

```
1 func int main()
2 {
3
4     struct house *my_house;
5     struct condo *my_condo;
6     int a;
7     int b;
8     int c;
9
10    my_house = new struct house;
11    my_condo = new struct condo;
12
13    my_house->set_price(100);
14    my_condo->set_price(59);
15
16    a = my_house->get_price();
17    b = my_condo->get_price();
18
19    c = a - b;
20
21    print(c);
22
23
24
25    return 0;
26 }
27
28
29 struct house {
30     int price;
31
32     method void set_price(int x)
33     {
34         price = x;
35     }
36
37     method int get_price()
38     {
39         return price;
40     }
41
42 };
43
44
45 struct condo {
46     int price;
47
48     method void set_price(int x)
49     {
50         price = x;
51     }
52
53     method int get_price()
54     {
55         return price;
56     }
57 }
```

58

```
};
```

test-class3.out

1

```
41
```

test-for1.jt

```
1 func int main()
2 {
3     int i;
4     for (i = 0; i < 5; i = i + 1) {
5         print(i);
6     }
7     return 0;
8 }
```

test-for1.out

```
1 0
2 1
3 2
4 3
5 4
```

test-free1.jt

```
1 func int main()
2 {
3     struct person *sam;
4
5     sam = new struct person;
6
7     sam->age = 100;
8     sam->height = 100;
9     sam->gender = 100;
10
11     free(sam);
12
13     print("freed");
14
15
16     return 0;
17 }
18
19 struct person {
20     int age;
21     int height;
22     int gender;
23 };
```

test-for1.out

```
1 0
2 1
3 2
4 3
5 4
```

test-free1.jt

```
1 func int main()
2 {
3     struct person *sam;
4
5     sam = new struct person;
6
7     sam->age = 100;
8     sam->height = 100;
9     sam->gender = 100;
10
11     free(sam);
12
13     print("freed");
14
15
16     return 0;
17 }
18
19 struct person {
20     int age;
21     int height;
22     int gender;
23 };
```

test-free1.out

```
1 freed
```

test-func1.jt

```
1 func int main()
2 {
3     int sum;
4     sum = add(10,10);
5     if (sum == 20) {
6         print("passed");
7     } else {
8         print("failed");
9     }
10    return 0;
11 }
12
13 func int add(int x, int y)
14 {
15     return x + y;
16 }
```

test-func1.out

```
1 passed
```

test-func2.jt

```
1 int global_var;
2
3 func int main()
4 {
5     global_var = 0;
6     add_to_global();
7     if (global_var == 1) {
8         print("passed");
9     } else {
10        print("failed");
11    }
12
13 }
14
15 func void add_to_global()
16 {
17     global_var = global_var + 1;
18 }
```

test-func2.out

```
1 passed
```


test-func3.jt

```
1 func int main()
2 {
3     int a;
4     struct person *sam;
5     sam = new struct person;
6     update_age(sam);
7
8     a = sam->age;
9
10    if (a == 10) {
11        print("passed");
12    }
13
14    return 0;
15 }
16
17 func void update_age(struct person *p)
18 {
19     p->age = 10;
20 }
21
22 struct person {
23     int age;
24     int height;
25 };
```

test-func3.out

```
1 passed
```

test-gcd1.jt

```
1 func int main()
2 {
3     int a;
4     int b;
5     int c;
6
7     c = gcd(15,27);
8
9     if (c == 3) {
10         print("passed");
11     }
12
13     return 0;
14 }
15
16
17 func int gcd(int a, int b)
18 {
19     while (a != b) {
20         if (a > b) {
21             a = a - b;
22         }
23         else {
24             b = b - a;
25         }
26     }
27     return a;
28 }
```

test-gcd1.out

```
1 passed
```

test-lib1.jt

```
1
2 #include_jtlib <math.jt>
3
4 func int main()
5 {
6     int a;
7     int b;
8     int c;
9     a = 10;
10    b = 3;
11
12    c = add(a,b);
13    if (c == 13) {
14        print("passed");
15    }
16 }
```

test-lib1.out

```
1 passed
```

test-linkedlist1.jt

```
1 #include_jtlib <int_list.jt>
2
3 func int main()
4 {
5
6     struct int_list *my_list;
7     my_list = int_list_initialize();
8     int_list_insert(my_list,9);
9     int_list_insert(my_list,5);
10    int_list_insert(my_list,8);
11    int_list_insert(my_list,10);
12    int_list_insert(my_list,40);
13    int_list_insert(my_list,11);
14    int_list_insert(my_list,0);
15    int_list_insert(my_list,9);
16    int_list_insert(my_list,478);
17    int_list_print(my_list);
18
19    return 0;
20 }
```

test-linkedlist1.out

```
1 9
2 5
3 8
4 10
5 40
6 11
7 0
8 9
9 478
```

test-linkedlist2.jt

```
1 #include_jtlib <int_list.jt>
2
3 func int main()
4 {
5     struct int_list *header;
6     header = int_list_initialize();
7     int_list_insert(header,2);
8     int_list_insert(header,2);
9     int_list_insert(header,3);
10    int_list_insert(header,9);
11    int_list_insert(header,100);
12    int_list_insert(header,61);
13
14    if (int_list_contains(header,100) == true) {
15        print("passed contains test");
16    }
17
18    return 0;
19 }
```

test-linkedlist2.out

```
1 passed contains test
```

test-malloc1.jt

```
1 func int main()
2 {
3
4     struct person *andy;
5     int *a;
6     int b;
7     int zipcode;
8
9     andy = new struct person;
10
11     b = 25;
12
13     a = &b;
14
15     andy->age = *a;
16     andy->height = 100;
17     andy->zipcode = 10027;
18
19
20     zipcode = andy->zipcode;
21
22     if (zipcode == 10027) {
23         print("passed");
24     }
25
26     *a = andy->age;
27
28     if (*a == 25) {
29         print("word up");
30     }
31
32     return 0;
33 }
34
35
36
37 struct person {
38     int age;
39     int zipcode;
40     int height;
41 };
```

test-malloc1.out

```
1 passed
2 word up
```

test-pointer1.jt

```
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 }
```

test-pointer1.out

```
1 passed
```

test-struct-access1.jt

```
1 func int main()
2 {
3     struct house my_house;
4     int a;
5     int b;
6     int c;
7
8     a = 99;
9     my_house.price = a;
10    c = my_house.price;
11    my_house.age = 10;
12    b = my_house.age;
13
14    print(c);
15    print(b);
16
17    return 0;
18 }
19
20 struct house {
21     int price;
22     int age;
23 };
```

test-struct-access1.out

```
1 99
2 10
```


test-testcase1.jt

```
1 func int main()
2 {
3     int i;
4     i = add(2,3);
5     if (i == 5) {
6         print("passed");
7     }
8     return 0;
9 }
10
11
12 func int add(int x, int y)
13 {
14     return x + y;
15 } with test {
16     assert(a == a);
17 } using {
18     int a;
19     int b;
20     a = 10;
21     b = 5;
22 }
```

test-testcase1.out

```
1 passed
```

test-testcase2.jt

```
1 func int main()
2 {
3     int a;
4     int b;
5     int c;
6
7     a = 10;
8     b = 5;
9     c = 0;
10
11    a = b - c;
12    if (a == 5) {
13        print("passed");
14    }
15    return 0;
16 }
17
18
19 func int sub(int x, int y)
20 {
21     return x - y;
22 } with test {
23     assert(a == b - 5);
24 } using {
25     int a;
26     int b;
27     a = 5;
28     b = 10;
29 }
```

test-testcase2.out

```
1 passed
```

test-testcase3.jt

```
1 func int main()
2 {
3     int a;
4     int b;
5     int c;
6
7     a = 10;
8     b = 23;
9
10    c = max(a, b);
11
12    if (c == 23) {
13        print("passed");
14    }
15
16    return 0;
17 }
18
19 func int max(int x, int y)
20 {
21     if (x > y) {
22         return x;
23     }
24     return y;
25 } with test {
26     assert((max(a,b) == 10));
27 } using {
28     int a;
29     int b;
30     a = 10;
31     b = 9;
32 }
```

test-testcase3.out

```
1 passed
```

test-while1.jt

```
1 func int main()
2 {
3     int i;
4     int sum;
5     i = 0;
6     while (i < 10) {
7         print("looping");
8         i = i + 1;
9     }
10
11     return 0;
12 }
```

test-while1.out

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

11 Conclusion

11.1 Lessons Learned

11.1.1 Andrew

11.1.2 Jemma

11.1.3 Jared

11.1.4 Jake

12 Code

12.1 scanner.mll

```
1 { open Parser }
2
3 (* Regex shorthands *)
4 let digit = ['0' - '9']
5 let my_int = digit+
6 let double = (digit+) ['.'] digit+
7 let my_char = '''['a' - 'z' 'A' - 'Z']'''
8 let newline = '\n'
9 let my_string = '"' ([ 'a' - 'z' ] | [ ' ' ] | [ 'A' - 'Z' ] | [ '_' ] | [ '!' ] | [ ',' ] )+ '"'
10
11 rule token = parse
12   [ ' ' '\t' '\r' '\n' ] { token lexbuf } (* White space *)
13   | "/"* { comment lexbuf }
14   | '(' { LPAREN }
15   | ')' { RPAREN }
16   | '{' { LBRACE }
17   | '}' { RBRACE }
18   | ',' { COMMA }
19   | ';' { SEMI }
20   | '#' { POUND }
21
22   (*Header files *)
23   | "include_jtlib" { INCLUDE }
24
25   (* Operators *)
26   | "+" { PLUS }
27   | "-" { MINUS }
28   | "*" { STAR }
29   | "/" { DIVIDE }
30   | "%" { MODULO }
31   | "^" { EXPO }
32   | "=" { ASSIGN }
33   | "==" { EQ }
34   | "!=" { NEQ }
35   | "!" { NOT }
36   | "&&" { AND }
37   | "&" { AMPERSAND }
38   | "||" { OR }
39   | "<" { LT }
40   | ">" { GT }
41   | "<=" { LEQ }
42   | ">=" { GEQ }
43   | "[" { LBRACKET }
44   | "]" { RBRACKET }
45   | "." { DOT }
46   | "->" { POINTER_ACCESS }
47
48   (* Control flow *)
49   | "if" { IF }
50   | "else" { ELSE }
51   | "return" { RETURN }
52   | "while" { WHILE }
53   | "for" { FOR }
54   | "assert" { ASSERT }
55
56   (* Datatypes *)
57   | "void" { VOID }
```

```

58 | "struct"      { STRUCT }
59 | "method"     { METHOD }
60 | "double"     { DOUBLE }
61 | "int"        { INT }
62 | "char"       { CHAR }
63 | "string"     { STRING }
64 | "bool"       { BOOL }
65 | "true"       { TRUE }
66 | "false"      { FALSE }
67 | "func"       { FUNC }
68 | "new"        { NEW }
69 | "free"       { FREE }
70 | "NULL"       { NULL }
71 | "DUBS"       { DUBS }
72
73 (* Testing keywords *)
74 | "with test"  { WTEST }
75 | "using"     { USING }
76
77 | ['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' ]* as lxm { ID(lxm)}
78 | ['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' ]* ".jt" as lxm { INCLUDE_FILE(
    lxm) }
79 | my_int as lxm      { INT_LITERAL(int_of_string lxm)}
80 | double as lxm      { DOUBLE_LITERAL((float_of_string lxm)) }
81 | my_char as lxm     { CHAR_LITERAL(String.get lxm 1) }
82 | ''' {let buffer = Buffer.create 1 in STRING_LITERAL(string_find buffer lexbuf)
    }
83
84 | eof { EOF }
85 | _ as char { raise (Failure ("illegal character " ^
86   Char.escaped char))}
87
88
89 (* Whitespace*)
90 and comment = parse
91   "*/" { token lexbuf }
92   | _ { comment lexbuf }
93
94 and string_find buffer = parse
95   ''' {Buffer.contents buffer }
96   | _ as chr { Buffer.add_char buffer chr; string_find buffer lexbuf }
97
98 \newpage
99
100 \subsection{parser.mly}
101 %{ open Ast %}
102
103 /*
104   Tokens/terminal symbols
105 */
106 %token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA SEMI POUND INCLUDE
107 %token PLUS MINUS STAR DIVIDE ASSIGN NOT MODULO EXPO AMPERSAND
108 %token FUNC
109 %token WTEST USING STRUCT DOT POINTER_ACCESS METHOD
110 %token EQ NEQ LT LEQ GT GEQ AND OR TRUE FALSE
111 %token INT DOUBLE VOID CHAR STRING BOOL NULL
112 %token INT_PT DOUBLE_PT CHAR_PT STRUCT_PT
113 %token ARRAY
114 %token NEW FREE DUBS

```



```

115 %token RETURN IF ELSE WHILE FOR ASSERT
116
117 /*
118     Tokens with associated values
119 */
120 %token <int> INT_LITERAL
121 %token <float> DOUBLE_LITERAL
122 %token <char> CHAR_LITERAL
123 %token <string> STRING_LITERAL
124 %token <string> ID
125 %token <string> INCLUDE_FILE
126 %token EOF
127
128 /*
129     Precedence rules
130 */
131 %nonassoc NOELSE
132 %nonassoc ELSE
133 %right ASSIGN
134 %left OR
135 %left AND
136 %left EQ NEQ
137 %left LT GT LEQ GEQ
138 %left PLUS MINUS
139 %left STAR DIVIDE MODULO
140 %right EXPO
141 %right NOT NEG AMPERSAND
142 %right RBRACKET
143 %left LBRACKET
144 %right DOT POINTER_ACCESS
145
146 /*
147     Start symbol
148 */
149
150 %start program
151
152 /*
153     Returns AST of type program
154 */
155
156 %type<Ast.program> program
157
158 %%
159
160 /*
161     Use List.rev on any rule that builds up a list in reverse. Lists are built in
162     reverse
163     for efficiency reasons
164 */
165
166 program: includes var_decls func_decls struc_decls EOF { ($1, List.rev $2, List.
167     rev $3, List.rev $4) }
168
169 includes:
170     /* noting */ { [] }
171     | includes include_file { $2 :: $1 }
172
173 include_file:

```

```

172     POUND INCLUDE STRING_LITERAL { (Curr, $3) }
173 | POUND INCLUDE LT INCLUDE_FILE GT      { (Standard,$4) }
174
175 var_decls:
176     /* nothing */ { [] }
177 | var_decls vdecl { $2::$1 }
178
179 func_decls:
180     fdecl {[ $1]}
181 | func_decls fdecl { $2::$1}
182
183 mthd:
184     METHOD any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
185     RBRACE {{
186     typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
187     $9; tests = None }}
188
189 struc_func_decls:
190     /* nothing */ { [] }
191 | struc_func_decls mthd { $2::$1 }
192
193 struc_decls:
194     /*nothing*/ { [] }
195 | struc_decls sdecl { $2::$1}
196
197 prim_typ:
198 | STRING { String }
199 | DOUBLE { Double }
200 | INT { Int }
201 | CHAR { Char }
202 | BOOL { Bool }
203
204 void_typ:
205 | VOID { Void }
206
207 struct_typ:
208 | STRUCT ID { $2 }
209
210 array_typ:
211     prim_typ LBRACKET INT_LITERAL RBRACKET { ($1, $3) }
212 | prim_typ LBRACKET RBRACKET { ($1, 0) }
213
214 pointer_typ:
215 | prim_typ STAR { Primitive($1) }
216 | struct_typ STAR { Struct_typ($1) }
217 | array_typ STAR { Array_typ(fst $1, snd $1) }
218
219 double_pointer_typ:
220 | pointer_typ STAR { Pointer_typ($1) }
221
222
223 any_typ:
224     prim_typ { Primitive($1) }
225 | struct_typ { Struct_typ($1) }
226 | pointer_typ { Pointer_typ($1) }
227 | double_pointer_typ { Pointer_typ($1) }
228 | void_typ { Primitive($1) }
229 | array_typ { Array_typ(fst $1, snd $1) }

```

```

230
231
232 any_typ_not_void:
233     prim_typ      { Primitive($1) }
234     | struct_typ   { Struct_typ($1) }
235     | pointer_typ   { Pointer_typ($1) }
236     | double_pointer_typ { Pointer_typ($1) }
237     | array_typ     { Array_typ(fst $1, snd $1) }
238
239 /*
240 Rules for function syntax
241 */
242 fdecl:
243     FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
244     RBRACE {{
245         typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
246         $9; tests = None }}
247     | FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
248     RBRACE testdecl {{
249         typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
250         $9; tests = Some({asserts = $11; using = { uvdecls = []; stmts = [] }}) }}
251     | FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list func_body
252     RBRACE testdecl usingdecl {{
253         typ = $2; fname = $3; formals = $5; vdecls = List.rev $8; body = List.rev
254         $9; tests = Some({asserts = $11; using = { uvdecls = (fst $12); stmts = (snd
255         $12)}}) }}
256
257 /*
258 "with test" rule
259 */
260 testdecl:
261     WTEST LBRACE stmt_list RBRACE { $3 }
262
263 /*
264 "using" rule
265 */
266 usingdecl:
267     USING LBRACE vdecl_list stmt_list RBRACE { (List.rev $3, List.rev $4) }
268
269 /*
270 Formal parameter rules
271 */
272 formal_opts_list:
273     /* nothing */ { [] }
274     | formal_opt { $1 }
275
276 formal_opt:
277     any_typ_not_void ID {[(($1,$2)]}
278     | formal_opt COMMA any_typ_not_void ID {(($3,$4):: $1)}
279
280 actual_opts_list:
281     /* nothing */ { [] }
282     | actual_opt { $1 }
283
284 actual_opt:
285     expr { [$1] }
286     | actual_opt COMMA expr { $3:: $1 }

```

```

285  /*
286  Rule for declaring a list of variables, including variables of type struct x
287  */
288  vdecl_list:
289      /* nothing */ { [] }
290      | vdecl_list vdecl { $2::$1 }
291
292  /*
293  Includes declaring a struct
294  */
295
296  vdecl:
297      any_typ_not_void ID SEMI { ($1, $2) }
298
299  /*
300  Rule for defining a struct
301  */
302  sdecl:
303      STRUCT ID LBRACE vdecl_list struc_func_decls RBRACE SEMI {{
304          sname = $2; attributes = List.rev $4; methods = List.rev $5 }}
305
306
307  func_body:
308      stmt_list    {[Block(List.rev $1)]}
309
310  stmt_list:
311      /* nothing */ { [] }
312      | stmt_list stmt { $2::$1 }
313
314  /*
315  Rule for statements. Statements include expressions
316  */
317  stmt:
318      expr SEMI                                { Expr $1 }
319      | LBRACE stmt_list RBRACE                { Block(List.rev $2) }
320      | RETURN SEMI                           { Return Noexpr }
321      | RETURN expr SEMI                       { Return $2 }
322      | IF LPAREN expr RPAREN stmt ELSE stmt   { If($3, $5, $7) }
323      | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([]))
324      ) }
325      | WHILE LPAREN expr RPAREN stmt           { While($3, $5) }
326      | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt { For($3, $5, $7,
327      $9)}
328      | ASSERT LPAREN expr RPAREN SEMI         { Assert($3) }
329
330  /*
331  Rule for building expressions
332  */
333  expr:
334      INT_LITERAL      { Lit($1)}
335      | STRING_LITERAL { String_lit($1) }
336      | CHAR_LITERAL   { Char_lit($1) }
337      | DOUBLE_LITERAL  { Double_lit($1) }
338      | TRUE            { BoolLit(true) }
339      | FALSE           { BoolLit(false) }
340      | ID              { Id($1) }
341      | LPAREN expr RPAREN { $2 }
342      | expr PLUS expr    { Binop($1, Add, $3) }
343      | expr MINUS expr   { Binop($1, Sub, $3) }

```

```

342 | expr STAR expr { Binop($1, Mult, $3)}
343 | expr DIVIDE expr { Binop($1, Div, $3)}
344 | expr EQ expr { Binop($1, Equal, $3)}
345 | expr EXPO expr { Binop($1, Exp, $3)}
346 | expr MODULO expr { Binop($1, Mod, $3)}
347 | expr NEQ expr { Binop($1, Neq, $3)}
348 | expr LT expr { Binop($1, Less, $3)}
349 | expr LEQ expr { Binop($1, Leq, $3)}
350 | expr GT expr { Binop($1, Greater, $3)}
351 | expr GEQ expr { Binop($1, Geq, $3)}
352 | expr AND expr { Binop($1, And, $3)}
353 | expr OR expr { Binop($1, Or, $3)}
354 | NOT expr { Unop(Not, $2) }
355 | AMPERSAND expr { Unop(Addr, $2) }
356 | expr ASSIGN expr { Assign($1, $3) }
357 | expr DOT expr { Struct_access($1, $3)}
358 | expr POINTER_ACCESS expr { Pt_access($1, $3)}
359 | STAR expr { Dereference($2) }
360 | expr LBRACKET INT_LITERAL RBRACKET { Array_access($1, $3)}
361 | NEW prim_typ LBRACKET INT_LITERAL RBRACKET { Array_create($4, $2) }
362 | NEW STRUCT ID { Struct_create($3)}
363 | FREE LPAREN expr RPAREN { Free($3) }
364 | ID LPAREN actual_opts_list RPAREN { Call($1, $3)}
365 | NULL LPAREN any_typ_not_void RPAREN { Null($3) }
366 | DUBS { Dubs }
367 expr_opt:
368     /* nothing */ { Noexpr }
369 | expr { $1 }

```

12.2 ast.ml

```
1 type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And
  | Or | Mod | Exp
2 type uop = Neg | Not | Addr
3 type prim = Int | Double | String | Char | Void | Bool
4 type typ = Primitive of prim | Struct_typ of string | Func_typ of string |
  Pointer_typ of typ | Array_typ of prim * int | Any
5 type bind = typ * string
6
7 type dir_location = Curr | Standard
8
9 type header = dir_location * string
10
11 type expr =
12   Lit      of int
13   | String_lit of string
14   | Char_lit of char
15   | Double_lit of float
16   | Binop    of expr * op * expr
17   | Unop     of uop * expr
18   | Assign   of expr * expr
19   | Noexpr
20   | Id of string
21   | Struct_create of string
22   | Struct_access of expr * expr
23   | Pt_access of expr * expr
24   | Dereference of expr
25   | Array_create of int * prim
26   | Array_access of expr * int
27   | Free of expr
28   | Call of string * expr list
29   | BoolLit of bool
30   | Null of typ
31   | Dubs
32
33 type stmt =
34   Block of stmt list
35   | Expr of expr
36   | If of expr * stmt * stmt
37   | While of expr * stmt
38   | For of expr * expr * expr * stmt
39   | Return of expr
40   | Assert of expr
41
42 type with_using_decl = {
43   uvdecls : bind list;
44   stmts : stmt list;
45 }
46
47 type with_test_decl = {
48   asserts : stmt list;
49   using : with_using_decl;
50 }
51
52 (* Node that describes a function *)
53 type func_decl = {
54   typ : typ;
55   fname : string;
```

```

56     formals : bind list;
57     vdecls  : bind list;
58     body    : stmt list;
59     tests   : with_test_decl option;
60 }
61
62 (* Node that describes a given struct *)
63 type struct_decl = {
64     sname      : string;
65     attributes : bind list;
66     methods    : func_decl list;
67 }
68
69 (* Root of tree. Our program is made up three things 1) list of global variables
70    2) list of functions 3) list of struct definition *)
71 type program = header list * bind list * func_decl list * struct_decl list

```

12.3 semant.ml

```
1 (* Semantic checker code. Takes Ast as input and returns a Sast *)
2
3 module A = Ast
4 module S = Sast
5 module StringMap = Map.Make(String)
6
7 type variable_decls = A.bind;;
8
9 (* Hashtable of valid structs. This is filled out when we iterate through the user
   defined structs *)
10 let struct_types:(string, A.struct_decl) Hashtbl.t = Hashtbl.create 10
11 let func_names:(string, A.func_decl) Hashtbl.t = Hashtbl.create 10
12
13 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 }
14
15 (* Symbol table used for checking scope *)
16 type symbol_table = {
17   parent : symbol_table option;
18   variables : (string, A.typ) Hashtbl.t;
19 }
20
21 (* Environment*)
22 type environment = {
23   scope : symbol_table;
24   return_type : A.typ option;
25   func_name : string option;
26   in_test_func : bool;
27   in_struct_method : bool;
28   struct_name : string option
29 }
30
31 (* For debugging *)
32 let rec string_of_typ t =
33   match t with
34   | A.Primitive(A.Int) -> "Int"
35   | A.Primitive(A.Double) -> "Double"
36   | A.Primitive(A.String) -> "String"
37   | A.Primitive(A.Char) -> "Char"
38   | A.Primitive(A.Void) -> "Void"
39   | A.Struct_typ(s) -> "struct " ^ s
40   | A.Pointer_typ(t) -> "pointer " ^ (string_of_typ t)
41   | A.Array_typ(p,_) -> "Array type " ^ (string_of_typ (A.Primitive(p)))
42   | _ -> "not sure"
43
44 (* Search symbol tables to see if the given var exists somewhere *)
45 let rec find_var (scope : symbol_table) var =
46   try Hashtbl.find scope.variables var
47   with Not_found ->
48     match scope.parent with
49     | Some(parent) -> find_var parent var
50     | _ -> raise (Exceptions.UndeclaredVariable var)
51
52 (* Helper function to reeturn an identifiers type *)
53 let type_of_identifier var env =
54   find_var env.scope var
```



```

55
56 (* Returns the type of the arrays elements. E.g. int[10] arr... type_of_array arr
   would return A.Int *)
57 let type_of_array arr _ =
58   match arr with
59   | A.Array_typ(p,_) -> A.Primitive(p)
60   | A.Pointer_typ(A.Array_typ(p, _)) -> A.Primitive(p)
61   | _ -> raise (Exceptions.InvalidArrayVariable)
62
63 (* Function is done for creating sast after semantic checking. Should only be
   called on struct or array access *)
64 let rec string_identifier_of_expr expr =
65   match expr with
66   | A.Id(s) -> s
67   | A.Struct_access(e1, _) -> string_identifier_of_expr e1
68   | A.Pt_access(e1, _) -> string_identifier_of_expr e1
69   | A.Array_access(e1, _) -> string_identifier_of_expr e1
70   | A.Call(s,_) -> s
71   | _ -> raise (Exceptions.BugCatch "string_identifier_of_expr")
72
73
74 let rec string_of_expr e env =
75   match e with
76   | A.Lit(i) -> string_of_int i
77   | A.String_lit(s) -> s
78   | A.Char_lit(c) -> String.make 1 c
79   | A.Double_lit(_) -> ""
80   | A.Binop(e1,op,e2) -> let str1 = string_of_expr e1 env in
81   let str2 = string_of_expr e2 env in
82   let str_op =
83     (match op with
84      | A.Add -> "+"
85      | A.Sub -> "-"
86      | A.Mult -> "*"
87      | A.Div -> "/"
88      | A.Equal -> "=="
89      | A.Neq -> "!="
90      | A.Less -> "<="
91      | A.Leq -> "<"
92      | A.Greater -> ">="
93      | A.Geq -> ">"
94      | A.And -> "&&"
95      | A.Or -> "||"
96      | A.Mod -> "%"
97      | A.Exp -> "^"
98     ) in (String.concat " " [str1;str_op;str2])
99   | A.Unop(u,e) -> let str_expr = string_of_expr e env in
100   let str_uop =
101     (match u with
102      | A.Neg -> "-"
103      | A.Not -> "!"
104      | A.Addr -> "&"
105     ) in
106   let str1 = String.concat " " [str_uop; str_expr] in str1
107   | A.Assign (_,_) -> ""
108   | A.Noexpr -> ""
109   | A.Id(s) -> s
110   | A.Struct_create(_) -> ""
111   | A.Struct_access(e1,e2) -> let str1 = string_of_expr e1 env in

```

```

112     let str2 = string_of_expr e2 env in
113     let str_acc = String.concat "." [str1; str2] in str_acc
114 | A.Pt_access(e1,e2) -> let str1 = string_of_expr e1 env in
115     let str2 = string_of_expr e2 env in
116     let str_acc = String.concat "->" [str1; str2] in str_acc
117
118 | A.Dereference(e) -> let str1 = string_of_expr e env in (String.concat "" ["*"; str1])
119 | A.Array_create(i,p) -> let str_int = string_of_int i in
120     let rb = "]" in
121     let lb = "[" in
122     let new_ = "new" in
123     let str_prim =
124     (match p with
125       | A.Int -> "int"
126       | A.Double -> "double"
127       | A.Char -> "char"
128       | _ -> raise (Exceptions.BugCatch "string_of_expr"))
129     ) in let str_ar_ac = String.concat "" [new_; " "; str_prim; lb; str_int; rb]
130     in str_ar_ac
131 | A.Array_access(e,i) -> let lb = "[" in
132     let rb = "]" in
133     let str_int = string_of_int i in
134     let str_expr = string_of_expr e env in
135     let str_acc = String.concat "" [str_expr; lb; str_int; rb] in str_acc
136 | A.Free(_) -> ""
137 | A.Call(s,le) -> let str1 = s ^ "(" in
138     let str_exprs_rev = List.map (fun n -> string_of_expr n env) le in
139     let str_exprs = List.rev str_exprs_rev in
140     let str_exprs_commas = (String.concat "," str_exprs) in
141     let str2 = (String.concat "" (str1::str_exprs_commas::[")"])) in str2
142 | A.BoolLit (b) ->
143     (match b with
144       | true -> "true"
145       | false -> "false"
146     )
147 | A.Null(_) -> "NULL"
148 | A.Dubs -> ""
149
150 (* Function is done for creating sast after semantic checking. Should only be
151    called on struct fields *)
152 let string_of_struct_expr expr =
153     match expr with
154     | A.Id(s) -> s
155     | _ -> raise (Exceptions.BugCatch "string_of_struct_expr")
156
157 (* Helper function to check for dups in a list *)
158 let report_duplicate exceptf list =
159     let rec helper = function
160       | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
161       | _ :: t -> helper t
162       | [] -> ()
163     in helper (List.sort compare list)
164
165 (* Used to check include statments *)
166 let check_ends_in_jt str =
167     let len = String.length str in
168     if len < 4 then raise (Exceptions.InvalidHeaderFile str);
169     let subs = String.sub str (len - 3) 3 in

```

```

168 (match subs with
169   ".jt" -> ()
170 | _ -> raise (Exceptions.InvalidHeaderFile str)
171 )
172
173 let check_in_test e = if e.in_test_func = true then () else raise (Exceptions.
    InvalidAssert "assert can only be used in tests")
174
175 (* Helper function to check a typ is not void *)
176 let check_not_void exceptf = function
177   (A.Primitive(A.Void), n) -> raise (Failure (exceptf n))
178 | _ -> ()
179
180 (* Helper function to check two types match up *)
181 let check_assign lvaluet rvaluet err =
182   (match lvaluet with
183     A.Pointer_typ(A.Array_typ(p,0)) ->
184       (match rvaluet with
185         A.Pointer_typ(A.Array_typ(p2,_)) -> if p = p2 then lvaluet else raise
186         err
187       | _ -> raise err
188   )
189 | A.Primitive(A.String) -> (match rvaluet with A.Primitive(A.String) -> lvaluet
190 | A.Array_typ(A.Char,_) -> lvaluet | _ -> raise err)
191 | A.Array_typ(A.Char,_) -> (match rvaluet with A.Array_typ((A.Char),_) ->
192   lvaluet | A.Primitive(A.String) -> lvaluet | _ -> raise err)
193 | _ -> if lvaluet = rvaluet then lvaluet else raise err
194 )
195
196 (* Search hash table to see if the struct is valid *)
197 let check_valid_struct s =
198   try Hashtbl.find struct_types s
199 with | Not_found -> raise (Exceptions.InvalidStruct s)
200
201 (* Checks the hash table to see if the function exists *)
202 let check_valid_func_call s =
203   try Hashtbl.find func_names s
204 with | Not_found -> raise (Exceptions.InvalidFunctionCall (s ^ " does not exist.
    Unfortunately you can't just expect functions to magically exist"))
205
206 (* Helper function that finds index of first matching element in list *)
207 let rec index_of_list x l =
208   match l with
209   [] -> raise (Exceptions.BugCatch "index_of_list")
210 | hd::tl -> let (_,y) = hd in if x = y then 0 else 1 + index_of_list x tl
211
212 let index_helper s field env =
213   let struct_var = find_var env.scope s in
214   match struct_var with
215   A.Struct_typ(struc_name) ->
216     (let stru:(A.struct_decl) = check_valid_struct struc_name in
217     try let index = index_of_list field stru.A.attributes in index with |
218     Not_found -> raise (Exceptions.BugCatch "index_helper"))
219 | A.Pointer_typ(A.Struct_typ(struc_name)) ->
220   (let stru:(A.struct_decl) = check_valid_struct struc_name in
221   try let index = index_of_list field stru.A.attributes in index with |
222   Not_found -> raise (Exceptions.BugCatch "index_helper"))

```

```

220 | _ -> raise (Exceptions.BugCatch "struct_contains_field")
221
222
223 (* 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 *)
224 let index_of_struct_field stru expr env =
225   match stru with
226   | A.Id(s) -> (match expr with A.Id(s1) -> index_helper s s1 env | _ -> raise
    (Exceptions.BugCatch "index_of_struct"))
227   | _ -> raise (Exceptions.InvalidStructField)
228
229
230
231 (* Checks the relevant struct actually has a given field *)
232 let struct_contains_field s field env =
233   let struct_var = find_var env.scope s in
234   match struct_var with
235   | A.Struct_typ(struc_name) ->
236     (let stru:(A.struct_decl) = check_valid_struct struc_name in
237      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))
238   | A.Pointer_typ(A.Struct_typ(struc_name)) ->
239     (let stru:(A.struct_decl) = check_valid_struct struc_name in
240      try let (my_typ,_) = (List.find (fun (_,nm) -> if nm = field then true else
    false) stru.A.attributes) in my_typ with | Not_found ->
241      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))
242
243   | _ -> raise (Exceptions.BugCatch "struct_contains_field")
244
245 let struct_contains_method s methd env =
246   let struct_var = find_var env.scope s in
247   match struct_var with
248   | A.Pointer_typ(A.Struct_typ(struc_name)) ->
249     (let stru:(A.struct_decl) = check_valid_struct struc_name in
250      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))
251
252   | _ -> raise (Exceptions.BugCatch "struct_contains_field")
253
254
255 (* Checks that struct contains expr *)
256 let struct_contains_expr stru expr env =
257   match stru with
258   | A.Id(s) -> (match expr with
259     | A.Id(s1) -> struct_contains_field s s1 env
260     | A.Call(s1, _) -> struct_contains_method s s1 env
261     | _ -> raise (Exceptions.InvalidStructField))
262   | _ -> raise (Exceptions.InvalidStructField)
263
264 let struct_field_is_local str fiel env =
265   try (let _ = struct_contains_field str fiel env in false)
266   with | Exceptions.InvalidStructField -> true
267
268 let rec type_of_expr env e =

```

```

269 match e with
270   A.Lit(_) -> A.Primitive(A.Int)
271 | A.String_lit(_) -> A.Primitive(A.String)
272 | A.Char_lit(_) -> A.Primitive(A.Char)
273 | A.Double_lit(_) -> A.Primitive(A.Double)
274 | A.Binop(e1,_,_) -> type_of_expr env e1
275 | A.Unop (_,e1) -> type_of_expr env e1
276 | A.Assign(e1,_) -> type_of_expr env e1
277 | A.Id(s) -> find_var env.scope s
278 | A.Struct_create(s) -> A.Pointer_typ(A.Struct_typ(s))
279 | A.Struct_access(e1,e2) -> struct_contains_expr e1 e2 env
280 | A.Pt_access(e1,e2) -> let tmp_type = type_of_expr env e1 in
281   (match tmp_type with
282     A.Pointer_typ(A.Struct_typ(_)) ->
283       (match e2 with
284         A.Call(_,_) -> struct_contains_expr e1 e2 env
285       | A.Id(_) -> struct_contains_expr e1 e2 env
286       | _ -> raise (Exceptions.BugCatch "type_of_expr")
287       )
288   | _ -> raise (Exceptions.BugCatch "type_of_expr")
289   )
290 | A.Dereference(e1) -> let tmp_e = type_of_expr env e1 in
291   (
292     match tmp_e with
293     A.Pointer_typ(p) -> p
294   | _ -> raise (Exceptions.BugCatch "type_of_expr")
295   )
296 | A.Array_create(i,p) -> A.Pointer_typ(A.Array_typ(p,i))
297 | A.Array_access(e,_) -> type_of_array (type_of_expr env e) env
298 | A.Call(s,_) -> let func_info = (check_valid_func_call s) in func_info.A.typ
299 | A.BoolLit(_) -> A.Primitive(A.Bool)
300 | A.Null(t) -> t
301 | _ -> raise (Exceptions.BugCatch "type_of_expr")
302
303 (* convert expr to sast expr *)
304 let rec expr_sast expr env =
305   match expr with
306   A.Lit a -> S.SLit a
307 | A.String_lit s -> S.SString_lit s
308 | A.Char_lit c -> S.SChar_lit c
309 | A.Double_lit d -> S.SDouble_lit d
310 | A.Binop (e1, op, e2) -> let tmp_type = type_of_expr env e1 in S.SBinop (
311   expr_sast e1 env, op, expr_sast e2 env, tmp_type)
312 | A.Unop (u, e) -> S.SUnop(u, expr_sast e env)
313 | A.Assign (s, e) -> S.SAssign (expr_sast s env, expr_sast e env)
314 | A.Noexpr -> S.SNoexpr
315 | A.Id s -> (match env.in_struct_method with
316   true ->
317     (match env.struct_name with
318       Some(nm) -> let local_struct_field = struct_field_is_local nm s env in
319       (match local_struct_field with
320         true -> S.SId (s)
321       | false -> let tmp_id = A.Id(nm) in let tmp_pt_access = A.Pt_access(tmp_id
322   , A.Id(s)) in (expr_sast tmp_pt_access env)
323       )
324     | None -> raise (Exceptions.BugCatch "expr_sast")
325     )
326 | false -> S.SId (s)
327   )

```

```

326 | A.Struct_create s -> S.SStruct_create s
327 | A.Free e -> let st = (string_identifier_of_expr e) in S.SFree(st)
328 | A.Struct_access (e1, e2) -> let index = index_of_struct_field e1 e2 env in S.
    SStruct_access (string_identifier_of_expr e1, string_of_struct_expr e2, index)
329 | A.Pt_access (e1, e2) ->
330   (match e2 with
331   | A.Id(_) -> 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)
    in t
332   | 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
333     (match struct_decl with
334     | 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)))
335     | _ -> raise (Exceptions.BugCatch "expr_sast")
336     )
337   | _ -> raise (Exceptions.BugCatch "expr_sast")
338   )
339 | A.Array_create (i, p) -> S.SArray_create (i, p)
340 | A.Array_access (e, i) -> let tmp_string = (string_identifier_of_expr e) in
341   let tmp_type = find_var env.scope tmp_string in S.SArray_access (tmp_string, i
    , tmp_type)
342 | A.Dereference(e) -> S.SDereference(string_identifier_of_expr e)
343 | A.Call (s, e) -> S.SCall (s, (List.map (fun n -> expr_sast n env) e))
344 | A.BoolLit(b) -> S.SBoolLit((match b with true -> 1 | false -> 0))
345 | A.Null(t) -> S.SNull t
346 | A.Dubs -> S.SDubs
347
348
349 (* Convert ast struct to sast struct *)
350 let struct_sast r =
351   let tmp:(S.sstruct_decl) = {S.ssname = r.A.sname ; S.sattributes = r.A.
    attributes} in
352   tmp
353
354
355 (* function that adds struct pointer to formal arg *)
356 let add_pt_to_arg s f =
357   let tmp_formals = f.A.formals in
358   let tmp_type = A.Pointer_typ(A.Struct_typ(s.A.sname)) in
359   let tmp_string = "p" in
360   let new_formal:(A.bind) = (tmp_type, tmp_string) in
361   let formals_with_pt = new_formal :: tmp_formals in
362   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
    } in
363   new_func
364
365 (* Creates new functions whose first paramters is a pointer to the struct type
    that the method is associated with *)
366 let add_pts_to_args s fl =
367   let list_of_struct_funcs = List.map (fun n -> add_pt_to_arg s n) fl in
368   list_of_struct_funcs
369
370
371 (* Struct semantic checker *)
372 let check_structs structs =
373   (report_duplicate(fun n -> "duplicate struct " ^ n) (List.map (fun n -> n.A.
    sname) structs));

```

```

374 ignore (List.map (fun n -> (report_duplicate(fun n -> "duplicate struct field "
375 ^ n) (List.map (fun n -> snd n) n.A.attributes))) structs);
376
377 ignore (List.map (fun n -> (List.iter (check_not_void (fun n -> "Illegal void
378 field" ^ n)) n.A.attributes)) structs);
379 ignore(List.iter (fun n -> Hashtbl.add struct_types n.A.sname n) structs);
380 let tmp_funcs = List.map (fun n -> (n, n.A.methods)) structs in
381 let tmp_funcs_with_formals = List.fold_left (fun l s -> let tmp_l = (
382   add_pts_to_args (fst s) (snd s)) in l @ tmp_l) [] tmp_funcs in
383 (* Globa variables semantic checker *)
384 let check_globals globals env =
385   ignore(env);
386   ignore (report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd
387     globals));
388   List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;
389   (* Check that any global structs are actually valid structs that have been
390     defined *)
391   List.iter (fun (t,_) -> match t with
392     A.Struct_typ(nm) -> ignore(check_valid_struct nm); ()
393     | _ -> ())
394     globals;
395   (* Add global variables to top level symbol table. Side effects *)
396   List.iter (fun (t,s) -> (Hashtbl.add env.scope.variables s t)) globals;
397   globals
398
399 (* Main entry pointer for checking the semantics of an expression *)
400 let rec check_expr expr env =
401   match expr with
402   | A.Lit(_) -> A.Primitive(A.Int)
403   | A.String_lit(_) -> A.Primitive(A.String)
404   | A.Char_lit(_) -> A.Primitive(A.Char)
405   | A.Double_lit(_) -> A.Primitive(A.Double)
406   | A.Binop(e1,op,e2) -> let e1' = (check_expr e1 env) in
407     let e2' = (check_expr e2 env) in
408     (match e1' with
409     | A.Primitive(A.Int) | A.Primitive(A.Double) | A.Primitive(A.Char) ->
410       (match op with
411       | A.Add | A.Sub | A.Mult | A.Div | A.Exp | A.Mod when e1' = e2' && (e1' = A.
412         Primitive(A.Int) || e1' = A.Primitive(A.Double)) -> e1'
413       | A.Equal | A.Neq when e1' = e2' -> ignore("got equal"); A.Primitive(A.Int)
414       | A.Less | A.Leq | A.Greater | A.Geq when e1' = e2' && (e1' = A.Primitive(A.
415         Int) || e1' = A.Primitive(A.Double)) -> e1'
416       | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
417     )
418   | A.Pointer_typ(_) -> let e1' = (check_expr e1 env) in
419     let e2' = (check_expr e1 env) in
420     (match op with
421     | A.Equal | A.Neq when e1' = e2' && (e1 = A.Null(e2')) || e2 = A.Null(e1') ->
422       e1'
423     | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
424

```

```

425 )
426 | _ -> raise (Exceptions.InvalidExpr "Illegal binary op")
427 )
428 | A.Unop(uop,e) -> let expr_type = check_expr e env in
429   (match uop with
430     A.Not -> (match expr_type with A.Primitive(A.Bool) -> expr_type | _ ->
431       raise Exceptions.NotBoolExpr)
432     | A.Neg -> expr_type
433     | A.Addr -> A.Pointer_typ(expr_type)
434   )
435 | A.Assign(var,e) -> (let right_side_type = check_expr e env in
436   let left_side_type = check_expr var env in
437   check_assign left_side_type right_side_type Exceptions.IllegalAssignment)
438 | A.Noexpr -> A.Primitive(A.Void)
439 | A.Id(s) -> type_of_identifier s env
440 | A.Struct_create(s) -> (try let tmp_struct = check_valid_struct s in (A.
441   Pointer_typ(A.Struct_typ(tmp_struct.A.sname))) with | Not_found -> raise (
442   Exceptions.InvalidStruct s))
443 | A.Struct_access(e1,e2) -> struct_contains_expr e1 e2 env
444 | A.Pt_access(e1,e2) -> let e1' = check_expr e1 env in
445   (match e1' with
446     A.Pointer_typ(A.Struct_typ(_)) -> struct_contains_expr e1 e2 env
447     | A.Pointer_typ(A.Primitive(p)) -> (let e2' = check_expr e2 env in (
448     check_assign (A.Primitive(p)) e2') (Exceptions.InvalidPointerDereference))
449     | _ -> raise (Exceptions.BugCatch "hey")
450   )
451 | A.Dereference(i) -> let pointer_type = (check_expr i env) in (
452   match pointer_type with
453     A.Pointer_typ(pt) -> pt
454     | _ -> raise (Exceptions.BugCatch "Deference")
455   )
456 | A.Array_create(size,prim_type) -> A.Pointer_typ(A.Array_typ(prim_type, size))
457 | A.Array_access(e, _) -> type_of_array (check_expr e env) env
458 | A.Free(p) -> let pt = string_identifier_of_expr p in
459   let pt_typ = find_var env.scope pt in (match pt_typ with A.Pointer_typ(
460   _) -> pt_typ | _ -> raise (Exceptions.InvalidFree "not a pointer"))
461 | A.Call("print", el) -> if List.length el != 1 then raise Exceptions.
462   InvalidPrintCall
463   else
464     List.iter (fun n -> ignore(check_expr n env); ()) el; A.Primitive(A.Int)
465 | A.Call(s,el) -> let func_info = (check_valid_func_call s) in
466   let func_info_formals = func_info.A.formals in
467   if List.length func_info_formals != List.length el then
468     raise (Exceptions.InvalidArgumentsToFunction (s ^ " is supplied with wrong
469     args"))
470 else
471   List.iter2 (fun (ft,_) e -> let e = check_expr e env in ignore(check_assign ft
472   e (Exceptions.InvalidArgumentsToFunction ("Args to functions " ^ s ^ " don't
473   match up with it's definition")))) func_info_formals el;
474   func_info.A.typ
475 | A.BoolLit(_) -> A.Primitive(A.Bool)
476 | A.Null(t) -> t
477 | A.Dubs -> A.Primitive(A.Void)
478
479 (* Checks if expr is a boolean expr. Used for checking the predicate of things
480   like if, while statements *)
481 let check_is_bool expr env =
482   ignore(check_expr expr env);

```



```

474 match expr with
475   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,_) -> ()
476
477 | _ -> raise (Exceptions.InvalidBooleanExpression)
478
479 (* Checks that return value is the same type as the return type in the function
    definition*)
480 let check_return_expr expr env =
481   match env.return_type with
482     Some(rt) -> if rt = check_expr expr env then () else raise (Exceptions.
        InvalidReturnType "return type doesnt match with function definition")
483   | _ -> raise (Exceptions.BugCatch "Should not be checking return type outside a
        function")
484
485 (* Main entry point for checking semantics of statements *)
486 let rec check_stmt stmt env =
487   match stmt with
488     A.Block(l) -> (let rec check_block b env2 =
489       (match b with
490         [A.Return _ as s] -> let tmp_block = check_stmt s env2 in ([tmp_block])
491         | A.Return _ :: _ -> raise (Exceptions.InvalidReturnType "Can't have any
            code after return statement")
492         | A.Block l :: ss -> check_block (l @ ss) env2
493         | l :: ss -> let tmp_block = (check_stmt l env2) in
            let tmp_block2 = (check_block ss env2) in ([tmp_block] @ tmp_block2)
494         | [] -> ([]))
495       in
496         let checked_block = check_block l env in S.SBlock(checked_block)
497       )
498   (*| A.Block(b) -> S.SBlock (List.map (fun n -> check_stmt n env) b) *)
499   | A.Expr(e) -> ignore(check_expr e env); S.SExpr(expr_sast e env)
500   | 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)
501   | A.While(e,s) -> ignore(check_is_bool e env); S.SWhile (expr_sast e env,
        check_stmt s env)
502   | 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)
503   | A.Return(e) -> ignore(check_return_expr e env);S.SReturn (expr_sast e env)
504   | A.Assert(e) -> ignore(check_in_test env); ignore(check_is_bool e env);
505     let str_expr = string_of_expr e env in
506     let then_stmt = S.SExpr(S.SCall("print", [S.SString_lit(str_expr ^ " passed"
507     )])) in
508     let else_stmt = S.SExpr(S.SCall("print", [S.SString_lit(str_expr ^ " failed"
509     )])) in S.SIf (expr_sast e env, then_stmt, else_stmt)
510
511 (* Converts 'using' code from ast to sast *)
512 let with_using_sast r env =
513   let tmp:(S.swith_using_decl) = {S.suvdecls = r.A.uvdecls; S.sstmts = (List.map (
514     fun n -> check_stmt n env) r.A.stmts)} in
515   tmp
516
517 (* Converts 'test' code from ast to sast *)
518 let with_test_sast r env =
519   let tmp:(S.swith_test_decl) = {S.sasserts = (List.map (fun n -> check_stmt n env
520     ) r.A.asserts) ; S.susing = (with_using_sast r.A.using env)} in
521   tmp
522

```

```

520 (* Here we convert the user defined test cases to functions which can subsequently
    be called by main in the test file *)
521 let convert_test_to_func using_decl test_decl env =
522   List.iter (fun n -> (match n with A.Assert(_) -> () | _ -> raise Exceptions.
    InvalidTestAsserts)) test_decl.A.asserts;
523   let test_asserts = List.rev test_decl.A.asserts in
524   let concat_stmts = using_decl.A.stmts @ test_asserts in
525   (match env.func_name with
526    Some(fn) -> let new_func_name = fn ^ "test" in
527    let new_func:(A.func_decl) = {A.typ = A.Primitive(A.Void); A.fname = (
    new_func_name); A.formals = []; A.vdecls = using_decl.A.vdecls; A.body =
    concat_stmts ; A.tests = None} in new_func
528
529   | None -> raise (Exceptions.BugCatch "convert_test_to_func")
530 )
531
532 (* Function names (aka can't have two functions with same name) semantic checker
    *)
533 let check_function_names functions =
534   ignore(report_duplicate (fun n -> "duplicate function names " ^ n) (List.map (
    fun n -> n.A.fname) functions));
535   (* Add the built in function(s) here. There shouldnt be too many of these *)
536   ignore(Hashtbl.add func_names built_in_print_string.A.fname
    built_in_print_string);
537   (* 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) *)
538   ignore(List.iter (fun n -> Hashtbl.add func_names n.A.fname n) functions); ()
539
540 let check_prog_contains_main funcs =
541   let contains_main = List.exists (fun n -> if n.A.fname = "main" then true else
    false) funcs in
542   (match contains_main with
543    true -> ()
544    | false -> raise Exceptions.MissingMainFunction
545   )
546
547 (* Checks programmer hasn't defined function print as it's reserved *)
548 let check_function_not_print names =
549   ignore(if List.mem "print" (List.map (fun n -> n.A.fname) names) then raise (
    Failure ("function print may not be defined")) else ()); ()
550
551 (* Check the body of the function here *)
552 let rec check_function_body funct env =
553   let curr_func_name = funct.A.fname in
554   report_duplicate (fun n -> "duplicate formal arg " ^ n) (List.map snd funct.A.
    formals);
555   report_duplicate (fun n -> "duplicate local " ^ n) (List.map snd funct.A.vdecls)
    ;
556   (* Check no duplicates *)
557
558   let in_struct = env.in_struct_method in
559   let formals_and_locals =
560     (match in_struct with
561      true ->
562       let (struct_arg_typ, _) = List.hd funct.A.formals in
563       (match struct_arg_typ with
564        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
565         | _ -> raise (Exceptions.BugCatch "check function body")
566         )
567         | false -> List.append funct.A.formals funct.A.vdecls
568         )
569     in
570
571 report_duplicate (fun n -> "same name for formal and local var " ^ n) (List.map
    snd formals_and_locals);
572 (* Check structs are valid *)
573 List.iter (fun (t,_) -> match t with
574     A.Struct_ttyp(nm) -> ignore(check_valid_struct nm); ()
575     | _ -> ())
576 ) formals_and_locals;
577 (* Create new environment -> symbol table parent is set to previous scope's
    symbol table *)
578 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
579 (* Add formals + locals to this scope symbol table *)
580 List.iter (fun (t,s) -> (Hashtbl.add new_env.scope.variables s t))
    formals_and_locals;
581 let body_with_env = List.map (fun n -> check_stmt n new_env) funct.A.body in
582 (* Compile code for test case iff a function has defined a with test clause *)
583 let sast_func_with_test =
584     (match funct.A.tests with
585     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
586     Some(check_function_body func_with_test new_env2)
587     | None -> None
588     )
589 in
590
591 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)} in
592 tmp
593
594 (* Entry point to check functions *)
595 let check_functions functions_with_env includes globals_add structs_add =
596     let function_names = List.map (fun n -> fst n) functions_with_env in
597
598     (check_function_names function_names);
599     (check_function_not_print function_names);
600     (check_prog_contains_main function_names);
601     let sast_funcs = (List.map (fun n -> check_function_body (fst n) (snd n))
        functions_with_env) in
602     (*let sprogram:(S.sprogram) = program_sast (globals_add, functions, structs_add)
        in *)
603     let sast = (includes, globals_add, sast_funcs, (List.map struct_sast structs_add
        )) in
604     sast
605     (* Need to check function test + using code here *)
606
607 let check_includes includes =

```

```

608 let headers = List.map (fun n -> snd n) includes in
609 report_duplicate (fun n -> "duplicate header file " ^ n) headers;
610 List.iter check_ends_in_jt headers;
611 ()
612
613
614 (*****)
615 (* Entry point for semantic checking AST. Output is SAST *)
616 (*****)
617 let check (includes, globals, functions, structs) =
618   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
619   let _ = check_includes includes in
620   let (structs_added, struct_methods) = check_structs structs in
621   let globals_added = check_globals globals prog_env in
622   let functions_with_env = List.map (fun n -> (n, prog_env)) functions in
623   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
624   let sast = check_functions (functions_with_env @ methods_with_env) includes
        globals_added structs_added in
625   sast

```

12.4 sast.ml

```
1 open Ast
2
3 type var_info = (string * typ)
4
5 type sexpr =
6   SLit      of int
7   | SString_lit of string
8   | SChar_lit of char
9   | SDouble_lit of float
10  | SBinop    of sexpr * op * sexpr * typ
11  | SUnop     of uop * sexpr
12  | SAssign   of sexpr * sexpr
13  | SNoexpr
14  | SId of string
15  | SStruct_create of string
16  | SStruct_access of string * string * int
17  | SPt_access of string * string * int
18  | SArray_create of int * prim
19  | SArray_access of string * int * typ
20  | SDereference of string
21  | SFree of string
22  | SCall of string * sexpr list
23  | SBoolLit of int
24  | SNull of typ
25  | SDubs
26
27 type sstmt =
28   SBlock of sstmt list
29   | SExpr of sexpr
30   | SIf of sexpr * sstmt * sstmt
31   | SWhile of sexpr * sstmt
32   | SFor of sexpr * sexpr * sexpr * sstmt
33   | SReturn of sexpr
34
35 type swith_using_decl = {
36   suvdecls : bind list;
37   sstmts : sstmt list;
38 }
39
40 type swith_test_decl = {
41   sasserts : sstmt list;
42   susing : swith_using_decl;
43 }
44
45 (* Node that describes a function *)
46 type sfunc_decl = {
47   styp : typ;
48   sfname : string;
49   sformals : bind list;
50   svdecls : bind list;
51   sbody : sstmt list;
52   stests : sfunc_decl option;
53 }
54
55 (* Node that describes a given struct *)
56 type sstruct_decl = {
57   ssname : string;
```

```
58     sattributes    : bind list;  
59 }  
60  
61 (* Root of tree. Our program is made up three things 1) list of global variables  
62    2) list of functions 3) list of struct definition *)  
type sprogram = header list * bind list * sfunc_decl list * sstruct_decl list
```

12.5 codegen.ml

```
1
2 module L = Llvm
3 module A = Ast
4 module S = Sast
5 module C = Char
6 module StringMap = Map.Make(String)
7
8 let context = L.global_context ()
9 (* module is what is returned from this file aka the LLVM code *)
10 let main_module = L.create_module context "Jateste"
11 let test_module = L.create_module context "Jateste-test"
12
13 (* Defined so we don't have to type out L.i32_type ... every time *)
14 let i32_t = L.i32_type context
15 let i64_t = L.i64_type context
16 let i8_t = L.i8_type context
17 let i1_t = L.i1_type context
18 let d_t = L.double_type context
19 let void_t = L.void_type context
20 let str_t = L.pointer_type i8_t
21
22 (* Hash table of the user defined structs *)
23 let struct_types:(string, L.lltype) Hashtbl.t = Hashtbl.create 10
24 (* Hash table of global variables *)
25 let global_variables:(string, L.llvalue) Hashtbl.t = Hashtbl.create 50
26
27 (* Helper function that returns L.lltype for a struct. This should never fail as
   semantic checker should catch invalid structs *)
28 let find_struct_name name =
29   try Hashtbl.find struct_types name
30   with | Not_found -> raise(Exceptions.InvalidStruct name)
31
32 let rec index_of_list x l =
33   match l with
34   [] -> raise (Exceptions.InvalidStructField)
35   | hd::tl -> let (_,y) = hd in if x = y then 0 else 1 + index_of_list x tl
36
37
38 (* Code to declare struct *)
39 let declare_struct s =
40   let struct_t = L.named_struct_type context s.S.ssname in
41   Hashtbl.add struct_types s.S.ssname struct_t
42
43
44 let prim_ltype_of_typ = function
45   A.Int -> i32_t
46   | A.Double -> d_t
47   | A.Char -> i8_t
48   | A.Void -> void_t
49   | A.String -> str_t
50   | A.Bool -> i1_t
51
52
53 let rec ltype_of_typ = function
54   | A.Primitive(s) -> prim_ltype_of_typ s
55   | A.Struct_typ(s) -> find_struct_name s
56   | A.Pointer_typ(s) -> L.pointer_type (ltype_of_typ s)
```

```

57 | A.Array_typ(t,n) -> L.array_type (prim_ltype_of_typ t) n
58 | _ -> void_t
59
60 let type_of_llvalue v = L.type_of v
61
62 let string_of_expr e =
63   match e with
64   | S.SId(s) -> s
65   | _ -> raise (Exceptions.BugCatch "string_of_expr")
66
67 (* Function that builds LLVM struct *)
68 let define_struct_body s =
69   let struct_t = Hashtbl.find struct_types s.S.ssname in
70   let attribute_types = List.map (fun (t, _) -> t) s.S.sattributes in
71   let attributes = List.map ltype_of_typ attribute_types in
72   let attributes_array = Array.of_list attributes in
73   L.struct_set_body struct_t attributes_array false
74
75 (* Helper function to create an array of size i fille with l values *)
76 let array_of_zeros i l =
77   Array.make i l
78
79 let default_value_for_prim_type t =
80   match t with
81   | A.Int -> L.const_int (prim_ltype_of_typ t) 0
82   | A.Double -> L.const_float (prim_ltype_of_typ t) 0.0
83   | A.String -> L.const_string context ""
84   | A.Char -> L.const_int (prim_ltype_of_typ t) 0
85   | A.Void -> L.const_int (prim_ltype_of_typ t) 0
86   | A.Bool -> L.const_int (prim_ltype_of_typ t) 0
87
88 (* Here we define and initailize global vars *)
89 let define_global_with_value (t, n) =
90   match t with
91   | A.Primitive(p) ->
92     (match p with
93     | A.Int -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n
94       init main_module)
95     | A.Double -> let init = L.const_float (ltype_of_typ t) 0.0 in (L.
96       define_global n init main_module)
97     | A.String -> let init = L.const_pointer_null (ltype_of_typ t) in (L.
98       define_global n init main_module)
99     | A.Void -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n
100       init main_module)
101     | A.Char -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n
102       init main_module)
103     | A.Bool -> let init = L.const_int (ltype_of_typ t) 0 in (L.define_global n
104       init main_module)
105     )
106   | A.Struct_typ(s) -> let init = L.const_named_struct (find_struct_name s) [||]
107     in (L.define_global n init main_module)
108
109   | A.Pointer_typ(_) -> let init = L.const_pointer_null (ltype_of_typ t) in (L.
110     define_global n init main_module)
111
112   | A.Array_typ(p,i) -> let init = L.const_array (prim_ltype_of_typ p) (
113     array_of_zeros i (default_value_for_prim_type ((p)))) in (L.define_global n
114     init main_module)

```



```

106 | A.Func_typ(_) ->let init = L.const_int (ltype_of_typ t) 0 in (L.
    define_global n init main_module)
107 | A.Any -> raise (Exceptions.BugCatch "define_global_with_value")
108
109
110 (* Where we add global variabes to global data section *)
111 let define_global_var (t, n) =
112     match t with
113     | A.Primitive(_) -> Hashtbl.add global_variables n (define_global_with_value (
        t,n))
114     | A.Struct_typ(_) -> Hashtbl.add global_variables n (define_global_with_value
        (t,n))
115     | A.Pointer_typ(_) -> Hashtbl.add global_variables n (
        define_global_with_value (t,n))
116     | A.Array_typ(_,_) -> Hashtbl.add global_variables n (define_global_with_value
        (t,n))
117     | A.Func_typ(_) -> Hashtbl.add global_variables n (L.declare_global (
        ltype_of_typ t) n main_module)
118     | A.Any -> raise (Exceptions.BugCatch "define_global_with_value")
119
120
121 (* Translations functions to LLVM code in text section *)
122 let translate_function functions the_module =
123
124 (* Here we define the built in print function *)
125 let printf_t = L.var_arg_function_type i32_t [||] in
126 let printf_func = L.declare_function "printf" printf_t the_module in
127
128
129 (* Here we iterate through Ast.functions and add all the function names to a
    HashMap *)
130 let function_decls =
131     let function_decl m fdecl =
132         let name = fdecl.S.sfname
133         and formal_types =
134             Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.S.
                sformals)
135             in let ftype = L.function_type (ltype_of_typ fdecl.S.styp)
                formal_types in
136         StringMap.add name (L.define_function name ftype the_module, fdecl)
137     m in
138     List.fold_left function_decl StringMap.empty functions in
139
140 (* Create format strings for printing *)
141 let (main_function,_) = StringMap.find "main" function_decls in
142 let builder = L.builder_at_end context (L.entry_block main_function) in
143 (*let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in *)
144 let str_format_str = L.build_global_stringptr "%s\n" "fmt_string" builder in
145 let int_format_str = L.build_global_stringptr "%d\n" "fmt_int" builder in
146 let float_format_str = L.build_global_stringptr "%f\n" "fmt_float" builder in
147
148 (* Method to build body of function *)
149 let build_function_body fdecl =
150     let (the_function, _) = StringMap.find fdecl.S.sfname function_decls in
151     (* builder is the LLVM instruction builder *)
152     let builder = L.builder_at_end context (L.entry_block the_function) in
153

```

```

154 (* This is where we push local variables onto the stack and add them to a local
    HashMap*)
155 let local_vars =
156   let add_formal m(t, n) p = L.set_value_name n p;
157   let local = L.build_alloca (ltype_of_ttyp t) n builder in
158   ignore (L.build_store p local builder);
159   StringMap.add n local m in
160
161   let add_local m (t, n) =
162     let local_var = L.build_alloca (ltype_of_ttyp t) n builder
163     in StringMap.add n local_var m in
164
165 (* This is where we push formal arguments onto the stack *)
166 let formals = List.fold_left2 add_formal StringMap.empty fdecl.S.sformals
167   (Array.to_list (L.params the_function)) in
168   List.fold_left add_local formals fdecl.S.svdecls in
169
170
171 (* Two places to look for a variable 1) local HashMap 2) global HashMap *)
172 let find_var n = try StringMap.find n local_vars
173   with Not_found -> try Hashtbl.find global_variables n
174   with Not_found -> raise (Failure ("undeclared variable " ^ n))
175   in
176
177 (*
178   let type_of_expr e =
179   let tmp_type = L.type_of e in
180   let tmp_string = L.string_of_lltype tmp_type in ignore(print_string
181   tmp_string);
182   match tmp_string with
183     "i32*" -> A.Primitive(A.Int)
184     | "i32" -> A.Primitive(A.Int)
185     | "i8" -> A.Primitive(A.Char)
186     | "i8*" -> A.Primitive(A.Char)
187     | "i1" -> A.Primitive(A.Bool)
188     | "i1*" -> A.Primitive(A.Bool)
189     | "double" -> A.Primitive(A.Double)
190     | "double*" -> A.Primitive(A.Double)
191     | _ -> raise (Exceptions.BugCatch ("type_of_expr"))
192   in
193   *)
194
195 (* Format to print given arguments in print(...) *)
196 let print_format e =
197   (match e with
198     (S.SString_lit(_)) -> str_format_str
199     | (S.SLit(_)) -> int_format_str
200     | (S.SDouble_lit(_)) -> float_format_str
201     | (S.SId(i)) -> let i_value = find_var i in
202       let i_type = L.type_of i_value in
203       let string_i_type = L.string_of_lltype i_type in
204       (match string_i_type with
205         "i32*" -> int_format_str
206         | "i8*" -> str_format_str
207         | "float*" -> float_format_str
208         | "double*" -> float_format_str
209         | _ -> raise (Exceptions.InvalidPrintFormat))
210     | _ -> raise (Exceptions.InvalidPrintFormat)
211   )

```

```

211     in
212
213     (* Returns address of i. Used for lhs of assignments *)
214     let rec addr_of_expr i builder =
215     match i with
216     | S.SLit(_) -> raise Exceptions.InvalidLhsOfExpr
217     | S.SString_lit(_) -> raise Exceptions.InvalidLhsOfExpr
218     | S.SChar_lit(_) -> raise Exceptions.InvalidLhsOfExpr
219     | S.SId(s) -> find_var s
220     | S.SBinop(_,_,_,_) -> raise (Exceptions.UndeclaredVariable("Unimplemented
221                               addr_of_expr"))
222     | S.SUnop(_,e) -> addr_of_expr e builder
223     | S.SStruct_access(s,_,index) -> let tmp_value = find_var s in
224       let deref = L.build_struct_gep tmp_value index "tmp" builder in deref
225     | S.SPt_access(s,_,index) -> let tmp_value = find_var s in
226       let load_tmp = L.build_load tmp_value "tmp" builder in
227       let deref = L.build_struct_gep load_tmp index "tmp" builder in deref
228     | S.SDereference(s) -> let tmp_value = find_var s in
229       let deref = L.build_gep tmp_value [|L.const_int i32_t 0|] "tmp" builder in L
230       .build_load deref "tmp" builder
231
232     | S.SArray_access(ar,index, t) -> let tmp_value = find_var ar in
233       (match t with
234       | A.Array_typ(_) -> let deref = L.build_gep tmp_value [|L.const_int i32_t 0 ;
235         L.const_int i32_t index|] "arrayvalueaddr" builder in deref
236       | A.Pointer_typ(_) -> let loaded_value = L.build_load tmp_value "tmp" builder
237         in
238         let deref = L.build_gep loaded_value [|L.const_int i32_t 0 ; L.const_int
239         i32_t index|] "arrayvalueaddr" builder in deref
240       | _ -> raise Exceptions.InvalidArrayAccess)
241     | _ -> raise (Exceptions.UndeclaredVariable("Invalid LHS of assignment"))
242
243     in
244     let add_terminal builder f =
245       match L.block_terminator (L.insertion_block builder) with
246       | Some _ -> ()
247       | None -> ignore (f builder) in
248
249     (* This is where we build LLVM expressions *)
250     let rec expr builder = function
251     | S.SLit l -> L.const_int i32_t l
252     | S.SString_lit s -> let temp_string = L.build_global_stringptr s "str" builder
253       in temp_string
254     | S.SChar_lit c -> L.const_int i8_t (C.code c)
255     | S.SDouble_lit d -> L.const_float d_t d
256     | S.SBinop(e1, op, e2,t) ->
257       let e1' = expr builder e1
258       and e2' = expr builder e2 in
259       (match t with
260       | A.Primitive(A.Int) | A.Primitive(A.Char) -> (match op with
261       | A.Add -> L.build_add
262       | A.Sub -> L.build_sub
263       | A.Mult -> L.build_mul
264       | A.Equal -> L.build_icmp L.Icmp.Eq
265       | A.Neq -> L.build_icmp L.Icmp.Ne
266       | A.Less -> L.build_icmp L.Icmp.Slt
267       | A.Leq -> L.build_icmp L.Icmp.Sle
268       | A.Greater -> L.build_icmp L.Icmp.Sgt
269       | A.Geq -> L.build_icmp L.Icmp.Sge

```

```

264 | _ -> raise (Exceptions.BugCatch "Binop")
265 )e1' e2' "add" builder
266 | A.Primitive(A.Double) ->
267 (match op with
268   A.Add -> L.build_fadd
269 | A.Sub -> L.build_fsub
270 | A.Mult -> L.build_fmul
271 | A.Equal -> L.build_fcmp L.Fcmp.Oeq
272 | A.Neq -> L.build_fcmp L.Fcmp.One
273 | A.Less -> L.build_fcmp L.Fcmp.Olt
274 | A.Leq -> L.build_fcmp L.Fcmp.Ole
275 | A.Greater -> L.build_fcmp L.Fcmp.Ogt
276 | A.Geq -> L.build_fcmp L.Fcmp.Oge
277 | _ -> raise (Exceptions.BugCatch "Binop")
278 ) e1' e2' "addfloat" builder
279 | A.Primitive(A.Bool) ->
280 (
281 match op with
282   A.And -> L.build_and
283 | A.Or -> L.build_or
284 | A.Equal -> L.build_icmp L.Icmp.Eq
285 | _ -> raise (Exceptions.BugCatch "Binop")
286 ) e1' e2' "add" builder
287 | A.Pointer_typ(_) ->
288 (match op with
289   A.Equal -> L.build_is_null
290 | A.Neq -> L.build_is_not_null
291 | _ -> raise (Exceptions.BugCatch "Binop")
292 )e1' "add" builder
293 | _ -> raise (Exceptions.BugCatch "Binop"))
294
295 | S.SUnop(u,e) ->
296 (match u with
297   A.Neg -> let e1 = expr builder e in L.build_not e1 "not" builder
298 | A.Not -> let e1 = expr builder e in L.build_not e1 "not" builder
299 | A.Addr -> let iden = string_of_expr e in
300   let lvalue = find_var iden in lvalue
301 )
302 | S.SAssign (l, e) -> let e_temp = expr builder e in
303 ignore(let l_val = (addr_of_expr l builder) in (L.build_store e_temp l_val
builder)); e_temp
304 | S.SNoexpr -> L.const_int i32_t 0
305 | S.SId (s) -> L.build_load (find_var s) s builder
306 | S.SStruct_create(s) -> L.build_malloc (find_struct_name s) "tmp" builder
307 | S.SStruct_access(s,_,index) -> let tmp_value = find_var s in
308   let deref = L.build_struct_gep tmp_value index "tmp" builder in
309   let loaded_value = L.build_load deref "dd" builder in loaded_value
310 | S.SPt_access(s,_,index) -> let tmp_value = find_var s in
311   let load_tmp = L.build_load tmp_value "tmp" builder in
312   let deref = L.build_struct_gep load_tmp index "tmp" builder in
313   let tmp_value = L.build_load deref "dd" builder in tmp_value
314 | 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
315 | S.SArray_access(ar,index,t) -> let tmp_value = find_var ar in
316 (match t with
317   A.Pointer_typ(_) -> let loaded_value = L.build_load tmp_value "loaded"
builder in
318   let deref = L.build_gep loaded_value [|L.const_int i32_t 0 ; L.const_int
i32_t index|] "arrayvalueaddr" builder in

```

```

319     let final_value = L.build_load deref "arrayvalue" builder in final_value
320 | 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
321     let final_value = L.build_load deref "arrayvalue" builder in final_value
322 | _ -> raise Exceptions.InvalidArrayAccess)
323 | S.SDereference(s) -> let tmp_value = find_var s in
324     let load_tmp = L.build_load tmp_value "tmp" builder in
325     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
326
327 | S.SFree(s) -> let tmp_value = L.build_load (find_var s) "tmp" builder in L.
    build_free (tmp_value) builder
328 | S.SCall("print", [e]) | S.SCall("print_int", [e]) -> L.build_call printf_func
    [(print_format e); (expr builder e)] "printresult" builder
329 | S.SCall(f, args) -> let (def_f, fdecl) = StringMap.find f function_decls in
330     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
331 | S.SBoolLit(b) -> L.const_int i1_t b
332 | S.SNull(t) -> L.const_null (ltype_of_type t)
333 | S.SDubs -> let tmp_call = S.SCall("print", [(S.SString_lit("dubs!"))]) in expr
    builder tmp_call
334 in
335
336
337 (* This is where we build the LLVM statements *)
338 let rec stmt builder = function
339     S.SBlock b -> List.fold_left stmt builder b
340 | S.SExpr e -> ignore (expr builder e); builder
341
342
343 | S.SIf(pred, then_stmt, else_stmt) ->
344     (*let curr_block = L.insertion_block builder in *)
345     (* the function (of type llvalue that we are currently in *)
346     let bool_val = expr builder pred in
347     let merge_bb = L.append_block context "merge" the_function in
348     (* then block *)
349     let then_bb = L.append_block context "then" the_function in
350
351     add_terminal (stmt (L.builder_at_end context then_bb) then_stmt) (L.build_br
    merge_bb);
352     (* else block*)
353     let else_bb = L.append_block context "else" the_function in
354     add_terminal (stmt (L.builder_at_end context else_bb) else_stmt) (L.build_br
    merge_bb);
355     ignore (L.build_cond_br bool_val then_bb else_bb builder);
356     L.builder_at_end context merge_bb
357 | S.SWhile(pred, body_stmt) ->
358     let pred_bb = L.append_block context "while" the_function in
359     ignore (L.build_br pred_bb builder);
360     let body_bb = L.append_block context "while_body" the_function in
361     add_terminal (stmt (L.builder_at_end context body_bb) body_stmt) (L.build_br
    pred_bb);
362     let pred_builder = L.builder_at_end context pred_bb in
363     let bool_val = expr pred_builder pred in
364     let merge_bb = L.append_block context "merge" the_function in
365     ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
366     L.builder_at_end context merge_bb
367

```

```

368 | S.SFor(e1,e2,e3,s) -> ignore(expr builder e1); let tmp_stmt = S.SExpr(e3) in
369 |   let tmp_block = S.SBlock([s] @ [tmp_stmt]) in
370 |   let tmp_while = S.SWhile(e2, tmp_block) in stmt builder tmp_while
371 | S.SReturn r -> ignore (match fdecl.S.styp with
372 |   A.Primitive(A.Void) -> L.build_ret_void builder
373 |   _ -> L.build_ret (expr builder r) builder); builder
374 in
375
376 (* Build the body for this function *)
377 let builder = stmt builder (S.SBlock fdecl.S.sbody) in
378
379 add_terminal builder (match fdecl.S.styp with
380 |   A.Primitive(A.Void) -> L.build_ret_void
381 |   _ -> L.build_ret (L.const_int i32_t 0) )
382 in
383
384 (* Here we go through each function and build the body of the function *)
385 List.iter build_function_body functions;
386 the_module
387
388 (* Create a main function in test file - main then calls the respective tests *)
389 let test_main functions =
390 |   let tests = List.fold_left (fun l n -> (match n.S.stests with Some(t) -> l @ [t]
391 |   | None -> l)) [] functions in
392 |   let names_of_test_calls = List.fold_left (fun l n -> l @ [(n.S.sfname)]) []
393 |   tests in
394 |   let sast_calls = List.fold_left (fun l n -> l @ [S.SExpr(S.SCall("print",[S.
395 |   S.String_lit(n ^ " tests:")))] @ [S.SExpr(S.SCall(n,[[]]))] []
396 |   names_of_test_calls in
397 |   let print_stmt = S.SExpr(S.SCall("print",[S.SString_lit("Tests:"))]) in
398 |   let tmp_main:(S.sfunc_decl) = { S.styp = A.Primitive(A.Void); S.sfname = "main";
399 |   S.sformals = []; S.svdecls = []; S.sbody = print_stmt::sast_calls; S.stests=
400 |   None; } in tmp_main
401
402
403 let func_builder f b =
404 |   (match b with
405 |   true -> let tests = List.fold_left (fun l n -> (match n.S.stests with Some(t)
406 |   -> l @ [n] @ [t] | None -> l)) [] f in (tests @ [(test_main f)])
407 |   false -> f
408 |   )
409
410 (*****
411 (* Entry point for translating Ast.program to LLVM module *)
412 (*****
413 let gen_llvm (_, input_globals, input_functions, input_structs) gen_tests_bool =
414 |   let _ = List.iter declare_struct input_structs in
415 |   let _ = List.iter define_struct_body input_structs in
416 |   let _ = List.iter define_global_var input_globals in
417 |   let the_module = (match gen_tests_bool with true -> test_module | false ->
418 |   main_module) in
419 |   let _ = translate_function (func_builder input_functions gen_tests_bool)
420 |   the_module in
421 |   the_module

```

12.6 myprinter.ml

12.7 exceptions.ml

```
1 (* Program structure exceptions *)
2 exception MissingMainFunction
3
4 exception InvalidHeaderFile of string
5
6 (* Struct exceptions*)
7 exception InvalidStruct of string
8
9 (* Variable exceptions*)
10 exception UndeclaredVariable of string
11
12 (*Expression exceptions *)
13 exception InvalidExpr of string
14 exception InvalidBooleanExpression
15 exception IllegalAssignment
16 exception InvalidFunctionCall of string
17 exception InvalidArgumentsToFunction of string
18 exception InvalidArrayVariable
19 exception InvalidStructField
20 exception InvalidFree of string
21 exception InvalidPointerDereference
22 exception NotBoolExpr
23 exception InvalidArrayAccess
24
25 (* Print exceptions *)
26 exception InvalidPrintCall
27 exception InvalidPrintFormat
28
29 (* Statement exceptions*)
30 exception InvalidReturnType of string
31 exception InvalidLhsOfExpr
32
33 (* Bug catcher *)
34 exception BugCatch of string
35
36 (* Input *)
37 exception IllegalInputFormat
38 exception IllegalArgument of string
39
40 (* Test cases *)
41 exception InvalidTestAsserts
42 exception InvalidAssert of string
```


12.8 jatest.ml

```
1 open Printf
2 module A = Ast
3 module S = Sast
4
5
6 let standard_library_path = "/home/plt/JaTeste/lib/"
7 let current_dir_path = "./"
8
9 type action = Scan | Parse | Ast | Sast | Compile | Compile_with_test
10
11 (* Determines what action compiler should take based on command line args *)
12 let determine_action args =
13   let num_args = Array.length args in
14   (match num_args with
15    | 1 -> raise Exceptions.IllegalInputFormat
16    | 2 -> Compile
17    | 3 -> let arg = Array.get args 1 in
18      (match arg with
19       | "-t" -> Compile_with_test
20       | "-l" -> Scan
21       | "-p" -> Parse
22       | "-se" -> Sast
23       | "-ast" -> Ast
24       | _ -> raise (Exceptions.IllegalArgument arg)
25      )
26
27   | _ -> raise (Exceptions.IllegalArgument "Can't recognize arguments")
28   )
29
30 (* Create executable filename *)
31 let executable_filename filename =
32   let len = String.length filename in
33   let str = String.sub filename 0 (len - 3) in
34   let exec = String.concat "" [str ; ".ll"] in
35   exec
36
37 (* Create test executable filename *)
38 let test_executable_filename filename =
39   let len = String.length filename in
40   let str = String.sub filename 0 (len - 3) in
41   let exec = String.concat "" [str ; "-test.ll"] in
42   exec
43
44 (* Just scan input *)
45 let scan input_raw =
46   let lexbuf = Lexing.from_channel input_raw in (print_string "Scanned\n"); lexbuf
47
48 (* Scan, then parse input *)
49 let parse input_raw =
50   let input_tokens = scan input_raw in
51   let ast:(A.program) = Parser.program Scanner.token input_tokens in (print_string
52     "Parsed\n"); ast
53
54 (* Process include statements. Input is ast, and output is a new ast *)
55 let process_headers ast:(A.program) =
56   let (includes,_,_,_) = ast in
57   let gen_header_code (incl,globals, current_func_list, structs) (path, str) =
```

```

57     let tmp_path = (match path with A.Curr -> current_dir_path | A.Standard ->
58     standard_library_path) in
59     let file = tmp_path ^ str in
60     let ic =
61     try open_in file with _ -> raise (Exceptions.InvalidHeaderFile file) in
62     let (_,_,funcs,structs) = parse ic in
63     let new_ast:(A.program) = (incl, globals, current_func_list @ funcs, structs @
64     structs) in
65     new_ast
66 in
67 let modified_ast:(A.program) = List.fold_left gen_header_code ast includes in
68 modified_ast
69
70 (* Scan, parse, and run semantic checking. Returns Sast *)
71 let semant input_raw =
72     let tmp_ast = parse input_raw in
73     let input_ast = process_headers tmp_ast in
74     let sast:(S.sprogram) = Semant.check input_ast in (print_string "Semantic check
75     passed\n"); sast
76
77 (* Generate code given file. @bool_tests determines whether to create a test file
78 *)
79 let code_gen input_raw exec_name bool_tests =
80     let input_sast = semant input_raw in
81     let file = exec_name in
82     let oc = open_out file in
83     let m = Codegen.gen_llvm input_sast bool_tests in
84     Llvm_analysis.assert_valid_module m;
85     fprintf oc "%s\n" (Llvm.string_of_llmodule m);
86     close_out oc;
87     ()
88
89 let get_ast input_raw =
90     let ast = parse input_raw in
91     ast
92
93 (*****
94 (* Entry pointer for Compiler *)
95 (*****
96 let _ =
97     (* Read in command line args *)
98     let arguments = Sys.argv in
99     (* Determine what the compiler should do based on command line args *)
100     let action = determine_action arguments in
101     let source_file = open_in arguments.((Array.length Sys.argv - 1)) in
102     (* Create a file to put executable in *)
103     let exec_name = executable_filename arguments.((Array.length Sys.argv - 1)) in
104     (* Create a file to put test executable in *)
105     let test_exec_name = test_executable_filename arguments.((Array.length Sys.argv
106     - 1)) in
107
108     let _ = (match action with
109     Scan -> let _ = scan source_file in ()
110     | Parse -> let _ = parse source_file in ()
111     | Ast -> let _ = parse source_file in ()
112     | Sast -> let _ = semant source_file in ())

```

```
111 | Compile -> let _ = code_gen source_file exec_name false in ()
112 | Compile_with_test -> let _ = code_gen source_file exec_name false in
113     let source_test_file = open_in arguments.((Array.length Sys.argv - 1)) in
        let _ = code_gen source_test_file test_exec_name true in ()
114 ) in
115 close_in source_file
```