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

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 an imperative, C-like language, with a few object oriented features added, that makes it easy to add test cases to one's code. The syntax is very similar to C, but with added capability of adding methods directly to "structs". Furthermore, test cases are appended to user-defined functions, that enable the user to define test cases right away. By appending the keyword "with test" onto the end of a function, the programmer is able to test his or her code straight away. Code samples will be supplied through the report.

1.3   Running the JaTeste Compiler

At compile time, The JaTesté compiler generates two files: (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. All code is compiled into LLVM, a portable assembly-like language. To run the compiled LLVM code, we use 'lli", an LLVM interpreter.

For the first file, the compiler completely disregards the test cases and thus produces an executable without any of the test cases code. This enables the programmer to produce a regular executable without the overhead of the test cases when he or she desired.

For the second file, the compiler turns the test cases into functions, and precedes to run each function from a completely brand new "main" method. "main" essentially runs through each function, each of which runs the user-defined tests. Furthermore, the compiler adds "printf" calls to each test letting the user known whether a given test passed or failed.

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

The optional -options are:

- No arguments If run without arguments, the compiler ignores the test cases and creates one executable.

- "-t" Compile with test This results in the compiler creating two LLVM files: 1) a regular executable named "source_file.ll" 2) a test file named "source_file-test.ll"

- "-l" Scan only This results in the compiler simply scanning the source code

- "-p" Parse only This results in the compiler simply parsing the source code

- "-se" SAST This results in the compiler running the semantic checker on the source code and the stopping.

- "-ast" AST This also results in the compiler running the semantic checker on the source code and the stopping.

2 Short Tutorial

2.1 JaTesté Overview

Any given JaTesté program can be broken down into three segments: 1) global variable declarations 2) function definitions 3) struct definitions.

Global variable declarations are exactly like in C.

Function definitions are similar to C, except the keyword "func" is needed before the return type. Furthermore, all variable declarations must be done at the top of a function. Any function with return type other than void, must use the keyword "return" to return the given value; code cannot be written after a return statement.

Structs are also similar to C, except the programmer can define methods within the struct.

2.2 Samples Programs

1. Here's the first example of a JaTesté program. As illustrated, the syntax is very similar to C. Note the keyword "func" that is needed for defining functions.

```
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(add(a,0)  == 10);
17  } using {
18         int a;
19         int b;
20         a = 10;
21         b = 5;
22  }
```

As can be seen the "add" function has a snippet of code directly preceding it. This is an example of using test cases. The code within the "with test" block defines the test cases for the add function. Furthermore, note the code following the test case that starts with "using ...". The block is used to setup the environment for the test cases. In this example, the test single test case "assert(a == 10);" references the variable "a"; it is within "using " block scope that a is defined.

2. Here's another example program:

```
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(sub(10,5) == b - 5);
24               assert(sub(b,d) == 1);
25               assert(sub(c,d) == 4);
26      } using {
27               int a;
28               int b;
29               int c;
30               int d;
31               a = 5;
32               b = 10;
33               c = 13;
34               d = 9;
35      }
```

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

3. Here's another example that uses structs. The syntax is very similar to C:

```
 1      int global_var;
 2
 3      func int main()
 4      {
 5               int tmp;
 6               struct rectangle *rec_pt;
 7               rec_pt = new struct rectangle;
 8               update_rec(rec_pt, 6);
 9               tmp = rec_pt->width;
10
11               print(tmp);
12
13               return 0;
14      }
15
16      func void update_rec(struct rectangle *p, int x)
```

```
17   {
18           p->width = x;
19   } with test {
20           assert(t->width == 10);
21   } using {
22           struct rectangle *t;
23           t = new struct rectangle;
24           update_rec(t, 10);
25   }
26
27   struct rectangle {
28           int width;
29           int height;
30   };
```

Note the syntax here: global variables are declared at the top, functions are defined in the middle, and structs are defined at the bottom. This is required for all JaTesté 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, bool, if, else, for, while, with test, using, assert, true, false, func, method, malloc, free, NULL, 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 = (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:

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

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 }
```

3.8 Data Types

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

3.9 Primitives

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

3.9.1 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 deceleration is:

```
typ ID
```

The grammar that recognizes an integer initialization is:

```
ID ASSIGN expr
```

3.9.2 bool Types

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

```
1  bool my_bool;
2  my_bool = true;
```

3.9.3 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 deceleration is:

```
typ ID
```

The grammar that recognizes a double initialization is:

```
ID ASSIGN expr
```

3.9.4 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 deceleration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

3.9.5 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.

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

The grammar that recognizes a char deceleration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

3.10 Structures

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

3.10.1 Defining Structures

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

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

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

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

};

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

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

```
STRUCT ID LBRACE vdecl_list struc\_func\_decls RBRACE SEMI
```

3.10.2 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". When using the "new" keyword, the memory is allocated on the heap for the given structs. Structs can also be allocated on the stack as follows:

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

3.10.3 Accessing Structure Members

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

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

If the struct is allocated on the stack, use:

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

```
expr DOT expr
```

3.11 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.

3.11.1 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
```

3.11.2 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.

3.11.3 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:

```
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.

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

4   Expressions and Operators

4.1   Expressions

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

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

The grammar for expressions is:

```
expr:
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
```

4.2   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).

4.3   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 to 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.

4.4   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

- $\wedge$ 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
```

4.5 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
```

4.6 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
```

4.7 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
```

4.8 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.

5 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
```

5.1 If Statement

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

```
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
```

5.2 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
```

5.3 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
```

5.4 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.

5.5 Return Statement

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

```
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.

6   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.

6.1   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 */
```

6.2   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.

6.3 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 */
```

6.4 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;
```

6.5   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.

6.6   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 enviornment 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 enviornment for it's 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.

7 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.

7.1 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

7.2 Workflow

7.3 Github Stats

7.4 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....

7.5 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

8 Architecture

8.1 Block Diagram

8.2 The Compiler

8.3 The Scanner

8.4 The Parser

8.5 The Semantic Checker

8.6 The Code Generator

8.7 Supplementary Code

9 Testing

9.1 Test Plan

9.2 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!
======= 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!
=========== All Tests Passed! =============

9.3   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

9.4   Tests

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

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

   toyota = new struct car;

   toyota->priice;

   return 0;
}

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

struct-access-fail1.out

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

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

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

  my_bool = true;
  my_bool2 = false;

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

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

  return 0;
}
```

test-bool1.out

```
or passed
and passed
```

test-bool2.jt

```
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-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

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

  sam = new struct person;

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

  free(sam);

  print("freed");


  return 0;
}

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

test-for1.out

```
0
1
2
3
4
```

test-free1.jt

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

  sam = new struct person;

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

  free(sam);

  print("freed");


  return 0;
}

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

test-free1.out

```
freed
```

test-func1.jt

```
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

```
#include_jtlib <math.jt>

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

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

test-lib1.out

```
passed
```

test-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
```

10   Conclusion

10.1   Lessons Learned

10.1.1   Andrew

10.1.2   Jemma

10.1.3   Jared

10.1.4   Jake

11   Code