Lorax Language

Doug Bienstock (dmb2168)
Chris D'Angelo (cd2665)
Zhaarn Maheswaran (zsm2103)
Tim Paine (tkp2108)
Kira Whitehouse (kbw2116)

Columbia University

December 19, 2013

"I am the Lorax. I speak for the trees." - Dr. Seuss 1971

Table of Contents

Introduction

Project Overview

Lorax is an imperative tree manipulation language. While tree based operations can be accomplished in most popular languages through the use of a standard library, we wanted to design a language with the tree as the central structure and an minimal syntax that would make it easier not only to program tree based algorithms, but to understand them as well.

Language Goals

Trees are often taught in the context of a given language's standard libraries. They are often misunderstood by students, who resort to using predesigned tree data structures, rather than building their own. With Lorax, we present an environment which promotes the use of trees, providing an intuitive syntax to aid in programmer understanding, while abstracting the more complex pointer operations being done "under the hood".

Language Tutorial

Installing the Compiler

Installation of the Lorax compiler requires the git version control tool, as well as the suite of OCaml compilers. For full compilation, a GCC compiler is also required. Alternatively, the Lorax compiler executable can be downloaded using the following git command:

```
git clone https://github.com/mychrisdangelo/LoraxLanguageCompiler
```

A First Example

Here is a sample Lorax program which prints "Hello, World". In Lorax, "strings" are simply wrappers for a tree of characters, so the internal representation is that of a tree.

```
int main() {
      print("hello, world");
}
```

Running the compiler

Our compiler runs with a number of flags, which allow the user to inspect the compiler's output from one of four primary phases. The Lorax programmer may also compile directly to machine code using the -b flag which invokes the native gcc compiler.

```
-a source.lrx                (Print AST of source)
-t source.lrx                (Print Symbol Table of source)
-s source.lrx                (Run Semantic Analysis over source)
-c source.lrx [target.c]     (Compile to c. Second argument optional)
-b source.lrx [target.out]   (Compile to executable)
```

Language Reference Manual

Introduction

This manual describes the Lorax programming language. The Lorax language provides a syntax that enables the easy creation and manipulation of the tree abstract data type. Trees are a native data type of the language. Each tree encloses a value of a Lorax primitive type. Tree's branching factor is dynamically typed and value data type is statically typed. Language operators allow you to insert trees, traverse their structure, access their node contents, and compare data items within tree nodes. The programmer can create and manipulate these trees while the Lorax language handles memory management and tree structural consistency under the hood.

Lexical Conventions

Comments

In-line comments are preceded by `//`. Block comments are delimited by `/*` and `*/`. Block comments can be written on a single line or can span multiple lines. Nesting is not allowed.

Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. If identifiers are a length greater than 10 characters the behavior is undefined.

Keywords

The following identifiers are reserved for the use as keywords, and may not be used otherwise:

```
int         root     char
float       mod      degree
string      print    while
return      if       tree
for         else     bool
break       true     null
continue    false
```

Constants

A constant is a literal numeric or character value, such as 5 or 'm'. All constants are of a particular data type.

Integer Constants

An integer constant is a sequence of digits, starting with a non-zero digit. All integer constants are assumed to be decimal (base 10). Decimals values may use digits from 0 to 9.

```
459
0
8
```

Character Constants

A character constant is a single ASCII character enclosed within single quotation marks, such as ‘Q’. Some characters, such as the single quotation mark character itself cannot be represented using only one character. To represent such characters there are several “escape sequences” that you can use:

Sequence	Definition
`\n`	New line.
`\t`	tab.
`\\`	Backslash

Floating Point Constants

A floating point constant is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or “whole”) part of the number, a decimal point, and a sequence of digits which represents the fractional part. Either the integer part or the fractional part may be omitted, but not both. The decimal point may not be omitted. Here are some examples:

```
float a;
float b;
float c;
float d;
a = 4.7;
b = 4.;
c = .7;
d = 0.7;
```

String Constants

A string constant is a sequence of zero or more ASCII characters, or escape sequences enclosed within double quotation marks. A string constant is of type “array of characters”. Strings are stored as a 1 dimensional tree of characters. For more on the structure of the string object see *Tree Types* section below. Here are some example of string constants:

```
// this is a single string constant
“tutti frutti ice cream”
// this one uses two escape sequences
“\”hello, world!\””
/* to insert a newline character into a string, so that when the
 * string is printed it will on two different lines you can use
 * the newline escape sequence ‘\n’
 */
print(“Hello\nGoodbye”);
```

Boolean Constants

There are only two boolean constants, true and false. They must be typed in all lowercase letters. An example of declaring a boolean from a constant:

```
bool success;
success = true;
```

Tree Constant

A tree constant is expressed as a sequence of values of a consistent primitive data type (choice of char, int, float, bool). A tree constant begins with the first value representing the root node value, followed by square brackets containing the root's children separated by commas. Trees maintain a single data type in all of the tree node values. Lorax strictly enforces the type it first recognizes in the root. Lorax will display an error in the case of a type mismatch amidst a tree constant.

```
/* a is a tree of depth 3, degree 2, of integer data type value of the
 * root node is int 1, and its children are of value 2 and 3
 * respectively. The child with value of 2 has no children.
 * The child of value 3 has two children of value 4 and 5 respectively.
 * The nodes of value 4 and 5 have no children
 */
1[2, 3[4, 5]]
```

Data Types

Lorax promotes the use of tree structures as much as possible. There are four basic types that escape this norm. Declarations of data types must occur at the beginning of a function block or at the beginning of a file for global declarations.

Atom Types

Integers

Integers (`int`) are represented in 32-bit 2's complement notation. The default value of an integer variable is 0.

Floating Point Numbers

Single precision floating point (`float`) quantities have a magnitude in the range of approximately 10^(+ or - 38) or 0; their precision is 24 bits or about seven decimal digits. The default value of an float variable is 0.0.

Booleans

Booleans can be either `true` or `false`. The default value of a boolean variable is `false`.

Characters

A character, or char, is any single ASCII character. The default value for a char is `'\0'`.

Tree Types

As stated previously, Lorax encourages the use of tree structures as much as possible. Trees may contain any primitive data type as their tree node value. A string in Lorax is a tree of char data type node values that has its own definition syntax as we shall see but can be expressed as a tree constant as well.

Declaring Trees

You declare a tree using the tree keyword, followed by whitespace, followed by less than symbol, followed by a primitive data type in Lorax representing the node values of this tree, followed by the greater than symbol, followed by the identifier name being declared, followed by open parentheses, expression resulting in integer representing the branching factor, and finally closed parentheses. The default root value of a tree is the default value of its atom type (see above for the specific initial value). Tree atom type is declared statically at compile time. Tree degree (a.k.a. branching factor) may be defined at compile time using an integer literal or at runtime using an expression resulting in a positive integer value. Upon declaration trees are auto initialized with their atom type's respective initial value with children equal to `null`. Here is an example that declares a tree that has a degree (a.k.a branching factor) of 4 of `int` type:

```
tree <int>e(4);
```

Initializing Trees

You can initialize the elements in a tree by listing the initialized values, separated by commas, in a set of square braces. When declaring a tree without defining it, you must specify the type and branching factor in the declaration. Here is an example declaration with definition.

```
tree <int>a(2);
a = 1[2, 3[4, 5]];
```

Accessing Tree Children

You can access the child of a tree by specifying the tree name, followed by the percent symbol, followed by the child index. The child index begins at zero. Attempting to access a child outside of the branching factor of a node will result in a failed assertion at runtime. Accessing tree children against a tree literal or string literal is not possible. Here is an example statement of accessing the 4th index (5th child) of tree `a`:

```
a%4;
```

null is No-Child Indicator

`null` is a keyword without an explicit value in Lorax. It cannot be explicitly assigned to any data type or tree in Lorax. It is used only in order to answer the question: does a tree exist? In the below example we test if this tree has a child:

```
tree <int>a(1);
a = 42[];
bool b;
b = (a%0 == null); // b is true
```

Accessing Tree Node Values

You can access the node value of a tree by specifying the tree name followed by the @ symbol. This can be combined with the child accessing facility presented above. Accessing tree data members of a tree literal or string literal is possible but not on the left hand side of an assignment. Here is an example statement of accessing the 4th index (5th child) of tree `a` and setting the value stored in that child to integer 5:

```
a%4@ = 5;
```

Strings

In Lorax a special keyword and syntax is provided to make the declaration of strings straightforward, though under the hood they are no different than trees. Strings are combinations of characters that are delimited by double quotes. Strings are initialized as a tree structure with branching factor of one terminated by the end of the tree having no child (`null`). Each tree node encapsulates a single character and has a single child for the next letter in the string. Below is an example of declaring and defining a string using convenient syntax:

```
string simple;
simple = "Hello";

// the above may also be represented this way
tree <char>complicated(1);
complicated = 'H'['e'['l'['l'['o']]]];

/* notice this statement prints true (comparison operators discussed
 * below)
 */
print(simple == complicated);
```

Expressions and Operations

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values. Here are some examples:

```
47
2 + 2
cosine(3.14159)
```

Parentheses group sub expressions. Innermost expressions are evaluated first:

```
( 2 * ( ( 3 + 10 ) - ( 2 * 6 ) ) )
```

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are type and value of the right operand.

Assignment Operators

Assignment operators store primitive values in variables, copy the reference of a tree to a tree variable, or assign a value to a tree nodes value. The Lorax assignment operator is =. It is a binary operator and is right-associative. When assignment is taken, the value of the expression on the right is assigned to the left value, and the new value of the left value is returned, which allows chaining of assignments. Assignment can take some of these example forms:

```
// where a is declared as int a;
a = 4;
/* where b is declared as tree b; and c is a previously declared
 * and defined tree
 */
b = c;
// where d is previously declared as a tree containing int value types
d%0@ = 5;
/* Tree reference assignment. Where t and t2 are previously declared
 * and defined trees this assignment will set t's first child
 * (reference 0) to the root tree of t2
 */
t%0 = t2;
// value assignment to a tree node
a = b@;
```

In this section we describe the built-in operators for Lorax, and define what constitutes an expression in our language. Operators are listed in order of precedence.

Arithmetic Operators / Tree Operators

Lorax provides operators for standard arithmetic operations: addition (+), subtraction (-), multiplication (*), and division (/), along with modular division (`mod`) and negation (-) . Usage of these operators is straightforward when using primitive types. Arithmetic operations are not valid among the `bool` type. With two `char` operands only the addition and subtraction operators are valid. Here are some examples using arithmetic operators with primitives:

```
x = 5 + 3; // where x is of type int
y = 10.23 + 37.332; // where y is of type float
z = 'a' + ('c' - 'a'); // where z of type char
```

You use the modulus operator `mod` to obtain the remainder produced by dividing its two operands. The mod operator may only be used between two integer values.

```
x = 5 mod 3;
```

You use the negation operator on a `float` or `int` type.

```
x = -4;
```

Of the arithmetic operators, trees may only use the addition operator. Like all arithmetic operators the tree addition operation must contain trees of the same data type on either side. When the addition operator is used, both tree operands are checked to have the same data type at compile time. In this operation usage of a consistent degree for both tree operands is left to the programmer. A mismatch will cause a failed assertion at runtime. When the addition operator is used, a new tree is constructed from the tree operand on the left hand side of the `+` symbol. The tree operand on the right hand side of the `+` symbol is copied into the first available `null` position (breadth first position) on the newly formed tree. This rule allows for the easy concatenation of two trees representing strings. Examples of this operation below:

```
tree <int>a(2);
tree <int>b(2);
a = 1[2, 3[4, 5]]; // tree of degree 2, depth 3, int data type
/* after the below operation a new tree will be created
 * and assigned to b representing 1[6[7, 8], 3[4, 5]]
 */
b = a%0 + 6[7, 8];
```

Comparison Operators

You use the comparison operators to determine how two operands relate to each other: are they equal to each other, is one larger than the other, is one smaller than the other, and so on. When you use any of the comparison operators, the result is boolean value `true` or `false`. Comparison operators are all binary operators and are left-associative. The compiler will issue a warning if the operators == and != are used with two tree operands of differing atom type. Tree operands may contain differing degrees and atom types when used with comparison operators. In the case of comparing trees the definition of this comparison is indicated below:

Operator	Primitive Types Definition	Tree Type Definition
`>`	Greater than.	LHS # of nodes > RHS # of nodes
`>=`	Greater than or equals.	LHS # of nodes >= RHS # of nodes
`==`	Equal to.	LHS tree structure and data is equal to RHS tree structure and data Can also be used to compare to `null`
`!=`	Not equal to.	LHS tree structure and date is not equal to RHS tree structure and data Can also be used to compare to `null`
`<=`	Less than or equals.	LHS # of nodes <= RHS # of nodes
`<`	Less than or equals.	LHS # of nodes <= RHS # of nodes

Logical Operators

Logical operators test the truth value of a pair of operands. The following logical operators `&&` (logical and) and `||` (logical or) are binary operators and left associative. They take two operands of type boolean, and return a boolean value. `!` is a unary operator and appears on the left side of the operand. The type of the operand must be of type boolean and return type is also a boolean value. Short circuit evaluation is not supported.

Operator Precedence

The following is a list of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right.

```
()
% @
!
* / mod
+ -
> < >= <=
== !=
&&
||
=
,
```

Statements

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form:

expression `;`

Compound Statement

So that several statements can be used where one is expected, the compound statement is provided:

compound-statement:
`{` *statement-list* `}`

statement-list:
statement
statement, statement-list

Conditional Statement

The two forms of the conditional statement are:

`if (` *expression* `) {` *statement* `}`
`if (` *expression* `) {` *statement* `} else {` *statement* `}`

In both cases the expression is evaluated and if it is `true` the first sub-statement is executed. In the second case the second sub-statement is executed if the expression is `false`. As usual the "else" ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

While Statement

The `while` statement has the form:

> `while (` *expression* `) {` *statement* `}`

The sub-statement is executed repeatedly so long as the value of the expression remains `true`. The test takes place before each execution of the statement.

For Statement

The `for` statement has the form:

> `for (` *expression_1* `;` *expression_2* `;` *expression_3* `) {` *statement* `}`

This statement is equivalent to:

```
expression_1
while ( expression_2 ) {
        statement
        expression;
}
```

Return Statement

A function returns to its caller by means of the `return` statement, which has one of the forms:

```
return;
return expression;
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. If required the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

Functions

Function Definition

The Lorax language supports user defined functions. Every function declaration must be followed immediately by the definition of that function. Every function declaration must begin by specifying the return type of the function. The return type is followed by an identifier and comma-separated list of formal parameters enclosed within parentheses. A function may have any number of parameters, and all parameters are passed by value with the exception of tree types. The implementation details of the function follow immediately within braces. Every function may have a single return statement that returns a value consistent with its return type. Functions without an explicit return statement will return the default value for the return data type for that function. Functions may only return primitive types. A function is called using its identifier followed by its parameters in parentheses separated by commas. If there are no required parameters, the function is called using its identifier followed by empty parentheses. Lorax does not support function overloading. However, the built-in function `print` accepts variable arguments of all types which is described below in Built-in Functions. Functions in Lorax may recursively call themselves. Here is an example of a user-defined function in Lorax:

```
int square(int x) {
    return x * x;
}

int main() {
    int x = 4;
    int s = square(x);
    return 0;
}
```

main Function

In Lorax there is an entry function where the program starts. There must be one main function and should be defined like this:

```
int main() {
      statement-list
}
```

Built-in Functions

print Function

The print function provided accepts a variable number of arguments of any of the Lorax data types. Presenting `print` with any of the primitive types will print the type in its most natural form. Presenting print with a tree argument will print the tree in a kind of debug format unless the data type for the tree is of 1-degree `char` type in which case it will print a string. The print function has return value. Examples below:

```
print("hello, world"); // will print hello, world
print(3); // will print 3
print(3.14); // will print 3.14
print('a'); // will print a
tree <int>t(2);
t = 1[2, 3];
print(t); // will print 1[2, 3]
```

parent Function

The parent function takes a single tree argument. The return value of the function is the parent of the argument. Example below:

```
tree <int>grandFather(2);
grandFather = 1[2, 3[4, 5]];
tree grandChild <int>(2);
grandChild = (grandFather%2)%0; // referencing the child with value 4
tree middleChild <int>(2);
// middle refers to node with value 3
middleChild = parent(grandChild);
```

root Function

The root function takes a single tree argument. The return value of the function is the greatest parent of the argument. Example below:

```
tree <int>grandFather(2);
tree <int>grandFatherPtr(2);
tree <int>grandChild(2);
grandFather = 1[2, 3[4, 5]];
grandChild = (t%2)%0; // referencing the child with value 4
// grandFatherPtr refers to node with value 1
grandFatherPtr = root(grandChild);
```

degree Function

The degree function takes a single tree argument. The return value of the function is `int` type. The function returns the defined or inferred degree of the tree. Example below:

```
print(degree(3[4, 5])); // prints 2
```

Because tree literals of the form *expression*`[]` represents a single root node without children the degree interpretation of such an expression is flexible. In operations where such an expression is paired with another tree expression where the tree literal's degree is explicitly known (such as in the above example) then the single node tree literal expression will be assumed to be of the degree that is explicitly known. In the below example we demonstrate a case that is unusual. When a tree literal is a single node literal and it is not in conjunction with another tree literal with an explicitly known degree then the single node literal cannot be inferred and is assumed to have a degree of 0. Therefore the result of the below expression is integer 0.

```
degree(6[]);
```

Scope

Lorax is closed and statically scoped. Local primitive types are passed to their functions by value. Tree identifiers hold a reference to their tree structure and the tree reference may be passed from function to function. Tree objects are allocated at run time and deallocated when there is no tree identifier left in scope and referencing them. Lorax manages reference counting of all tree objects.

Sample Programs

Depth First Search

```
bool dfs(tree <int>t(2), int val) {
	int child;
	bool match;
	match = false;

	if (t == null) {
		return false;
	}

	if (t@ == val) {
		return true;
	}

	for (child = 0; child < degree(t); child = child + 1) {
		if (t%child != null) {
			if(t%child@ == val){
				return true;
			}
			else{
				match = dfs(t%child, val);
			}
		}
	}

	return match;
}

int main() {
	tree <int>t(2);
	t = 1[2, 3[4, 5]];
	if (dfs(t, 3)) {
		print("found it\n");
	} else {
		print("its not there\n");
	}
}
```

Hello World

```
int main() {
	print(“hello, world\n”);
}
```

Euclid's GCD

```
int gcd(int x, int y){
    int check;
    while (x != y) {
        if (x < y) {
            check = y - x;
            if (check > x) {
                x = check;
            } else {
                y = check;
            }
        } else {
            check = x - y;
            if (check > y) {
                y = check;
            } else {
                x = check;
            }
        }
    }
    return x;
}

int main() {
    print(gcd(25, 15));
}
```

Huffman Tree

```
int main () {
      tree <char> codingtree (2);
      codingtree = '$'['$'['$'['$'['c', '$'['t','m']],'r'],
            '$'['$'['$'['o','u'],'$'['k','n']],'a']],
            '$'['$'['$'['z','s'],'i'],'$'['$'['g','d'], 'h']]];
      decode("1000", codingtree);
      decode("111", codingtree);
      decode("011", codingtree);
      decode("011", codingtree);
      decode("001", codingtree);
      decode("01011", codingtree);
      print("\n------\n");
      decode("0000", codingtree);
      decode("111", codingtree);
      decode("001", codingtree);
      decode("101", codingtree);
      decode("1001", codingtree);
      print("\n------\n");
      decode("00010", codingtree);
      decode("101", codingtree);
      decode("00011", codingtree);
```

```
	print("\n------\n");
	decode("01010", codingtree);
	decode("101", codingtree);
	decode("001", codingtree);
	decode("011", codingtree);
	print("\n------\n");
	decode("1101", codingtree);
	decode("01000", codingtree);
	decode("01001", codingtree);
	decode("1100", codingtree);
	print("\n------\n");
}

int decode(tree <char> letter (1), tree <char> codingtree (2)){
	tree <char> a (1);
	tree <char> b (2);
	a = letter;
	b = codingtree;
	while(true) {
		if(b%0 == null){
			print(b@);
			return 0;
		}
		if(a@ == '0') {
		/*	print(a@); */
			b = b%0;
			a = a%0;
		}
		else {
		/*	print(a@); */
			b = b%1;
			a = a%0;
		}
	}

}
```

Using Trees as an Array

```
/* Inserts an element into the array */
int insert_array(tree <int>t(1), int index, int val) {
	tree <int> a(1);
	int i;
	a = t;
	if (a == null) {
		return -1;
	}
	for (i = 0; i < index; i=i+1) {
		a = a%0;
		if(a == null){
```

```
                return -1; //invalid access
            }
        }
        a@ = val;
        return 0;
}

/* Accesses an element in the array */
int access_array(tree<int>t(1), int index) {
        tree <int> a(1);
        int i;
        a = t;
        if (a == null) {
            print("Invalid access");
            return -1;
        }
        for (i = 0; i < index; i = i+1) {
            a = a%0;
            if(a == null){
                print("Invalid access");
                return -1;
            }
        }
        return a@;
}

/* Gets the size of the array */
int size_array(tree <int> t(1)) {
        int i;
        tree <int> a (1);
        a = t;
        i = 0;
        while( a != null) {
            a = a%0;
            i = i + 1;
        }
        return i;
}

int main() {
        tree <int>t(1);
        int size;
        int i;
        int p;
        t = 0[0[0[0[0[0]]]]];
        /* size = 6; */
        /* init_array(t, size); */
        for (i = 0; i < size_array(t); i = i + 1) {
            insert_array(t, i, i);
```

```
            p = access_array(t, i);
            print(p);

      }
      print("\n");
      print(t);
}
```

Project Plan

Team Responsibilities

The Lorax project was developed with a five person team. Preliminary language design work was performed as a team. Development began in earnest beginning with semantic analysis on November 17th. From this point until December 15th, Kira Whitehouse and Chris worked on average six to eight hours each day. Below is a listing of the essential documents associated with this project and the members that contributed to these files in order of greatest to least contributor.

```
Project Proposal:                    Chris, Kira, Doug, Zhaarn, Tim
Language Reference Manual:           Chris, Doug, Kira, Tim, Zhaarn
Lorax.ml:                            Chris
Scanner.ml:                          Chris, Doug, Zhaarn
Ast.ml:                              Chris
Symtab.mll:                          Tim, Chris
Parser.mly:                          Chris, Doug
Check.ml:                            Chris, Kira
Intermediate.ml                      Kira, Chris, Zhaarn
Output.ml:                           Kira, Chris
lrxlib.h:                            Kira, Doug, Tim, Chris
Makefile:                            Chris
Tests / testall.sh:                  Chris, Zhaarn, Kira
Sample Programs:                     Zhaarn, Chris
Final Report:                        Chris, Tim, Kira, Doug
Presentation:                        Chris
```

Style Guide

We adhered to the OCaml Style witnessed in Stephen Edward's MicroC example:

```
(* comment *)

(*
 * Long Comment
 * Comments proceed the code thought
 *)

match (* pattern matching aligns with c of match *)
   a -> b
```

```
 | c -> d
 | e ->
   f (* also acceptable to begin return on aligned next line *)

let x = in
print_string x; (* statements utilizing let statement are aligned *)

if true then
  print_string “two space indentation” (* two spaces *)
else
  print_string “here too”

No regard for column length.

Self documenting variable names. Self evident 1 or 2 char variable
names. cl = “child list”, c = “child”.

under_scores. No CamelCase.
```

We adhered to the K&R C Style for lrxlib.h.

Project Timeline

Below is a schedule of deadlines. The project development log reflects the actual work history.

Date	Scheduled Deadline
9/13	Project Started
9/25	Language proposal finalized
10/28	Language Reference Manual finalized
10/28	Scanner, Parser, AST
11/17	“Hello World” Code Generation
11/27	Connection of complete compiler path
11/30	Semantic Analysis Complete
12/4	lrxlib.h and sample programs complete
12/15	Compiler complete. Documentation completed.
12/19	Final Presentation

Project Development Log

Below are the dates of actual project developments. Highlighted are project start dates, end dates, and milestones. The Language Reference Manual was adjusted continuously as development progressed. Tests were added to the test suites continuously as new features of the language were added.

Date	Event
10/14	Work begins on Scanner
10/28	LRM submitted. Partial Sample program set complete.
11/10	Work begins on Parser, AST
11/12	Test suite built to accommodate each module

```
11/13       Scanner, Parser, AST is completed
11/16       Work begins on Semantic Analysis. SymTab work begins.
11/17       Work begins on Code Generation
11/22       Symtab completed.
11/23       Semantic Analysis compiling producing output.
11/24       Tree literal type/degree checking completed.*
11/27       Semantic Analysis completed. Code Generation is rewritten.
12/7        Tree literal to C decl/def completed.*
12/13       Tree assignment in known cases possible.*
12/15       Tree addition operator completed.

* Major technical milestones
```

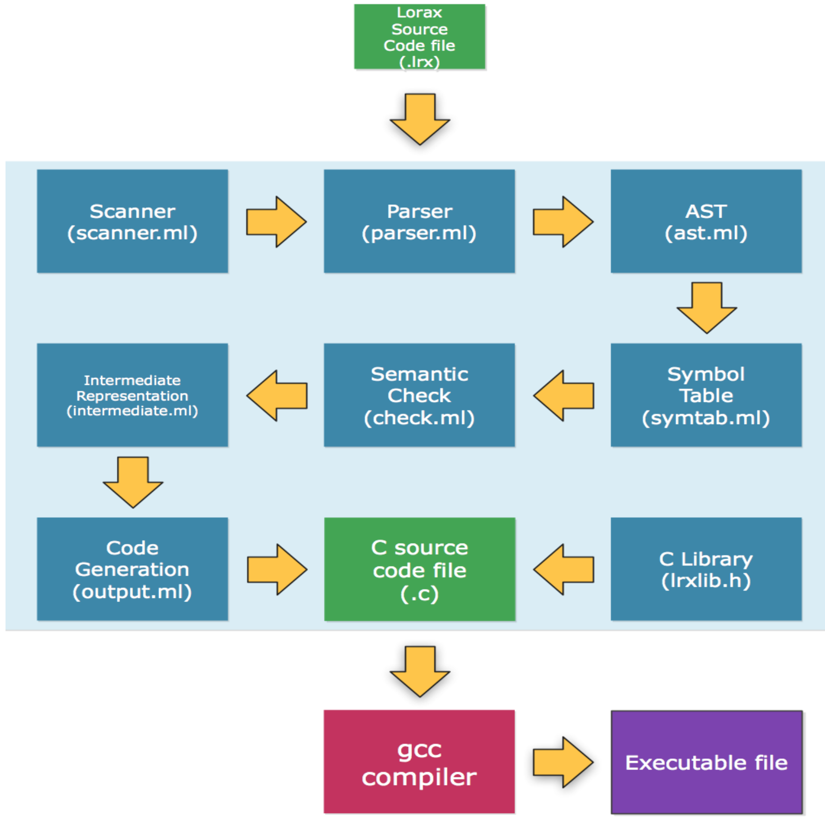
Development Environment

Lorax was developed in the Mac OS X environment. The compiler was written in OCaml, using the ocamlc, ocamllex, and ocamlyacc tools from the 4.01 distribution. The generated C code is compiled using the gcc compiler by default. The build process was automated with Makefiles. Testing and verification was run with shell scripts (bash). Testing was also performed on Ubuntu 12.04. The team mostly worked using Sublime Text 2 and Vim as text editor. Memory management testing in Valgrind was performed in a virtual environment with the use of Virtual Box and the tool Vagrant. Trello was used for task management. Github was used for git repository hosting. Google Drive was used to store and collectively edit documentation. Google Hangout was useful for remote meetings allowing face to face interaction with screen sharing and shared whiteboard space. Apple's Keynote was used for writing the presentation.

Architectural Design

Overview

The Lorax compiler takes a single Lorax source program as input, and outputs C source code, which is then compiled with the gcc compiler. The compiler consists of the following phases: scanning, parsing, symbol table generation, semantic analysis, intermediate representation generation, and output code generation. Code generation always includes a library of our creation, lrxlib.h, to allow for the functions required for tree declaration, definition, manipulation, reference counting, and deallocation.



Scanner (scanner.mll)

The scanner goes through the .lrx source file and converts it into a stream of tokens, using ocamllex.

Parser (parser.mly)

The parser is used to analyze the stream of tokens given by the scanner, and decide whether or not they are in the language that is specified by our context free grammar. During parsing, a scope number is bundled with blocks, types are deduced, and an abstract syntax tree is generated. This output tree defines the structure of a given program.

Abstract Syntax Tree (ast.ml)

The AST defines the structure of the Lorax language, including the various types and structures, like variables, blocks, functions, and the program itself.

Symbol Table (symtab.ml)

The symbol table takes the abstract syntax tree generated in previous steps, and generates a table of the declared variables and functions, including their scope number. This table is used to enforce unique function and variable names within each scope, and to verify whether or not a variable or function is visible within the current scope.

Static Semantic Checker (check.ml)

The semantic checker performs semantic analysis of the program. Here, we check that variables are declared within a given scope. We also check for correspondence between atom types within assignment, binary and unary operations, and function calls. For tree literals, this requires extensive checking, as we must verify recursively that the degree and type of all subtrees agree. Multiple calls to obtain children from a tree also proved challenging (e.g. t%0%1) as we had to determine that the leftmost article of the expression was a valid id of a declared tree. In this file we additionally check to ensure calls to functions match the declarations of these functions, with respect to return type and arguments.

Intermediate Representation (intermediate.ml)

The intermediate representation unravels the semantically checked abstract syntax tree and generates code that is close to the output in structure, but requires a final translation into actual C code. Here we generate jump and return labels for if blocks, while loops, and for loops, such that output can flatten blocks within functions. Because C does not allow for declarations to occur after labels all variables must be declared at the top of a function body. We accomplish this by pulling all declaration types to the top of the function before we send the list of statements to output. Here we also generate temporary variables to hold the result of expressions, which we later use to assign user-declared variables. This process requires a global count so that all variables are defined with unique names. We generate a default return value that is called at the end of a function, and will be used unless the user writes a return. This quiets warnings within gcc about functions without return statements.

Finally, we generate calls for creating and cleaning up memory within trees. Trees are defined as structures within our c library; thus, each individual subtree within a tree literal must be declared and defined. As such, a tree literal with n nodes will have n calls to lrx_declare_tree and n calls to lrx_define_tree. We also generate calls to cleanup the memory used by these trees; whenever a tree is defined we create a corresponding lrx_destroy_tree call to that variable, and pull these destroy calls out to be placed before any return statement. This ensures that memory is handled properly whenever we exit the scope of a function.

Output (output.ml)

Output takes the code generated in the intermediate representation, and converts it to the C output code. Here we match types of expressions with atom vs. tree; if an atom we insert the c equivalent of the syntax, if a tree we call functions we write in the lorax library lrxlib.h. The most challenging part about this file was dealing with pointers in c. Though we pass atom types by value within functions, we pass tree types by reference, which means that at each point a tree is used within a function call it must be marked as a triple pointed structure (struct tree ***) and dereferenced accordingly. There were similarly complex issues with defining trees, as we had to generate child arrays of trees (struct tree *children[n]), fill them with their children, and send this data alongside a pointer to a root type into lrxlib.h. This required creating temporary variables for children arrays as well as root values. It also required editing intermediate and introducing a few new types (tree declarations, pointers to atom types) so that we could properly recognize these instances and use pointers and arrays accordingly.

Lorax Library (lrxlib.h)

The lorax library provides low level implementation of the tree based operations required in Lorax. This includes all aspects of memory management (construction and destruction with reference counting), tree typing, tree based operations, and printing. Tree structures, when declared (e.g. tree <int> t(3)) are placed onto the heap and instantiated with a call to lrx_declare_tree. This tree can be defined later with an assignment (t = 5[6, 7, 8]) which calls lrx_define_tree. Trees are destroyed at the end of a function block with a call to lrx_destroy_tree. We use reference counting to properly designate usage and destroy elements only when appropriate. The most challenging part of memory was twofold: 1) we decided to allow tree literals of degree 0 to be used (e.g. 6[]) and 2) we decided to allow tree addition. Both of these parts of our language presented complications within memory. For 1), we define and declare a temporary tree to hold the tree literal 6[]. Because we don't know what degree this literal is, we cannot malloc space for a given number of children, so we leave its array of children null. Only later, when this literal is used within assignment or addition (which brings us to the second challenge), do we malloc space for an appropriate number of children. This relies on the degree of the tree that it is being added with or assigned to. For instance, the operation 6[] + 7[8,9] would malloc space for 2 children because the rhs of the operation has degree 2. The bizarre case is when we add two trees of degree 0, such as 5[] + 6[]. Here we evaluate both trees to degree 1. This means that the code 5[] + 6[] + 7[8, 9] will assert a failure because adding trees 5[] and 6[] will result in a tree degree of 1, and we do not allow addition between trees of different types. 2) When trees are created with addition, we do not have access to their internals. Thus, they cannot be deleted like an ordinary tree with lrx_destroy_tree; they have their own destroy function, lrx_destroy_add_tree. This practice results in minor memory loses when we create a tree via addition and later assign this tree to another element (e.g. t = v + s; t = m). This is because within lrx_assign_tree_direct we call lrx_destroy_tree on the lhs of the assignment, to free the memory before we reassign the pointer. Because we have two different types of destroy calls, this practice results in memory loss.

Command Line Interface (lorax.ml)

The lorax command line allows the user to inspect output at major phases of the compilation process, by specifying one of a variety of flags. This is a helpful debugging tool, and can provide insight on how the Lorax compiler translates down to C code.

Testing Plan

The Test Suite

Frequent and thorough testing was an essential component of our build process. We performed a number of tests at every stage of development. For three core functions of the compiler (parser, semantic analysis, code generation) we wrote tests to prove the validity of those individual sections. These were regression tests that compile code up to the latest module (i.e. source->scanner->parser/ast), and compared this to expected output. As all modules neared completion, we ran end-to-end tests as a means of both verifying functionality, and making sure no previous modules were broken. As development was ending we also added several failure tests to demonstrate that each module is capable of catching failures as well. We ran C based tests with Valgrind to experiment and correct bad read/write errors in the Lorax C library. Shell scripts allowed us to run tests in bulk, and a full suite of end-to-end tests was run before any changes were committed to the master branch.

Lessons Learned

Advice for Future Groups

The project takes time everyday. It is not possible to learn everything you will need to know from any lecture. If you get started early you may feel like you don't know what you're doing but later in the semester you will also feel this way. It is only by getting started early that you will ever begin to "know what you're doing." Mistakes that we made are typical of any large design project. Hindsight is 20/20. Below are some things we would have liked to have known before making these mistakes:

- Think hard about the types you use to represent your language before you enter the code generation phase. Example: You may think that OCaml char is useful for representing characters in your language but perhaps not. A scanner reads an escape character as a string (e.g. '\' 'n').
- If you are planning to use pointers to manipulate objects as we did think long and hard about how you declare/define/access/dereference those variables in code generation. We began writing an intermediate representation temporary generator as if we were writing a vanilla C-Language without the existence of pointers. Late in development a major breakthrough that our system required a comprehensive solution not hacks here and there.
- If you can try to use OCaml records. Use tuples to carry information in your language sparingly.
- Steal from the best. Stephen Edward's MicroC and Dara Hazeghi's (2011) strlang were extremely helpful in understanding and designing our language. Dara's code provided a template for our design. This was our greatest source of instruction.

Doug Bienstock

When trying to come up with an idea for the language, think not only abstractly but also take some time to think of explicit use cases and the specific mechanics of your language. I think we came up with a language that in the abstract could be very useful, but given the time constraints and our skill set ended up being very difficult to actually implement. Even small domain-specific languages like MySQL were developed over a long period of time by some very smart people, so it is OK if your language is a bit limited in its scope or function. This is mostly a learning experience and even though our language is somewhat functionally limited it was still a powerful and fun educational experience. Something I would advise is to constantly keep thinking ahead when you are planning out the language, but also when developing. Think about what your decisions mean for implementing the AST, for checking the AST, for generating IR, and for outputting code. Everything is very tightly linked and a decision that seems superfluous could influence the entire language. This was my first long-term and large scale software development project so that was also interesting. One difficult thing is judging the complexity and the time needed for a task without really understanding the exact task and the problems that will inevitably crop up along the way. I think a big difficulty was also needing to learn while also doing at the same time. Don't expect to be given all the set of tools on the first day and get to a lecture where you think to yourself "I can start now!" This doesn't happen and it’s important to just dive in and kind of learn as you go on your own, because if you wait until you think there is going to be some magical start moment you will be disappointed.

Chris D’Angelo

“Whenever there is any doubt, there is no doubt.” - Ronin, David Mamet

This has been an incredible experience. Writing the compiler was challenging and extremely rewarding. In many ways it is not that different from other software development. In one aspect compiler writing is passing information dressed in some format between interfaces and understanding the state of that information/format throughout. OCaml is ideal in this respect. Baked in pattern matching and type checking provided almost prescient guardrails allowing us to make fewer mistakes when passing all of our types around. It was a pleasure from day one to use. In stark contrast developing lrxlib with very deep pointer assignment was at times nearly impossible to debug. Knowing your type in C is not so straightforward.

The experience building this compiler also solidified my adoration for debuggers. I have now begun an interest in building a debugger myself. The OCaml debugger was instrumental to our development and understanding the state of the inputs and outputs of each module. I also learned how valuable testing can be. Perpetual writing of tests allowed us to verify the features we were writing and also verify that the features we were adding afterwards were not breaking previous work. I learned more clearly than ever before the true meaning of compile-time vs runtime. The trees in our language are checked statically for their datatype but the degree is verified at runtime. The realization of what this really meant was eye opening for me.

With any software development, there is a high from developing something, running it, and then seeing the result is what you expected. Writing a compiler was that feeling of elation amplified. It is exciting the day that your compiler knows right from wrong in your language. For the programmer a compiler error is a mistake. For the compiler writer a parse error or a type mismatch can be a success story.

Kira Whitehouse and I worked together in person almost every day from November 22nd to December 15th. Her contribution to the project was invaluable. We worked closely on all sections but essential features requiring solutions with very challenging algorithms were born out of Kira’s creativity and very hard work. It was a privilege to work with her and she deserves the highest possible grade. Amazing was the difficulty of the things we wrote. Equally amazing are the things we didn’t have to write because of the miracle of recursion. To quote Kira in the midst of our struggle to type check our first tree literal: “We’re losing our sense of recursion.” Special thanks to Jonathan Balsano who in the 11th hour provided his expertise in debugging the addition operator between trees.

Zhaarn Maheswaran

I think the biggest mistake that we made was designing the language around an abstract concept (the use of trees) rather than a set of concrete use cases. As a result, the end product doesn't have much utility as a language, although it provides an interesting theoretical exercise. It was fun to see how concisely I could express a standard algorithm in an environment where the only data structure at my disposal is a tree. Along those same lines, I think working off of specific use cases would have helped us narrow the scope of the language.

Tim Paine

Its important to keep the scope of the project narrow, while still answering some interesting questions and challenging yourself. Limiting yourself to only working with integers, for example, makes your life a little bit easier, and for us at least, would not have significantly reduced the novelty of our language. Many features that were assumed to be easy turned out not to be so. In the end, features were cut out at multiple stages of the build process, and though the end project still answers some interesting questions, there is still work to be done. Oh, and I learned some OCaml.

Kira Whitehouse

Programming is tainted by a queer sense of spirituality. It is associated with rituals (all nighters, Star Wars, and video games), feasts (coke, pizza, and pad thai), and scriptures (C man-pages, Java API, and pydoc). There are a variety of versions of a higher being that coders worship and argue amongst themselves about. I myself look to C for the answers to the universe. But this semester, I became a polytheist. While writing this compiler some sort of serious enlightenment went on; I am now a believer in OCaml.

To quote ocaml.org, OCaml is an "industrial strength programming language." Its pattern matching offered an elegant solution to examining and type checking symbolic data. And its debugging facilities were imperative to validating the output of each piece of our project. Though its structure seemed unintuitive at first, I have come to embrace its motto "recurse-or-die."

I feel incredibly luck to have had Chris D'Angelo on board with me for this project. His management skills and work ethic kept our project on track, allowing us to finish in time. His enthusiasm and passion for programming was infectious. Many nights we would stay up for "just ten more minutes." Only a few hours later would we head to bed, both half-delirious, with goofy grins on our faces.

After sweat, tears, and a couple peanut-butter cookies, I look back on this semester with a big "wow, that was fun." I have a newfound appreciation for memory management in C. And I am now eager to continue to explore lower level systems engineering. Writing a compiler was not something I ever dreamt I would do, but it has been one of the most satisfying experiences writing code that I have ever had. I look forward to round two.

Appendix

Presentation Slides

A presentation demonstrating a basic tutorial and explanation of the design of the Lorax Compiler can be found here: http://bit.ly/theloraxpresentation

Complete Code Reference

Source controlled documents associated with The Lorax Language Compiler can be found here: http://bit.ly/theloraxcode

Root Directory

ast.ml

```
(*
 * Authors:
 * Chris D'Angelo
 * Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
 * which provided background knowledge.
 *)

type op =
    Add
  | Sub
  | Mult
  | Div
  | Mod
  | Equal
  | Neq
  | Less
  | Leq
  | Greater
  | Geq
  | Child
  | And
  | Or

type uop =
    Neg
  | Not
  | At
  | Pop

type expr =
    Int_Literal of int
  | Float_Literal of float
  | String_Literal of string
  | Char_Literal of char
  | Bool_Literal of bool
  | Null_Literal
  | Id of string
  | Binop of expr * op * expr
  | Unop of expr * uop
  | Tree of expr * expr list
  | Assign of expr * expr
  | Call of string * expr list
  | Noexpr

type atom_type =
    Lrx_Int
  | Lrx_Float
  | Lrx_Bool
  | Lrx_Char

type tree_decl = {
    datatype : atom_type;
```

```
    degree : expr;
}

type var_type =
    Lrx_Tree of tree_decl
  | Lrx_Atom of atom_type

type var = string * var_type

(*
 * wrappers for use in symtab
 * scope_var_decl =
 *              <<identifier name>> *
 *              <<data type>> *
 *              <<block id to be assigned in symtab>>
 *
 * scope_func_decl =
 *               <<identifier name>> *
 *               <<return data type>> *
 *               <<formal arg list>> *
 *               <<block id to be assigned in symtab>>
 *)
type scope_var_decl = string * var_type * int

type scope_func_decl = string * var_type * var_type list * int

type stmt =
    CodeBlock of block
  | Expr of expr
  | Return of expr
  | If of expr * block * block
  | For of expr * expr * expr * block
  | While of expr * block
  | Continue
  | Break

and block = {
    locals : var list;
    statements: stmt list;
    block_id: int;
}

type func = {
    fname : string;
    ret_type : var_type;
    formals : var list;
    fblock : block;
}

type program = var list * func list

type decl =
    SymTab_FuncDecl of scope_func_decl
  | SymTab_VarDecl of scope_var_decl

(* used by check.ml *)
let string_of_unop = function
    Neg -> "-"
  | Not -> "!"
  | At -> "@"
  | Pop -> "--"

let string_of_binop = function
        Add -> "+"
      | Sub -> "-"
      | Mult -> "*"
      | Div -> "/"
      | Mod -> "mod"
      | Child -> "%"
      | Equal -> "=="
      | Neq -> "!="
      | Less -> "<"
      | Leq -> "<="
```

```
      | Greater -> ">"
      | Geq -> ">="
      | And -> "&&"
      | Or -> "||"

let rec string_of_expr = function
    Int_Literal(l) -> string_of_int l
  | Float_Literal(l) -> string_of_float l
  | String_Literal(l) -> "\"" ^ l ^ "\""
  | Char_Literal(l) -> "\'" ^ (String.make 1) l ^"\'"
  | Bool_Literal(l) -> string_of_bool l
  | Null_Literal -> "null"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      string_of_binop o ^ " " ^
      string_of_expr e2
  | Unop(e, o) ->
      (match o with
          Neg -> "-" ^ string_of_expr e
        | Not -> "!" ^ string_of_expr e
        | At -> string_of_expr e ^ "@"
        | Pop -> string_of_expr e ^ "--")
  | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Tree(r, cl) -> string_of_expr r ^ "[" ^ String.concat ", " (List.map string_of_expr cl) ^ "]"
  | Noexpr -> ""

let string_of_atom_type = function
    Lrx_Int -> "int"
  | Lrx_Float -> "float"
  | Lrx_Bool -> "bool"
  | Lrx_Char -> "char"

let string_of_vdecl v =
    (match (snd v) with
        Lrx_Atom(t) -> string_of_atom_type t ^ " " ^ fst v
      | Lrx_Tree(t) -> "tree <" ^ string_of_atom_type t.datatype ^ ">" ^ fst v ^ "(" ^ string_of_expr t.degree ^ ")"
    )

let rec string_of_stmt = function
    CodeBlock(b) -> string_of_block b
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, b1, b2) ->
    (match b2.statements with
        [] -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_block b1
      | _  -> "if (" ^ string_of_expr e ^ ")\n" ^
              string_of_block b1 ^ "else\n" ^ string_of_block b1)
  | For(e1, e2, e3, b) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_block b
  | While(e, b) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_block b
  | Break -> "break;"
  | Continue -> "continue;"

and string_of_block (b:block) =
  "{\n" ^
  String.concat ";\n" (List.map string_of_vdecl b.locals) ^ (if (List.length b.locals) > 0 then ";\n" else "") ^
  String.concat "" (List.map string_of_stmt b.statements) ^
  "}\n"

let string_of_var_type = function
    Lrx_Atom(t) -> string_of_atom_type t
  | Lrx_Tree(t) -> "tree <" ^ string_of_atom_type t.datatype ^ ">(" ^ string_of_expr t.degree ^ ")" (* only for use within fdecl formals *)

let string_of_fdecl fdecl =
  (string_of_var_type fdecl.ret_type) ^ " " ^
```

```
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_vdecl fdecl.formals) ^ ")\n" ^
  string_of_block fdecl.fblock

let string_of_program (vars, funcs) =
  String.concat ";\n" (List.map string_of_vdecl vars) ^ (if (List.length vars) > 0 then ";\n" else "") ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

check.ml

```
(*
 * Authors:
 * Chris D'Angelo
 * Kira Whitehouse
 * Special thanks to Dara Hazeghi's strlang which provided background knowledge.
 *)

open Ast

let fst_of_three (t, _, _) = t
let snd_of_three (_, t, _) = t
let fst_of_four (t, _, _, _) = t

(*expressions from Ast but with typing added*)
type c_expr =
    C_Int_Literal of int
  | C_Float_Literal of float
  | C_String_Literal of string
  | C_Char_Literal of char
  | C_Bool_Literal of bool
  | C_Null_Literal
  | C_Id of var_type * string * int
  | C_Binop of var_type * c_expr * op * c_expr
  | C_Unop of var_type * c_expr * uop
  | C_Tree of var_type * int * c_expr * c_expr list
  | C_Assign of var_type * c_expr * c_expr
  | C_Call of scope_func_decl * c_expr list
  | C_Noexpr

(*statements from Ast but with typing added*)
type c_stmt =
    C_CodeBlock of c_block
  | C_Expr of c_expr
  | C_Return of c_expr
  | C_If of c_expr * c_block * c_block
  | C_For of c_expr * c_expr * c_expr * c_block
  | C_While of c_expr * c_block
  | C_Continue
  | C_Break

(* tree declaration from Ast but with typing added *)
and c_tree_decl = {
    c_datatype: atom_type;
    c_degree: c_expr;
}

and c_block = {
    c_locals : scope_var_decl list;
    c_statements: c_stmt list;
    c_block_id: int;
}

type c_func = {
    c_fname : string;
    c_ret_type : var_type;
    c_formals : scope_var_decl list;
    c_fblock : c_block;
}

type c_program = scope_var_decl list * c_func list

(* structures the 'main' function *)
let main_fdecl (f:c_func) =
```

```
  if f.c_fname = "main" && f.c_ret_type = Lrx_Atom(Lrx_Int) && f.c_formals = []
        then true else false

(*called to get the Atom/Tree type of an expresion*)
let type_of_expr = function
    C_Int_Literal(i) -> Lrx_Atom(Lrx_Int)
  | C_Float_Literal(f) -> Lrx_Atom(Lrx_Float)
  | C_String_Literal(s) -> Lrx_Tree({datatype = Lrx_Char; degree = Int_Literal(1)})
  | C_Char_Literal(c) -> Lrx_Atom(Lrx_Char)
  | C_Bool_Literal(b) -> Lrx_Atom(Lrx_Bool)
  | C_Binop(t,_,_,_) -> t
  | C_Unop(t,_,_) -> t
  | C_Id(t,_,_) -> t
  | C_Assign(t,_,_) -> t
  | C_Tree(t, d, _, _) ->
    (match t with
        Lrx_Atom(t) -> Lrx_Tree({datatype = t; degree = Int_Literal(d)})
      | _ -> raise (Failure "Tree type must be Lrx_atom"))
  | C_Call(f,_) -> let (_,r,_,_) = f in r
  | (C_Noexpr | C_Null_Literal) -> raise (Failure("Type of expression called on Null_Literal or Noexpr"))

(* error raised for improper binary operation *)
let binop_error (t1:var_type) (t2:var_type) (op:op) =
  raise(Failure("operator " ^ (string_of_binop op) ^ " not compatible with expressions of type " ^
    (string_of_var_type t1) ^ " and " ^ (string_of_var_type t2)))


(* check binary operators *)
let check_binop (c1:c_expr) (c2:c_expr) (op:op) =
  match (c1, c2) with
      (C_Null_Literal, C_Null_Literal) ->
      (match op with
          (Equal | Neq) -> C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
        | _ -> raise (Failure ("operator " ^ string_of_binop op ^ " not compatible with types null and null")))
    | ((C_Null_Literal, t) | (t, C_Null_Literal)) ->
      (match (type_of_expr t) with
          Lrx_Tree(l) ->
          (match op with
              (Equal | Neq) -> C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
            | _ -> raise (Failure ("operator " ^ string_of_binop op ^ " not compatible with types null and tree")))
        | _ -> raise (Failure ("null cannot be compared with non-tree type")))
    | _ ->
    let (t1, t2) = (type_of_expr c1, type_of_expr c2) in
    match (t1, t2) with
       (Lrx_Atom(Lrx_Int), Lrx_Atom(Lrx_Int)) ->
       (match op with
           (Add | Sub | Mult | Div | Mod) -> C_Binop(Lrx_Atom(Lrx_Int), c1, op, c2)
         | (Equal | Neq | Less | Leq | Greater | Geq) -> C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
         | _ -> binop_error t1 t2 op)
     | (Lrx_Atom(Lrx_Float), Lrx_Atom(Lrx_Float)) ->
       (match op with
           (Add | Sub | Mult | Div) ->   C_Binop(Lrx_Atom(Lrx_Float), c1, op, c2)
         | (Equal | Neq | Less | Leq | Greater | Geq) -> C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
         | _ -> binop_error t1 t2 op)
     | (Lrx_Atom(Lrx_Bool), Lrx_Atom(Lrx_Bool)) ->
       (match op with
           (And | Or | Equal | Neq) ->
               C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
             | _ -> binop_error t1 t2 op)
     | (Lrx_Atom(Lrx_Char), Lrx_Atom(Lrx_Char)) ->
       (match op with
           (Add | Sub) -> C_Binop(Lrx_Atom(Lrx_Char), c1, op, c2)
         | (Equal | Neq | Less | Leq | Greater | Geq) -> C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
         | _ -> binop_error t1 t2 op)
     | (Lrx_Tree(t), Lrx_Atom(Lrx_Int)) ->
          (if op = Child then
```

```
            C_Binop(Lrx_Tree(t), c1, op, c2)
          else binop_error t1 t2 op)
     | (Lrx_Tree(l1), Lrx_Tree(l2)) ->
          (match op with
              Add -> if l1.datatype = l2.datatype then C_Binop(Lrx_Tree(l1), c1, op, c2)
              else raise (Failure ("Cannot add type " ^ string_of_var_type t1 ^ " with type " ^ string_of_var_type t2))
            | (Equal | Neq) -> if l1.datatype = l2.datatype then C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
              else ((prerr_string ("Warning: comparison of " ^ string_of_var_type t1 ^ " with type " ^ string_of_var_type t2))
                 ; C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2))
            | (Less | Greater | Leq | Geq) -> C_Binop(Lrx_Atom(Lrx_Bool), c1, op, c2)
            | _ -> binop_error t1 t2 op)
     | _ -> binop_error t1 t2 op



let unop_error (t:var_type) (op:Ast.uop) =
  raise(Failure("operator " ^ (string_of_unop op) ^ " not compatible with expression of type " ^ (string_of_var_type t)))

let check_unop (c:c_expr) (op:Ast.uop) =
  let te = type_of_expr c in
  match te with
     Lrx_Atom(Lrx_Int) ->
     (match op with
         Neg -> C_Unop(Lrx_Atom(Lrx_Int), c, op)
       | _ -> unop_error te op)
   | Lrx_Atom(Lrx_Float) ->
     (match op with
         Neg -> C_Unop(Lrx_Atom(Lrx_Float), c, op)
       | _ -> unop_error te op)
   | Lrx_Atom(Lrx_Bool) ->
     (match op with
         Not -> C_Unop(Lrx_Atom(Lrx_Bool), c, op)
       | _ -> unop_error te op)
   | Lrx_Tree(t) ->
     (match op with
         Pop -> C_Unop(Lrx_Tree(t), c, op)
       | At -> C_Unop(Lrx_Atom(t.datatype), c, op)
       | _ -> unop_error te op)
   | _ -> unop_error te op

(*compares argument list*)
let rec compare_arglists formals actuals =
    match (formals,actuals) with
        ([],[]) -> true
      | (head1::tail1, head2::tail2) ->
     (match (head1, head2) with
         (Lrx_Tree(t1), Lrx_Tree(t2)) -> (t1.datatype = t2.datatype) && compare_arglists tail1 tail2
       | _ -> (head1 = head2) && compare_arglists tail1 tail2)
      | _ -> false

(*checks that a function declaration and calling is proper, such that a function is called with the proper number and type of arguments*)
and check_fun_call (name:string) (cl:c_expr list) env =
  (*if name == print, match type with symtab print_type*)
    let decl = Symtab.symtab_find name env in
    let fdecl =
  (match decl with
      SymTab_FuncDecl(f) -> f
           | _ -> raise(Failure("symbol " ^ name ^ " is not a function"))) in
      let (fname,ret_type,formals,id) = fdecl in
      let actuals = List.map type_of_expr cl in
      match name with
         "print" -> C_Call((fname, ret_type, actuals, id), cl)
       | ("degree" | "root" | "parent") ->
         if ((List.length actuals) = 1) then
           let tree_arg = List.hd actuals in
           match tree_arg with
              Lrx_Tree(t) ->
```

```
              if name = "degree" then C_Call((fname, ret_type, actuals, id), cl)
              else C_Call((fname, tree_arg, actuals, id), cl)
            | _ -> raise(Failure("function degree expects tree"))
        else raise(Failure("function " ^ name ^ " expects a single tree as an argument"))
      | _ ->
       if (List.length formals) = (List.length actuals) then
                 if compare_arglists formals actuals then C_Call(fdecl, cl)
                 else raise(Failure("function " ^ name ^ "'s argument types don't match its formals"))
         else raise(Failure("function " ^ name ^ " expected " ^ (string_of_int (List.length actuals)) ^
                  " arguments but called with " ^ (string_of_int (List.length formals))))

let rec check_id_is_valid (id_name:string) env =
     let decl = Symtab.symtab_find id_name env in
     let id = Symtab.symtab_get_id id_name env in
     (match decl with
         SymTab_VarDecl(v) -> (snd_of_three v, fst_of_three v, id)
       | _ -> raise (Failure("symbol " ^ id_name ^ " is not a variable")))

and extract_l_value (l:c_expr) env =
    match l with
       C_Id(t,s,_) -> s
     | C_Binop(t,l,o,r) -> extract_l_value l env
     | C_Unop(t,l,o) -> extract_l_value l env
     | _ -> raise (Failure ("Cannot dereference expression without id"))

and check_l_value (l:expr) env =
    match l with
       Id(s) -> let (t, e, id) = check_id_is_valid s env in C_Id(t,e, id)
     | _ -> let ce = (check_expr l env) in
       match ce with
          C_Binop(_,_,op,_) ->
          (if op = Child then
            (let s = (extract_l_value ce env) in
            let (t, e, _) = check_id_is_valid s env in
            ignore t; ignore e; ce)
           else raise (Failure ("Left hand side of assignment operator is improper type")))
         | C_Unop(_,_,op) ->
           (if op = At then
              (let s = (extract_l_value ce env) in
              ignore (check_id_is_valid s env); ce)
           else raise (Failure ("Left hand side of assignment operator is improper type")))
         | _ -> raise (Failure ("Left hand side of assignment operator is improper type"))

 and check_tree_literal_is_valid (d:int) (t:var_type) (el:expr list) env =
     match el with
        [] -> []
      | head :: tail ->
        let checked_expr = check_expr head env in
        match checked_expr with
           C_Tree(tree_type, tree_degree, child_e, child_el) ->
           if (tree_degree = d || tree_degree = 0) && tree_type = t then
             C_Tree(tree_type, d, child_e, child_el) :: check_tree_literal_is_valid d t tail env
           else raise (Failure ("Tree type is not consistent: expected <" ^ string_of_var_type t ^ ">(" ^ string_of_int d ^ ") but received <" ^ string_of_var_type tree_type ^ ">(" ^ string_of_int tree_degree ^ ")"))
         | _ ->
           let child_type = (type_of_expr checked_expr) in
           if child_type = t then
             checked_expr :: check_tree_literal_is_valid d t tail env
           else raise (Failure ("Tree literal type is not consistent: expected <" ^ string_of_var_type t ^ "> but received <" ^ string_of_var_type child_type ^">"))

and check_tree_literal_root_is_valid (e:expr) (el: expr list) env =
  let checked_root = check_expr e env in
  let type_root = type_of_expr checked_root in
  match type_root with
     (Lrx_Atom(Lrx_Int) | Lrx_Atom(Lrx_Float) | Lrx_Atom(Lrx_Char) | Lrx_Atom(Lrx_Bool)) ->
     let degree_root = List.length el in
     let checked_tree = check_tree_literal_is_valid degree_root type_root el env in
     (type_root, degree_root, checked_root, checked_tree)
```

```
   | _ -> raise (Failure ("Tree root cannot be of non-atom type: " ^ string_of_var_type type_root))

and check_expr (e:expr) env =
      match e with
        Int_Literal(i) -> C_Int_Literal(i)
      | Float_Literal(f) -> C_Float_Literal(f)
      | String_Literal(s) -> C_String_Literal(s)
      | Char_Literal(c) -> C_Char_Literal(c)
      | Bool_Literal(b) -> C_Bool_Literal(b)
      | Tree(e, el) -> let (t, d, e, el) = check_tree_literal_root_is_valid e el env in
           C_Tree(t, d, e, el)
      | Id(s) -> let (t, e, id) = check_id_is_valid s env in
           C_Id(t,e, id)
      | Binop(e1, op, e2) ->
        let (c1, c2) = (check_expr e1 env, check_expr e2 env) in
         check_binop c1 c2 op (* returns C_Binop *)
      | Assign(l, r) ->
        let checked_r = check_expr r env in
        let checked_l = check_l_value l env in
        let t_r = type_of_expr checked_r in
        let t_l =  type_of_expr checked_l in
        (match (t_l, t_r) with
        | (Lrx_Atom(a1), Lrx_Atom(a2)) ->
           if t_r = t_l then C_Assign(t_l, checked_l, checked_r) else
             raise(Failure("assignment not compatible with expressions of type " ^ string_of_var_type t_l ^ " and " ^ string_of_var_type t_r))
        | (Lrx_Tree(t1), Lrx_Tree(t2)) ->
          if t1.datatype = t2.datatype then C_Assign(t_l, checked_l, checked_r) else
          raise(Failure("assignment not compatible with expressions of type " ^ string_of_var_type t_l ^ " and " ^ string_of_var_type t_r))
        | _ -> raise(Failure("assignment not compatible with expressions of type " ^ string_of_var_type t_l ^ " and " ^ string_of_var_type t_r)) )
      | Unop(e, op) ->
           let checked = check_expr e env in
           check_unop checked op (* returns C_Unop *)
      | Null_Literal -> C_Null_Literal
      | Call(n, el) ->
           let checked = check_exprlist el env in
           check_fun_call n checked env
      | Noexpr -> C_Noexpr

and check_exprlist (el:expr list) env =
      match el with
         [] -> []
       | head :: tail -> (check_expr head env) :: (check_exprlist tail env)


(* check a single statement *)
let rec check_statement (s:stmt) ret_type env (in_loop:int) =
      match s with
         CodeBlock(b) ->
        let checked_block = check_block b ret_type env in_loop in
        C_CodeBlock(checked_block)
      | Return(e) ->
        let checked = check_expr e env in
        let t = type_of_expr checked in
        if t = ret_type then C_Return(checked) else
        raise (Failure("function return type " ^ string_of_var_type t ^ "; type " ^ string_of_var_type ret_type ^ "expected"))
      | Expr(e) -> C_Expr(check_expr e env)
      | If(e, b1, b2) ->
         let c = check_expr e env in
         let t = type_of_expr c in
         (match t with
           Lrx_Atom(Lrx_Bool) -> C_If(c, check_block b1 ret_type env in_loop, check_block b2 ret_type env in_loop)
          | _ -> raise (Failure "If statement must evaluate on boolean expression"))
      | For(e1, e2, e3, b) ->
        let (c1, c2, c3) = (check_expr e1 env, check_expr e2 env, check_expr e3 env) in
        if(type_of_expr c2 = Lrx_Atom(Lrx_Bool)) then
        C_For(c1, c2, c3, check_block b ret_type env (in_loop + 1))
               else raise(Failure("for loop condition must evaluate on boolean expressions"))
```

```
        | While(e, b) ->
        let c = check_expr e env in
              if type_of_expr c = Lrx_Atom(Lrx_Bool) then
        C_While(c, check_block b ret_type env (in_loop + 1))
              else raise(Failure("while loop must evaluate on boolean expression"))
      | Continue ->
        if in_loop = 0 then raise (Failure "continue statement not within for or while loop")
        else C_Continue
      | Break ->
        if in_loop = 0 then raise (Failure "break statement not within for or while loop")
        else C_Break

and check_is_fdecl (f:string) env =
     let fd = Symtab.symtab_find f env in
      match fd with
             SymTab_VarDecl(v) -> raise(Failure("symbol is not a function"))
        | SymTab_FuncDecl(f) -> f

(* returns a verified statement list *)
and check_statement_list (s:stmt list) (ret_type:var_type) env (in_loop:int)=
     match s with
        [] -> []
      | head :: tail -> check_statement head ret_type env in_loop :: check_statement_list tail ret_type env in_loop

(* returns verified c_block record *)
and check_block (b:block) (ret_type:var_type) env (in_loop:int) =
     let vars = check_is_vardecls b.locals (fst env, b.block_id) in
     let stmts = check_statement_list b.statements ret_type (fst env, b.block_id) in_loop in
     { c_locals = vars; c_statements = stmts; c_block_id = b.block_id }

(* returns c_func record *)
and check_function (f:func) env =
     let checked_block = check_block f.fblock f.ret_type env 0 in
     let checked_formals = check_is_vardecls f.formals (fst env, f.fblock.block_id) in
     let checked_scope_func_decl = check_is_fdecl f.fname env in
     { c_fname = fst_of_four checked_scope_func_decl; c_ret_type = f.ret_type; c_formals = checked_formals; c_fblock = checked_block }

(* returns list of verified function declarations *)
and check_functions (funcs:func list) env =
      match funcs with
         [] -> []
       | head :: tail -> check_function head env :: check_functions tail env

and check_main_exists (f:c_func list) =
      if (List.filter main_fdecl f) = [] then false else true

(* returns list of verified global variable declarations *)
and check_is_vardecls (vars: var list) env =
     match vars with
        [] -> []
      | head :: tail ->
        let decl = Symtab.symtab_find (fst head) env in
        let id = Symtab.symtab_get_id (fst head) env in
        match decl with
            SymTab_FuncDecl(f) -> raise(Failure("symbol is not a variable"))
          | SymTab_VarDecl(v) ->
          let var = snd_of_three v in
          match var with
             Lrx_Tree(t) ->
             let checked_degree = check_expr t.degree env in
             let type_of_degree = type_of_expr checked_degree in
             (match type_of_degree with
                 Lrx_Atom(Lrx_Int) -> (fst_of_three v, snd_of_three v, id) :: check_is_vardecls tail env
               | _ -> raise (Failure ("Tree degree must be of type int")))
         | Lrx_Atom(a) -> (fst_of_three v, snd_of_three v, id) :: check_is_vardecls tail env


(*
 * returns (<<verified list of global variable declarations>>, <<verified list of function declarations>>)
```

```
 *)
let check_program (p:program) env =
    let gs = fst p in
    let fs = snd p in
     let vdecllst = check_is_vardecls gs env in
     let fdecllst = check_functions fs env in
     if (check_main_exists fdecllst) then (vdecllst, fdecllst)
     else raise (Failure("function main not found"))
```

intermediate.ml

```
(*
 * Authors:
 * Kira Whithouse
 * Chris D'Angelo
 * Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
 * which provided background knowledge.
 *)

open Ast
open Check

let tmp_reg_id = ref 0
let label_id = ref 0

let string_of_tmp_var_type  = function
    Lrx_Atom(t) -> string_of_atom_type t
  | Lrx_Tree(t) -> "tree_datatype_" ^ string_of_atom_type t.datatype ^ "_degree_" ^ string_of_expr t.degree

let gen_tmp_var t u =
  let x = tmp_reg_id.contents in
  let prefix = "__tmp_" ^ string_of_tmp_var_type t in
  tmp_reg_id := x + 1; (prefix, t, x, u)


let gen_tmp_label (s:unit) =
  let x = label_id.contents in
  label_id := x + 1; "__LABEL_" ^ (string_of_int x)

(* == scope_var_decl * int *)
type ir_var_decl = string * var_type * int * int

(* Ir_String_Literal unecessary. Converted to Ir_Tree_Literal here. *)
type ir_expr =
    Ir_Int_Literal of ir_var_decl * int
  | Ir_Float_Literal of ir_var_decl * float
  | Ir_Char_Literal of ir_var_decl * char
  | Ir_Bool_Literal of ir_var_decl * bool
  | Ir_Unop of ir_var_decl * uop * ir_var_decl
  | Ir_Binop of ir_var_decl * op * ir_var_decl * ir_var_decl
  | Ir_Id of ir_var_decl * ir_var_decl
  | Ir_Assign of ir_var_decl * ir_var_decl
  | Ir_Tree_Literal of ir_var_decl * ir_var_decl * ir_var_decl (* 4[3, 2[]]*)
  | Ir_Call of ir_var_decl * scope_func_decl * ir_var_decl list
  | Ir_Null_Literal of ir_var_decl
  | Ir_Noexpr

type ir_stmt =
  | Ir_If of ir_var_decl * string
  | Ir_Jmp of string
  | Ir_Label of string
  | Ir_Decl of ir_var_decl
  | Ir_Null_Decl of ir_var_decl
  | Ir_Tree_Destroy of ir_var_decl
  | Ir_Tree_Add_Destroy of ir_var_decl
  | Ir_Ret of ir_var_decl * string * string
  | Ir_Expr of ir_expr
  | Ir_Ptr of ir_var_decl * ir_var_decl
  | Ir_At_Ptr of ir_var_decl
  | Ir_Leaf of ir_var_decl * int
  | Ir_Internal of ir_var_decl * int * ir_var_decl
  | Ir_Child_Array of ir_var_decl * int
```

```
  | Ir_Decl_Umbilical of ir_var_decl

type ir_func = {
  ir_header: var_type * string * scope_var_decl list;
  ir_vdecls: ir_stmt list;
  ir_stmts: ir_stmt list;
  ir_destroys: ir_stmt list;
}

type ir_fheader = {
  ir_name: string;
  ir_ret_type: var_type;
  ir_formals: var_type list;
}

type ir_program = {
  ir_globals: scope_var_decl list;
  ir_headers: ir_fheader list;
  ir_bodies: ir_func list;
}

let is_destroy (s: ir_stmt) =
  match s with
     (Ir_Tree_Destroy(_) | Ir_Tree_Add_Destroy(_))-> true
    | _ -> false

let is_not_destroy (s:ir_stmt) =
  not (is_destroy s)

let is_decl (s: ir_stmt) =
  match s with
    ( Ir_Decl(_) | Ir_At_Ptr(_) | Ir_Null_Decl(_)) -> true
    | _ -> false

let is_not_decl (s:ir_stmt) =
  not (is_decl s)

let gen_ir_default_ret (t: var_type) =
  let tmp = (gen_tmp_var t 0) in
  let start_cleanup = gen_tmp_label () in
  let end_cleanup = gen_tmp_label () in
  [Ir_Decl(tmp); Ir_Ret(tmp, start_cleanup, end_cleanup)]

let is_atom t =
   let (_, t2, _, _) = t in
   match t2 with
      Lrx_Tree(_) -> false
    | _ -> true

let is_tree t =
    not (is_atom t)

let gen_tmp_internal child tree_type child_number child_array =
  [Ir_Internal(child_array, child_number, child)]

let rec gen_tmp_internals children tree_type array_access child_array =
 match children with
    [] -> []
  | head :: tail -> gen_tmp_internal head tree_type array_access child_array @ gen_tmp_internals tail tree_type (array_access + 1) child_array

let gen_tmp_child child tree_type tree_degree =
  if (is_atom child) then
    let tmp_root_data = (gen_tmp_var tree_type 0) in
    let d =
    (match tree_type with
        Lrx_Atom(a) -> a
      | Lrx_Tree(t) -> raise(Failure "Tree type as tree data item. (Error 3)")) in
    let tmp_leaf_children = (gen_tmp_var (Lrx_Tree({datatype = d; degree = Int_Literal(tree_degree)})) 0) in
    let tmp_leaf_root = (gen_tmp_var (Lrx_Tree({datatype = d; degree = Int_Literal(tree_degree)})) 0) in
    ([Ir_At_Ptr(tmp_root_data);
```

```
      Ir_Ptr(tmp_root_data, child);
      Ir_Leaf(tmp_leaf_children, tree_degree);
      Ir_Decl(tmp_leaf_root);
      Ir_Tree_Destroy(tmp_leaf_root);
      Ir_Expr(Ir_Tree_Literal(tmp_leaf_root, tmp_root_data, tmp_leaf_children))], tmp_leaf_root)
  else
    ([], child)

let rec gen_tmp_children children tree_type tree_degree =
  match children with
  [] -> []
  | head :: tail -> gen_tmp_child head tree_type tree_degree :: gen_tmp_children tail tree_type tree_degree

let gen_tmp_tree tree_type tree_degree root children_list tmp_tree =
  let children = gen_tmp_children children_list tree_type tree_degree in
  let (decls, tmp_children) = (List.fold_left (fun (a, b) (c, d) -> ((c @ a), (d :: b))) ([],[]) (List.rev children)) in
  let d =
  (match tree_type with
      Lrx_Atom(a) -> a
    | Lrx_Tree(t) -> raise(Failure "Tree type as tree data item. (Error 1)")) in
  let child_array = gen_tmp_var (Lrx_Tree({datatype = d; degree = Int_Literal(tree_degree)})) 0 in
  let internals = gen_tmp_internals tmp_children tree_type 0 child_array in
  let tmp_root_ptr = gen_tmp_var tree_type 0 in
  decls @ [Ir_Child_Array(child_array, tree_degree)] @ internals @ [Ir_At_Ptr(tmp_root_ptr); Ir_Ptr(tmp_root_ptr, root)] @ [Ir_Expr(Ir_Tree_Literal(tmp_tree, tmp_root_ptr, child_array))]

let rec char_list_to_c_tree cl =
    match cl with
       [t] -> C_Tree(Lrx_Atom(Lrx_Char), 1, C_Char_Literal(t), [])
     | h :: t ->
       if h = '\\' then
          let h2 = (List.hd t) in
          let escape_char =
          match h2 with
             'n' -> '\n'
           | 't' -> '\t'
           | '\\' -> '\\'
           | _ -> raise (Failure "Invalid escape sequence used in string literal") in
          if (List.length (List.tl t)) = 0 then C_Tree(Lrx_Atom(Lrx_Char), 1, C_Char_Literal(escape_char), [])
          else C_Tree(Lrx_Atom(Lrx_Char), 1, C_Char_Literal(escape_char), [(char_list_to_c_tree (List.tl t))])
       else
          C_Tree(Lrx_Atom(Lrx_Char), 1, C_Char_Literal(h), [(char_list_to_c_tree t)])
     | _ -> raise (Failure "Cannot create an empty string literal")

let string_to_char_list s =
  let rec exp i l = if i < 0 then l else exp (i - 1) (s.[i] :: l) in
  exp (String.length s - 1) []

let rec gen_ir_expr_list (el:c_expr list) (args:scope_var_decl list) =
  match el with
  [] -> []
  | head :: tail -> gen_ir_expr head args :: gen_ir_expr_list tail args

and gen_ir_expr (e:c_expr) (args:scope_var_decl list) =
  match e with
     C_Int_Literal(i) ->
     let tmp = gen_tmp_var (Lrx_Atom(Lrx_Int)) 0 in
     ([Ir_Decl(tmp); Ir_Expr(Ir_Int_Literal(tmp, i))], tmp)
   | C_Float_Literal(f) ->
     let tmp = gen_tmp_var (Lrx_Atom(Lrx_Float)) 0 in
     ([Ir_Decl(tmp); Ir_Expr(Ir_Float_Literal(tmp, f))], tmp)
   | C_Char_Literal(c) ->
     let tmp = gen_tmp_var (Lrx_Atom(Lrx_Char)) 0 in
     ([Ir_Decl(tmp); Ir_Expr(Ir_Char_Literal(tmp, c))], tmp)
   | C_Bool_Literal(b) ->
     let tmp = gen_tmp_var (Lrx_Atom(Lrx_Bool)) 0 in
     ([Ir_Decl(tmp); Ir_Expr(Ir_Bool_Literal(tmp, b))], tmp)
```

```
    | C_Unop(v, e, o) ->
      let (s, r) = gen_ir_expr e args in
      (match o with
          Pop -> raise (Failure "TEMPORARY: Pop not implemented.")
        | At -> let tmp = gen_tmp_var v 1 in ([Ir_At_Ptr(tmp)] @ s @ [Ir_Expr(Ir_Unop(tmp, o, r))], tmp)
        | _ -> let tmp = gen_tmp_var v 0 in ([Ir_Decl(tmp)] @ s @ [Ir_Expr(Ir_Unop(tmp, o, r))], tmp))
    | C_Binop(v, e1, o, e2) ->
      let (s1, r1) = gen_ir_expr e1 args in
      let (s2, r2) = gen_ir_expr e2 args in
     let tmp =
        (match o with
            Child -> gen_tmp_var v 1
          | _ -> gen_tmp_var v 0 )
      in (match (v, o) with
             (Lrx_Tree(t), Add) -> ([Ir_Decl(tmp); Ir_Tree_Add_Destroy(tmp)] @ s1 @ s2 @ [Ir_Expr(Ir_Binop(tmp, o, r1, r2))], tmp)
           | _ -> ([Ir_Decl(tmp)] @ s1 @ s2 @ [Ir_Expr(Ir_Binop(tmp, o, r1, r2))], tmp))
    | C_Id(t, s, i) ->
       (match t with
       Lrx_Tree(_) -> if (List.exists (fun (s1, t1, i1) -> (s1 = s && t1 = t && i1 = i)) args) then ([], (s, t, i, 3)) else ([], (s, t, i, 0))
        | _ -> ([], (s, t, i, 0)))
    | C_Assign(t, l, r) ->
      let (s1, r1) = gen_ir_expr l args in
      let (s2, r2) = gen_ir_expr r args in
      (s1 @ s2 @ [Ir_Expr(Ir_Assign(r1, r2))], r2)
    | C_Tree(t, d, e, el) ->
      let (s, r) = gen_ir_expr e args in
      let ir_el = gen_ir_expr_list el args in
      let (sl, rl) = (List.fold_left (fun (sl_ir, rl_ir) (s_ir, r_ir) -> (sl_ir @ s_ir, rl_ir@[r_ir])) ([],[]) ir_el) in
      let i  =
      (match t with
          Lrx_Atom(a) -> a
        | Lrx_Tree(t) -> raise (Failure "Tree type as tree data item. (Error 2)")) in
      let tmp = (gen_tmp_var (Lrx_Tree({datatype = i; degree = Int_Literal(d)})) 0) in
      let tmp_tree =  gen_tmp_tree t d r rl tmp in
      ([Ir_Decl(tmp); Ir_Tree_Destroy(tmp)] @ sl @ s @ tmp_tree, tmp)
    | C_Call(fd, el) ->
      let (n, rt, fm, s) = fd in
      let ir_el = gen_ir_expr_list el args in
      let tmp =
      (match n with
          ("parent" | "root") -> gen_tmp_var rt 1
        | _ -> gen_tmp_var rt 0)
      in
      let (sl, rl) = (List.fold_left (fun (sl_ir, rl_ir) (s_ir, r_ir) -> (sl_ir @ s_ir, rl_ir@[r_ir])) ([],[]) ir_el) in
      (Ir_Decl(tmp) :: sl @ [Ir_Expr(Ir_Call(tmp, fd, rl))], tmp)
    | C_String_Literal(s) -> let result = (char_list_to_c_tree (string_to_char_list s)) in
      gen_ir_expr result args
    | C_Null_Literal -> let tmp = (gen_tmp_var (Lrx_Tree({datatype = Lrx_Int; degree = Int_Literal(1)})) 2) in
      ([Ir_Null_Decl(tmp); Ir_Expr(Ir_Null_Literal(tmp))], tmp)
    | C_Noexpr -> ([Ir_Expr(Ir_Noexpr)], ("void_tmp_unused", Lrx_Atom(Lrx_Int), -1, -1))

let decl_and_destroy_local (v:scope_var_decl) =
  let (n, t, s) = v in
  (match t with
      Lrx_Tree(_) -> [Ir_Tree_Destroy(n, t, s, 0); Ir_Decl(n, t, s, 0)]
    | _ -> [Ir_Decl(n, t, s, 0)])

let rec decl_and_destroy_locals (vl:scope_var_decl list) =
  match vl with
      [] -> []
    | head :: tail -> decl_and_destroy_local head @ decl_and_destroy_locals tail

let rec gen_ir_block (b: c_block) (args:scope_var_decl list) =
  let decls = decl_and_destroy_locals b.c_locals in
  decls @ (gen_ir_stmtlist b.c_statements args)
```

```
and gen_ir_stmt (s: c_stmt) (args:scope_var_decl list) =
   match s with
      C_CodeBlock(b) -> gen_ir_block b args
    | C_Return(e) ->
      let (s, r) = gen_ir_expr e args in
      let start_cleanup = gen_tmp_label () in
      let end_cleanup = gen_tmp_label () in
     s @ [Ir_Ret(r, start_cleanup, end_cleanup)]
    | C_Expr(e) -> fst (gen_ir_expr e args)
    | C_If(e, b1, b2) ->
      let (s, r) = gen_ir_expr e args in
      let irb1 = gen_ir_block b1 args in
      let irb2 = gen_ir_block b2 args in
      let startlabel = gen_tmp_label () in
      let endlabel = gen_tmp_label () in
      s @ [Ir_If(r, startlabel)] @ irb2 @ [Ir_Jmp(endlabel); Ir_Label(startlabel)] @ irb1 @ [Ir_Label(endlabel)]
    | C_For(e1, e2, e3, b) ->
      let (s1, r1) = gen_ir_expr e1 args in
      let (s2, r2) = gen_ir_expr e2 args in
      let (s3, r3) = gen_ir_expr e3 args in
      let irb = gen_ir_block b args in
      let startlabel = gen_tmp_label () in
      let endlabel = gen_tmp_label () in
      s1 @ [Ir_Jmp(endlabel); Ir_Label(startlabel)] @ irb @ s3 @ [Ir_Label(endlabel)] @ s2 @ [Ir_If(r2, startlabel)]
    | C_While(e, b) ->
      let (s, r) = gen_ir_expr e args in
      let irb = gen_ir_block b args in
      let startlabel = gen_tmp_label () in
      let endlabel = gen_tmp_label () in
      [Ir_Jmp(endlabel); Ir_Label(startlabel)] @ irb @ [Ir_Label(endlabel)] @ s @ [Ir_If(r, startlabel)]
    | C_Continue -> raise (Failure "TEMPORARY: Continue not implemented.")
    | C_Break -> raise (Failure "TEMPORARY: Break not implemented.")

and gen_ir_stmtlist (slist: c_stmt list) (args:scope_var_decl list) =
  match slist with
     [] -> []
   | head :: tail -> gen_ir_stmt head args @ gen_ir_stmtlist tail args

and gen_ir_body (f: c_func) =
  let header = (f.c_ret_type, f.c_fname, f.c_formals) in
  let default_ret = gen_ir_default_ret f.c_ret_type in
  let body = gen_ir_block f.c_fblock f.c_formals @ default_ret in
  let decls = List.filter is_decl body in
  let stmts = List.filter is_not_decl body in
  let destroys = List.filter is_destroy stmts in
  let stmts = List.filter is_not_destroy stmts in
  {ir_header = header; ir_vdecls = decls; ir_stmts = stmts; ir_destroys = destroys}

and gen_ir_fbodys (flist:c_func list) =
  match flist with
     [] -> []
   | head :: tail -> gen_ir_body head :: gen_ir_fbodys tail

and gen_ir_fdecls (flist:c_func list) =
  match flist with
     [] -> []
   | head :: tail ->
  {ir_name = head.c_fname; ir_ret_type = head.c_ret_type; ir_formals = List.map snd_of_three head.c_formals} :: gen_ir_fdecls tail

let rec intermediate_rep_program (p:c_program) =
  let ir_fdecls = gen_ir_fdecls (snd p) in
  let ir_fbodys = gen_ir_fbodys (snd p) in
  {ir_globals = fst p; ir_headers = ir_fdecls; ir_bodies = ir_fbodys}
```

lorax.ml

```
(*
 * Authors:
 * Chris D'Angelo
```

```
 * Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
 * which provided background knowledge.
 *)

open Unix

let c_compiler = "gcc"
let c_warnings = "-w"
let c_debug = "-Wall"
let c_includes = "-I"

type action = Ast | Symtab | SAnalysis | Compile | Binary | Help

let usage (name:string) =
  "usage:\n" ^ name ^ "\n" ^
    "        -a source.lrx              (Print AST of source)\n" ^
    "        -t source.lrx              (Print Symbol Table of source)\n" ^
    "        -s source.lrx              (Run Semantic Analysis over source)\n" ^
    "        -c source.lrx [target.c]   (Compile to c. Second argument optional)\n" ^
    "        -b source.lrx [target.out] (Compile to executable)\n"

let get_compiler_path (path:string) =
  try
    let i = String.rindex path '/' in
    String.sub path 0 i
  with _ -> "."

let _ =
  let action =
  if Array.length Sys.argv > 1 then
    (match Sys.argv.(1) with
        "-a" -> if Array.length Sys.argv == 3 then Ast else Help
      | "-t" -> if Array.length Sys.argv == 3 then Symtab else Help
      | "-s" -> if Array.length Sys.argv == 3 then SAnalysis else Help
      | "-c" -> if (Array.length Sys.argv == 3) || (Array.length Sys.argv == 4) then Compile else Help
      | "-b" -> if (Array.length Sys.argv == 3) || (Array.length Sys.argv == 4) then Binary else Help
      | _ -> Help)
  else Help in

  match action with
      Help -> print_endline (usage Sys.argv.(0))
    | (Ast | Symtab | SAnalysis | Compile | Binary ) ->
      let input = open_in Sys.argv.(2) in
      let lexbuf = Lexing.from_channel input in
      let program = Parser.program Scanner.token lexbuf in
      (match action with
          Ast -> let listing = Ast.string_of_program program
                 in print_string listing
        | Symtab -> let env = Symtab.symtab_of_program program in
                    print_string (Symtab.string_of_symtab env)
        | SAnalysis -> let env = Symtab.symtab_of_program program in
                    let checked = Check.check_program program env in
                    ignore checked; print_string "Passed Semantic Analysis.\n"
        | Compile -> let env = Symtab.symtab_of_program program in
                     let checked = Check.check_program program env in
                     let inter_pgrm = Intermediate.intermediate_rep_program checked in
                     let compiled_program = Output.c_of_inter_pgrm inter_pgrm in
                     if Array.length Sys.argv == 3 then print_endline compiled_program
                     else let out = open_out Sys.argv.(3) in output_string out compiled_program; close_out out
        | Binary ->  let env = Symtab.symtab_of_program program in
                     let checked = Check.check_program program env in
                     let inter_pgrm = Intermediate.intermediate_rep_program checked in
                     let compiled_program = Output.c_of_inter_pgrm inter_pgrm in
                     let tmp_c_file = Sys.argv.(2) ^ "_lrxtmp.c" in
                     let exec_file_name = if Array.length Sys.argv == 3 then "a.out" else Sys.argv.(3) in
                     let out = open_out tmp_c_file in
                     output_string out compiled_program; close_out out;
                    execvp c_compiler [|c_compiler; c_warnings; c_debug; c_includes ^ (get_compiler_path Sys.argv.(0)); tmp_c_file; "-o"; exec_file_name|]
```

```
        | Help -> print_endline (usage Sys.argv.(0))) (* impossible case *)

```

lrxlib.h

```
/*
 * Authors:
 * Kira Whitehouse
 * Chris D'Angelo
 * Doug Bienstock
 * Tim Paine
 */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stdint.h>
#define false 0
#define true !false

//#define LRXDEBUG
#ifdef LRXDEBUG
#define LrxLog( ... ) fprintf(stderr, __VA_ARGS__ )
#else
#define LrxLog( ... )
#endif

typedef enum {
    _GT_,
    _GTE_,
    _LT_,
    _LTE_,
    _EQ_,
    _NEQ_,
} Comparator;

typedef enum {
    _BOOL_,
    _INT_,
    _FLOAT_,
    _CHAR_,
    _STRING_ /* when tree is char type with degree = 1 */
} Atom;

typedef int bool;

typedef union Root {
    char char_root;
    int int_root;
    bool bool_root;
    float float_root;
} Root;

typedef struct tree{
    int degree;
    Atom datatype;
    Root root;

    /* array of children, which are themselves tree pointers */
    struct tree **children;
    struct tree *parent;

    /* leaf == childless */
    bool leaf;
    /* isNull == has been declared but not defined */
    bool is_null;
    /* reference count (smart pointer) */
    int *count;
} tree;

int lrx_print_bool(bool b) {
    if (b) {
```

```c
        fprintf(stdout, "true");
    } else {
        fprintf(stdout, "false");
    }
    return 0;
}

int lrx_print_tree(struct tree *t) {
    // Occurs when tree is imbalanced (one child is instantiated and not the others)
    if (t == NULL) {
        fprintf(stdout, "null");
        return 0;
    }

    LrxLog("datatype: %d\n", t->datatype);
    switch (t->datatype){
        case _INT_:
            fprintf(stdout, "%d", t->root.int_root);
            LrxLog("%d\n", t->root.int_root);
            break;
        case _FLOAT_:
            fprintf(stdout, "%f", t->root.float_root);
            break;
        case _CHAR_:
        case _STRING_:
            fprintf(stdout, "%c", t->root.char_root);
            break;
        case _BOOL_:
            lrx_print_bool(t->root.bool_root);
            break;
    }

    if (t->children) {
        int i;
        if (t->datatype != _STRING_ ) {
            fprintf(stdout, "[");
        }
        for (i = 0; i < t->degree; ++i) {

            if (t->children[i] == NULL && t->degree == 1 && (t->datatype == _CHAR_ || t->datatype == _STRING_)) {
                break;
            }
            lrx_print_tree(t->children[i]);

            if (t->datatype != _STRING_ && i != t->degree - 1){
                fprintf(stdout, ",");
            }
        }
        if (t->datatype != _STRING_) {
            fprintf(stdout, "]");
        }
    }
    return 0;
}

void lrx_destroy_add_tree(struct tree *t) {
    if (t == NULL){
        return;
    }

    if (t->children){
        int i;
        for(i = 0; i < t->degree; ++i){
            lrx_destroy_add_tree(t->children[i]);
        }
        free(t->children);
    }

    free(t->count);
    free(t);
}
```

```
void lrx_destroy_tree(struct tree *t) {

    if (t == NULL) {
        return;
    }

    *(t->count) -= 1;
    if (*(t->count) == 0) {

         if (t->children){
            int i;
            for (i = 0; i < t->degree; ++i){
                lrx_destroy_tree(t->children[i]);
            }
            free(t->children);
         }

          free(t->count);
          free(t);

    }
}

struct tree *lrx_declare_tree(Atom type, int deg) {
    assert(deg >= 0);
    struct tree *t = (struct tree *)malloc(sizeof(struct tree));
    assert(t);

    t->degree = deg;
    t->datatype = type;
    t->count = (int *)malloc(sizeof(int));
    assert(t->count);
    *(t->count) = 1;

    switch (type) {
        case _BOOL_:
            t->root.bool_root = false;
            break;
        case _INT_:
            t->root.int_root = 0;
            break;
        case _FLOAT_:
            t->root.float_root = 0.0;
            break;
        case _CHAR_:
        case _STRING_:
           if (t->degree == 1) {
               LrxLog("Declare string\n");
                 t->datatype = _STRING_;
           }
            t->root.char_root = '\0';
            break;
    }

    t->is_null = true;
    t->leaf = true;
    if (t->degree > 0) {
        t->children = (struct tree **)malloc(sizeof(struct tree *) * t->degree);
        assert(t->children);
        memset((t->children), 0, sizeof(struct tree*) * t->degree);
    }

    t->parent = NULL;
    return t;
}

struct tree *lrx_define_tree(struct tree *t, void *root_data, struct tree **children){
    /* set root data */
    switch (t->datatype){
        case _BOOL_:
            t->root.bool_root = *((bool *)root_data);
            break;

```

```c
        case _INT_:
            t->root.int_root = *((int *)root_data);
            break;

        case _FLOAT_:
            t->root.float_root = *((float *)root_data);
            break;
        case _CHAR_:
        case _STRING_:
            t->root.char_root = *((char *)root_data);
            break;
    }

    t->is_null = false;

    if (children == NULL){
        return t;
    }

    /* set pointers to children */
    int num_children = t->degree;
    int i;
    int null = 0;
    for (i = 0; i < num_children; ++i) {
        if (children[i] != NULL){
            children[i]->parent = t;
            *(children[i]->count) += 1;
            t->children[i] = children[i];
        }
        else {
            null +=1;
        }
    }
    if(null != num_children) {
        t->leaf = false;
    }

    return t;
}

/* data = t@; */
bool *lrx_access_data_at_bool (struct tree **t) {
    assert(*t != NULL);
    return &(*t)->root.bool_root;
}

int *lrx_access_data_at_int (struct tree **t) {
    assert(*t != NULL);
    return &((*t)->root.int_root);
}

float *lrx_access_data_at_float (struct tree **t) {
    assert(*t != NULL);
    return &(*t)->root.float_root;
}

char *lrx_access_data_at_char (struct tree **t) {
    assert(*t != NULL);
    return &(*t)->root.char_root;
}

/* t@ = data */
bool lrx_assign_data_at_bool (struct tree **t, const bool data) {
    assert(*t != NULL);
    return (*t)->root.bool_root = data;
}

int lrx_assign_data_at_int (struct tree **t, const int data) {
    assert(*t != NULL);
    return (*t)->root.int_root = data;
}

float lrx_assign_data_at_float (struct tree **t, const float data) {
```

```
    assert(t != NULL);
    return (*t)->root.float_root = data;
}

char lrx_assign_data_at_char (struct tree **t, const char data) {
    assert(t != NULL);
    return (*t)->root.char_root = data;
}

/* t1 = t2%0 */
struct tree **lrx_access_child (struct tree **t, const int child) {
    assert(*t);
    assert(child < (*t)->degree);

    /* ptr to the parent's ptr to it's children */
    return &((*t)->children[child]);
}

/* t1 = t2. Lhs is the tree pointer we need without dereference */
struct tree **lrx_assign_tree_direct(struct tree **lhs, struct tree **rhs) {
    if(lhs == rhs)
        return lhs;
    if(lhs && rhs && *rhs && *lhs){
        if((*rhs)->degree == 0) {
            int lhs_degree = (*lhs)->degree;
            (*rhs)->degree = lhs_degree;
            (*rhs)->children = (struct tree **)malloc(sizeof(struct tree *) * lhs_degree);
            assert((*rhs)->children);
            memset(((*rhs)->children), 0, sizeof(struct tree*) * lhs_degree);
        }
        assert((*lhs)->degree == (*rhs)->degree);
    }

    if(*lhs){
        if((*lhs)->parent){
            ((*lhs)->parent)->leaf = false;
        }
    }

    lrx_destroy_tree(*lhs);
    *lhs = *rhs;
    if(*rhs){
        if((*rhs)->count)
            *((*rhs)->count) += 1;
    }

    return lhs;
}

int _lrx_count_nodes( struct tree *t ) {
    int count = 0;
    int i;
    if(t == NULL ) {
        return 0;
    }
    if(t->leaf) {
        return 1;
    }
    count += 1;
    for(i = 0; i < t->degree; i++) {
        count +=  _lrx_count_nodes( t->children[i] );
    }
    return count;
}


void lrx_copy_construct_tree(struct tree **target, struct tree **source,
    int depth, int *insert, struct tree ***position) {

    void *root;
    switch((*source)->datatype){
        case _BOOL_:
            root = &(*source)->root.bool_root;
```

```
            break;

        case _INT_:
            root = &(*source)->root.int_root;
            break;

        case _FLOAT_:
            root = &(*source)->root.float_root;
            break;
        case _CHAR_:
        case _STRING_:
            root = &(*source)->root.char_root;
            break;
    }

    int degree = (*source)->degree;
    struct tree *children[degree];

    int i;
    for (i = 0; i < degree; ++i) {
        children[i] = NULL;

        if (!(*source)->leaf && (*source)->children && (*source)->children[i] != NULL){
            struct tree *child = lrx_declare_tree((*source)->datatype, degree);
            lrx_copy_construct_tree(&child, &(*source)->children[i], depth + 1, insert, position);
            children[i] = child;
        }
        else if (depth < *insert){
            *insert = depth;
            (*target)->leaf = false;
            *position = &((*target)->children[i]);
        }
    }
    *target = lrx_define_tree(*target, root, children);
}

/** concatenation
* appends t2 to the first available child sport in t1
* if no such spot is available
*/
void lrx_add_trees(struct tree **target, struct tree **lhs, struct tree **rhs) {
    if (lhs && rhs && *rhs && *lhs) {
        assert((*lhs)->datatype == (*rhs)->datatype);

        int rhs_degree = (*rhs)->degree;
        int lhs_degree = (*lhs)->degree;
        if (rhs_degree == 0 && lhs_degree == 0){
            (*rhs)->degree = 1;
            (*lhs)->degree = 1;
        }
        if (rhs_degree == 0) {
           (*rhs)->degree = (*lhs)->degree;

             (*rhs)->children = (struct tree **)malloc(sizeof(struct tree *) * (*rhs)->degree);
             assert((*rhs)->children);
             memset(((*rhs)->children), 0, sizeof(struct tree *) * (*rhs)->degree);
        }
        if (lhs_degree == 0) {
            (*lhs)->degree = (*rhs)->degree;
            (*lhs)->children = (struct tree **)malloc(sizeof(struct tree *) * (*lhs)->degree);
            assert((*lhs)->children);
            memset(((*lhs)->children), 0, sizeof(struct tree *) * (*lhs)->degree);

            (*target)->degree = (*rhs)->degree;
            (*target)->children = (struct tree **)malloc(sizeof(struct tree *) * (*lhs)->degree);
            assert((*target)->children);
            memset(((*target)->children), 0, sizeof(struct tree *) * (*lhs)->degree);
        }
        assert((*lhs)->degree == (*rhs)->degree);
```

```
    }

    /* copy construct lhs */
    int max_nodes_lhs = _lrx_count_nodes(*lhs);
    struct tree **pos;
    lrx_copy_construct_tree(target, lhs, 0, &max_nodes_lhs, &pos);

    /* copy construct rhs */
    struct tree **trash;
    int max_nodes_rhs = _lrx_count_nodes(*rhs);
    struct tree *rhs_copy = lrx_declare_tree((*rhs)->datatype, (*rhs)->degree); /* Ir_Decl */
    lrx_copy_construct_tree(&rhs_copy, rhs, max_nodes_rhs, &max_nodes_rhs, &trash);

    *pos = rhs_copy;
}

struct tree **lrx_get_root(struct tree **t){
    if ((*t)->parent == NULL) {
            return t;
    }
    return lrx_get_root(&(*t)->parent);
}

struct tree **lrx_get_parent(struct tree **t) {
    assert(t && *t);
    return &((*t)->parent);
}

int _lrx_check_equals(struct tree *lhs, struct tree *rhs ) {
    if (lhs == NULL && rhs == NULL)
        return true;
    if (lhs == NULL || rhs == NULL)
        return false;

    int equals = 1;
    if (lhs->datatype != rhs->datatype || lhs->degree != rhs->degree) return !equals;

    switch (lhs->datatype) {
            case _INT_:
                    equals = lhs->root.int_root == rhs->root.int_root;
                    break;
            case _BOOL_:
                    equals = lhs->root.bool_root == rhs->root.bool_root;
                    break;
            case _FLOAT_:
                    equals = lhs->root.float_root == rhs->root.float_root;
                    break;
            case _CHAR_:
        case _STRING_:
                    equals = lhs->root.char_root == rhs->root.char_root;
                    break;
    }

    if (!equals) return equals;

    int i;
    for (i = 0; i < lhs->degree; i++) {
            equals = _lrx_check_equals(lhs->children[i], rhs->children[i]);
            if (!equals) return equals;
    }

    return equals;
}

bool lrx_compare_tree(struct tree *lhs, struct tree *rhs, Comparator comparison) {
    int lhs_nodes = _lrx_count_nodes( lhs );
    int rhs_nodes = _lrx_count_nodes( rhs );
    int value;

    LrxLog("%d vs %d\n", lhs_nodes, rhs_nodes);
    LrxLog("Comparator = %d\n", comparison);
    #ifdef LRXDEBUG
    lrx_print_tree(lhs);
```

```
    printf("\n");
    lrx_print_tree(rhs);
    printf("\n");
    #endif

    switch (comparison) {
            case _LT_:
                    value = lhs_nodes < rhs_nodes;
                    break;
            case _LTE_:
                    value = lhs_nodes <= rhs_nodes;
                    break;
            case _GT_:
                    value = lhs_nodes > rhs_nodes;
                    break;
            case _GTE_:
                    value = lhs_nodes >= rhs_nodes;
                    break;
            case _EQ_:
                    value = _lrx_check_equals( lhs, rhs );
                    break;
            case _NEQ_:
                    value = !_lrx_check_equals( lhs, rhs );
             break;
    }

    return value;
}

int lrx_get_degree(struct tree **t) {
    return (*t)->degree;
}
```

Makefile

```
#
# Authors:
# Chris D'Angelo
# Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
# which provided background knowledge.
#

OBJS = ast.cmo symtab.cmo check.cmo intermediate.cmo output.cmo parser.cmo scanner.cmo lorax.cmo

lorax : $(OBJS)
	ocamlc -o lorax -g unix.cma $(OBJS)

.PHONY : test
test : lorax testall.sh
	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

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

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

.PHONY : clean
clean :
	rm -rf lorax parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff *~ *_lrxtmp.c a.out.dSYM examples/*lrxtmp.c

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
symtab.cmo: ast.cmo
```

```
symtab.cmx: ast.cmx
check.cmo: symtab.cmo
check.cmx: symtab.cmx
intermediate.cmo: check.cmo
intermediate.cmx: check.cmx
output.cmo: intermediate.cmo
output.cmx: intermediate.cmx
lorax.cmo: scanner.cmo parser.cmi ast.cmo symtab.cmo check.cmo intermediate.cmo output.cmo
lorax.cmx: scanner.cmx parser.cmx ast.cmx symtab.cmx check.cmx intermediate.cmx output.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```

output.ml

```
(*
 * Authors:
 * Kira Whitehouse
 * Chris D'Angelo
 * Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
 * which provided background knowledge.
 *)

open Ast
open Check
open Intermediate

let c_of_var_type = function
      Lrx_Atom(Lrx_Int) -> "int"
    | Lrx_Atom(Lrx_Float) -> "float"
    | Lrx_Atom(Lrx_Bool) -> "bool"
    | Lrx_Atom(Lrx_Char) -> "char"
    | Lrx_Tree(t) -> "tree *"

let c_of_func_decl_var_type = function
    Lrx_Atom(Lrx_Int) -> "int"
  | Lrx_Atom(Lrx_Float) -> "float"
  | Lrx_Atom(Lrx_Bool) -> "bool"
  | Lrx_Atom(Lrx_Char) -> "char"
  | Lrx_Tree(t) -> "tree **"

let c_of_var_def (v:ir_var_decl) =
    let (_ ,t, _,u) = v in match t with
      Lrx_Atom(Lrx_Int) -> "0"
    | Lrx_Atom(Lrx_Float) -> "0.0"
    | Lrx_Atom(Lrx_Bool) -> "false"
    | Lrx_Atom(Lrx_Char) -> "\'\\0\'"
    | Lrx_Tree(l) ->
  if u = 1 then "NULL" else
  "lrx_declare_tree(_" ^ String.uppercase (string_of_atom_type l.datatype) ^ "_, " ^ string_of_expr l.degree ^ ")"

let c_of_var_decl (v:ir_var_decl) =
    let (n,t,s,u) = v in
  let pointer_galaga = if u = 1 then "*" else "" in
     c_of_var_type t ^ pointer_galaga ^ " " ^ n ^ "_" ^ string_of_int s

let c_of_null_decl (v:ir_var_decl) =
  let (n,_,s,_) = v in
  "void * " ^ n ^ "_" ^ string_of_int s

let c_of_ir_var_decl (v:scope_var_decl) =
  let (n,t,s) = v in
   c_of_var_type t ^ " " ^ n ^ "_" ^ string_of_int s

let rec c_of_var_umbilical_decl (v:ir_var_decl) =
    let (n,t,s,u) = v in
     c_of_var_type t ^ "*" ^ n ^ "_" ^ string_of_int s

let c_of_ptr_decl (v:ir_var_decl) =
    let (n,t,s,u) = v in
```

```
    c_of_var_type t ^ " *" ^ n ^ "_" ^ string_of_int s

let c_of_ir_var_decl_list = function
    [] -> ""
  | vars -> (String.concat (";\n") (List.map c_of_ir_var_decl vars)) ^ ";\n\n"

let c_of_var_decl_list = function
      [] -> ""
    | vars -> (String.concat (";\n") (List.map c_of_var_decl vars)) ^ ";\n\n"

let c_of_func_actual (v:ir_var_decl) =
    let(n,t,s,u) = v in
   let prefix =
   (match t with
     Lrx_Tree(_) -> if (u = 3 || u = 1) then "" else "&"
    | _ -> if u = 1 then "*" else "") in
    prefix ^ n ^ "_" ^ string_of_int s

let c_of_func_decl_args = function
      [] -> ""
    | args -> String.concat (", ") (List.map c_of_func_actual args)

let c_of_ir_var_decl (v:scope_var_decl) =
   let (n,t,s) = v in
   match t with
       Lrx_Tree(_)->  c_of_func_decl_var_type t ^ " " ^ n ^ "_" ^ string_of_int s
     | _ -> c_of_func_decl_var_type t ^ " " ^ n ^ "_" ^ string_of_int s


let c_of_func_def_formals = function
      [] -> ""
    | args -> String.concat (", ") (List.map c_of_ir_var_decl args)

let c_of_var_arg (v:ir_var_decl) =
    let (n,t,s, u) = v in
   let prefix =
   (match t with
       Lrx_Tree(_)-> if u = 1 then "" else if u = 3 then "" else "&"
     | Lrx_Atom(_) -> if u = 1 then "*" else "") in
    prefix ^ n ^ "_" ^ string_of_int s

let c_of_tree_null (v:ir_var_decl) =
   let (n, t, s, u) = v in
   let prefix = if u = 1 then "*" else "" in
   prefix ^ n ^ "_" ^ string_of_int s

let c_of_var_name (v:ir_var_decl) =
   let (n,_,s, _) = v in
   n ^ "_" ^ string_of_int s

let c_of_print_var (arg :ir_var_decl) =
    let (n ,t, s, u) = arg in
    (match t with
              Lrx_Atom(Lrx_Int) -> "fprintf(stdout, \"%d\", " ^ c_of_var_arg arg ^ ")"
       | Lrx_Atom(Lrx_Float) -> "fprintf(stdout, \"%f\", " ^ c_of_var_arg arg ^ ")"
       | Lrx_Atom(Lrx_Char) -> "fprintf(stdout, \"%c\", " ^ c_of_var_arg arg ^ ")"
       | Lrx_Atom(Lrx_Bool) -> "lrx_print_bool(" ^ c_of_var_arg arg ^ ")"
       | Lrx_Tree(l) ->
       let prefix = if u = 1 then "*" else if u = 3 then "*" else "" in
       let name = n ^ "_" ^ string_of_int s in
       "lrx_print_tree(" ^ prefix ^ name ^ ")")

let c_of_print_call = function
      [] -> ""
    | print_args -> String.concat (";\n") (List.map c_of_print_var print_args)

let unescape_char c =
    match c with
       '\n' -> "\\n"
     | '\t' -> "\\t"
     | '\\' -> "\\\\"
     | _ -> String.make 1 c

```

```
let c_of_tree_comparator = function
      Greater -> "_GT_"
    | Less -> "_LT_"
    | Leq -> "_LTE_"
    | Geq -> "_GTE_"
    | Equal -> "_EQ_"
    | Neq -> "_NEQ_"
    | _ -> raise (Failure "Not a valid tree comparator")


let rec c_of_expr = function
      Ir_Int_Literal(v, i) -> c_of_var_name v ^ " = " ^ string_of_int i
    | Ir_Float_Literal(v, f) ->  c_of_var_name v ^ " = " ^ string_of_float f
    | Ir_Char_Literal(v, c) -> c_of_var_name v ^ " = " ^ "\'" ^ unescape_char c ^ "\'"
    | Ir_Bool_Literal(v, b) -> c_of_var_name v ^ " = " ^ string_of_bool b
     | Ir_Null_Literal(n) -> c_of_var_name n ^ " = NULL; /* Ir_Null_Literal */"
    | Ir_Unop(v1, op, v2) ->
      (match op with
             (Neg | Not) -> c_of_var_name v1 ^ " = " ^ string_of_unop op ^ c_of_var_name v2
        | At -> let (_,t,_, u) = v1 in
         (match t with
             Lrx_Atom(Lrx_Int) -> c_of_var_name v1 ^ " = lrx_access_data_at_int(" ^ c_of_var_arg v2 ^ ")"
           | Lrx_Atom(Lrx_Float) -> c_of_var_name v1 ^ " = lrx_access_data_at_float(" ^ c_of_var_arg v2 ^ ")"
           | Lrx_Atom(Lrx_Char) -> c_of_var_name v1 ^ " = lrx_access_data_at_char(" ^ c_of_var_arg v2 ^ ")"
           | Lrx_Atom(Lrx_Bool) -> c_of_var_name v1 ^ " = lrx_access_data_at_bool(" ^ c_of_var_arg v2 ^ ")"
           | _ -> raise (Failure "Return type of access data member cannot be tree."))
        | Pop -> raise (Failure "TEMPORARY: Pop not implemented."))
     | Ir_Binop(v1, op, v2, v3) ->
      let (_,t1,_, u1) = v2 in
      let (_,t2,_, u2) = v3 in
      (match (t1, t2) with
          (Lrx_Tree(_), Lrx_Tree(_)) ->
          if u1 = 2 || u2 = 2 then
            (match op with
                Equal -> c_of_var_name v1 ^ " = (" ^ c_of_tree_null v2 ^ " == " ^ c_of_tree_null v3 ^ ")"
              | Neq -> c_of_var_name v1 ^ " = (" ^ c_of_tree_null v2 ^ " != " ^ c_of_tree_null v3 ^ ")"
              | _ -> raise (Failure "Impossible null/tree binop null/tree") )
          else
            (match op with
                (Less | Leq | Greater | Geq | Equal | Neq ) ->
                c_of_var_name v1 ^ " = lrx_compare_tree(" ^ c_of_tree_null v2 ^ ", " ^ c_of_tree_null v3 ^ ", " ^ c_of_tree_comparator op ^ ")"
              | Add -> "lrx_add_trees(" ^ c_of_var_arg v1 ^ ", " ^ c_of_var_arg v2 ^ ", " ^ c_of_var_arg v3 ^ ")"
              | _ -> raise (Failure "Operation not available between two tree types."))
        | (Lrx_Atom(_), Lrx_Atom(_)) ->
          (match op with
              Mod -> c_of_var_name v1 ^ " = " ^ c_of_var_arg v2 ^ " % " ^ c_of_var_arg v3
            | _ -> c_of_var_name v1 ^ " = " ^ c_of_var_arg v2 ^ " " ^ string_of_binop op ^ " " ^ c_of_var_arg v3)
        | (Lrx_Tree(_), Lrx_Atom(_)) -> c_of_var_name v1 ^ " = lrx_access_child(" ^ c_of_var_arg v2 ^ ", " ^ c_of_var_name v3 ^ ")"
        | _ -> raise (Failure "Invalid expression. There is no atom operator tree expression."))
    | Ir_Id(v1, v2) -> c_of_var_name v1 ^ " = " ^ c_of_var_name v2
    | Ir_Assign(v1, v2) ->
      let (_,t1,_,u1) = v1 in
       let (_,t2,_,u2) = v2 in
      (match (t1, t2) with
              (Lrx_Tree(_), Lrx_Tree(_)) -> "lrx_assign_tree_direct(" ^ c_of_var_arg v1 ^ ", " ^ c_of_var_arg v2 ^ ")"
            | (Lrx_Atom(_), Lrx_Atom(_)) -> c_of_var_arg v1 ^ " = " ^ c_of_var_arg v2
            | _ -> raise (Failure "Tree cannot be assigned to atom type."))
    | Ir_Tree_Literal(v, root, children) -> "lrx_define_tree(" ^ c_of_var_name v ^ ", " ^
           c_of_var_name root ^ ", " ^ c_of_var_name children ^ ")"
    | Ir_Call(v1, v2, vl) ->
    let func_name = fst_of_four v2 in
```

```
    (match func_name with
        "print" -> (c_of_print_call vl)
      | "degree" -> c_of_var_name v1 ^ " = " ^ "lrx_get_degree(" ^ c_of_func_decl_args vl ^ ")"
      | "parent" -> c_of_var_name v1 ^ " = lrx_get_parent(" ^ c_of_var_arg (List.hd vl) ^ ")"
      | "root" -> c_of_var_name v1 ^ " = lrx_get_root(" ^ c_of_var_arg (List.hd vl) ^ ")"
      | _ -> c_of_var_name v1 ^ " = " ^ fst_of_four v2 ^ "( " ^ c_of_func_decl_args vl ^ " )")
  | Ir_Noexpr -> ""

let c_of_ref (r:ir_var_decl) =
  let (n2,_, s2, u2) = r in
  let prefix = if u2 = 1 then "" else "&" in
   prefix ^ n2 ^ "_" ^ string_of_int s2

let rec c_of_leaf (n:string) (d:int) =
    if d < 0 then "" else
    n ^ "[" ^ string_of_int d ^ "] = NULL; /* c_of_leaf */\n" ^ c_of_leaf n (d - 1)


let c_of_stmt (v:ir_stmt) (cleanup:string) =
    match v with
       Ir_Decl(d) -> c_of_var_decl d ^ " = " ^ c_of_var_def d ^ "; /* Ir_Decl */"
     | Ir_Decl_Umbilical(d) -> c_of_var_umbilical_decl d ^ " = NULL ; /* Ir_Decl_Umbilical */"
     | Ir_Null_Decl(d) -> c_of_null_decl d ^ " = NULL; /* Ir_Null_Decl */"
   | Ir_Leaf(p, d) -> c_of_var_decl p ^ "[" ^ string_of_int d ^ "]; /* Ir_Leaf */\n" ^
       c_of_leaf (c_of_var_name p) (d - 1)
     | Ir_Child_Array(d, s) -> c_of_var_decl d ^ "[" ^ string_of_int s ^ "]; /* Ir_Child_Array */\n" ^
        "/* Filling with NULL preemptively */\n" ^ c_of_leaf (c_of_var_name d) (s - 1)
     | Ir_Internal(a, c, t) -> c_of_var_name a ^ "[" ^ string_of_int c ^ "] = " ^ c_of_var_name t ^ "; /* Ir_Internal */"
     | Ir_Ptr(p, r) -> c_of_var_name p ^ " = " ^ c_of_ref r ^ "; /* Ir_Ptr */"
   | Ir_At_Ptr(p) -> c_of_ptr_decl p ^ " = NULL; /* Ir_At_Ptr */"
   | Ir_Ret(v, s, e) -> "goto " ^ s ^ ";\n" ^ e ^ ":\nreturn " ^ c_of_var_arg v ^ ";\n" ^
       s ^ ":\n" ^ cleanup ^ "goto " ^ e ^ ";\n"
     | Ir_Expr(e) -> c_of_expr e ^ ";\n"
     | Ir_If(v, s) -> "if(" ^ c_of_var_name v ^ ") goto " ^ s ^ "" ^ ";"
     | Ir_Jmp(s) -> "goto " ^ s ^ ";"
     | Ir_Label(s) -> s ^ ":"
     | _ -> raise (Failure ("Ir_Tree_Destroy should be impossible here"))

let c_of_destroy (v:ir_stmt) =
  match v with
     Ir_Tree_Destroy(d) -> "lrx_destroy_tree(" ^ c_of_var_name d ^ ");"
   | Ir_Tree_Add_Destroy(d) -> "lrx_destroy_add_tree(" ^ c_of_var_name d ^ ");"
   | _ -> raise (Failure ("only Ir_Tree_Destroy should be possible here"))

let c_of_destroys destroys =
  String.concat ("\n") (List.map c_of_destroy destroys) ^ "\n\n"

let rec c_of_stmt_list stmts cleanup =
  match stmts with
      [] -> []
     | head :: tail ->  c_of_stmt head cleanup :: c_of_stmt_list tail cleanup

let c_of_func (f: ir_func) =
    let (t, n, sl) = f.ir_header in
  let cleanup = c_of_destroys f.ir_destroys in
    c_of_var_type t ^ " " ^ n ^ "(" ^ c_of_func_def_formals sl ^ ")\n{\n" ^
    String.concat "\n" (c_of_stmt_list f.ir_vdecls cleanup) ^ "\n\n" ^ String.concat "\n" (c_of_stmt_list f.ir_stmts cleanup) ^ "}"

let c_of_func_list = function
     [] -> ""
   | funcs -> String.concat ("\n") (List.map c_of_func funcs)

let c_of_func_decl_formals = function
    [] -> ""
   | formals -> String.concat (", ") (List.map c_of_func_decl_var_type formals)

let c_of_func_decl (f:ir_fheader) =
```

```
    (c_of_var_type f.ir_ret_type) ^ " " ^ f.ir_name ^
    "(" ^ (c_of_func_decl_formals f.ir_formals) ^ ");"

let c_of_func_decl_list = function
      [] -> ""
    | fdecls -> String.concat ("\n") (List.map c_of_func_decl fdecls) ^ "\n\n"

let c_of_inter_pgrm (p:ir_program) =
    "#include \"lrxlib.h\"\n" ^
    c_of_ir_var_decl_list p.ir_globals ^
  c_of_func_decl_list p.ir_headers ^
  c_of_func_list p.ir_bodies
```

parser.mly

```
/*
 * Authors:
 * Chris D'Angelo
 * Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
 * which provided background knowledge.
 */

%{ open Ast

let scope_id = ref 1

let inc_block_id (u:unit) =
    let x = scope_id.contents in
    scope_id := x + 1; x

%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN POP
%token AND OR NOT
%token EQ NEQ LT LEQ GT GEQ
%token LBRACKET RBRACKET
%token CHAR BOOL INT FLOAT STRING TREE
%token BREAK CONTINUE AT CHILD
%token TRUE FALSE NULL
%token RETURN IF ELSE FOR WHILE
%token <int> INT_LITERAL
%token <bool> BOOL_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <char> CHAR_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left NEG NOT
%left AT CHILD POP

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program global_vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   var_type ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
```

```
     { { fname = $2;
         ret_type = $1;
           formals = $4;
           fblock = {locals = List.rev $7; statements = List.rev $8; block_id = inc_block_id ()} } }

block:
    LBRACE stmt_list RBRACE { {locals = []; statements = List.rev $2; block_id = inc_block_id ()} }


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

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

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

global_vdecl:
  vdecl SEMI { $1 }

vdecl:
    var_type ID { ($2, $1) }
  | TREE LT INT GT ID LPAREN expr RPAREN { ($5, Lrx_Tree({datatype = Lrx_Int; degree = $7})) }
  | TREE LT CHAR GT ID LPAREN expr RPAREN { ($5, Lrx_Tree({datatype = Lrx_Char; degree = $7})) }
  | TREE LT BOOL GT ID LPAREN expr RPAREN { ($5, Lrx_Tree({datatype = Lrx_Bool; degree = $7})) }
  | TREE LT FLOAT GT ID LPAREN expr RPAREN { ($5, Lrx_Tree({datatype = Lrx_Float; degree = $7})) }
  | STRING ID { ($2, Lrx_Tree({datatype = Lrx_Char; degree = Int_Literal(1)})) }

var_type:
    INT    { Lrx_Atom(Lrx_Int) }
  | CHAR   { Lrx_Atom(Lrx_Char) }
  | BOOL   { Lrx_Atom(Lrx_Bool) }
  | FLOAT  { Lrx_Atom(Lrx_Float) }

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

stmt:
    block { CodeBlock($1) }
  | expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | IF LPAREN expr RPAREN block %prec NOELSE { If($3, $5, {locals = []; statements = []; block_id = inc_block_id ()}) }
  | IF LPAREN expr RPAREN block ELSE block { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN block { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN block { While($3, $5) }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }

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

expr:
    literal                        { $1 }
  | tree                           { $1 }
  | ID                             { Id($1) }
  | expr PLUS   expr               { Binop($1, Add, $3) }
  | expr MINUS  expr               { Binop($1, Sub, $3) }
  | expr TIMES  expr               { Binop($1, Mult, $3) }
  | expr DIVIDE expr               { Binop($1, Div, $3) }
  | expr MOD    expr               { Binop($1, Mod, $3) }
```

```
  | expr EQ     expr             { Binop($1, Equal, $3) }
  | expr NEQ    expr             { Binop($1, Neq, $3) }
  | expr LT     expr             { Binop($1, Less, $3) }
  | expr LEQ    expr             { Binop($1, Leq, $3) }
  | expr GT     expr             { Binop($1, Greater, $3) }
  | expr GEQ    expr             { Binop($1, Geq, $3) }
  | expr AND    expr             { Binop($1, And, $3) }
  | expr OR     expr             { Binop($1, Or, $3) }
  | MINUS expr %prec NEG         { Unop($2, Neg) }
  | NOT expr                     { Unop($2, Not) }
  | expr CHILD expr              { Binop($1, Child, $3) }
  | expr POP                     { Unop($1, Pop) }
  | expr AT                      { Unop($1, At) }
  | expr ASSIGN expr             { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

literal:
    INT_LITERAL    { Int_Literal($1) }
  | FLOAT_LITERAL  { Float_Literal($1) }
  | STRING_LITERAL { String_Literal($1) }
  | CHAR_LITERAL   { Char_Literal($1) }
  | BOOL_LITERAL   { Bool_Literal($1) }
  | NULL           { Null_Literal }

node_expr:
      literal             { $1 }
  | ID                  { Id($1) }
   | LPAREN expr RPAREN { $2 }

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

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

tree:
    node_expr LBRACKET nodes RBRACKET { Tree($1, $3) }

nodes:
    /* nothing */    { [] }
  | expr             { [$1] }
  | expr COMMA nodes { $1 :: $3 } /* nodes are kept in order */
```

README.md

```
Lorax Programming Language
==========================
Compiler for Lorax, a language focused on making tree operations simple. Authors: Doug Beinstock (dmb2168), Chris D'Angelo (cd2665), Zhaarn Maheswaran (zsm2103), Tim Paine (tkp2108), Kira Whitehouse (kbw2116)

Requirements
===========
[OCaml](http://ocaml.org/), [Unix](http://www.ubuntu.com/), [gcc](http://gcc.gnu.org/)
Quick Start
===============
```
$ cat hello.lrx
$ int main() { print("hello, world\n"); }
$ make
$ ./lorax -b hello.lrx
$ ./a.out
$ hello, world
$
```
Compiler Flags
==============
* `-a` Print the Abstract Syntax Tree digested source code.
* `-t` Print an alphabetical list of the symbol table created from source code.
* `-s` Run Semantic Analysis on source code.
```

```
* `-c` Compile source code to target c language. Default to stdout, or written to filename present in third command line argument.
* `-b` Compile source code to binary ouput. By default to a.out, or the filename present in third command line argument.

Running Tests
=============
```
$ make
$ ./testall.sh
$
```
Examples
========
If you're interested in some real world examples of the lorax language check out the `examples`
directory.

User Guides
===========
[Language Reference Manual](http://bit.ly/theloraxmanual), [Lorax Language Presentation](http://bit.ly/theloraxpresentation)
```

scanner.ml

```
(*
 * Authors:
 * Chris D'Angelo
 *)

{
    open Parser
    exception LexError of string

    let verify_escape s =
            if String.length s = 1 then (String.get s 0)
            else
            match s with
               "\\n" -> '\n'
             | "\\t" -> '\t'
             | "\\\\" -> '\\'
             | c -> raise (Failure("unsupported character " ^ c))
}

(* Regular Definitions *)

let digit = ['0'-'9']
let decimal = ((digit+ '.' digit*) | ('.' digit+))

(* Regular Rules *)

(*
 * built-in functions handled as keywords in semantic checking
 * print, root, degree
 *)

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"        { block_comment lexbuf }
| "//"            { line_comment lexbuf }
| '('         { LPAREN }
| ')'         { RPAREN }
| '{'         { LBRACE }
| '}'         { RBRACE }
| ']'         { RBRACKET }
| '['         { LBRACKET }
| ';'         { SEMI }
| ','         { COMMA }
| '+'         { PLUS }
| '-'         { MINUS }
| "--"        { POP }
| '*'         { TIMES }
| "mod"       { MOD }
```

```
| '/'         { DIVIDE }
| '='         { ASSIGN }
| "=="        { EQ }
| "!="        { NEQ }
| '<'         { LT }
| "<="        { LEQ }
| ">"         { GT }
| ">="        { GEQ }
| "if"        { IF }
| "else"      { ELSE }
| "for"       { FOR }
| "while"     { WHILE }
| "return"    { RETURN }
| "int"       { INT }
| "float"     { FLOAT }
| "string"    { STRING }
| "bool"      { BOOL }
| "tree"      { TREE }
| "break"     { BREAK }
| "continue"  { CONTINUE }
| "null"      { NULL }
| "char"      { CHAR }
| "!"                { NOT }
| "&&"               { AND }
| "||"               { OR }
| '@'            { AT }
| '%'                { CHILD }
| digit+ as lxm                              { INT_LITERAL(int_of_string lxm) }
| decimal as lxm                             { FLOAT_LITERAL(float_of_string lxm) }
| '\"' ([^'\"']* as lxm) '\"'    { STRING_LITERAL(lxm) }
| '\'' ([^'\'']* as lxm ) '\''     { CHAR_LITERAL((verify_escape lxm)) }
| ("true" | "false") as lxm                  { BOOL_LITERAL(bool_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and block_comment = parse
  "*/" { token lexbuf }
| eof  { raise (LexError("unterminated block_comment!")) }
| _    { block_comment lexbuf }

and line_comment = parse
| ['\n' '\r'] { token lexbuf }
| _           { line_comment lexbuf }
```

symtab.ml

```
(*
 * Authors:
 * Tim Paine
 * Chris D'Angelo
 * Special thanks to Dara Hazeghi's strlang and Stephen Edward's MicroC
 * which provided background knowledge.
 *)

open Ast

(*
 * SymMap contains string : Ast.decl pairs representing
 * identifiername_scopenumber : decl
 *)
module SymMap = Map.Make(String)

let scope_parents = Array.create 1000 0


(* string_of_vdecl from ast.ml *)
let string_of_decl = function
      SymTab_VarDecl(n, t, id)      -> string_of_vdecl (n, t)
    | SymTab_FuncDecl(n, t, f, id) ->
      (string_of_var_type t) ^ " " ^
       n ^ "(" ^
       String.concat ", " (List.map string_of_var_type f) ^ ")"
```

```
let string_of_symtab env =
    let symlist = SymMap.fold
            (fun s t prefix -> (string_of_decl t) :: prefix) (fst env) [] in
    let sorted = List.sort Pervasives.compare symlist in
    String.concat "\n" sorted

let rec symtab_get_id (name:string) env =
    let(table, scope) = env in
    let to_find = name ^ "_" ^ (string_of_int scope) in
    if SymMap.mem to_find table then scope
    else
            if scope = 0 then raise (Failure("symbol " ^ name ^ " not declared in current scope"))
            else symtab_get_id name (table, scope_parents.(scope))
(*
 * Look for the symbol in the given environment and scope
 * then recursively check in all ancestor scopes
 *)
let rec symtab_find (name:string) env =
    let(table, scope) = env in
    let to_find = name ^ "_" ^ (string_of_int scope) in
    if SymMap.mem to_find table then SymMap.find to_find table
    else
            if scope = 0 then raise (Failure("symbol " ^ name ^ " not declared in current scope"))
            else symtab_find name (table, scope_parents.(scope))

let rec symtab_add_decl (name:string) (decl:decl) env =
    let (table, scope) = env in (* get current scope and environment *)
    let to_find = name ^ "_" ^ (string_of_int scope) in
    if SymMap.mem to_find table then raise(Failure("symbol " ^ name ^ " declared twice in same scope"))
    else ((SymMap.add to_find decl table), scope)

(*
 * recursively add list of variables to the symbol table along with the scope of
 * the block in which they were declared
 *)
let rec symtab_add_vars (vars:var list) env =
    match vars with
      [] -> env
    | (vname, vtype) :: tail -> let env = symtab_add_decl vname (SymTab_VarDecl(vname, vtype, snd env)) env in (* name, type, scope *)
            symtab_add_vars tail env

(* add declarations inside statements to the symbol table *)
let rec symtab_add_stmts (stmts:stmt list) env =
    match stmts with
      [] -> env (* block contains no statements *)
    | head :: tail -> let env = (match head with
            CodeBlock(s) -> symtab_add_block s env (* statement is an arbitrary block *)
            | For(e1, e2, e3, s) -> symtab_add_block s env (* add the for's block to the record *)
            | While(e, s) -> symtab_add_block s env (* same deal as for *)
            | If(e, s1, s2) -> let env = symtab_add_block s1 env in symtab_add_block s2 env (* add both of if's blocks separately *)
        | _ -> env) in symtab_add_stmts tail env (* return, continue, break, etc *)

and symtab_add_block (b:block) env =
    let (table, scope) = env in
    let env = symtab_add_vars b.locals (table, b.block_id) in
    let env = symtab_add_stmts b.statements env in
     scope_parents.(b.block_id) <- scope; (* parent is block_id - 1 *)
     ((fst env), scope) (* return what we've made *)

and symtab_add_func (f:func) env =
    let scope = snd env in
    let args = List.map snd f.formals in (* gets name of every formal *)
    let env = symtab_add_decl f.fname (SymTab_FuncDecl(f.fname, f.ret_type, args, scope)) env in (* add current function to table *)
    let env = symtab_add_vars f.formals ((fst env), f.fblock.block_id) in (* add vars to the next scope in. scope_id is ahead by one *)
```

```
    symtab_add_block f.fblock ((fst env), scope) (* add body to symtable given current environment and scope *)

(* add list of functions to the symbol table *)
and symtab_add_funcs (funcs:func list) env =
    match funcs with
       [] -> env
      | head :: tail -> let env = symtab_add_func head env in
       symtab_add_funcs tail env

let add_builtins env =
     let env = symtab_add_decl "print" (SymTab_FuncDecl("print", Lrx_Atom(Lrx_Int), [], 0)) env in
     let env = symtab_add_decl "root" (SymTab_FuncDecl("root", Lrx_Atom(Lrx_Int), [], 0)) env in
     let env = symtab_add_decl "parent" (SymTab_FuncDecl("parent", Lrx_Atom(Lrx_Int), [], 0)) env in
     symtab_add_decl "degree" (SymTab_FuncDecl("degree", Lrx_Atom(Lrx_Int), [], 0)) env

(*
 * env: Ast.decl Symtab.SymMap.t * int = (<abstr>, 0)
 * the "int" is used to passed from function to function
 * to remember the current scope. it is not used outside this
 * file
 *)
let symtab_of_program (p:Ast.program) =
    let env = add_builtins (SymMap.empty, 0) in
    let env = symtab_add_vars (fst p) env in
   symtab_add_funcs (snd p) env
```

testall.sh

```
#!/bin/sh

#
# Authors:
# Chris D'Angelo
# Zhaarn Maheswaran
# Special thanks Stephen Edward's MicroC which provided background knowledge.
#

lorax="./lorax"
binaryoutput="./a.out"

# Set time limit for all operations
ulimit -t 30

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

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.lrx files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
   echo "FAILED"
   error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
```

```
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
        if [[ $5 != *fail* ]]; then
          SignalError "$1 failed on $*"
          return 1
        fi
    }
}

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.a.out" &&
    Run "$lorax" "-a" $1 ">" ${basename}.a.out &&
    Compare ${basename}.a.out ${reffile}.out ${basename}.a.diff

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

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.s.out" &&
    Run "$lorax" "-s" $1 ">" ${basename}.s.out &&
    Compare ${basename}.s.out ${reffile}.out ${basename}.s.diff

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

```
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

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

    echo -n "$basename..."

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

    generatedfiles=""

    # old from microc - interpreter
    # generatedfiles="$generatedfiles ${basename}.i.out" &&
    # Run "$lorax" "-i" "<" $1 ">" ${basename}.i.out &&
    # Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

    generatedfiles="$generatedfiles ${basename}.c.out" &&
    Run "$lorax" "-c" $1 ">" ${basename}.c.out &&
    Compare ${basename}.c.out ${reffile}.out ${basename}.c.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}
CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.lrx//'`
    reffile=`echo $1 | sed 's/.lrx$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    # old from microc - interpreter
    # generatedfiles="$generatedfiles ${basename}.i.out" &&
    # Run "$lorax" "-i" "<" $1 ">" ${basename}.i.out &&
    # Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

    generatedfiles="$generatedfiles ${basename}.c.out" &&
    {
        Run "$lorax" "-b" $1 "2>" ${basename}.c.out ||
        Run "$binaryoutput" ">" ${basename}.b.out
    } &&
    Compare ${basename}.c.out ${reffile}.out ${basename}.c.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
```

```
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

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

    echo -n "$basename..."

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

    generatedfiles=""
    tmpfiles=""

    # old from microc - interpreter
    # generatedfiles="$generatedfiles ${basename}.i.out" &&
    # Run "$lorax" "-i" "<" $1 ">" ${basename}.i.out &&
    # Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

    generatedfiles="$generatedfiles ${basename}.f.out" &&
    tmpfiles="$tmpfiles tests/${basename}.lrx_lrxtmp.c a.out" &&
    Run "$lorax" "-b" $1 &&
    Run "$binaryoutput" ">" ${basename}.f.out &&
    Compare ${basename}.f.out ${reffile}.out ${basename}.f.diff

    rm -f $tmpfiles

    # Report the status and clean up the generated files

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

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.lrx"
fi

for file in $files
do
```

```
    case $file in
    *test-parser*)
        CheckParser $file 2>> $globallog
        ;;
    *test-sa*)
        CheckSemanticAnalysis $file 2>> $globallog
        ;;
    *test-full*)
        TestRunningProgram $file 2>> $globallog
        ;;
    *test-fail*)
        CheckFail $file 2>> $globallog
        ;;
   *test-*)
       Check $file 2>> $globallog
       ;;
   *)
       echo "unknown file type $file"
       globalerror=1
       ;;
    esac
done

exit $globalerror
```

Examples

array.lrx

```
/*
 * Lorax Array Example
 * Author: Zhaarn Maheswaran
 */

/* Inserts an element into the array */
int insert_array(tree <int>t(1), int index, int val) {
    tree <int> a(1);
    int i;
    a = t;
    if (a == null) {
            return -1;
    }
    for (i = 0; i < index; i=i+1) {
            a = a%0;
            if(a == null){
                    return -1; //invalid access
            }
    }
    a@ = val;
    return 0;
}

/* Accesses an element in the array */
int access_array(tree<int>t(1), int index) {
    tree <int> a(1);
    int i;
    a = t;
    if (a == null) {
            print("Invalid access");
            return -1;
    }
    for (i = 0; i < index; i = i+1) {
            a = a%0;
            if(a == null){
                    print("Invalid access");
                    return -1;
            }
    }
    return a@;
}

/* Gets the size of the array */
```

```
int size_array(tree <int> t(1)) {
    int i;
    tree <int> a (1);
    a = t;
    i = 0;
    while( a != null) {
            a = a%0;
            i = i + 1;
    }
    return i;
}

int main() {
    tree <int>t(1);
    int size;
    int i;
    int p;
    t = 0[0[0[0[0[0]]]]];
    /* size = 6; */
    /* init_array(t, size); */
    for (i = 0; i < size_array(t); i = i + 1) {
            insert_array(t, i, i);
            p = access_array(t, i);
            print(p);

    }
    print("\n");
    print(t);
}
```

dfs.lrx

```
/*
 * Lorax Hello World
 * Author: Chris D'Angelo
 */

bool dfs(tree <int>t(2), int val) {
    int child;
    bool match;
    match = false;

    if (t == null) {
            return false;
    }

    if (t@ == val) {
            return true;
    }

    for (child = 0; child < degree(t); child = child + 1) {
            if (t%child != null) {
                    if(t%child@ == val){
                            return true;
                    }
                    else{
                            match = dfs(t%child, val);
                    }
            }
    }

    return match;
}

int main() {
    tree <int>t(2);
    t = 1[2, 3[4, 5]];
    if (dfs(t, 3)) {
            print("found it\n");
    } else {
            print("its not there\n");
    }
}
```

gcd.lrx

```
/*
 * Lorax GCD
 * Author: Chris D'Angelo
 */

int gcd(int x, int y){
    int check;
    while (x != y) {
        if (x < y) {
            check = y - x;
            if (check > x) {
                x = check;
            } else {
                y = check;
            }
        } else {
            check = x - y;
            if (check > y) {
                y = check;
            } else {
                x = check;
            }
        }
    }
    return x;
}

int main() {
    print(gcd(25, 15));
}
```

helloworld.lrx

```
/*
 * Lorax Hello World
 * Author: Chris D'Angelo
 */

int main() {
     print("hello, world\n");
}
```

huffman.lrx

```
/*
 * Lorax Huffman Example
 * Prints groupmembers' names according to a predetermined huffman encoding
 * Author: Zhaarn Maheswaran
 */

int main () {
    tree <char> codingtree (2);
    codingtree = '$'['$'['$'['$'['c', '$'['t','m']],'r'],
            '$'['$'['$'['o','u'],'$'['k','n']],'a']],
            '$'['$'['$'['z','s'],'i'],'$'['$'['g','d'], 'h']]];
    decode("1000", codingtree);
    decode("111", codingtree);
    decode("011", codingtree);
    decode("011", codingtree);
    decode("001", codingtree);
    decode("01011", codingtree);
    print("\n------\n");
    decode("0000", codingtree);
    decode("111", codingtree);
    decode("001", codingtree);
    decode("101", codingtree);
    decode("1001", codingtree);
    print("\n------\n");
    decode("00010", codingtree);
    decode("101", codingtree);
    decode("00011", codingtree);
    print("\n------\n");
```

```
    decode("01010", codingtree);
    decode("101", codingtree);
    decode("001", codingtree);
    decode("011", codingtree);
    print("\n------\n");
    decode("1101", codingtree);
    decode("01000", codingtree);
    decode("01001", codingtree);
    decode("1100", codingtree);
    print("\n------\n");
}

int decode(tree <char> letter (1), tree <char> codingtree (2)){
    tree <char> a (1);
    tree <char> b (2);
    a = letter;
    b = codingtree;
    while(true) {
            if(b%0 == null){
                    print(b@);
                    return 0;
            }
            if(a@ == '0') {
            /*      print(a@); */
                    b = b%0;
                    a = a%0;
            }
            else {
            /*      print(a@); */
                    b = b%1;
                    a = a%0;
            }
    }
}
```

Tests

test-fail1.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Checks that semantic analysis catches type mismatch
 */

int main () {
    print(1 + 1.0);
}
```

test-fail1.out

```
Fatal error: exception Failure("operator + not compatible with expressions of type int and float")
```

test-fail2.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests for faulty tree declaration
 */

int main () {
    tree <int> tree();
}
```

test-fail2.out

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

test-fail3.lrx

```
/*
 * Author: Zhaarn Maheswaran
```

```
 * Tests for scanner error
 */

int main () {

    int i;
    i = 1;
    i?2;
}
```

test-fail3.out

```
Fatal error: exception Failure("illegal character ?")
```

test-fail4.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test fucntion return types
 */

int function()
{
    return true;
}
int main () {
    function();
}
```

test-fail4.out

```
Fatal error: exception Failure("function return type bool; type intexpected")
```

test-fail5.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests failure of conditional statements
 */

int main () {

    int i;
    i = 1;
    while(i) {
            print(3);
    }
}
```

test-fail5.out

```
Fatal error: exception Failure("while loop must evaluate on boolean expression")
```

test-fail6.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests type mismatch of function arguments
 */

int function(int a, float b)
{
    print(a + b);
}
int main () {

    function(9, 7.0);

}
```

test-fail6.out

```
Fatal error: exception Failure("operator + not compatible with expressions of type int and float")
```

test-fail7.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests type mismatch of actual parameters
 */

int function(int a, int b)
{
    print(a + b);
}
int main () {

    function(9, 7.0);

}
```

test-fail7.out

```
1 Fatal error: exception Failure("function function's argument types don't match its formals")
```

test-fail8.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests incorrect tree decl
 */

int main () {

    tree <int> t (3);
    t = 1[2, 5[3, 5, 4, 3], 5];

}
```

test-fail8.out

```
1 Fatal error: exception Failure("Tree type is not consistent: expected <int>(3) but received
<int>(4)")
```

test-fail9.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests tree type mismatch
 */

int main () {

    tree <int> t (3);
    t = 1[2, 5[3, 5, 4.0], 5];

}
```

test-fail9.out

```
1 Fatal error: exception Failure("Tree literal type is not consistent: expected <int> but
received <float>")
```

test-full1.lrx

```
/*
 * Author: Chris D'Angelo
 * First end to end test of print
 */

int main()
{
    print(1);
    return 0;
}
```

test-full1.out

```
1 1
```

test-full10.lrx

```
/*
 * Author: Kira Whitehouse
 * Stress testing empty brace edge case tree definition
 */

int main()
{
    print("hello world\n\n");

    print(1[2, 3, 4, 5[], 6, 7[]], "\n");

    print('a'['b'['c'[]]], "\n\n\n");

    print(true[true[false, true], false[]], "\n");
}
```

test-full10.out

```
hello world


1[2[null,null,null,null,null,null],3[null,null,null,null,null,null],4[null,null,null,null,null,null],5[null,null,null,null,null,null],6[null,null,null,null,null,null],7[null,null,null,null,null,null]]
abc


true[true[false[null,null],true[null,null]],false[null,null]]
```

test-full11.lrx

```
/*
 * Authors:
 * Kira Whitehouse
 * Chris D'Angelo
 * End to end test of child operator (%) and assignment to tree node lhs
 */

int main()
{
    tree <int> t(2);
    tree <int> s(2);

    t = 3[4[9[101, 102], 10], 5];
    s = 6[7, 8];

    t%0%0%1 = s;
    print(t, '\n');
    t = s;
    print(t, '\n');
}
```

test-full11.out

```
3[4[9[101[null,null],6[7[null,null],8[null,null]]],10[null,null]],5[null,null]]
6[7[null,null],8[null,null]]
```

test-full12.lrx

```
/*
 * Authors:
 * Chris D'Angelo
 * End to end test of unop - and !
 */

int main()
{
    bool b;
    int a;
    a = -2;
    b = !false;

```

```
    print(a, '\n', b);
}
```

test-full12.out

```
-2
true
```

test-full13.lrx

```
/*
 * Authors:
 * Kira Whitehouse
 * Chris D'Angelo
 * End to end test of @ operator without child operator
 */

int main()
{
    tree <int>t(2);
    tree <float>t2(2);
    tree <bool>t3(2);
    tree <char>t4(2);
    t = 1[2, 3];
    t2 = 1.0[2.0, 3.0];
    t3 = true[true, false];
    t4 = 'a'['b', 'c'];
    print(t@, '\n', t2@, '\n', t3@, '\n', t4@);
}
```

test-full13.out

```
1
1.000000
true
a
```

test-full14.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests arithmetic. Adapted from Stephen Edwards microc.
 */

int main()
{
  print(39 + 3);
}
```

test-full14.out

```
42
```

test-full15.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests order of operations. Adapted from Stephen Edwards microc.
 */

int main()
{
  print(1 + 2 * 3 + 4);
}
```

test-full15.out

```
11
```

test-full16.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test left-to-right evaluation of expressions. Modified from Stephen Edwards microc.
 */
```

```

int a; /* Global variable */

int inca() { a = a + 1; return a; }  /* Increment a; return its new value */

int main() {
  a = 42;    /* Initialize a */
  print(inca() + a);
}
```

test-full16.out

```
86
```

test-full17.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test side-effect sequence in a series of statements. Modified from Stephen Edwards microc.
 */

int g;

int main() {
  int l;
  l = 1;
  print(l);
  g = 3;
  print(g);
  l = 5;
  print(l+100);
  g = 7;
  print(g+100);
}
```

test-full17.out

```
13105107
```

test-full18.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test for recursion. Modified from Stephen Edwards microc.
 */

int fib(int x)
{
  if (x < 2) { return 1; }
  return fib(x-1) + fib(x-2);
}

int main()
{
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
}
```

test-full18.out

```
112358
```

test-full19.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests for loop. Modified from Stephen Edwards microc.
 */

int main()
{
```

```
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(i);
  }
  print(42);
}
```

test-full19.out

```
0123442
```

test-full2.lrx

```
/*
 * Author: Chris D'Angelo
 * End to end test of assignment to all types
 */

int main()
{

    int a;
    float b;
    bool c;
    char d;
    a = 1;
    b = 3.14;
    c = true;
    d = 'a';
    print(a, b, c, d);
}
```

test-full2.out

```
13.140000truea
```

test-full20.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests functions. Modified from Stephen Edwards microc.
 */

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

int main()
{
  int a;
  a = add(39, 3);
  print(a);
}
```

test-full20.out

```
42
```

test-full21.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests functions calls with expressions. Modified from Stephen Edwards microc.
 */

int fun(int x, int y)
{
  return 0;
}

int main()
{
  int i;
  i = 1;
```

```

  fun(i = 2, i = i+1);

  print(i);
}
```

test-full21.out

```
3
```

test-full22.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests void function call. Modified from Stephen Edwards microc.
 */

int printem(int a, int b, int c, int d)
{
  print(a);
  print(b);
  print(c);
  print(d);
}

int main()
{
  printem(42,17,192,8);
}
```

test-full22.out

```
42171928
```

test-full23.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test left-to-right evaluation of arguments. Modified from Stephen Edwards microc.
 */

int a; /* Global variable */

int inca() { a = a + 1; return a; }  /* Increment a; return its new value */

int add2(int x, int y) { return x + y; }

int main() {
  a = 0;
  print(add2(inca(), a));
}
```

test-full23.out

```
2
```

test-full24.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests the GCD algorithm. Modified from Stephen Edwards microc.
 */

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

int main()
{
  print(gcd(2,14));
  print(gcd(3,15));
```

```
  print(gcd(99,121));
}
```

test-full24.out

```
2311
```

test-full25.lrx

```
/*
 * Authors:
 * Chris D'Angelo
 * Kira Whitehouse
 * Tests multiple child operator on lhs and rhs.
 */

int main()
{
    tree <int> t(2);
    tree <int> s(2);

    t = 1[2[3, 4], 5];
    s = 6[7[8,9], 10[]];

    t%0%0 = s;
    print(t, '\n');
    t = s%0%0;
    print(t, '\n');
}
```

test-full25.out

```
1[2[6[7[8[null,null],9[null,null]],10[null,null]],4[null,null]],5[null,null]]
8[null,null]
```

test-full26.lrx

```
/*
 * Authors:
 * Kira Whitehouse
 * Kitchen sink test of rhs, lhs % and @ operators.
 */

int main()
{
    tree <int> t(2);
    tree <int> s(2);
    tree <int> m(2);


    t = 1[2[3, 4], 5];
    s = 6[7[8,9], 10[]];
    m = 44[55[],66];

     t%0%0@ = s@;  // t = 1[2[6, 4], 5];
     print("t=\n", t , "\ns=\n", s, "\n\n");
     t = s%0%0;    //8[null, null]
     print("t=\n", t , "\ns=\n", s, "\n\n");
     t = s%0;              //7[8,9]
     print("t=\n", t , "\ns=\n", s, "\n\n");
     t%1%0 = m;
     print("t=\n", t , "\ns=\n", s, "\n\n");
     t = s%0%0%0;
     print("t=\n", t , "\ns=\n", s, "\n\n");
}
```

test-full26.out

```
t=
1[2[6[null,null],4[null,null]],5[null,null]]
s=
6[7[8[null,null],9[null,null]],10[null,null]]

t=
```

```
8[null,null]
s=
6[7[8[null,null],9[null,null]],10[null,null]]

t=
7[8[null,null],9[null,null]]
s=
6[7[8[null,null],9[null,null]],10[null,null]]

t=
7[8[null,null],9[44[55[null,null],66[null,null]],null]]
s=
6[7[8[null,null],9[44[55[null,null],66[null,null]],null]],10[null,null]]

t=
null
s=
6[7[8[null,null],9[44[55[null,null],66[null,null]],null]],10[null,null]]
```

test-full27.lrx

```
/*
 * Authors:
 * Chris D'Angelo
 * Kitchen sink test of trees, assignment, reassignment, for loop, print, strings
 */

int main()
{
    tree <int> t(2);
    tree <int> s(2);
    tree <int> u(2);
    string v;
    string w;
    int i;

    v = "abcdefg";
    w = "hijklmn";

    v%0%0@ = 'Z';
    v%0%0%0 = w;

    print(v, '\n');

    t = 1[2[-101, 102], 5];
    s = 6[7[8,9], 10[]];
    u = 1001[1002[1003, 1004], 1005[]];

    // print("s@ = ", s@, "\n,s@ = ", u%0%0@, "\n, s@ + t%0%1@ = ", s@ + t%0%1@, "\n");

    t%0@ = 201; // t = 1[201[-101, 102], 5];
    s = t%1 = u%0; //      t = 1[2[-101, 102], 1002[1003, 1004]];
    print("s = ", s, "\nt = ", t, "\nu = ", u, '\n');

    t = 1[2[-101, 102], 5];
    s = 6[7[8,9], 10[]];


    for (i = 0; i < 2; i = i+1) {
            t%0%i@ = i;
    }
    print(t, '\n', s);
}
```

test-full27.out

```
abZhijklmn
s = 1002[1003[null,null],1004[null,null]]
t = 1[201[-101[null,null],102[null,null]],1002[1003[null,null],1004[null,null]]]
u = 1001[1002[1003[null,null],1004[null,null]],1005[null,null]]
1[2[0[null,null],1[null,null]],5[null,null]]
6[7[8[null,null],9[null,null]],10[null,null]]
```

test-full28.lrx

```
/*
 * Authors:
 * Chris D'Angelo
 * tree child, and @ with char arithmetic
 */

int main()
{
    tree <char>t(2);
    t = 'E'['F', 'G'];

    print(t%0@ + ('B'-'A'));
}
```

test-full28.out

```
G
```

test-full29.lrx

```
/*
 * Authors:
 * Chris D'Angelo
 * Kira Whitehouse
 * tree child operator assignment
 */

int main()
{
    tree <int>t(2);

    t = 1[2, 3[4, 5]];
    t%0%1 = 102[103, 104];

    print("t = ", t, "\n");
}
```

test-full29.out

```
t = 1[2[null,102[103[null,null],104[null,null]]],3[4[null,null],5[null,null]]]
```

test-full3.lrx

```
/*
 * Author: Chris D'Angelo
 * End to end test of vanilla assignment and tree literals
 */

int main()
{
    tree <char>t(2);
    tree <int>t2(3);
    tree <float>t3(3);
    tree <bool>t4(2);
    t = 'h'['i'['j', 'o'], 'k'];
    t2 = 1[2, 3[4, 5, 6], 7];
    t3 = 1.0[2.1, 3.1, 4.1[5.2, 5.3, 5.4]];
    t4 = true[false[true, false], true];
    print(t, t2, t3, t4);
}
```

test-full3.out

```
h[i[j[null,null],o[null,null]],k[null,null]]1[2[null,null,null],3[4[null,null,null],5[null,null,null],6[null,null,null]],7[null,null,null]]1.000000[2.100000[null,null,null],3.100000[null,null,null],4.100000[5.200000[null,null,null],5.300000[null,null,null],5.400000[null,null,null]]]true[false[true[null,null],false[null,null]],true[null,null]]
```

test-full30.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests global variables. Adapted from Stephen Edwards microc.
 */

int a;
int b;

int printa()
{
  print(a);
}

int printb()
{
  print(b);
}

int incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
{
  a = 42;
  b = 21;
  printa();
  printb();
  incab();
  printa();
  printb();
}
```

test-full30.out

```
42214322
```

test-full31.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests integer operations. Adapted from Stephen Edwards microc.
 */

int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  print(1 == 2);
  print(1 == 1);
  print(99);
  print(1 != 2);
  print(1 != 1);
  print(99);
  print(1 < 2);
  print(2 < 1);
  print(99);
  print(1 <= 2);
  print(1 <= 1);
  print(2 <= 1);
  print(99);
  print(1 > 2);
  print(2 > 1);
  print(99);
  print(1 >= 2);
  print(1 >= 1);
  print(2 >= 1);
}
```

test-full31.out

```
3-125099falsetrue99truefalse99truefalse99truetruefalse99falsetrue99falsetruetrue
```

test-full32.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests float operations. Adapted from Stephen Edwards microc.
 */

int main()
{
  print(1.0 + 2.0);
  print(1.0 - 2.0);
  print(1.0 * 2.0);
  print(135.0 / 2.0);
  print(99.0);
  print(1.0 == 2.0);
  print(1.0 == 2.0);
  print(1.0 == 1.0);
  print(99.0);
  print(1.0 != 2.0);
  print(1.0 != 1.0);
  print(99.0);
  print(1.0 < 2.0);
  print(2.0 < 1.0);
  print(99.0);
  print(1.0 <= 2.0);
  print(1.0 <= 1.0);
  print(2.0 <= 1.0);
  print(99.0);
  print(1.0 > 2.0);
  print(2.0 > 1.0);
  print(99.0);
  print(1.0 >= 2.0);
  print(1.0 >= 1.0);
  print(2.0 >= 1.0);
}
```

test-full32.out

```
3.000000-
1.0000002.00000067.50000099.000000falsefalsetrue99.000000truefalse99.000000truefalse99.000000true
truefalse99.000000falsetrue99.000000falsetruetrue
```

test-full33.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test all statement forms. Adapted from Stephen Edwards microc.
 */

int foo(bool a, int b) {
  int i;
  if (a) {
    return b + 3;
  }
  else {
    for (i = 0 ; i < 5 ; i = i + 1) {
       b = b + 5;
    }
  }
  return b;
}

int main() {
  print(foo(true,42));
  print(foo(false,37));
}
```

test-full33.out

```
4562
```

test-full34.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests variable assignment. Adapted from Stephen Edwards microc.
 */

int main()
{
  int a;
  bool b;
  float c;
  char d;
  a = 33;
  b = true;
  c = 2.483;
  d = 'z';
  print(a);
  print(b);
  print(c);
  print(d);
}
```

test-full34.out

```
33true2.483000z
```

test-full35.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Tests variable assignment. Adapted from Stephen Edwards microc.
 */

int main()
{
  int a;
  int b;
  a = 42;
  b = 57;
  print(a + b * 3);
}
```

test-full35.out

```
213
```

test-full36.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test for variable assingment with global variables. Adapted from Stephen Edwards microc.
 */

int a;

int printxy(int x, int y) {
  print(x);
  print(y);
}

int main()
{
  int b;
  a = 42;
  b = 57;
  printxy(a + b * 3, 77);
}
```

test-full36.out

```
21377
```

test-full37.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing degree function
 */

int main()
{
    tree <int>t(2);
    int a;
    a = degree(t) + 100;
    print("should be degree 2 = ", degree(t), "\n");
    print("should be degree 3 = ", degree(1[2, 3[], 4[5, 6, 7]]), "\n");
    print("should be degree 1 = ", degree("Hello world\n"), "\n");
    print("should print 102 = ", a, "\n");
}
```

test-full37.out

```
should be degree 2 = 2
should be degree 3 = 3
should be degree 1 = 1
should print 102 = 102
```

test-full38.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * Test for while loop. Adapted from Stephen Edwards microc.
 */

int main()
{
  int i;
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
}
```

test-full38.out

```
5432142
```

test-full39.lrx

```
/*
 * Author: Chris D'Angelo
 * Test for parent function
 */

int main()
{
    tree <int>t(2);
    tree <int>t2(2);
    t = 1[2, 3[4, 5]];
    t2 = 0[1, 2];
    print("should print 3[4, 5] = ", parent(t%1%0), "\n");
    print("should print null = ", parent(1[2, 3]), "\n");
    print("should print 1[2, 3[4, 5] = ", parent(parent(t%1%0)), "\n");
    t2%0 = parent(parent(t%1%0));
    print("should print 0[1[2, 3[4, 5], null] = ", t2, "\n");
    // will cause assertion failure
    // print("what is this printing? ", parent(parent(t)), "\n");
}
```

test-full39.out

```
should print 3[4, 5] = 3[4[null,null],5[null,null]]
should print null = null
should print 1[2, 3[4, 5] = 1[2[null,null],3[4[null,null],5[null,null]]]
should print 0[1[2, 3[4, 5], null] = 
0[1[2[null,null],3[4[null,null],5[null,null]]],2[null,null]]
```

test-full4.lrx

```
/*
 * Author: Zhaarn Mathewsaan
 * End to end test of for loop
 */

int main()
{
	int i;
	for(i = 0; i < 10; i = i+1)
	{
		print(i);
	}
}
```

test-full4.out

```
0123456789
```

test-full40.lrx

```
/*
 * Author: Chris D'Angelo
 * Test for root function
 */

int main()
{
	tree <int>t(2);
	tree <int>s(2);
	string v;
	v = "Hello Kira";
	t = 1[2, 3[4, 5]];
	s = t;

	print("should print 1[2, 3[4, 5]] = ", parent(parent(t%1%1)), "\n");
	print("should print 1[2, 3[4, 5]] = ", root(t%1%1), "\n");
	print("should print 1[2, 3[4, 5]] = ", root(s%1%1), "\n");
	print("should print Hello Kira = ", root(v%0%0%0), "\n");
	print("should print 1[2, 3, 4] = ", root(1[2, 3, 4]));
}
```

test-full40.out

```
should print 1[2, 3[4, 5]] = 1[2[null,null],3[4[null,null],5[null,null]]]
should print 1[2, 3[4, 5]] = 1[2[null,null],3[4[null,null],5[null,null]]]
should print 1[2, 3[4, 5]] = 1[2[null,null],3[4[null,null],5[null,null]]]
should print Hello Kira = Hello Kira
should print 1[2, 3, 4] = 1[2[null,null,null],3[null,null,null],4[null,null,null]]
```

test-full41.lrx

```
/*
 * Author: Chris D'Angelo
 * Test for mod operator
 */

int main()
{
	print("should print 2 = ", 7 mod 5, "\n");
}
```

test-full41.out

```
should print 2 = 2
```

test-full42.lrx

```
/*
 * Author: Chris D'Angelo
 * Stress testing plus operator with [] edge cases
 */

int main()
{
    tree <int> t(2);
    tree <int> s(2);
    tree <int> m(1);

    t = 6[] + 4[2, 3];
    print(t, "\n");
    m = 4[] + 6[];
    print(t, "\n");
    t = 4[2, 3] + 6[];
    print(t, "\n");

    t = 4[2[5,6], 3] + 6[7, 8];
    print(t, "\n");

    s = 5[10, 9];
    t = t + s + s + t ;
    print(t, "\n");
}
```

test-full42.out

```
6[4[2[null,null],3[null,null]],null]
6[4[2[null,null],3[null,null]],null]
4[2[6[null,null],null],3[null,null]]
4[2[5[null,null],6[null,null]],3[6[7[null,null],8[null,null]],null]]
4[2[5[5[10[null,null],9[null,null]],4[2[5[null,null],6[null,null]],3[6[7[null,null],8[null,null]],null]]],6[null,null]],3[6[7[null,null],8[null,null]],5[10[null,null],9[null,null]]]]
```

test-full5.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * End to end test of if/else statements
 */

int main()
{
     int i;
     float j;
     char k;
     bool l;
     i = 1;
     j = 2.0;
     k = 'a';
     l = true;

     if(i == 1)
     {
             print(1);
     }
     else
     {
             print(0);
     }

     if(j == 2.0)
     {
             print(2);
     }
     else
     {
             print(0);
```

```
    }

    if(k == 'a')
    {
            print(3);
    }
    else
    {
            print(0);
    }

    if(l == true)
    {
            print(4);
    }
    else
    {
            print(0);
    }

    if(i == 1)
    {
            print(5);
    }

    if(i == 2)
    {
            print(0);
    }
    else
    {
            print(6);
    }
    return 0;
}
```

test-full5.out

```
123456
```

test-full6.lrx

```
/*
 * Author: Chris D'Angelo
 * End to end test of escape chars
 */

int main()
{
    char a;
    char b;
    char c;
    char d;
    char e;
    char f;
    string s;
    string s2;

    s = "hello, world\n";
    s2 = 'b'['y'['e'['\n']]];

    a = '\t';
    b = 'b';
    c = '\n';
    d = 'd';
    f = '\\';

    print(a, b, c, d, f, '\n');
    print(s);
    print(s2);
}
```

test-full6.out

```
	b
d\
hello, world
bye
```

test-full7.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * End to end test of tree passing, function call
 */

int test_tree(tree <int>t(2))
{
    print(t);
    return 0;
}

int main()
{
    tree <int>t(2);
    t = 1[2, 3[4, 5]];
    test_tree(t);
}
```

test-full7.out

```
1[2[null,null],3[4[null,null],5[null,null]]]
```

test-full8.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * End to end test of comparison operators with trees and atoms
 */

int main()
{

    tree <int>t(2);
    tree <int>t2(2);
    tree <int>t3(1);
    tree <char>t4(1);
    int a;
    int c;
    bool b;
    a = 3;
    c = 4;
    t = 1[2, 3];
    t3 = 9[10[11]];
    t2 = 4[5, 6[7, 8]];
    b = t2 > t;
    print(b);
    b = t2 <= t;
    print(b);

    print(t2 < t, '\n', t >= t2, '\n', t <= t3);

    print('\n', t == t3);

    print('\n', 2[3[]] == 2[3]);
    print('\n', 2[3[]] == 2[3[]]);
    print('\n', 1[2[],3[]] != t);
    print('\n', "hello\n" == "hello\n");
}
```

test-full8.out

```
truefalsefalse
false
true
false
true
true
```

```
false
true
```

test-full9.lrx

```
/*
 * Author: Zhaarn Maheswaran
 * End to end test of while loop
 */

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

test-full9.out

```
0123456789
```

test-parser1.lrx

```
int main()
{
print(39 + 3);
}
```

test-parser1.out

```
int main()
{
print(39 + 3);
}
```

test-parser2.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int do() {
print(1);
}

int do2() {
print(2);
}

int main()
{
do();
do2();
}
```

test-parser2.out

```
int main()
{
do();
do2();
}

int do2()
{
print(2);
}

int do()
{
```

```
print(1);
}
```

test-parser3.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int main() {
    bool b;
    a = b = c;
    z%0%3@ = 4;
    a = z%0;
    t%0 = t2; // t and t2 are both trees. t2 is now being assigned the first child of t (accessed by %0)
    t%0 = t3%0; // similar to above but now t's first child is t3's first child.
    t%0@ = 3; // @ dereferences the value in that node. Now the t's first child's value is 3
    t%3+4@ = 4;
    normal_int = t%0@; // now we're assigning a normal int var the value from inside t's first child value
    t@ = 4; // this is assigning the root nodes value as 4
    t%0%1@ = 5; // this is assigning t's first child's, second child node value to 5
    t3 = t%0%1--; // this is popping t's first child's second child node from the tree t and returns t to assign to t3
    t%toyfunc()%3@-- = t2--;
    t = 5[];
    b = (a%0 == null);
}
```

test-parser3.out

```
int main()
{
bool b;
a = b = c;
z % 0 % 3@ = 4;
a = z % 0;
t % 0 = t2;
t % 0 = t3 % 0;
t % 0@ = 3;
t % 3 + 4@ = 4;
normal_int = t % 0@;
t@ = 4;
t % 0 % 1@ = 5;
t3 = t % 0 % 1--;
t % toyfunc() % 3@-- = t2--;
t = 5[];
b = a % 0 == null;
}
```

test-parser4.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int main() {
    string a;
    a = "hello, world";
    print(a);
    return 0;
}
```

test-parser4.out

```
int main()
{
tree <char>a(1);
a = "hello, world";
print(a);
return 0;
```

```
}
```

test-parser5.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int main()
{
    int a;
    int b;
    while(true) {
           a = 1;
           b = 2;
           print(a);
    }
}
```

test-parser5.out

```
int main()
{
int a;
int b;
while (true) {
a = 1;
b = 2;
print(a);
}
}
```

test-parser6.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int func_test(int a, char b, tree<bool>t(3), bool c)
{
    tree <int>t(2);
    print(a);
}

bool main() {
    string s;
    tree <char>t(1);
    s = "hello, world";
    t = ','[' '['w'['o'['r'['l'['d']]]]]];
    s@;
    t%3;
    t%x;
    t%(5 + 6);
    print(s);
}
```

test-parser6.out

```
bool main()
{
tree <char>s(1);
tree <char>t(1);
s = "hello, world";
t = ','[' '['w'['o'['r'['l'['d']]]]]];
s@;
t % 3;
t % x;
t % 5 + 6;
print(s);
}

int func_test(int a, char b, tree <bool>t(3), bool c)
```

```
{
tree <int>t(2);
print(a);
}
```

test-parser7.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int main()
{
    tree <int>t(1);
    t%1 == null;
}
```

test-parser7.out

```
int main()
{
tree <int>t(1);
t % 1 == null;
}
```

test-parser8.lrx

```
/*
 * Author: Chris D'Angelo
 * Lorax Parser Kitchen Sink
 * Testing valid parseable file
 */

// parsing requires global variables must be declared first
int a;
tree <float>b(3);
float c;
char d;
string e;

int inc (int x) {
    return x + 1;
}

char capitalize_letter_a (char a) {
    if (a == 'a') {
            return 'A';
    }
    return '0';
}

int change_first_child_letter_to_p(tree <char>r(2)) {
    r%(1-1)@ = 'w'; // for fun
    // r%1-1@ = 'w'; // this is acceptable syntax but not semantics
    r%0@ = 'p';
    return 0;
}

int change_letter_to_q(tree <char>n(2)) {
    n@ = 'q';
    return 0;
}

int print_for_me_please(string s)
{
    print(s + "\n");
}

int capitalize_all_of_me(string s)
{
    string tmp;
    tmp = s;
    while (tmp%0 != null) {
```

```
        if (tmp@ < 'z') {
            // lowercase
            tmp@ = tmp@ + 'A' - 'a';
        }
        tmp = tmp%0;
    }
    return 0;
}

int main() {
    // parsing requires function locals must be declared first
    tree <float>g(3);
    tree <char>k(2);
    int l;
    char m;
    bool s;
    bool t;
    string v;
    tree <char>z(2);
    int y;
    l = 2;
    a = 4;

    while (l < a) {
        inc(l);
        break;
    }
    y = -1;
    print(a mod 3);
    g = 1.1[2.1, 2.2[2.21, 2.22, 2.23], 2.3[2.31, 2.32]];

    k = 'z'['x', 'y'['b', 'a']];
    print(capitalize_letter_a(k%1%1));

    change_letter_to_q(k%1%1);
    print(k%1%1@); // should print 0 and tree should be 'z'['x', 'q'['b', 'a']];
    change_first_child_letter_to_p(k);
    print(k); // should print 0 and tree should be 'z'['p', 'q'['b', 'a']];

    print(t = (!s || false && true));

    print_for_me_please("hello");

    v = "hello";
    capitalize_all_of_me(v);
    print(v);

    z = k + 'm'['n', 'o']; // will give 'a' a child 'm' (which itself has two children)

    z%1--; // pop the second (ref: 1) child off of z
    z = 'm'['n', 'o']--; // will nullify this tree
    z = 'm'['n', 'o']%0--; // will pop the 'n' from the tree

    for (l = 0; l < 42; l = l + 1) {
        print(l);
    }

    a = b = c;
    z%0%3@ = 4;

    a = z%0;

    t%0 = t2; // t and t2 are both trees. t2 is now being assigned the first child of t (accessed by %0)
    t%0 = t3%0; // similar to above but now t's first child is t3's first child.
    t%0@ = 3; // @ dereferences the value in that node. Now the t's first child's value is 3
    t%z@ = 4;
    normal_int = t%0@; // now we're assigning a normal int var the value from inside t's first child value
    t@ = 4; // this is assigning the root nodes value as 4
    t%0%1@ = 5; // this is assigning t's first child's, second child node value to 5
    t3 = t%0%1--; // this is popping t's first child's second child node from the tree t and returns t to assign to t3
```

```
    t%toyfunc()%3-- = t2;
}
```

test-parser8.out

```
tree <char>e(1);
char d;
float c;
tree <float>b(3);
int a;
int main()
{
tree <float>g(3);
tree <char>k(2);
int l;
char m;
bool s;
bool t;
tree <char>v(1);
tree <char>z(2);
int y;
l = 2;
a = 4;
while (l < a) {
inc(l);
break;}
y = -1;
print(a mod 3);
g = 1.1[2.1, 2.2[2.21, 2.22, 2.23], 2.3[2.31, 2.32]];
k = 'z'['x', 'y'['b', 'a']];
print(capitalize_letter_a(k % 1 % 1));
change_letter_to_q(k % 1 % 1);
print(k % 1 % 1@);
change_first_child_letter_to_p(k);
print(k);
print(t = !s || false && true);
print_for_me_please("hello");
v = "hello";
capitalize_all_of_me(v);
print(v);
z = k + 'm'['n', 'o'];
z % 1--;
z = 'm'['n', 'o']--;
z = 'm'['n', 'o'] % 0--;
for (l = 0 ; l < 42 ; l = l + 1) {
print(l);
}
a = b = c;
z % 0 % 3@ = 4;
a = z % 0;
t % 0 = t2;
t % 0 = t3 % 0;
t % 0@ = 3;
t % z@ = 4;
normal_int = t % 0@;
t@ = 4;
t % 0 % 1@ = 5;
t3 = t % 0 % 1--;
t % toyfunc() % 3-- = t2;
}

int capitalize_all_of_me(tree <char>s(1))
{
tree <char>tmp(1);
tmp = s;
while (tmp % 0 != null) {
if (tmp@ < 'z')
{
tmp@ = tmp@ + 'A' - 'a';
}
tmp = tmp % 0;
}
return 0;
```

```
}

int print_for_me_please(tree <char>s(1))
{
print(s + "\n");
}

int change_letter_to_q(tree <char>n(2))
{
n@ = 'q';
return 0;
}

int change_first_child_letter_to_p(tree <char>r(2))
{
r % 1 - 1@ = 'w';
r % 0@ = 'p';
return 0;
}

char capitalize_letter_a(char a)
{
if (a == 'a')
{
return 'A';
}
return '0';
}

int inc(int x)
{
return x + 1;
}
```

test-parser9.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid parseable file
 */

int main()
{
t = 1[2[1], 3];
return 0;
}
```

test-parse9.out

```
int main()
{
t = 1[2[1], 3];
return 0;
}
```

test-sa1.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

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

test-sa1.out

```
Passed Semantic Analysis.
```

test-sa10.lrx

```
/*
```

```
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    tree <int>t(2);
    tree <char>t2(2);
    tree <int>t3(2);
    tree <int>t4(2);
    int a;
    a = 5;
    t4 = (2+3)[a, (5-7)];
    t--;
    t3 = t%0%1;
    return 0;
}
```

test-sa10.out

```
Passed Semantic Analysis.
```

test-sa11.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    int a;
    bool b;
    float c;
    a = 1;
    b = true;
    c = 3.14;
    a = -a;
    b = !b;
    c = -17.0;
    return 0;
}
```

test-sa11.out

```
Passed Semantic Analysis.
```

test-sa12.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    tree <int>t(2);
    t = 3[4[], 5];
    t = 3[];
    return 0;
}
```

test-sa12.out

```
Passed Semantic Analysis.
```

test-sa13.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
```

```
    tree <int>t(2);
    int a;
    a = 2;
    t = 1[2, 3[4, 5]];
    t%1%0@ = a;
    t%1%0@ = 42;
    return 0;
}
```

test-sa13.out

```
Passed Semantic Analysis.
```

test-sa14.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    int n;
    bool b;
    tree <int>t(2);
    tree <char>t2(4);
    tree <int> t3(n);
    t = t + t3;
    b = t < t2;
    b = t == t3;

    return 0;
}
```

test-sa14.out

```
Passed Semantic Analysis.
```

test-sa15.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    int n;
    tree <int> t(2);
    tree <char> t2(4);
    tree <int> t3(n);

    null == null;
    t == null;
    t + t == t3;
    t != t2;

    return 0;
}
```

test-sa15.out

```
Passed Semantic Analysis.
```

test-sa16.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    bool b;
    int a;
```

```
	int c;
	if(b)
	{
		b = true;
	}

	else{
		c = 4;
	}
}
```

test-sa16.out

```
Passed Semantic Analysis.
```

test-sa17.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
	bool b;
	int i;
	while(b)
	{
		for(i = 0; i < 4; i = i + 1)
		{

		}
	}
}
```

test-sa17.out

```
Passed Semantic Analysis.
```

test-sa18.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
	bool b;
	int i;
	while(b)
	{
		for(i = 0; i < 4; i = i + 1)
		{
			break;
			break;
		}
		break;
		break;
	}
}
```

test-sa18.out

```
Passed Semantic Analysis.
```

test-sa19.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int d;

```

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

int chris(int a, int c, int f)
{
    int d;
    while(true) {
            a = 4;
    }
    d = 5;
}

int main()
{
    int b;
    chris(b, 4, add());
}
```

test-sa19.out

```
Passed Semantic Analysis.
```

test-sa2.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int a; // global scope

int main()
{
    int b;
    int a; // scope of main function
    return 0;
}
```

test-sa2.out

```
Passed Semantic Analysis.
```

test-sa3.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    3 + 4;
    3.0 - 7.0;
    'a' + 'z';
    true && false;
    return 0;
}
```

test-sa3.out

```
Passed Semantic Analysis.
```

test-sa4.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    int a;
    float b;
```

```
    char c;
    bool d;
    3 + a;
    3.0 - b;
    'a' + c;
    true && d;
    return 0;
}
```

test-sa4.out

```
Passed Semantic Analysis.
```

test-sa5.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    int a;
    float b;
    char c;
    bool d;

    a = 4;
    b = 5.0;
    c = 'b';
    d = false;

    3 + a;
    3.0 - b;
    'a' + c;
    true && d;
    return 0;
}
```

test-sa5.out

```
Passed Semantic Analysis.
```

test-sa6.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    1[2, 3[4, 5]];
    return 0;
}
```

test-sa6.out

```
Passed Semantic Analysis.
```

test-sa7.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    tree <int>t(2);
    t = 2[3, 4[5, 6]];
    return 0;
}
```

test-sa7.out

```
Passed Semantic Analysis.
```

test-sa8.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    tree <int>t(2);
    tree <int>t2(2);
    t%0 = 2[3, 4[5, 6]];
    t%0%1 = t2%0;
    return 0;
}
```

test-sa8.out

```
Passed Semantic Analysis.
```

test-sa9.lrx

```
/*
 * Author: Chris D'Angelo
 * Testing valid semantic analysis
 */

int main()
{
    tree <int>t(2);
    tree <char>t2(2);
    t@;
    t%0%1@;
    return 0;
}
```

test-sa9.out

```
Passed Semantic Analysis.
```