PLTree: Final Report

Table of Contents

Introduction

PLTree is a language for usage and manipulation of trees where the main data type is a tree. The language makes it easy to create and edit trees with functions such as adding a new item at a certain position in the tree or deleting items. The language will makes it simple to manipulate trees with common tree functions such as pruning, grafting, finding the root, and searching for an item. A selection of relevant tree functions are provided, a standard library is available, and user-defined functions may also be created for working with trees.

Language Tutorial

Environment Setup

At present PLTree is only supported on *nix systems, including Mac OS X and GNU/Linux. Certain dependencies must be installed in order to compile and run programs in PLTree.

OCaml

PLTree is written in the OCaml programming language. As such, to compile and run PLTree code, OCaml must be installed. Instructions for OCaml download and installation on different operating systems can be found on the OCaml site: http://ocaml.org/docs/install.html

For recent Mac OS X systems, OCaml can be installed with Homebrew. First, XCode command line tools needs to be installed, which can be done with the following command:

```
xcode-select –install
```

Next, Homebrew can be installed via curl with the following command:

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Finally, OCaml can be installed with the following command:

```
brew install ocaml
```

More information about Homebrew can be found at Homebrew website: http://brew.sh/

For Linux systems which utilize the Advanced Packaging Tool (APT) package manager, such as Debian- or Ubuntu-based systems, OCaml can be simply installed with the following command:

```
sudo apt-get install ocaml
```

More information about APT can be found on the Debian website: https://wiki.debian.org/Apt

PLTree

A makefile has been provided in order to automatically install the essential components to run PLTree. The makefile currently requires administrator privileges to run on a computer. On *nix systems, the makefile installs the standard library at /usr/local/bin/pltree_std

GCC

PLTree compiles to the C programming language. As such, to run programs written in PLTree after compilation, one must have a means of compiling C code installed. We recommend the GNU Compiler Collection (GCC) which may be downloaded at: https://gcc.gnu.org

Sample Program: Hello World

Code

```
$stdio.tree$
string str "hello\n";
print : str;
```

Walkthrough

The program starts with an import statement:

```
$stdio.tree$
```

This particular statement imports the standard library.

Following that is the first execution statement of the program:

```
string str "hello\n";
```

This line declares a string variable named str. As with all data types in PLTree, a string is a type of tree. A string tree holds characters, each of which is an individual single node tree.

This particular line makes use of syntactic sugar, a feature of the PLTree language. The string declaration statement in this sample program is syntactically equivalent to this line of code:

```
string str ['h' 'e' 'l' 'l' 'o' '\n'];
```

which represents the declaration of a string variable named str which is a tree of characters with six leaf nodes, each of which contains one character in "hello\n". The '\n' is a newline character, and indicates the end of text on that line and the start of a new line.

The next and final execution statement in this program is a print statement:

```
print : str;
```

This line simply calls the print function on str, to print the str string.

Expected Output

The expected output for this program would be:

```
hello
```

This output would be written by the print statement.

Sample Program: GCD (Iterative)

Code

```
$stdio.tree$

gcd : any args [
	int a args->0;
	int b args->1;
	int c 0;
	while : a != 0 [
		c = a;
		a = b%a;
		b = c;
	]
	return:b;
]

print:"Testing iterative gcd with 65 and 195\n";

print:[gcd:[65 195] "\n"];
```

Walkthrough

The program starts with an import statement:

```
$stdio.tree$
```

This particular statement imports the standard library.

gcd : any args [begins the declaration of a function named gcd. It accepts an argument of any type, and can refer to the argument as args within the function body.

int a args->0; int b args->1; int c 0; declares three variables, all of type int. The variable named a is set equal to the value of the function's argument's first branch. Similarly, b is set equal to the value of the second branch. c is set to 0.

while : a != 0 [begins a while loop that continues as long as the value of a is not 0.

c = a; a = b%a; b = c; assigns c the same value as a. a is set to the value of b mod a and b is set to the same value as c.

After the while loop terminates, return:b; causes gcd to return the value of b.

print:"Testing iterative gcd with 65 and 195\n"; prints the sentence “Testing iterative gcd with 65 and 195”, followed by a newline.

print:[gcd:[65 195] "\n"]; executes the newly-defined gcd function, passing as an argument a tree whose first branch is equal to 65 and whose second equals 195. The return value of that function call is then the first branch of an argument passed to print, and the newline character is the second. Simply, this statement prints the return value of gcd:[65 195] followed by a newline.

Expected Output
The expected output for this program would be:

```
65
```

This output would be written by the print statement.

Sample Program: GCD (Recursive)

Code

```
$stdio.tree$

gcdr : any args [
	int a args->0;
	int b args->1;

	if : a == 0 [
		return:b;
	]

	return:gcdr:[b%a a];
]

print:"Testing recursive gcd with 14 and 21\n";

print:[gcdr:[14 21] "\n"];
```

Walkthrough
This is similar to the above function. The main difference is this line:

```
return:gcdr:[b%a a];
```

This line recursively calls the function gcdr on b%a and a and returns the resulting value. This accomplishes the same result but recursively.

Expected Output
The expected output for this program would be:

This output would be written by the print statement.

Execution
To compile a .tree file into a .c file, execute a command in the following way:
$ ./pltree filename.tree filename.c
filename.c can then be compiled as a normal c file. Alternatively, a provided Makefile can be used to compile directly to the appropriate executable. From the directory containing the Makefile and the filename.tree file, execute
$ make filename
and then run the program with
$ ./filename

Language Reference Manual
Lexical Conventions
Tokens
The classes of tokens are: identifiers, keywords, constants, literals, and operators.

Reserved Keywords
int
double
char
string
any
args
if
ifelse
while
return
void
tree
width

Comments
The characters /* introduce a comment, which terminates with the characters */. Comments do not nest, and they cannot occur within a string or character literals. Comments can span multiple lines; multi-line comments are written in the same way as single-line comments.

Single line comments are written as follows:
/* this is a single-line comment */

Multi line comments are written similarly:
/* this is
a
multi-line comment */

Nested comments are not presently supported.

Identifiers
An identifier is a sequence of letters and digits, where the underscore (_) is included as a letter. Identifiers must begin with a letter and may be of any length. Upper and lowercase letters are treated differently, and thus PLTree is case-sensitive.

Constants
Constants are fixed values that do not change, which are built into the system. Standard mathematical constants are included. Constants are of various primitive data types. Primitive types are listed and described in the 'Types' section.

Literals
Literals are also items that may be of various primitive data types. The value of a variable that has been declared as equal to a literal, unlike a constant, may change. Primitive types are listed and described in the 'Types' section.

Types
All elements, including primitive types and collections, are trees. When a new variable is declared, a new tree is created. The smallest unit of the language is a single node. A node has a data member - for example, the integer 5 - and may have any number of children. All trees are built from these nodes. Tree roots and nodes may be of various data types. In this manual, we use node when we are not concerned with the children of that node, and tree elsewhere.

Primitive Types
Literals may be of the following primitive types:

Type	**Keyword**
character	char
integer	int
double	double

The boolean primitive type is not specifically included; boolean true/false values are indicated by int values '1' (if true) or '0' (if false).

Primitive types may form parts of expressions. Variables must be declared as of a particular type. See 'Variable Declaration' for explanation and syntax.

Collections
Collections are sets of primitive types held together in the same data structure. The fundamental data structure, on which all other data structures are built in this language, is the tree.

A string is a type of collection which contains a set of characters of type char. This collection is identified with the keyword string. A character string is a type of collection which is tree with a single root node of type char whose children are also of type char.

User-defined types are not presently supported.

Declarations

Variable Declaration

Every variable is treated as a tree. When a new variable is declared, a tree of a single node is created containing the data that variable is assigned to hold, of the type that the variable is declared as. For example, a string in our language would be a tree with leaves of characters.

Variables may be declared in the following syntax:

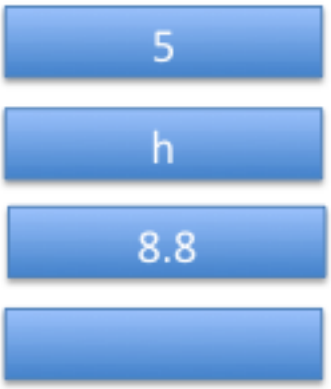
type name literal_value;

where *type* is one of the primitive types or void, *name* is an optional identifier for this tree, *literal_value* is either a literal or an expression of the data which evaluates to the appropriate type. These are stand-alone variable trees of a single node with no branches.

For example:

```
int a 5;
char b ‘h’;
double c 8.8;
void t []; /*empty tree*/
```

All of these are variable trees of a single node with no branches.



Variables may also be declared in the following syntax:

type name {tree_root}[children];

where *type* is one of the primitive types or void, *name* is an optional identifier for this tree, *tree_root* is either a literal or an expression of the data which the root of the variable tree will store which evaluates to the appropriate type and *{children}* is an optional argument that declares the children of the root of the variable tree; these may be values or variables.

When a variable is declared to be of type void, if there is a *tree_root* and *children*, *tree_root* is ignored. If only one argument is present, it is interpreted as *children*.

For example:

```
char str {'h'} ['e' 'l' 'l' 'o' '\n'];
```

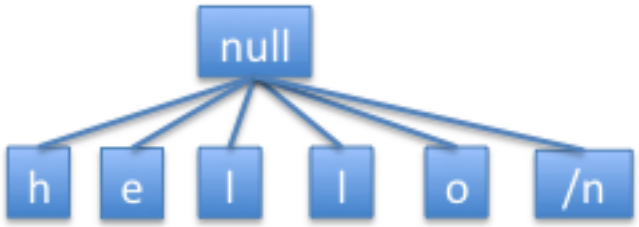
This is a character tree-in other words, a string-with root ‘h’ and children ‘e’, ‘l’, ‘l, ‘o’, and ‘/n’.

Variables may also contain trees. This is the case of strings, for example, which hold a tree with branches of nodes, each representing a single character in the string, as part of the tree.

For example:

```
char str ['h' 'e' 'l' 'l' 'o' '\n'];
```

holds a tree with branches that spell the string ‘hello\n’, with ‘\n’ being the new line character.



Type Inference

All primitive types can be written as is, without a type declaration or identifier.

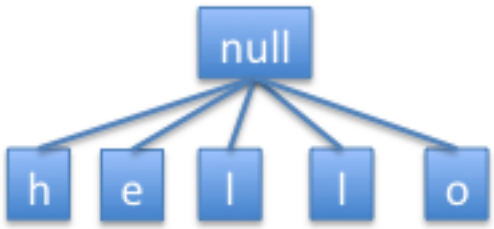
For example:

int 5; can be written as 5;
char ‘a’; can be written as ‘a’;
double 3.5; can be written as 3.5;

In addition, strings, which are really trees of char, can be declared in the following manner:

[‘h’ ‘e’ ‘l’ ‘l’ ‘o’]; can be written as “hello”;

Note the use of double quotes around strings and single quotes around chars.



Expressions and Statements

Operators

Arithmetic operators:

Addition	Binary	+
Subtraction	Binary	-
Multiplication	Binary	*

Division	Binary	/
Modulo	Binary	%
Minus	Unary	-

Numerical relational operators:

Is equal to	==
Is not equal to	!=
Is less than	<
Is less than or equal to	<=
Is greater than	>
Is greater than or equal to	>=

Logical operators:

And	Binary	&&
Or	Binary	||

Other operators:

Width	Unary (prefix)	#
Branch Accessor	Binary (postfix)	->i where i is the ith branch to be accessed

Operator Precedence

Operator precedence follows standard order of operations for all operators. Innermost parentheses always come first. Arithmetic operators take precedence over numerical relational operators. Numerical relational operators take precedence over logical operators. Accessor, width, and branch accessor operators take precedence over all other types of operators. Where operators share the same precedence, predecessors take precedence over successors.

Operator Usage

Operator functions are also available in addition to the standard operators. Operator functions are to be used for objects, while standard operators are to be used for primitive types.

All unary operators immediately precede their operand (e.g. -1, !true), except for the one referred to as Branch Accessor, which immediately follows its operand. Binary operators separate their operands (e.g 0 - 1, false && true). Operators that act on primitives automatically access the data member of the operand(s). For all binary operators, both

operands must be of the same type. All arithmetic operators expect Unary Minus are binary. They are all (including Unary Minus) to be used for either ints or doubles. The same type will be returned.

Numerical relational operators are to be used for either ints or doubles. A boolean value will be returned. In cases where numerical relational operators are used with other types, the value of the variables of other types will be converted to an appropriate numerical value for the purpose of comparison with a numerical relational operator.

The logical operators may only be used for ints. A integer value will be returned.

Width (#) is a unary prefix operator, shorthand for the function width, referring to all branches at the root of the present node. Its operand is a tree of any type. It returns an int equal to the number of branches, also known as the degree, of that tree.

The Branch Accessor Operator (->i) is used to access the ith branch of its operand where i is an integer. Branches are 0-indexed counting from the leftmost branch at the root. For example,

my_tree->2

returns branch 2 of tree my_tree, which is the 3rd branch with 0 as the starting index. Attempting to access a branch that does not exist is undefined.

Control Flow

Support for conditional branching in the form of if-statements and if-else statements, as well as loops in the form of while loops, are included. Conditions are boolean statements used in conditional branching and loops.

Conditional Branching

If/Else

If/else statements are written like this:

```
if : condition [
                /* condition met; do something */
] else [
                /* condition not met; do something */
]
```

Loops

While

While loops are written like this:

```
while : condition [
                /* condition met; do something */
]
```

Functions

Functions are actually a specialized type of tree. Children of function trees are statements, each of which is itself a tree. Nested functions form inner nodes of the same function tree, as each nested function is a tree whose children are statements. Execution of a function always proceeds in depth-first search order through the function tree.

Evaluation of a function occurs in applicative order, upon reaching relevant nodes involved.

As an example, a function call to the gcd function, with the int values 65 and 195 as the arguments of the gcd function, may look like this:

```
print: gcd: [65 195];
```

Functions may be called by other functions, with values of other functions serving as arguments for these functions.

As an example, a function call to print the result of the gcd function, with the int values 65 and 195 as the arguments of the gcd function, may look like this:

```
print: gcd: [65 195];
```

Built-In Functions

Built in functions are provided to all files by the standard library. The standard library is provided in the 'Appendix' section of this document.

User-Defined Functions

All functions accept as an argument any number of trees, each of which must be declared to be of a particular type. A function is declared in the following syntax:

```
name: argument-type t [
        /* do something */
        return:value;
]
```

where return-type is the return type, name is the name of the function, {argument-type t} is the input argument which must be of type argument-type, and the brackets [/*do something */] surround an execution block consisting of any number of statements.

A function may return any valid type. A function must always return a value of the appropriate type. If a function never returns, its return type should be declared as void; in this case, upon completion of the execution block, the function execution terminates without a returned value.

Nested Functions

Functions may be defined within functions; these are known as nested functions. As functions are trees in PLTree, nested functions further extend the tree with additional branching. The nested function is itself a function tree, and the root of the nested function extends the branch of the existing function tree for the function in which the nested function is nested.

File Input/Output

Importing Files

External files may be imported in the following way:

```
$filename$
```

where filename is the name of the external file to be imported in, including the .tree extension.

Program Structure and Scope

Structure

A program may exist entirely within a single source file or within multiple source files. A source file may include and link with files from existing libraries or other files. By convention import statements are typically to be included in the header of a source code file.

Valid Programs

A program must at minimum have a single valid execution statement.

For example,

```
put_t : {‘a’};
```

is a simple, complete program that executes put_t on the tree {‘a’}
This writes ‘a’ to the console.

Customarily there will also be at least one import statement:

```
$stdio.tree$
```

to import the standard library.

For example:

```
$stdio.tree$
print:”Hello, World!\n”;
```

Prints “Hello, World!” to the console with a newline

Scope Rules

Nodes can see data in child nodes below on the same branch, but not in sibling nodes on the same level as the nodes, and not in parent/ancestor nodes above the nodes.

Compilation

A compiler is provided for source code written in PLTree. Source code compiles to C.

File Extension

File extension for all PLTree programs shall be .tree

Standard Library

A standard library is included with our programming language with select built-in functions.

Code Examples

```
/* classic “Hello, World” */
$stdio.tree$
print:“Hello, World!\n”;
$stdio.tree$
/* iterative gcd function*/
gcd : any args [
        int a args->0;
        int b args->1;
        int c 0;
```

```
        while : a != 0 [
                c = a;
                a = b%a;
                b = c;
        ]
        return:b;
]

print:"Testing iterative gcd with 65 and 195\n";
print:[gcd:[65 195] "\n"];

/* recursive gcd function */
gcdr : any args [
        int a args->0;
        int b args->1;

        if : a == 0 [
                return:b;
        ]
        return:gcdr:[b%a a];
]

print:"Testing recursive gcd with 14 and 21\n";
print:[gcdr:[14 21] "\n"];
```

Project Plan

Project Processes

Planning

We met at minimum twice weekly (once with TA on a fixed date at a fixed time, once as a team with a target set date/time set every week, aimed to be regular but open to flexibility to accommodate individual needs), but often met more frequently, particularly when our discussions needed to go longer than the allotted time and/or around deadline times. Subteams also met separately on an as-needed basis.

For communication and collaboration we utilized Git (for version control), Slack (for collaboration, file sharing, and online discussion), Google Docs/Google Drive (for collaborative document editing, file sharing, and online discussion through commenting and the Google Docs chat panel), Google Chats/Hangouts, email, and mobile phone (both voice calls and SMS) to stay connected while we were away from each other. Email was used rather infrequently. We almost always met in person, but during certain occasions (such as Fall Break-during which two of our group mates travelled out) when we could not meet together in-person, we were able to collaborate online and continue working remotely while staying in touch. Fortunately it was only rarely that this needed to be done, and we usually worked together in-person.

We utilized a roughly ‘waterfall’ approach to project management, whereby we completed the entire project from start to finish in distinct linear stages. However, our ‘planning’ phase was relatively short, and we were more flexible and open to immediate change than is typically allowed in the usual ‘waterfall’ style of project management.

Specification

The Language Reference Manual (LRM) served as our primary specification document. We started writing the LRM from very early on in the project. However, our LRM underwent continuous revisions and significant changes throughout the project as our language evolved. We do not regret early writing of our LRM, however, as it made sure that we kept the important elements of our specification in mind constantly throughout the process. Back and forth discussion with the TA and some advice from Prof. Stephen Edwards helped significantly to evolve our language, and our LRM, into what it eventually became-more logical, more consistent, more robust.

Development

Development started with defining a grammar-first on paper, then typed-and then by implementation of a significant part of the lexer/parser and scanner. Code generation in the C programming language followed. This started very simply, with a basic 'hello world' functional program followed by a working 'print' function and then followed by the ability to 'pretty print' trees in a hierarchal structure. From there we moved onto more complex functions. Semantic analysis followed as did more complex code generation and improvements to all components. New functionality and revisions to syntax and grammar were included iteratively with time.

Testing

Unit testing, integration testing, and system testing were all used. Testing was performed continuously right from the start to ensure validity of code and to catch bugs/fix problems early on. We compared the expected code with the generated code and their results initially, but as our code became more robust, we compared the expected outputs with generated outputs.

Style Guide

Guidelines that we aimed to follow included the following:

- One statement per line where practical
- Use indentation to clearly mark code blocks
- Use indentation to clearly mark what falls within what sets of parentheses
- Use tab indentation rather than space indentation
- Helper/library functions may be written for commonly used code
- Encourage-but do not require-use of comments liberally to explain code well
- Use lowercase and/or camelcase for names (of functions, variables, files, etc.)

These were applied not as strict rules to be followed obligatorily but as general guides.

Team Responsibilities

Roles

Official team roles are assigned as follows:

Jacob Graff - System Architect
Shruti Kulkarni - Manager
Luiz ‘Bert’ Ramirez - Tester
Justin Walters - Language Guru

Subteams

Originally assigned subteams are as follows:

Jacob Graff & Justin Walters	syntax, grammar, scanner, parser, code generation
Luis 'Bert' Ramirez & Shruti Kulkarni	syntax, code generation, test plans, diagramming, planning, language tutorial, documentation

In reality, these roles and subteams were quite fluid as team members were willing and able to help out with what were originally other team members' assigned parts (i.e., people on the scanner/parser subteam also helped out with documentation, people on the documentation/test plans subteam discussed with the group what the grammar and syntax should look like, etc.). Team members did end up doing some thing(s) outside of their originally assigned parts at some point in the development and/or documentation process.

Project Timeline

Approximate timeline

Week of 09/23/2015 – Finalize proposal
09/30/2015 – Proposal submitted
Week of 10/15/2015 – Rough draft of LRM; work on grammar, lexer/parser
10/26/2015 – LRM Submitted
Week of 11/2/2015 – Work on lexer/parser, basic AST, early version of "Hello World", Test Plans
Week of 11/9/2015 – Better AST; update LRM, finalize "Hello World", test plans
11/16/2015 – Hello World completed
Week of 11/25 – Improve AST, SAST, test plans
Week of 12/2/2015 – Continue work on SAST, code generation, test plans
Week of 12/9/2015 – Finalize code, test plans, presentation slides
Week of 12/16/2015 – Fix bugs, presentation slides, final reports
12/22/2015 – Final report submitted

Development Environment

PLTree was built on a combination of Mac OS X and Linux platforms. Git was used as a distributed version control system. Vim, emacs, TextWrangler, and Sublime Text were among the text editors used. Most of the project code was done in OCaml. A makefile is used for installation and compilation. Tests were run from a bash script.

Project Log

We integrated Git with Slack to keep of project commits. This way every member of our team would be notified in case any change was made-however big or small-automatically. Some very early commits were lost due to mistakes in setup of Git; however, this was resolved early on. We also utilized Google Docs for collaborative document editing (in certain cases) which allowed us to keep track of who was working on what within the document at what time(s).

Selected excerpts and representative samples of our project log commits are included below:

Starting the project & initial development
----- November 8th -----

sgk2118 [10:39 PM]

enabled an integration in this channel: github

----- November 9th -----

github BOT [2:01 PM]
[PLTree:master] 2 new commits by Jacob:
3d7a159: Variable declaration, assignment (note: used = for assignment indication, might be necessary). Updated Makefile - Jacob
227976d: Fix merge conflicts - Jacob

[2:03]
[PLTree:master] 1 new commit by Jacob:
007a77e: Bring back Makefile - Jacob

jag2302 [2:04 PM]
joined #code

github BOT [2:15 PM]
[PLTree:master] 2 new commits by Jacob:
adea28f: More comprehensive test program - Jacob
e3acd3e: Makefile: hello depends on helloWorld.tree - Jacob

----- November 10th -----

github BOT [3:59 PM]
[PLTree:master] 1 new commit by Shruti Kulkarni:
d02b52e: test - Shruti Kulkarni

Parsing & revised test scripts
----- November 14th -----

github BOT [1:00 AM]
[PLTree:trees] 1 new commit by Jacob:
bf25aa1: Add support for comparisons, subtraction, addition - Jacob

github BOT [1:36 PM]
[PLTree:trees] 1 new commit by Jacob:
1451f69: Better tree printing - Jacob

[1:36]
[PLTree:master] 7 new commits by Jacob:
b4f7386: Start adding tree support - Jacob
449f006: Working hello world with trees - Jacob
e418e38: Improve support for trees, strings suck now though - Jacob
eb77253: Much more extensive tree support - Jacob
4160f94: remove test.py - Jacob Show more...

github BOT [1:51 PM]
[PLTree] New branch "printing" was pushed by jagraff

github BOT [2:08 PM]
[PLTree:printing] 1 new commit by Jacob:
8ec249b: improved printing - Jacob

github BOT [2:14 PM]
[PLTree:master] 2 new commits by Jacob:
d5999ca: Better pretty printing - Jacob
8ec249b: improved printing - Jacob

github BOT [4:36 PM]
[PLTree:master] 2 new commits by Luis Ramirez:
3955aaa: v1 of test suite: see comments at top of tester.sh - Luis Ramirez
6707795: Merge branch 'master' of PLTree - Luis Ramirez

github BOT [5:32 PM]
[PLTree:master] 1 new commit by Jacob:
37c4bb4: Changes to hello.tree and expected output - Jacob

github BOT [9:58 PM]
[PLTree] New branch "prog-rep" was pushed by JustWalters

github BOT [10:15 PM]
[PLTree:master] 2 new commits by Luis Ramirez:
7128e82: update tester script; accepts dirs & multi args - Luis Ramirez
e475f66: Merge branch 'master' of PLTree - Luis Ramirez

Further revisions to code & test plans

----- November 18th -----

github BOT [1:13 PM]
[PLTree:master] 1 new commit by Jacob:
1b1b51e: Modify loop printing slightly - Jacob

----- November 21st -----

github BOT [10:51 PM]
[PLTree:prog-rep] 24 new commits by Jacob and 2 others:

b4f7386: Start adding tree support - Jacob
449f006: Working hello world with trees - Jacob
e418e38: Improve support for trees, strings suck now though - Jacob
eb77253: Much more extensive tree support - Jacob
4160f94: remove test.py - Jacob Show more...

----- November 23rd -----

github BOT [4:30 PM]
[PLTree] New branch "Syntax" was pushed by jagraff

----- November 25th -----

github BOT [8:53 PM]
[PLTree] New branch "syntax" was pushed by jagraff

----- November 26th -----

github BOT [3:12 PM]
[PLTree:master] 5 new commits by Jacob:
ca7b144: Saving changes to parser - Jacob
6de432c: Remove .swp file - Jacob
25b2861: Working parser with new syntax - Jacob
9e54429: Fix a shift/reduce and update tests - Jacob
1efa54a: Fix all shift/reduce conflicts - Jacob

----- December 1st -----

github BOT [2:26 AM]
[PLTree:prog-rep] 2 new commits by Justin Walters:
2e02e54: Transformation from AST to SAST outputs rigt type - Justin Walters
e92c1c5: Slight changes to SAST printing - Justin Walters

github BOT [3:09 AM]
[PLTree:prog-rep] 3 new commits by Justin Walters:
2fcd890: Transformation returns more than last statement - Justin Walters
c1be084: Collect top-level var decs - Justin Walters
9c1283b: Preparing for top-level func declarations - Justin Walters

github BOT [3:18 AM]
[PLTree:prog-rep] 1 new commit by Justin Walters:
2e76edd: = and < return bool, not int - Justin Walters

github BOT [5:10 PM]
[PLTree:master] 1 new commit by Jacob:
035de74: Modified tester; grammer fixed - Jacob

----- December 2nd -----

github BOT [4:48 PM]
[PLTree:master] 3 new commits by Luis Ramirez:
128e26e: Added 'ignore whitespace' to tester - Luis Ramirez
4eaea2c: Added 'ignore whitespace' to tester - Luis Ramirez
d73ddae: Merge branch 'testsuite' - Luis Ramirez

Finalizing code
-- December 16th -----

github BOT [10:16 PM]
[PLTree] New branch "imports" was pushed by jagraff

github BOT [10:21 PM]
[PLTree:imports] 1 new commit by Jacob:
732a824: Fix some printing issues - Jacob

github BOT [10:37 PM]
[PLTree:imports] 1 new commit by Jacob:
69e6436: Add stdio library - Jacob

----- December 17th -----

github BOT [12:04 AM]
[PLTree:imports] 1 new commit by Jacob:
e99df2e: Add double import checking - Jacob

github BOT [12:45 AM]
[PLTree:imports] 3 new commits by Jacob:
3d95b9d: Making sure everything imports properly - Jacob
1f8f452: get rid of extra files - Jacob
7a82b52: Update Makefile to remove c files - Jacob

github BOT [12:56 AM]
[PLTree:imports] 1 new commit by Jacob:
a9eb22d: Get rid of all menhir warnings - Jacob

github BOT [11:04 AM]
[PLTree:imports] 1 new commit by Jacob:
90c70cf: Get tester working with new compiler - Jacob

github BOT [2:01 PM]
[PLTree:imports] 1 new commit by Jacob:
c8f1f63: Fix some warnings - Jacob

github BOT [2:36 PM]
[PLTree:master] 11 new commits by Jacob:
c82d1d0: working imports - Jacob
93abcb1: Get rid of unnecessary files - Jacob
732a824: Fix some printing issues - Jacob
69e6436: Add stdio library - Jacob
e99df2e: Add double import checking - Jacob Show more...

github BOT [3:17 PM]
[PLTree:master] 1 new commit by Jacob:
f96d35f: Modify print function slightly - Jacob

github BOT [6:42 PM]
[PLTree:master] 1 new commit by Jacob:
93976b7: Add install target to Makefile - Jacob

github BOT [6:49 PM]
[PLTree:master] 1 new commit by Jacob:
383a999: Remove void as vtype - Jacob

github BOT [6:54 PM]
[PLTree:master] 1 new commit by Jacob Graff:
052529e: Get rid of more unecessary vtype stuff - Jacob Graff

github BOT [10:50 PM]
[PLTree:master] 1 new commit by Justin Walters:
bc9ce72: All returns in a function must be the same type - Justin Walters

github BOT [11:56 PM]
[PLTree:master] 1 new commit by Justin Walters:
38c682d: FuncDec was returning its internal environment - Justin Walters

----- December 18th -----

github BOT [5:23 AM]
[PLTree:master] 1 new commit by Justin Walters:
7325f88: Can't redeclare a function - Justin Walters

github BOT [12:30 PM]
[PLTree:master] 1 new commit by Jacob:
f131b4a: Add uninstall target, change pretty_print name - Jacob

github BOT [12:44 PM]
[PLTree:master] 1 new commit by Shruti Kulkarni:
a2ebac4: presentation code - Shruti Kulkarni

Initial documentation
September 30, 9:38 PM
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:37 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:37 PM
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:30 PM
Jacob Abraham Graff
September 30, 9:30 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 9:30 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:28 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 9:28 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:26 PM
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:24 PM
Justin Walters
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:24 PM
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:24 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:23 PM
Luis Alberto Ramirez

Jacob Abraham Graff
September 30, 9:21 PM
Luis Alberto Ramirez
Justin Walters
Jacob Abraham Graff
September 30, 9:20 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 9:16 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 9:16 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 9:14 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 8:56 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 8:54 PM
Jacob Abraham Graff
September 30, 8:53 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 8:53 PM
Jacob Abraham Graff
September 30, 8:51 PM
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 8:47 PM
Luis Alberto Ramirez
Jacob Abraham Graff
Shruti Gopal Kulkarni
September 30, 8:47 PM
Luis Alberto Ramirez
Jacob Abraham Graff
September 30, 8:43 PM
Luis Alberto Ramirez
September 30, 8:25 PM
Luis Alberto Ramirez
Shruti Gopal Kulkarni
September 30, 5:01 PM
Shruti Gopal Kulkarni
September 29, 8:13 PM
Shruti Gopal Kulkarni
September 29, 8:13 PM
Shruti Gopal Kulkarni

Finalizing documentation

December 22, 11:45 PM
Shruti Gopal Kulkarni
December 22, 3:29 PM
Jacob Abraham Graff
December 21, 6:38 PM
Jacob Abraham Graff
December 18, 2:16 PM
Shruti Gopal Kulkarni
December 17, 10:26 PM

Shruti Gopal Kulkarni
December 17, 12:41 AM
Shruti Gopal Kulkarni
December 16, 11:30 PM
Shruti Gopal Kulkarni
December 16, 5:30 PM
Jacob Abraham Graff
Shruti Gopal Kulkarni
December 16, 4:58 PM
Shruti Gopal Kulkarni
December 16, 4:53 PM
Shruti Gopal Kulkarni

Language Evolution

Our language began with many parentheses, and so it looked a bit like Lisp. As in Lisp, where everything is a list, everything in PLTree is a tree, and so we wanted to illustrate that in code. It turned out to be unwieldy and confusing, and it received negative feedback, and so we shifted to a completely different syntax that no longer has any connection to the underlying tree concept.

Instead, we moved to a simple syntax without too many extraneous symbols. We did preserve the consistency between different kinds of statements that we had in mind with the original syntax. For instance, while, if, and function declarations look similar, with a keyword or id followed by a colon, an expression or argument type and name, followed by a list of statements enclosed in square brackets.

Major changes in language design were largely spurred by either a desire to ease development in PLTree or a realization that our plans were too technically complex for us to implement well given our time constraints. For instance, our language reference manual stated that user defined functions must declare the type that they return. We later removed that requirement because we found it unnecessary and in order to simplify the compilation process. Several weeks later, when we started adding support for passing arguments, we realized that functions should know and control the type of their argument and so added support for that. At the same time, we allowed the naming of the argument. Because all functions have at most one argument, we could have reserved arg, or something similar as a keyword, but instead allow the programmer to name it because it improves their experience and was very simple to implement.

Two major features that we ended up not implementing were representing functions as trees and typedef. We pictured functions as something like their AST representation, and the programmer would be able to manipulate the statements of a function. This may have had some interesting uses, but was not needed and so was always somewhat of a low priority. Unfortunately, this means not everything in the language is a tree. This is a slight inconsistency, but we do not believe it substantially detracts from PLTree. Typedef would have made it possible to name a tree of a particular structure. For instance, a user could declare a type called binaryTree, where objects of this type may have no more than two children per node. This would have fit nicely with the function declaration syntax we settled on, but it did not come together. Users are fully able to define functions that check the structure of their argument and either coerce it into an acceptable form or simply inform the caller that it is not what was expected. Typedef could have simplified this for the programmer, but it had to be sacrificed in the interest of time.

We did add some features that were not originally planned. We had not intended to support the import of external files, but we also wanted to minimize the number of

built-in functions. As our standard library began to grow, and we saw how large generated files were in comparison to the original .tree file, partially due to the standard library functions we were including in all files, we decided to extract that and have the user explicitly include them.

Another change we made that is the switch from compiling to LLVM to C. Early on, we realized that representing our tree-based language in LLVM would be more difficult than first anticipated. Rather than cripple our language in any way, we changed our target language. Because all of us have some familiarity with C, as opposed to LLVM, which none of us have experience with, we were better able to gauge the difficulty of implementing various features, and this allowed us to concentrate on parts of the language that were both important and feasible.

Future versions of PLTree would hopefully include the features we were forced to leave out as well as some additional ones that we did not even consider implementing this semester. Syntactic sugar like an incrementer operator (++) and for loops are two simple examples. Programmers should also be able to directly access the root of a tree. This was discussed during initial design sessions, but forgotten as we moved to implementation. Built in arithmetic and comparison operators and the default print function use the root, and programmers can by convention dedicate a branch to hold data if they need to modify it themselves, so it is neither useless nor necessary. However, it would obviously make sense to allow read and write access after the tree has already been created.

A function type was beyond our scope, but would greatly enhance the language. Currently, any function that needs to traverse a tree must implement the walk itself. Instead, passing a function to a depth-first traversal that calls the function on each node should be an option. We would also like to see domain-specific libraries. Those using PLTree as an educational tool or general purpose language will have different needs than those using it for those using it to work with decision trees. However, these groups will have common operations within themselves, and so it would be nice if they have specific libraries.

Anything is possible in the current incarnation of PLTree, but some routine tasks could definitely be made easier.

Translator Architecture

Block Diagram

IMPORT PREPROCESSOR:

Components

Scanner:

The scanner is fairly simple. It reads through the input stream, character by character,

and aggregates them into tokens. These tokens may have some data attached as well,

but not all of them do.

Parser:

The parser converts the tokens into an Abstract Syntax Tree. The different acceptable patterns of tokens are given by a context free grammar; parsing is done by building up a parse tree from the tokens. The Abstract Syntax Tree (AST) is built from this.

Semantic Checker:

The semantic checker is used to make sure that scoping rules are followed, and types aren't mismatched. In order to do this, the checker translates the AST to and SAST, and at each step updates an environment container. This environment maintains a list of global and local variables and function names. As a result, all scoping rules can be checked (for example, a statement that uses the variable 'x' must have a local or global variable named 'x' within its scope). In addition, the type of a variable is maintained, so any type mismatches cause an error.

Translator:

From the SAST, we generate a C AST, a tree of syntactically and semantically valid C

statements and expressions. During this translation, function declarations are moved to the global scope, since C does not allow nested functions.

C Pretty Printer:

The C pretty printer is fairly simple. It simply takes our CAST, which contains our representation of a C program, and prints out each element as a C statement.

Import Preprocessor:

The import preprocessor is another scanner/parser pair. The scanner looks for statements surrounded by '$'s, and reads in whatever is inside of those as import statements.

From there, each import statement is processed. For every import statement found, the same process is run on each file that needs to be imported, and a list maintained of all processes that have been imported (to prevent double imports). All of this is written into a .tmp file, which contains all the same code, with imports expanded.

Test Plan and Scripts

The tester suite was created using a bash script to automate actions and a number of support files. The files were initially the expected abstract syntax trees created by programs in PLTree. As the programs grew more complex, we switched to using the expected output of programs after compilation. The test programs were ran and compared with the bash script: it first runs the "make" command, then compiles the program, runs it and saves the output, and finally uses the "diff" command to compare it to the expected output.

The following program files were used to test the language:

- hello.tree: "Hello World" implementation
- fact.tree: factorial function; tests if loops
- fibo.tree: calculate Fibonacci sequence
- full.tree: used to test full system; tries to use all language features
- func_test.tree: test function declaration
- gcd.tree: test recursive functions
- unit.tree: incrementally test language features, such as operators

Testing Phases

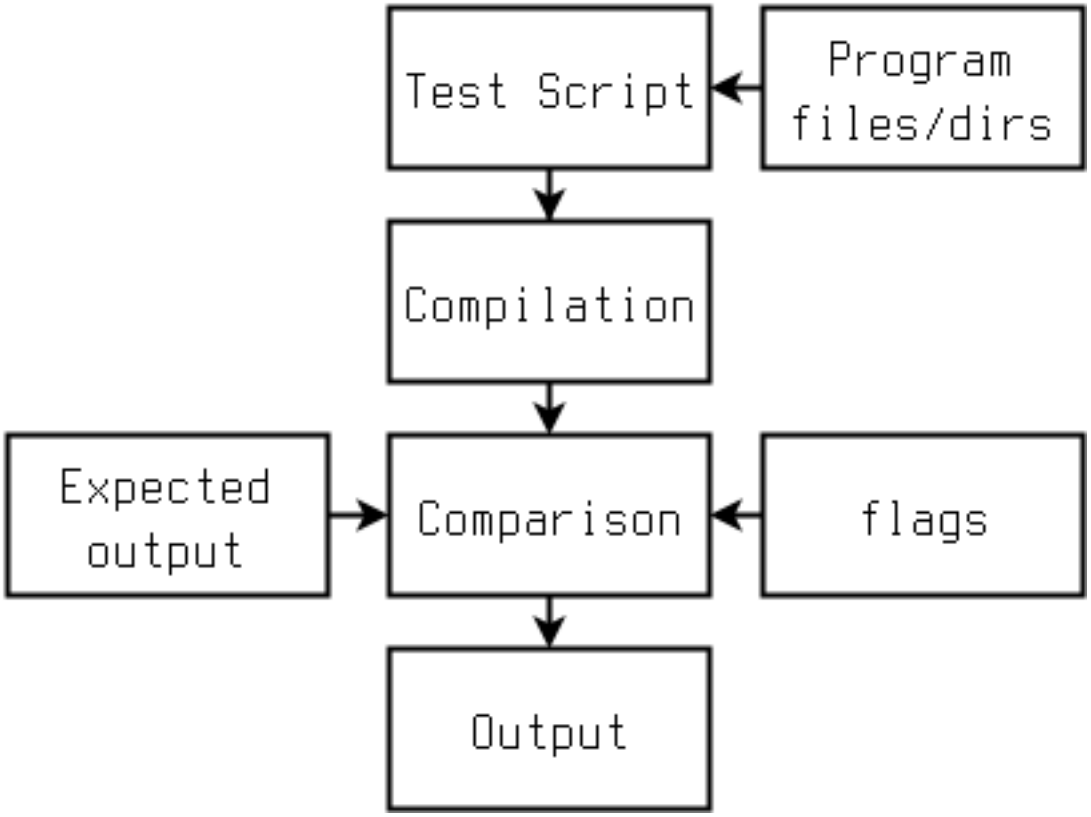
Unit Testing
The "unit" program incrementally tested variables and simple operators as they were added to the language. It would define different types and try to operate on combinations, in order to ensure proper behavior and to test if errors are caught. It also tested the conditional loops once they were implemented.

Integration Testing
This was tested by the "hello" program. As the most used module that we created was the stdio library, this program tested all the code in this module to ensure it was imported correctly and was producing the expected output.

System Testing
The "full" program attempted to test the language in its entirety. Almost all language features, such as definitions, operations, and standard functions are tested in the same program. Some code is repeated from the "unit" and "hello" programs, but some new scenarios are also added to inspect every side of the language.

Automation & Implementation

The tester script takes in files and directories as arguments. It also accepts the “-c” flag to compile and run the program. As stated before, it initially used abstract syntax trees to test programs. The “-c” flag was added to bypass this and compare the output of the compiled program.
The script goes through every argument given. If the “-c” flag is present, it switches to compile mode. If it is not, it goes through every given file and every file in a given directory. It checks to make sure the expected AST is present, runs it through the compiler, and compares the expected AST to the produced AST. If it is as expected, “SUCCESS” is printed in green. If it is not, “FAILED” is printed in red.
In compile mode, the script goes through every given file and every file in a given directory. It compiles it into a c file, executes it, and saves the output of the program. This output is compared to a pre-existing file to ensure that the program is outputting what it should. The following diagram illustrates it more clearly.



In its final iteration, the test script was run by simply entering:

```
$ ./tester.sh
```

This would automatically look in the tests/programs directory, assume that it was filled with .tree files, and test each file one by one, but only if no flags or arguments are given.

Test Scripts

tester.sh:

```
#!/bin/bash
#
# This script tests .tree files.
# usage: ./tester.sh [-c] [FILE.tree | treedir/]?
# the '-c' flag compiles the program and compares expected output
#
# File structure:
#    .tree files go in tests/programs
#     expected AST files go in tests/output_exp/FILE_exp
#     expected compiled output files go in tests/output_res/FILE_res
# All other directories are only used by the script
#


# define suffixes
out='_out'
exp='_exp'
# define directories
```

```
progdict='tests/programs/'
outdict='tests/output_out/'
expdict='tests/output_exp/'
# define default option
options='./pltree'

# Make sure there's arguments
if [ $# -eq 0 ]
then
  expdict='tests/output_res/'
  exp='_res'
  for f in $(ls $progdict)
        do
        # remove .tree extension
        progname="${f%.*}"
        # Make sure an expected output file is present
        if ! [ -a ${expdict}${progname}${exp} ]
        then
                echo "Place ${progname}${exp} file in ${expdict} directory."
                exit 2;
        else
                # make all the files and save output
                make $progdict$progname &> /dev/null
                $progdict$progname > "${outdict}${progname}${out}"

                # check if there are diffs
                if ! [[ $(diff -bw ${outdict}${progname}${out} ${expdict}${progname}${exp})
]]
                then
                        printf "${progdict}${progname}: \033[0;32mSUCCESS\033[0m\n"
                else
                        printf "${progdict}${progname}: \033[0;31mFAILED\033[0m\n"
                fi

                # clean directory of all non-.tree files
                find $progdict ! -name "*.tree" -type f -delete
        fi
  done
fi

make_calc() {
  # run "make"; surpress output
  make install &> /dev/null
  # double check that calc was created
  if ! [ -a pltree ]
  then
        echo 'Invoke lexer/parser to produce calc file.'
        exit 2;
  fi
}
```

```
# First, check for -c (compile) flag
while getopts ":c" opt; do
  case $opt in
        c)
                # change variables if the -c flag is present
                outdict='tests/output_com/'
                expdict='tests/output_res/'
                rundict='tests/output_run/'
                out='_run'
                exp='_res'
                options='./pltree'
                make_calc
                for arg
                do
                        # if it's a regular file
                        if [ -f $arg ]
                        then
                                # remove .tree extension
                                progname="${arg%.*}"
                                # Make sure an expected output file is present
                                if ! [ -a ${expdict}${progname}${exp} ]
                                then
                                        echo "Place ${progname}${exp} file in ${expdict}
directory."
                                        exit 2;
                                else
                                        # compile given file; save output
                                        $options $arg "${outdict}${progname}.c"
                                        gcc -Wall "${outdict}${progname}.c" -o
"${outdict}${progname}"
                                        rm -rf *.tmp
                                        touch "${rundict}${progname}${out}"
                                        "./${outdict}${progname}" >
"${rundict}${progname}${out}"

                                        # check if there are diffs
                                        if ! [[ $(diff -bw ${rundict}${progname}${out}
${expdict}${progname}${exp}) ]]
                                        then
                                                printf "${progname}:
\033[0;32mSUCCESS\033[0m\n"
                                        else
                                                printf "${progname}:
\033[0;31mFAILED\033[0m\n"
                                        fi
                                fi
                        # if it's a directory
                        elif [ -d $arg ]
                        then
                                for f in $(ls $arg)
```

```
                                do
                                        # remove .tree extension
                                        progname="${f%.*}"
                                        # Make sure an expected output file is present
                                        if ! [ -a ${expdict}${progname}${exp} ]
                                        then
                                                echo "Place ${progname}${exp} file in
${expdict} directory."
                                                exit 2;
                                        else
                                                # compile given file; save output
                                                $options $arg/$f  "${outdict}${progname}.c"
                                                gcc -Wall "${outdict}${progname}.c" -o
"${outdict}${progname}"
                                                rm -rf $arg/*.tmp
                                                touch "${rundict}${progname}${out}"
                                                "./${outdict}${progname}" >
"${rundict}${progname}${out}"

                                                # check if there are diffs
                                                if ! [[ $(diff -bw ${rundict}${progname}${out}
${expdict}${progname}${exp}) ]]
                                                then
                                                        printf "${arg}${progname}:
\033[0;32mSUCCESS\033[0m\n"
                                                else
                                                        printf "${arg}${progname}:
\033[0;31mFAILED\033[0m\n"
                                                fi
                                        fi
                                done
                        fi
                done
                exit 2;
                ;;

        \?)
                # exit on invalid flag
                echo "Invalid option: -$OPTARG"
                exit 2;
                ;;
  esac
done

make_calc

  for arg
  do
        # if it's a regular file
        if [ -f $arg ]
```

```
        then
                # remove .tree extension
                progname="${arg%.*}"
                # Make sure an expected output file is present
                if ! [ -a ${expdict}${progname}${exp} ]
                then
                        echo "Place ${progname}${exp} file in ${expdict} directory."
                        exit 2;
                else
                        # run parser on input file; save output
                        cat $arg | $options > "${outdict}${progname}${out}"

                        # check if there are diffs
                        if ! [[ $(diff -bw ${outdict}${progname}${out}
${expdict}${progname}${exp}) ]]
                        then
                                printf "${progname}: \033[0;32mSUCCESS\033[0m\n"
                        else
                                printf "${progname}: \033[0;31mFAILED\033[0m\n"
                        fi
                fi
        # if it's a directory
        elif [ -d $arg ]
        then
                for f in $(ls $arg)
                do
                        # remove .tree extension
                        progname="${f%.*}"
                        # Make sure an expected output file is present
                        if ! [ -a ${expdict}${progname}${exp} ]
                        then
                                echo "Place ${progname}${exp} file in ${expdict} directory."
                                exit 2;
                        else
                                # run parser on input file; save output
                                cat $arg/$f | $options > "${outdict}${progname}${out}"

                                # check if there are diffs
                                if ! [[ $(diff -bw ${outdict}${progname}${out}
${expdict}${progname}${exp}) ]]
                                then
                                        printf "${arg}/${progname}:
\033[0;32mSUCCESS\033[0m\n"
                                else
                                        printf "${arg}/${progname}:
\033[0;31mFAILED\033[0m\n"
                                fi
                        fi
                done
        fi
  done
```

Sample Test Programs

This program did the unit testing. It defines variables and tests while and for loops. The penultimate line references an undefined variable. The compiler recognizes as such and returns an error during compilation.

```
$stdio.tree$

int a 2;
double b 3.0;
char c 'a';
print: a;

if: 2 > 1 [ return:3; ]
  else [ return:2; ]

while: a < 6 [
  a = a + 1; ]

char b z;
print: b;
```

The following output is displayed when the compilation is attempted:

```
Fatal error: exception Failure("z does not exist or is not visible")
/usr/lib/gcc/x86_64-unknown-linux-gnu/5.3.0/../../../../lib/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

The compiler recognizes that the error is an undefined variable “z” and alerts the user of this.

The following program ensures that our functions were recursive; that is, that a function could call itself. It calculates the greatest common denominator of the given arguments.

```
gcd.tree:
$stdio.tree$

gcd : any args [
  int a args->0;
  int b args->1;
  int c 0;
  while : a != 0 [
          c = a;
          a = b%a;
          b = c;
  ]
```

```
  return:b;
]

print:"Testing iterative gcd with 65 and 195\n";

print:[gcd:[65 195] "\n"];


gcdr : any args [
  int a args->0;
  int b args->1;

  if : a == 0 [
        return:b;
  ]

  return:gcdr:[b%a a];
]

print:"Testing recursive gcd with 14 and 21\n";

print:[gcdr:[14 21] "\n"];


gcd.c (compiled code):
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>



struct List {
  struct node *head;
  struct node *tail;
};

struct node {
  void *data;
  struct node *next;
};

void init_list(struct List *list);

void traverse_list(struct List *list, void (*f)(void *));

void free_list(struct List *list);
```

```
void *get(struct List *list, int i);

void append(struct List *list, void *data);

void init_list(struct List *list) {
  list->head = NULL;
  list->tail = NULL;
}

void *get(struct List *list, int n) {
  int i = 0;
  struct node *curr;

  curr = list->head;

  while (i < n && curr != NULL) {
        curr = curr->next;
        i++;
  }

  return curr->data;
}




void traverse_list(struct List *list, void (*f)(void *))
{
  //Start at the beginning
  struct node *curr = list->head;

  //Iterate until we get to the end of the list
  while (curr != NULL)
  {
        f(curr->data);

        //Move to next node
        curr = curr->next;
  }
}


void free_list(struct List *list) {
  while (list->head != NULL) {
        struct node *curr = list->head;
        struct node *next = curr->next;
        free(curr);
        list->head = next;
```

```
  }
  free(list);
}



void append(struct List *list, void *data) {
  struct node *curr;
  struct node *new;

  curr = list->tail;

  if (curr == NULL) {
        curr = malloc(sizeof(struct node));
        curr->next = NULL;
        curr->data = data;
        list->head = curr;
        list->tail = curr;
  } else {
        new = malloc(sizeof(struct node));

        new->data = data;

        new->next = NULL;

        curr->next = new;

        list->tail = new;
  }
}


typedef enum {INT, CHAR, DOUBLE, BOOL, VOID, TREE} data_type;

union data_u {
  int i;
  char c;
  double d;
  struct tree *t;
};

struct tree {
  data_type type;
  union data_u data;
  int width;
  int refcount;
  struct List *children;
};

void put_t (struct tree *t);
```

```
struct tree *get_width_t(struct tree *t);

void init_tree (struct tree *root);

void free_tree(struct tree *t);

struct tree* inc_refcount(struct tree *t);
struct tree* dec_refcount(struct tree *t);

int equal(struct tree *lhs, struct tree *rhs);
int nequal(struct tree *lhs, struct tree *rhs);
int lt(struct tree *lhs, struct tree *rhs);
int gt(struct tree *lhs, struct tree *rhs);
int lte(struct tree *lhs, struct tree *rhs);
int gte(struct tree *lhs, struct tree *rhs);

struct tree *sub(struct tree *lhs, struct tree *rhs);
struct tree *add(struct tree *lhs, struct tree *rhs);
struct tree *mult(struct tree *lhs, struct tree *rhs);
struct tree *divd(struct tree *lhs, struct tree *rhs);
struct tree *mod(struct tree *lhs, struct tree *rhs);

struct tree *int_treemake(int i_data, struct tree *child, ...);
struct tree *char_treemake(char c_data, struct tree *child, ...);
struct tree *double_treemake(int d_data, struct tree *child, ...);
struct tree *tree_treemake(struct tree *t_data, struct tree *child, ...);
struct tree *void_treemake(struct tree *child, ...);

struct tree *treemake(data_type type, union data_u data, struct tree *child, va_list args);
int add_sibling (struct tree *root, struct tree *sibling, int n);
int add_child (struct tree *root, struct tree *child);
void set_type (struct tree *root, data_type type);
data_type get_type(struct tree *root);
struct tree *get_ith_sibling(struct tree *root, int i);
struct tree *get_branch_t(struct tree *root, struct tree *branch);
struct tree *get_branch(struct tree *root, int branch);


void put_t(struct tree *str) {

  switch(str->type) {
        case CHAR:
                putchar(str->data.c);
                break;
        case INT:
                printf("%d", str->data.i);
                break;
        case DOUBLE:
                printf("%f", str->data.d);
```

```
                break;
        case TREE:
                put_t(str->data.t);
                break;
        default:
                break;
  }
}

struct tree *get_width_t(struct tree *t) {
  return int_treemake(t->width, NULL);
}


int equal(struct tree *lhs, struct tree *rhs) {

  int retval;
  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = 0;
  } else {
        switch(lhs->type) {
                case CHAR:
                        retval = lhs->data.c == rhs->data.c;
                        break;
                case INT:
                        retval = lhs->data.i == rhs->data.i;
                        break;
                case DOUBLE:
                        retval = lhs->data.d == rhs->data.d;
                        break;
                default:
                        retval = 0;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
  return retval;
}

int nequal(struct tree *lhs, struct tree *rhs) {
  return !equal(lhs, rhs);
}
int lt(struct tree *lhs, struct tree *rhs) {
  int retval;
  inc_refcount(lhs);
  inc_refcount(rhs);
```

```
  if (lhs->type != rhs->type) {
        fprintf(stderr, "TYPE MISMATCH!\n");
        retval = 0;
  } else {
        switch(lhs->type) {
              case CHAR:
                    retval = lhs->data.c < rhs->data.c;
                    break;
              case INT:
                    retval = lhs->data.i < rhs->data.i;
                    break;
              case DOUBLE:
                    retval = lhs->data.d < rhs->data.d;
                    break;
              default:
                    retval = 0;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
  return retval;
}
int gt(struct tree *lhs, struct tree *rhs) {
  int retval;
  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = 0;
  } else {
        switch(lhs->type) {
              case CHAR:
                    retval = lhs->data.c > rhs->data.c;
                    break;
              case INT:
                    retval = lhs->data.i > rhs->data.i;
                    break;
              case DOUBLE:
                    retval = lhs->data.d > rhs->data.d;
                    break;
              default:
                    retval = 0;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
  return retval;
```

```
}
int lte(struct tree *lhs, struct tree *rhs) {
  return gt(rhs, lhs);
}
int gte(struct tree *lhs, struct tree *rhs) {
  return lt(rhs, lhs);
}
void dec_refcount_child(void *child) {
  dec_refcount((struct tree *)child);
}

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

  traverse_list(t->children, dec_refcount_child);
  free_list(t->children);
  if (t->type == TREE) {
        dec_refcount(t->data.t);
  }
  free(t);
}



struct tree* inc_refcount(struct tree *t) {
  if (t) {
        t->refcount++;
  }

  return t;
}

struct tree *dec_refcount(struct tree *t) {
  if (t) {
        if (--(t->refcount) <= 0) {
              free_tree(t);
              t = NULL;
        }
  }

  return t;
}

struct tree *sub(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);
```

```
  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c - rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i - rhs->data.i, NULL);
                    break;
              case DOUBLE:
                    retval =  double_treemake(lhs->data.d - rhs->data.d, NULL);
                    break;
              default:
                    retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;
}
struct tree *add(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c + rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i + rhs->data.i, NULL);
                    break;
              case DOUBLE:
                    retval =  double_treemake(lhs->data.d + rhs->data.d, NULL);
                    break;
              default:
                    retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
```

```
  return retval;
}

struct tree *mult(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c * rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i * rhs->data.i, NULL);
                    break;
              case DOUBLE:
                    retval =  double_treemake(lhs->data.d * rhs->data.d, NULL);
                    break;
              default:
                    retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;

}

struct tree *divd(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c / rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i / rhs->data.i, NULL);
```

```
                        break;
                case DOUBLE:
                        retval =  double_treemake(lhs->data.d / rhs->data.d, NULL);
                        break;
                default:
                        retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;
}

struct tree *mod(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
                case CHAR:
                        retval = char_treemake(lhs->data.c % rhs->data.c, NULL);
                        break;
                case INT:
                        retval =  int_treemake(lhs->data.i % rhs->data.i, NULL);
                        break;
                default:
                        retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;
}
void init_tree(struct tree *root) {
  root->children = malloc(sizeof(struct List));
  root->type = VOID;
  root->width = 0;
  root->refcount = 0;
  init_list(root->children);
}

struct tree *int_treemake(int i_data, struct tree *child, ...) {
```

```
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.i = i_data;
  root = treemake(INT, data, child, args);
  va_end(args);


  return root;
}


struct tree *char_treemake(char c_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.c = c_data;
  root = treemake(CHAR, data, child, args);
  va_end(args);

  return root;
}


struct tree *double_treemake(int d_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.d = d_data;
  root = treemake(DOUBLE, data, child, args);
  va_end(args);

  return root;
}

struct tree *tree_treemake(struct tree *t_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.t = t_data;
  root = treemake(TREE, data, child, args);
  va_end(args);
```

```
  return root;
}

struct tree *void_treemake(struct tree *child, ...) {

  va_list args;
  struct tree *root;
  union data_u data;

  data.t = NULL;

  va_start(args, child);
  root = treemake(VOID, data, child, args);
  va_end(args);

  return root;
}

struct tree *treemake(data_type type, union data_u data, struct tree *child,  va_list args) {
  struct tree *root = malloc(sizeof(struct tree));

  init_tree(root);
  root->type = type;
  root->data = data;


  while (child != NULL) {
          add_child(root, child);
          inc_refcount(child);
          child = va_arg(args, struct tree *);
  }

  return root;
}

int add_child (struct tree *root, struct tree *child) {
  if (root == NULL) {
          return -1;
  }
  inc_refcount(child);
  append(root->children, child);
  root->width += 1;
  return root->width;
}

void set_type (struct tree *root, data_type t) {
  root->type = t;
}
```

```
data_type get_type(struct tree *t) {
  return t->type;
}

struct tree *get_branch_t(struct tree *root, struct tree *branch) {
  struct tree *child;
  inc_refcount(branch);
  child = (struct tree *)get_branch(root, branch->data.i);
  dec_refcount(branch);
  return child;
}


struct tree *get_branch(struct tree *root, int branch) {
  return (struct tree *)get(root->children, branch);
}

struct tree *print(struct tree *data){inc_refcount(data);
  struct tree * n = get_width_t(data); inc_refcount(n);;
  struct tree * i = int_treemake(0, NULL); inc_refcount(i);;
  put_t(
        data);
  i = int_treemake(0, NULL); inc_refcount(i);;
  while (lt(i, n)) {
        print(
              get_branch_t(data, i));
        i = add(i, int_treemake(1, NULL)); inc_refcount(i);;

  };
  return data;;
return NULL;}
struct tree *pretty_print(struct tree *t){inc_refcount(t);
  struct tree * num_tabs = get_branch_t(t, int_treemake(0, NULL)); inc_refcount(num_tabs);;
  struct tree * data = get_branch_t(t, int_treemake(1, NULL)); inc_refcount(data);;
  struct tree * n = get_width_t(data); inc_refcount(n);;
  struct tree * i = int_treemake(0, NULL); inc_refcount(i);;
  if (gt(num_tabs, int_treemake(0, NULL))) {
        while (lt(i, num_tabs)) {
              print(
                    void_treemake(
                          char_treemake('\t', NULL),
                          NULL));
              i = add(i, int_treemake(1, NULL)); inc_refcount(i);;

        };

  };
  put_t(
        data);
```

```
  print(
        void_treemake(
              char_treemake('\n', NULL),
              NULL));
  i = int_treemake(0, NULL); inc_refcount(i);;
  while (lt(i, n)) {
        pretty_print(
              void_treemake(
                    add(num_tabs, int_treemake(1, NULL)),
                    get_branch_t(data, i),
                    NULL));
        i = add(i, int_treemake(1, NULL)); inc_refcount(i);;

  };
  return void_treemake(NULL);;
return NULL;}
struct tree *gcd(struct tree *args){inc_refcount(args);
  struct tree * a = get_branch_t(args, int_treemake(0, NULL)); inc_refcount(a);;
  struct tree * b = get_branch_t(args, int_treemake(1, NULL)); inc_refcount(b);;
  struct tree * c = int_treemake(0, NULL); inc_refcount(c);;
  while (nequal(a, int_treemake(0, NULL))) {
        c = a; inc_refcount(c);;
        a = mod(b, a); inc_refcount(a);;
        b = c; inc_refcount(b);;

  };
  return b;;
return NULL;}
struct tree *gcdr(struct tree *args){inc_refcount(args);
  struct tree * a = get_branch_t(args, int_treemake(0, NULL)); inc_refcount(a);;
  struct tree * b = get_branch_t(args, int_treemake(1, NULL)); inc_refcount(b);;
  if (equal(a, int_treemake(0, NULL))) {
        return b;;

  };
  return gcdr(
        void_treemake(
              mod(b, a),
              a,
              NULL));;
return NULL;}
int main(int argc, char **argv) {
;
;
;
  print(
        void_treemake(
              char_treemake('T', NULL),
              char_treemake('e', NULL),
              char_treemake('s', NULL),
```

```
                char_treemake('t', NULL),
                char_treemake('i', NULL),
                char_treemake('n', NULL),
                char_treemake('g', NULL),
                char_treemake(' ', NULL),
                char_treemake('i', NULL),
                char_treemake('t', NULL),
                char_treemake('e', NULL),
                char_treemake('r', NULL),
                char_treemake('a', NULL),
                char_treemake('t', NULL),
                char_treemake('i', NULL),
                char_treemake('v', NULL),
                char_treemake('e', NULL),
                char_treemake(' ', NULL),
                char_treemake('g', NULL),
                char_treemake('c', NULL),
                char_treemake('d', NULL),
                char_treemake(' ', NULL),
                char_treemake('w', NULL),
                char_treemake('i', NULL),
                char_treemake('t', NULL),
                char_treemake('h', NULL),
                char_treemake(' ', NULL),
                char_treemake('6', NULL),
                char_treemake('5', NULL),
                char_treemake(' ', NULL),
                char_treemake('a', NULL),
                char_treemake('n', NULL),
                char_treemake('d', NULL),
                char_treemake(' ', NULL),
                char_treemake('1', NULL),
                char_treemake('9', NULL),
                char_treemake('5', NULL),
                char_treemake('\n', NULL),
                NULL));
  print(
        void_treemake(
                gcd(
                        void_treemake(
                                int_treemake(65, NULL),
                                int_treemake(195, NULL),
                                NULL)),
                void_treemake(
                        char_treemake('\n', NULL),
                        NULL),
                NULL));
;
  print(
        void_treemake(
                char_treemake('T', NULL),
```

```
                char_treemake('e', NULL),
                char_treemake('s', NULL),
                char_treemake('t', NULL),
                char_treemake('i', NULL),
                char_treemake('n', NULL),
                char_treemake('g', NULL),
                char_treemake(' ', NULL),
                char_treemake('r', NULL),
                char_treemake('e', NULL),
                char_treemake('c', NULL),
                char_treemake('u', NULL),
                char_treemake('r', NULL),
                char_treemake('s', NULL),
                char_treemake('i', NULL),
                char_treemake('v', NULL),
                char_treemake('e', NULL),
                char_treemake(' ', NULL),
                char_treemake('g', NULL),
                char_treemake('c', NULL),
                char_treemake('d', NULL),
                char_treemake(' ', NULL),
                char_treemake('w', NULL),
                char_treemake('i', NULL),
                char_treemake('t', NULL),
                char_treemake('h', NULL),
                char_treemake(' ', NULL),
                char_treemake('1', NULL),
                char_treemake('4', NULL),
                char_treemake(' ', NULL),
                char_treemake('a', NULL),
                char_treemake('n', NULL),
                char_treemake('d', NULL),
                char_treemake(' ', NULL),
                char_treemake('2', NULL),
                char_treemake('1', NULL),
                char_treemake('\n', NULL),
                NULL));
  print(
        void_treemake(
                gcdr(
                        void_treemake(
                                int_treemake(14, NULL),
                                int_treemake(21, NULL),
                                NULL)),
                void_treemake(
                        char_treemake('\n', NULL),
                        NULL),
                NULL));
  return 0;
}
```

Program output:
Testing iterative gcd with 65 and 195
65
Testing recursive gcd with 14 and 21
7

Lessons Learned

Jacob Graff

Our project could have benefitted from a more complete model of our translator from the beginning - before any code was written. We wrote our scanner, parser, and ast without a clear goal of where we would end up, which I think caused our final code to be somewhat complicated to meet with older expectations (for example, the import preprocessor could have been made part of the original lexer/parser).

One thing that our language specifically would have benefitted from would be building from the idea of trees from the ground up; that is to say, write our parser to process trees specifically, and from there develop the concept of trees to write programs. Instead, our parser was originally built to process imperative programming, and the building of trees is not perfect, both in terms of syntax and in terms of representation. For this reason, I think that for our program, the “Hello world” requirement may not have been the best approach (although I think that in general, it is a good intermediate step and seems like a good requirement for the PLT class). Had we focused more on our tree structure and less on the programming language aspect, we could have had a more robust tree implementation.

In terms of group dynamics, we could have benefitted from more regularly scheduled group meetings. It is difficult to plan meetings for four people on the fly, and so a more strict regularly scheduled time to meet might have been better.

Our coding definitely benefitted a lot from the use of git. Different, somewhat incompatible pieces of code can be developed simultaneously, through the use of branches, and a long commit history is very helpful for figuring out how code was developed. In addition, our messaging platform, Slack, allows github integration. This is helpful because we get messages after each commit, so we knew when any given task for the project was completed.

Overall, this was a very interesting project, and for the first time for any of us writing a compiler I think it went well. I am now confident that I can do well in whatever future compiler related classes or tasks I may undertake.

Finally, one decision I made, taking CS Theory concurrently with PLT, was very helpful. I was learning the theory in CS Theory a few classes before learning a practical application of the theory in PLT, and this meant that it was usually fresh on my mind. Furthermore, it reinforced the idea that theory was very useful, and helped me better retain the ideas.

Shruti Kulkarni

Overall I am rather pleased with how our team handled the project. Our group dynamics worked out pretty well for the most part. We took seriously Prof. Stephen Edwards’ warning that we must start early on our project, and that we should make sure to hold ourselves and each other accountable for our work. That said, there is definitely room for improvement, and we shall certainly benefit from learning from what we could have done better as well as what we did right. We cooperated fairly well as a group, and

we aimed to communicate well also. But perhaps our communication could have improved, particularly around hectic periods (such as midterm exams week) and holidays when we weren't all available at the usual times. Fortunately, even when communication was temporarily delayed or disrupted, eventually we all got back in touch and ensured that we stayed on track with our work throughout. While we trieWe used Slack and Google for online communication, in addition to mobile phones (text and voice calls), and I think that worked rather well for us. We used Git for version control and that was very helpful.

Personally, it would have been helpful to review Computer Science Theory more thoroughly prior to and while taking this class (Programming Languages and Translators, abbreviated PLT), as it had been a while since I had taken the class and studied the subject by the time I had enrolled in this class. The material is directly relevant to our coursework as Computer Science Theory covered the theoretical foundation and this programming language project covered the practical application of it. At least one of our teammates was actually enrolled in Computer Science Theory concurrently and found it quite helpful, and I would recommend to other students who aim to take this class, and/or to engage in a project such as this (writing a programming language) to study and/or review Computer Science Theory concurrently with or immediately before doing so.

Luis 'Bert' Ramirez

This project was very new for everyone on my team and although I believe we worked well together, there were a number of improvements we could have made. The segmenting of roles worked out well and we used all available technologies to improve our workflow, such as git and Slack messaging. However, the fragmented structure made some development difficult and I believe we could have benefitted from more meetings. Regardless, a git-tracked text document tracking the progress worked well for communicating language changes. This way, the new language additions were brought to my attention quickly so I could design tests.

The testing was a much larger responsibility than I realized. Unit tests are necessary to isolate problems quickly but this was my first encounter with designing them. It proved to be challenging but the rapid communication of our online services helped me stay on top of all the additions. I now realize the challenges of low level programming, which is something I had a hard time understanding before taking on this project.

Overall, this project was something very new and not something I would have attempted at any point outside of class. It really took me out of my comfort zone but I got a deeper understanding of why languages are designed the way they are, and I also learned the importance of early, iterative testing.

Justin Walters

It was not until the final two weeks when everything was coming together that I realized how strange and lacking our language is. The root node of a tree is only accessible by builtin operators and the print function, trees are incredibly difficult to modify, and applying a function to every node or leaf is much harder than it should be. We succeeded in designing and compiling a language that works, but it is strange.

In the first weeks, we brainstormed as a group some of the things that should and should not be included and we built it incrementally. However, we never sat down and checked for consistency more methodically, which I see now is something I should have taken lead on as Language Guru. If we had done this regularly, we could have

greatly improved the language. Instead, we only looked to see if something made sense as we implemented it, when it was already intertwined with everything that preceded it.

We started early as repeatedly advised, so it never felt like we weren't going to finish in time. However, our meetings went well at the beginning, but it was harder to get everyone together as the semester went on. We used Slack so there was some communication throughout the week, but more regular meetings would have been helpful. When we did have meetings, we all should have all had an idea of what to discuss. As it was, the meetings were either short because we didn't remember what we needed to talk about, or went longer than necessary because we weren't focused. We really should have had an agenda before hand.

Overall, I am pleased with our process and our results. We did the main things we wanted to do and compromised intelligently.

Code Listing

==> ast.ml <==

```
type vtype =
  Int | Char | Double | Bool | String | Any

type expr =
  Tree of expr * expr list
| IntLit of string
| ChrLit of string
| FltLit of string
| StrLit of string
| GetBranch of expr * expr
| Void
| FunCall of string * expr
| Eq of expr * expr
| Neq of expr * expr
| Lt of expr * expr
| Leq of expr * expr
| Gt of expr * expr
| Geq of expr * expr
| Add of expr * expr
| Minus of expr * expr
| Mul of expr * expr
| Div of expr * expr
| Mod of expr * expr
| Cast of vtype * expr
| GetWidth of expr
| Id of string

type stmt =
  While of expr * stmt
| If of expr * stmt
| IfElse of expr * stmt * stmt
| FuncDec of string * vtype * string * stmt
| VarDec of vtype * string * expr
| Assn of string * expr
```

```
|   Expr of expr
|   Return of expr
|   Seq of stmt list

type program = stmt list

let string_of_vtype = function
   Int -> "int"
| Char -> "char"
| Double -> "double"
| Bool -> "bool"
| String -> "string"
| Any -> "any"

let rec string_of_expr = function
   Tree(e,l) -> string_of_expr e ^ " {{ " ^
            String.concat ", " (List.map string_of_expr l) ^ "}}"
| IntLit(s) -> s
| ChrLit(s) -> s
| FltLit(s) -> s
| StrLit(s) -> s
| GetBranch(e1,e2) -> string_of_expr e1 ^ "." ^ string_of_expr e2
| GetWidth(e1) -> string_of_expr e1
| Void -> "void"
| FunCall(s,e) -> s ^ "(" ^ string_of_expr e ^ ")"
| Eq(e1, e2) -> string_of_expr e1 ^ "==" ^ string_of_expr e2
| Neq(e1, e2) -> string_of_expr e1 ^ "!=" ^ string_of_expr e2
| Lt(e1, e2) -> string_of_expr e1 ^ "<" ^ string_of_expr e2
| Leq(e1, e2) -> string_of_expr e1 ^ "<=" ^ string_of_expr e2
| Gt(e1, e2) -> string_of_expr e1 ^ ">" ^ string_of_expr e2
| Geq(e1, e2) -> string_of_expr e1 ^ ">=" ^ string_of_expr e2
| Add(e1, e2) -> string_of_expr e1 ^ "+" ^ string_of_expr e2
| Minus(e1, e2) -> string_of_expr e1 ^ "-" ^ string_of_expr e2
| Mul(e1, e2) -> string_of_expr e1 ^ "*" ^ string_of_expr e2
| Div(e1, e2) -> string_of_expr e1 ^ "/" ^ string_of_expr e2
| Mod(e1, e2) -> string_of_expr e1 ^ "%" ^ string_of_expr e2
| Cast(vt,e) -> string_of_vtype vt ^ " to " ^ string_of_expr e
| Id(s) -> s

(* mixing c-like and pltree-like syntax. not on purpose *)
let rec string_of_stmt = function
   While(e,s) -> "While(" ^ string_of_expr e ^ ") {" ^ string_of_stmt s ^ "}"
| If(e,s) -> "If(" ^ string_of_expr e ^ ") {" ^ string_of_stmt s ^ "}"
| IfElse(e,s1,s2) -> "If(" ^ string_of_expr e ^ ") {" ^ string_of_stmt s1 ^ "} Else {" ^ string_of_stmt s2 ^ "}"
| FuncDec(s,vt,vn,l) -> s ^ "[" ^ string_of_stmt l ^ "]"
| VarDec(v,s,e) -> s ^ " = " ^ string_of_expr e
| Assn(s,e) -> s ^ " = " ^ string_of_expr e
| Expr(e) -> string_of_expr e
| Return(e) -> "Return(" ^ string_of_expr e ^ ")"
| Seq(l) ->  String.concat ", " (List.map string_of_stmt l)

let string_of_program stmts =
 String.concat " " (List.map string_of_stmt stmts)
```

==> compile.ml <==

```
open Ast

module StringMap = Map.Make(String)

type env = {
  functions: Sast.vtype StringMap.t;
  globals: (Sast.expr * Sast.vtype) StringMap.t;
  locals: (Sast.expr * Sast.vtype) StringMap.t;
  statements: string StringMap.t;
}

(* prints the variables of an environment. For debugging. Could print funcs *)
let print_maps env =
  let printtf = (fun key (expr,t) -> Printf.printf "%s -> " key;
          print_endline (Sast.string_of_expr expr ^ " of " ^ Sast.string_of_vtype t)) in
  let printvals = (fun key value -> printtf key value) in
  print_string "globals:";
  StringMap.iter printvals env.globals;
  print_string "\nlocals:";
  StringMap.iter printvals env.locals; print_string "\n"
let avt_to_svt = function
  Ast.Int -> Sast.Int
|   Ast.Double -> Sast.Double
|   Ast.String -> Sast.String
|   Ast.Char -> Sast.Char
|   Ast.Any    -> Sast.Any
|   Ast.Bool -> Sast.Bool
let matching t1 t2 = t1 == t2 || t1 == Sast.Any || t2 == Sast.Any
let translate prog =
  let rec add_all m = function
          [] -> m
          | (name,vtype)::tl -> add_all (StringMap.add name vtype m) tl in
  let builtins = add_all StringMap.empty [("put_t",Sast.Any)] in

  let empty_env = {
          functions = builtins;
          globals = StringMap.empty;
          locals = StringMap.empty;
          statements = StringMap.empty } in

  (* If there is a key in both maps, keeps value from first arg *)
  let merge_maps =
          let f = (fun k xopt yopt -> match xopt, yopt with Some x, _ -> xopt
                  | None, yo -> yopt ) in StringMap.merge f in

  (* Extracts vardecs from a list of Sast.stmts *)
  let rec get_vars_list = function
          Sast.Seq([]) -> []
          | Sast.Seq(hd::tl) -> (match hd with Sast.VarDec(_,_,_) ->
                  let l = Sast.Seq(tl) in hd::get_vars_list l
                  | _ -> get_vars_list(Sast.Seq(tl)))
          | _ -> [] in
```

```
let rec get_rets_list = function
	Sast.Seq([]) -> []
	| Sast.Seq(hd::tl) -> (match hd with Sast.Return(_,t) ->
		let l = Sast.Seq(tl) in t::get_rets_list l
		| Sast.While(_,seq,_) -> let l = Sast.Seq(tl) in
			List.concat [get_rets_list seq; get_rets_list l]
		| Sast.If(_,seq,_) -> let l = Sast.Seq(tl) in
			List.concat [get_rets_list seq; get_rets_list l]
		| Sast.IfElse(_,seq,seq2,_) -> let l = Sast.Seq(tl) in
			List.concat [get_rets_list seq; get_rets_list seq2; get_rets_list l]
		| _ -> get_rets_list(Sast.Seq(tl)))
	| _ -> [] in

let ret_types seq = let typeL = get_rets_list seq in let head = List.hd typeL in
	List.for_all (fun t -> matching t head) typeL in

(* environment -> Ast.expr -> (Sast.expr, Sast.vtype) *)
let rec expr env = function
Tree(e,l) -> let l = List.map (fun e -> let (e,_) = expr env e in e) l in
	let (e,t) = expr env e in Sast.Tree(e,l), t
| IntLit(s) -> Sast.IntLit(s), Sast.Int
| ChrLit(s) -> Sast.ChrLit(s), Sast.Char
| FltLit(s) -> Sast.FltLit(s), Sast.Double
| StrLit(s) -> Sast.StrLit(s), Sast.String
| Void -> Sast.Void, Sast.Any
| GetBranch(e1,e2) ->
	let (se2,st) = expr env e2 in (match st with Sast.Int ->
		let (se1,t) = expr env e1 in Sast.GetBranch(se1,se2), t
	| _ -> raise(Failure("Can only access branches with an int")))
| GetWidth(e1) ->
	let (se1, st) = expr env e1 in Sast.GetWidth(se1), st
| FunCall(s,e) -> if StringMap.mem s env.functions then
let vt = StringMap.find s env.functions in
let (e,t) = expr env e in if (vt == Sast.Any || t == vt) then Sast.FunCall(s,e), Sast.Any
else raise(Failure(s ^ " expects an argument of type " ^
	Sast.string_of_vtype vt ^ ", not " ^ Sast.string_of_vtype t))
else raise(Failure(s ^ " does not exist or is not visible"))
| Eq(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then Sast.Eq(e1,e2), Sast.Bool else
let type_string = Sast.string_of_vtype t1 ^ " == " ^ Sast.string_of_vtype t2 in
raise (Failure("Different types: " ^ type_string))
| Neq(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then Sast.Neq(e1,e2), Sast.Bool else
let type_string = Sast.string_of_vtype t1 ^ " != " ^ Sast.string_of_vtype t2 in
raise (Failure("Different types: " ^ type_string))
| Lt(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then Sast.Lt(e1,e2), Sast.Bool else raise (Failure("Different types"))
| Leq(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then Sast.Leq(e1,e2), Sast.Bool else raise (Failure("Different types"))
| Gt(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then Sast.Gt(e1,e2), Sast.Bool else raise (Failure("Different types"))
| Geq(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then Sast.Geq(e1,e2), Sast.Bool else raise (Failure("Different types"))

(* Allow arithmetic on chars? *)
```

```
| Add(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then match t1 with Sast.Int | Sast.Any | Sast.Double -> Sast.Add(e1,e2), t1
        | _ -> raise(Failure("Addition operands must be of type int or double"))
else let type_string = Sast.string_of_vtype t1 ^ " + " ^ Sast.string_of_vtype t2 in
raise (Failure("Different types: " ^ type_string))
| Minus(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then match t1 with Sast.Int | Sast.Any | Sast.Double -> Sast.Minus(e1,e2), t1
        | _ -> raise(Failure("Subtraction operands must be of type int or double"))
else let type_string = Sast.string_of_vtype t1 ^ " - " ^ Sast.string_of_vtype t2 in
raise (Failure("Different types: " ^ type_string))
| Mul(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then match t1 with Sast.Int | Sast.Any | Sast.Double -> Sast.Mul(e1,e2), t1
        | _ -> raise(Failure("Multiplication operands must be of type int or double"))
else let type_string = Sast.string_of_vtype t1 ^ " * " ^ Sast.string_of_vtype t2 in
raise (Failure("Different types: " ^ type_string))
| Div(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then match t1 with Sast.Int | Sast.Any | Sast.Double -> Sast.Div(e1,e2), t1
        | _ -> raise(Failure("Divison operands must be of type int or double"))
else raise (Failure("Different types"))
| Mod(e1, e2) -> let (e1,t1) = expr env e1 in let (e2,t2) = expr env e2 in
if (matching t1 t2) then match t1 with Sast.Int | Sast.Any | Sast.Double -> Sast.Mod(e1,e2), t1
        | _ -> raise(Failure("Mod operands must be of type int or double"))
else raise (Failure("Different types"))
| Cast(vt, e) -> let (e,_) = expr env e in let vt = avt_to_svt vt in
Sast.Cast(vt, e), vt
| Id(s) -> if StringMap.mem s env.locals then
        let (e1, e2) = StringMap.find s env.locals in (Sast.Id(s), e2)
else if StringMap.mem s env.globals then
        let (e1, e2) = StringMap.find s env.globals in (Sast.Id(s), e2)
else raise(Failure(s ^ " does not exist or is not visible")) in

(* environment -> Ast.stmt -> (environment, Sast.stmt) *)
let rec transform_stmt env = function
        (* Change Ast stuff to Sast and keep track of vars new to this scope *)
        While(e,seq) -> env, let (e,t) = expr env e in
        if t = Sast.Bool then let locs = merge_maps env.locals env.globals in
        let env = {env with globals=locs; locals=StringMap.empty} in
        let (_,s) = transform_stmt env seq in let vars = get_vars_list s in
        Sast.While(e,s,vars)
        else raise(Failure("While predicates must be of type bool"))
| If(e,seq) -> env, let (e,t) = expr env e in
        if t = Sast.Bool then let locs = merge_maps env.locals env.globals in
        let env = {env with globals=locs; locals=StringMap.empty} in
        let (_,s) = transform_stmt env seq in let vars = get_vars_list s in
        Sast.If(e,s,vars)
        else raise(Failure("If predicates must be of type bool"))
| IfElse(e,seq,seq2) -> env, let (e,t) = expr env e in
        if t = Sast.Bool then let locs = merge_maps env.locals env.globals in
        let env = {env with globals=locs; locals=StringMap.empty} in
        let (_,s) = transform_stmt env seq in let vars = get_vars_list s in
        let (_,s2) = transform_stmt env seq2 in let vars2 = get_vars_list s2 in
        let allvars = List.concat [vars ; vars2] in
        Sast.IfElse(e,s,s2,allvars)
        else raise(Failure("If predicates must be of type bool"))
| FuncDec(s, vt, vn, seq) -> let sexp = (match vt with
                                        Int -> Sast.IntLit("0"), Sast.Int
```

```
                                        |   Char -> Sast.ChrLit("0"), Sast.Char
                                        |   Double -> Sast.FltLit("0.0"), Sast.Double
                                        |   String -> Sast.StrLit("0"), Sast.String
                                        |   Any -> Sast.IntLit("0"), Sast.Any
                                        |   Bool -> Sast.IntLit("0"), Sast.Any) in
            if StringMap.mem s env.functions then raise(Failure(s ^ " is already declared"))
            else let locs = StringMap.add vn sexp StringMap.empty in
            let svt (_, vt) = vt in
            let funcs = StringMap.add s (svt sexp) env.functions in
            let newenv = {env with globals=StringMap.empty; locals=locs; functions=funcs} in
            let (_,seq) = transform_stmt newenv seq in let vars = get_vars_list seq in
            let sameRetTs = ret_types seq in if sameRetTs then {env with functions=funcs},
            Sast.FuncDec(s, (svt sexp), vn, seq,vars) else
            raise(Failure(s ^ " must have consistent return types")) (* TODO *)
    | VarDec(vt,s,e) -> if StringMap.mem s env.locals then
    raise (Failure (s ^ " is already declared")) else let (r,t) = expr env e in
    let locs = StringMap.add s (r,avt_to_svt vt) env.locals in {env with locals=locs},
    Sast.VarDec(avt_to_svt vt,s,r)
    | Assn(s,e) -> let (eSast,tSast) = if StringMap.mem s env.locals then
    StringMap.find s env.locals else if StringMap.mem s env.globals then
    StringMap.find s env.globals else raise (Failure(s ^
            " has not been declared"))
    in let (r,t) = expr env e in if tSast = t then
    let locs = StringMap.add s (r,t) env.locals in
    {env with locals=locs}, Sast.Assn(s, r) else
    raise(Failure(s ^ " is defined as " ^ Sast.string_of_vtype tSast ^
            ", not " ^ Sast.string_of_vtype t))

    | Expr(e) -> env, let (e,_) = expr env e in Sast.Expr(e)
    | Return(e) -> env, let (e,t) = expr env e in Sast.Return(e,t)
    | Seq(l) -> let (env,l) = map_stmts env l in env, Sast.Seq(List.rev l) and

    (* environment -> Ast.stmt list -> (environment, Sast.stmt list) *)
    (* Maps ast stmts to sast stmts, passing environment along *)
    map_stmts env stmts =
            List.fold_left (fun (env, m) stmt ->
                    let (e,s) = transform_stmt env stmt in
                    let mapped = s::m in e, mapped) (env, []) stmts in

    let (env, transformed) = map_stmts empty_env prog in

    (* print_maps env; *)

    List.rev transformed
```

==> iast.ml <==

```
type import = Open of string | Other of char | String of string

type program = import list
```

==> pltree.ml <==

```
type action = Ast | Compile
```

```
let _ =
  if Array.length Sys.argv > 2 then
            let infile = Sys.argv.(1) in
            let outfile = Sys.argv.(2) in
            let tempfile = infile ^ ".tmp" in
            Imports.check_imports infile tempfile;
            let out = open_out outfile in
            let lexbuf = Lexing.from_channel (open_in tempfile) in
            let program = Parser.program Scanner.token lexbuf in
            Execute.execute_prog (Compile.translate program) out
  else raise(Failure("Must provide input and output file"))
(*
 let action = if Array.length Sys.argv > 1 then
           List.assoc Sys.argv.(1) [ ("-a", Ast);
                                     ("-c", Compile) ]
 else Compile in
 let lexbuf = Lexing.from_channel stdin in
 let program = Parser.program Scanner.token lexbuf in
 match action with
           Ast -> let listing = Ast.string_of_program program
           in print_endline listing
 (* | Bytecode -> let listing =
           Bytecode.string_of_prog (Compile.translate program)
           in print_endline listing *)
 | Compile -> Execute.execute_prog (Compile.translate program)
*)
```

==> read_tree.ml <==

```
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let result = eval program in
  print_endline result;
```

==> stubs.ml <==

```
let tree_c =
"
typedef enum {INT, CHAR, DOUBLE, BOOL, VOID, TREE} data_type;

union data_u {
  int i;
  char c;
  double d;
  struct tree *t;
};
```

```
struct tree {
  data_type type;
  union data_u data;
  int width;
  int refcount;
  struct List *children;
};

void put_t (struct tree *t);

struct tree *get_width_t(struct tree *t);

void init_tree (struct tree *root);

void free_tree(struct tree *t);

struct tree* inc_refcount(struct tree *t);
struct tree* dec_refcount(struct tree *t);

int equal(struct tree *lhs, struct tree *rhs);
int nequal(struct tree *lhs, struct tree *rhs);
int lt(struct tree *lhs, struct tree *rhs);
int gt(struct tree *lhs, struct tree *rhs);
int lte(struct tree *lhs, struct tree *rhs);
int gte(struct tree *lhs, struct tree *rhs);

struct tree *sub(struct tree *lhs, struct tree *rhs);
struct tree *add(struct tree *lhs, struct tree *rhs);
struct tree *mult(struct tree *lhs, struct tree *rhs);
struct tree *divd(struct tree *lhs, struct tree *rhs);
struct tree *mod(struct tree *lhs, struct tree *rhs);

struct tree *int_treemake(int i_data, struct tree *child, ...);
struct tree *char_treemake(char c_data, struct tree *child, ...);
struct tree *double_treemake(double d_data, struct tree *child, ...);
struct tree *tree_treemake(struct tree *t_data, struct tree *child, ...);
struct tree *void_treemake(struct tree *child, ...);

struct tree *treemake(data_type type, union data_u data, struct tree *child, va_list args);
int add_sibling (struct tree *root, struct tree *sibling, int n);
int add_child (struct tree *root, struct tree *child);
void set_type (struct tree *root, data_type type);
data_type get_type(struct tree *root);
struct tree *get_ith_sibling(struct tree *root, int i);
struct tree *get_branch_t(struct tree *root, struct tree *branch);
struct tree *get_branch(struct tree *root, int branch);



void put_t(struct tree *str) {

  switch(str->type) {
          case CHAR:
```

```
                putchar(str->data.c);
                break;
        case INT:
                printf(\"%d\", str->data.i);
                break;
        case DOUBLE:
                printf(\"%f\", str->data.d);
                break;
        case TREE:
                put_t(str->data.t);
                break;
        default:
                break;
  }
}

struct tree *get_width_t(struct tree *t) {
  return int_treemake(t->width, NULL);
}


int equal(struct tree *lhs, struct tree *rhs) {

  int retval;
  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = 0;
  } else {
        switch(lhs->type) {
                case CHAR:
                        retval = lhs->data.c == rhs->data.c;
                        break;
                case INT:
                        retval = lhs->data.i == rhs->data.i;
                        break;
                case DOUBLE:
                        retval = lhs->data.d == rhs->data.d;
                        break;
                default:
                        retval = 0;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
  return retval;
}

int nequal(struct tree *lhs, struct tree *rhs) {
  return !equal(lhs, rhs);
}
int lt(struct tree *lhs, struct tree *rhs) {
```

```
  int retval;
  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
          fprintf(stderr, \"TYPE MISMATCH!\\n\");
          retval = 0;
  } else {
          switch(lhs->type) {
                  case CHAR:
                          retval = lhs->data.c < rhs->data.c;
                          break;
                  case INT:
                          retval = lhs->data.i < rhs->data.i;
                          break;
                  case DOUBLE:
                          retval = lhs->data.d < rhs->data.d;
                          break;
                  default:
                          retval = 0;
          }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
  return retval;
}
int gt(struct tree *lhs, struct tree *rhs) {
  int retval;
  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
          retval = 0;
  } else {
          switch(lhs->type) {
                  case CHAR:
                          retval = lhs->data.c > rhs->data.c;
                          break;
                  case INT:
                          retval = lhs->data.i > rhs->data.i;
                          break;
                  case DOUBLE:
                          retval = lhs->data.d > rhs->data.d;
                          break;
                  default:
                          retval = 0;
          }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);
  return retval;
}
int lte(struct tree *lhs, struct tree *rhs) {
```

```
  return gt(rhs, lhs);
}
int gte(struct tree *lhs, struct tree *rhs) {
  return lt(rhs, lhs);
}
void dec_refcount_child(void *child) {
  dec_refcount((struct tree *)child);
}

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

  traverse_list(t->children, dec_refcount_child);
  free_list(t->children);
  if (t->type == TREE) {
        dec_refcount(t->data.t);
  }
  free(t);
}



struct tree* inc_refcount(struct tree *t) {
  if (t) {
        t->refcount++;
  }

  return t;
}

struct tree *dec_refcount(struct tree *t) {
  if (t) {
        if (--(t->refcount) <= 0) {
              free_tree(t);
              t = NULL;
        }
  }

  return t;
}

struct tree *sub(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
```

```
                retval = char_treemake(lhs->data.c - rhs->data.c, NULL);
                break;
            case INT:
                retval =  int_treemake(lhs->data.i - rhs->data.i, NULL);
                break;
            case DOUBLE:
                retval =  double_treemake(lhs->data.d - rhs->data.d, NULL);
                break;
            default:
                retval =  NULL;
        }
    }

    dec_refcount(rhs);
    dec_refcount(lhs);

    return retval;
}
struct tree *add(struct tree *lhs, struct tree *rhs) {
    struct tree *retval;

    inc_refcount(lhs);
    inc_refcount(rhs);

    if (lhs->type != rhs->type) {
        retval = NULL;
    } else {
        switch (lhs->type) {
            case CHAR:
                retval = char_treemake(lhs->data.c + rhs->data.c, NULL);
                break;
            case INT:
                retval =  int_treemake(lhs->data.i + rhs->data.i, NULL);
                break;
            case DOUBLE:
                retval =  double_treemake(lhs->data.d + rhs->data.d, NULL);
                break;
            default:
                retval =  NULL;
        }
    }

    dec_refcount(rhs);
    dec_refcount(lhs);

    return retval;
}

struct tree *mult(struct tree *lhs, struct tree *rhs) {
    struct tree *retval;

    inc_refcount(lhs);
    inc_refcount(rhs);
```

```
  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c * rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i * rhs->data.i, NULL);
                    break;
              case DOUBLE:
                    retval =  double_treemake(lhs->data.d * rhs->data.d, NULL);
                    break;
              default:
                    retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;

}

struct tree *divd(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c / rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i / rhs->data.i, NULL);
                    break;
              case DOUBLE:
                    retval =  double_treemake(lhs->data.d / rhs->data.d, NULL);
                    break;
              default:
                    retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;
}
```

```
struct tree *mod(struct tree *lhs, struct tree *rhs) {
  struct tree *retval;

  inc_refcount(lhs);
  inc_refcount(rhs);

  if (lhs->type != rhs->type) {
        retval = NULL;
  } else {
        switch (lhs->type) {
              case CHAR:
                    retval = char_treemake(lhs->data.c % rhs->data.c, NULL);
                    break;
              case INT:
                    retval =  int_treemake(lhs->data.i % rhs->data.i, NULL);
                    break;
              default:
                    retval =  NULL;
        }
  }

  dec_refcount(rhs);
  dec_refcount(lhs);

  return retval;
}
void init_tree(struct tree *root) {
  root->children = malloc(sizeof(struct List));
  root->type = VOID;
  root->width = 0;
  root->refcount = 0;
  init_list(root->children);
}

struct tree *int_treemake(int i_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.i = i_data;
  root = treemake(INT, data, child, args);
  va_end(args);


  return root;
}


struct tree *char_treemake(char c_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;
```

```
  va_start(args, child);
  data.c = c_data;
  root = treemake(CHAR, data, child, args);
  va_end(args);

  return root;
}


struct tree *double_treemake(double d_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.d = d_data;
  root = treemake(DOUBLE, data, child, args);
  va_end(args);

  return root;
}

struct tree *tree_treemake(struct tree *t_data, struct tree *child, ...) {
  va_list args;
  union data_u data;
  struct tree *root;

  va_start(args, child);
  data.t = t_data;
  root = treemake(TREE, data, child, args);
  va_end(args);

  return root;
}

struct tree *void_treemake(struct tree *child, ...) {

  va_list args;
  struct tree *root;
  union data_u data;

  data.t = NULL;

  va_start(args, child);
  root = treemake(VOID, data, child, args);
  va_end(args);

  return root;
}

struct tree *treemake(data_type type, union data_u data, struct tree *child,  va_list args) {
  struct tree *root = malloc(sizeof(struct tree));
```

```
  init_tree(root);
  root->type = type;
  root->data = data;


  while (child != NULL) {
        add_child(root, child);
        inc_refcount(child);
        child = va_arg(args, struct tree *);
  }

  return root;
}

int add_child (struct tree *root, struct tree *child) {
  if (root == NULL) {
        return -1;
  }
  inc_refcount(child);
  append(root->children, child);
  root->width += 1;
  return root->width;
}

void set_type (struct tree *root, data_type t) {
  root->type = t;
}

data_type get_type(struct tree *t) {
  return t->type;
}

struct tree *get_branch_t(struct tree *root, struct tree *branch) {
  struct tree *child;
  inc_refcount(branch);
  child = (struct tree *)get_branch(root, branch->data.i);
  dec_refcount(branch);
  return child;
}


struct tree *get_branch(struct tree *root, int branch) {
  return (struct tree *)get(root->children, branch);
}
"


let ll_c =
"


struct List {
  struct node *head;
  struct node *tail;
};
```

```
struct node {
  void *data;
  struct node *next;
};

void init_list(struct List *list);

void traverse_list(struct List *list, void (*f)(void *));

void free_list(struct List *list);

void *get(struct List *list, int i);

void append(struct List *list, void *data);

void init_list(struct List *list) {
  list->head = NULL;
  list->tail = NULL;
}

void *get(struct List *list, int n) {
  int i = 0;
  struct node *curr;

  curr = list->head;

  while (i < n && curr != NULL) {
        curr = curr->next;
        i++;
  }

  return curr->data;
}




void traverse_list(struct List *list, void (*f)(void *))
{
  //Start at the beginning
  struct node *curr = list->head;

  //Iterate until we get to the end of the list
  while (curr != NULL)
  {
        f(curr->data);

        //Move to next node
        curr = curr->next;
  }
}
```

```
void free_list(struct List *list) {
  while (list->head != NULL) {
        struct node *curr = list->head;
        struct node *next = curr->next;
        free(curr);
        list->head = next;
  }
  free(list);
}



void append(struct List *list, void *data) {
  struct node *curr;
  struct node *new;

  curr = list->tail;

  if (curr == NULL) {
        curr = malloc(sizeof(struct node));
        curr->next = NULL;
        curr->data = data;
        list->head = curr;
        list->tail = curr;
  } else {
        new = malloc(sizeof(struct node));

        new->data = data;

        new->next = NULL;

        curr->next = new;

        list->tail = new;
  }
}
"
```

==> cast.ml <==

```
type vtype =
  Int | Char | Double | Bool | String | Any

type expr =
  Tree of expr * expr list
|   IntLit of string
|   ChrLit of string
|   FltLit of string
|   StrLit of string
|   GetBranch of expr * expr
|   GetWidth of expr
|   Void
|   FunCall of string * expr
|   Eq of expr * expr
```

```
|   Neq of expr * expr
|   Lt of expr * expr
|   Leq of expr * expr
|   Gt of expr * expr
|   Geq of expr * expr
|   Add of expr * expr
|   Minus of expr * expr
|   Mul of expr * expr
|   Div of expr * expr
|   Mod of expr * expr
|   Cast of vtype * expr
|   Id of string

type stmt =
  While of expr * stmt * stmt list
|   If of expr * stmt * stmt list
|   IfElse of expr * stmt * stmt * stmt list
|   FuncDec of string * vtype * string * stmt * stmt list
|   VarDec of vtype * string * expr
|   Assn of string * expr
|   Expr of expr
|   Return of expr
|   Seq of stmt list

type program = stmt list

let string_of_vtype = function
  Int -> "int"
| Char -> "char"
| Double -> "double"
| Bool -> "bool"
| String -> "string"
| Any -> "any"

let rec string_tab n v = if n == 0 then v else string_tab (n-1) ("\t" ^ v)

let rec tree_list_from_string =
  function s ->
        let len = String.length s in
        if (len > 0) then
                Tree(ChrLit("'" ^ (Char.escaped (String.get s 0)) ^ "'") , []):: 
                tree_list_from_string (String.sub s 1 (len - 1))
        else
                []

let rec gen_c_expr =
  let rec gen_c_tree_list = function n -> function
        hd::tl -> "" ^ (gen_c_expr n hd) ^ ", \n" ^ string_tab n (gen_c_tree_list n tl)
  |   [] -> ""
  in
  function n ->
  function
  FunCall(x, y) -> ("" ^ x ^ "(\n" ^ string_tab (n + 1) (gen_c_expr (n+1) y) ^ ")" )
|   Tree(IntLit(x), children) ->
        "int_treemake(" ^ x ^ ", " ^
```

```
                if List.length children == 0 then
                        "NULL)"
                else
                        "\n" ^ string_tab (n+1) (gen_c_tree_list (n+1) children ^ "NULL)")
|   Tree(ChrLit(x), children) ->
        "char_treemake(" ^ x ^ ", " ^
                if List.length children == 0 then
                        "NULL)"
                else
                        "\n" ^ string_tab (n+1) (gen_c_tree_list (n+1) children ^ "NULL)")
|   Tree(FltLit(x), children) ->
        "double_treemake(" ^ x ^ ", " ^
                if List.length children == 0 then
                        "NULL)"
                else
                        "\n" ^ string_tab (n+1) (gen_c_tree_list (n+1) children ^ "NULL)")
|   Tree(Void, children) ->
        "void_treemake(" ^
                if List.length children == 0 then
                        "NULL)"
                else
                        "\n" ^ string_tab (n+1) (gen_c_tree_list (n+1) children ^ "NULL)")
|   Tree(StrLit(x), []) -> "void_treemake(\n" ^
                string_tab (n+1) (gen_c_tree_list (n+1) (tree_list_from_string x) ^ "NULL)")
|   Tree(expr, children) ->
        "tree_treemake(" ^ gen_c_expr n expr ^ ", " ^
                if List.length children == 0 then
                        "NULL)"
                else
                        "\n" ^ string_tab (n+1) (gen_c_tree_list (n+1) children ^ "NULL)")
| GetBranch(tree, expr) -> "get_branch_t(" ^ gen_c_expr n tree ^ ", " ^ gen_c_expr n expr ^ ")"
|   GetWidth(tree) -> "get_width_t(" ^ gen_c_expr n tree ^ ")"
|   Eq(v1, v2) -> ("equal(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Neq(v1, v2) -> ("nequal(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Lt(v1, v2) -> ("lt(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Leq(v1, v2) -> ("lte(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Gt(v1, v2) -> ("gt(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Geq(v1, v2) -> ("gte(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Add(v1, v2) -> ("add(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Minus(v1, v2) -> ("sub(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Mul(v1, v2) -> ("mult(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Div(v1, v2) -> ("divd(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Mod(v1, v2) -> ("mod(" ^ gen_c_expr n v1 ^ ", " ^ gen_c_expr n v2 ^ ")")
|   Cast(vt,e) -> ("cast(" ^ string_of_vtype vt ^ ", " ^ gen_c_expr n e ^ ")")
|   Id(v1) -> "" ^ v1
|   IntLit(x) -> x
|   ChrLit(x) -> x
|   FltLit(x) -> x
|   StrLit(x) -> x
|   Void -> ""

let rec gen_c = function n -> function
  While(x, y, l) -> string_tab n ("while ("^(gen_c_expr n x) ^ ") { \n" ^ (gen_c (n+1) y) ^ "\n" ^  string_tab
n "}")
|   If(x, y, l) -> string_tab n ("if ("^(gen_c_expr n x) ^ ") { \n" ^ (gen_c (n+1) y) ^ "\n" ^  string_tab n "}")
|   IfElse(x, s1,s2, l) ->
```

```
        string_tab n ("if ("^(gen_c_expr n x) ^ ") { \n" ^ (gen_c (n+1) s1) ^ "\n" ^  string_tab n "} else
{\n"
        ^ (gen_c (n+1) s2) ^ "\n" ^ string_tab n "}")
|   VarDec(v1, v2, v3) -> string_tab n ("struct tree * " ^ v2 ^ " = " ^ gen_c_expr n v3 ^ "; inc_refcount(" ^
v2 ^ ");")
|   FuncDec(str, vt, vn, stmt, l) -> "" (*str ^ "(){\n" ^ gen_c (n + 1) stmt ^ "} " *)
|   Assn(v1, v2) -> string_tab n ("" ^ v1 ^ " = " ^ gen_c_expr n v2 ^ "; inc_refcount(" ^ v1 ^ ");")
|   Expr(v1) -> string_tab n (gen_c_expr n v1)
|   Return(v1) -> string_tab n ("return " ^ gen_c_expr n v1 ^ ";")
|   Seq(v1) -> let rec gen_c_seq = function
                        hd::tl -> gen_c n hd ^ ";\n" ^ gen_c_seq tl
                |   [] -> "" in
        gen_c_seq v1

let rec eval_expr = function n ->
  let rec eval_tree_list = function n -> function
        hd::tl -> eval_expr n hd ^ "::\n" ^ string_tab (n) (eval_tree_list n tl)
  |   [] -> "[]"
  in
  function
  FunCall(x, y) -> ("FunCall(" ^ x ^ ", \n" ^ string_tab (n+1) ((eval_expr (n+1) y) ^ ")"))
|   Tree(StrLit(x), []) -> "Tree(Void, \n" ^ string_tab (n+1) (eval_tree_list (n+1) (tree_list_from_string x) ^
")")
|   Tree(expr, children) ->
        "Tree(" ^ eval_expr n expr ^
                if List.length children == 0 then
                        ", [])"
                else
                        ", \n" ^ string_tab (n+1) (eval_tree_list (n+1) children ^ ")")
|   GetBranch(tree, expr) -> "" (* TODO *)
|   GetWidth(tree) -> ""
|   Eq(v1, v2) -> ("Eq(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Neq(v1, v2) -> ("Neq(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Lt(v1, v2) -> ("Lt(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Leq(v1, v2) -> ("Leq(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Gt(v1, v2) -> ("Gt(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Geq(v1, v2) -> ("Geq(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Add(v1, v2) -> ("Add(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Minus(v1, v2) -> ("Minus(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Mul(v1, v2) -> ("Mul(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Div(v1, v2) -> ("Div(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Mod(v1, v2) -> ("Mod(" ^ eval_expr n v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Cast(vt,e) -> ("Cast(" ^ string_of_vtype vt ^ ", " ^ eval_expr n e ^ ")")
|   Id(v1) -> "Id(" ^ v1 ^ ")"
|   IntLit(x) -> x
|   ChrLit(x) -> x
|   FltLit(x) -> x
|   StrLit(x) -> x
|   Void -> "Void"

let rec eval = function n -> function
  While(x, y, l) -> string_tab n ("While("^(eval_expr n x) ^ ",\n" ^ (eval (n+1) y) ^ "\n" ^  string_tab n ")")
|   If(x, y, l) -> string_tab n ("If("^(eval_expr n x) ^ ",\n" ^ (eval (n+1) y) ^ "\n" ^  string_tab n ")")
|   IfElse(x, s1, s2, l) -> string_tab n ("IfElse("^(eval_expr n x) ^ ",\n" ^
                (eval (n+1) s1) ^ "\n" ^  string_tab n ")\n" ^
```

```
                    (eval (n+1) s2) ^ "\n" ^  string_tab n ")")
|   VarDec(v1, v2, v3) -> string_tab n ("VarDec(" ^ v2 ^ ", " ^ eval_expr n v3 ^ ")")
|   FuncDec(str, vt, vn, stmt, l) -> "" (* TODO *)
|   Assn(v1, v2) -> string_tab n ("Assn(" ^ v1 ^ ", " ^ eval_expr n v2 ^ ")")
|   Expr(v1) -> string_tab n (eval_expr n v1)
|   Return(v1) -> string_tab n ("Return(" ^ eval_expr n v1 ^ ")")
|   Seq(v1) -> let rec eval_seq = function
                                hd::tl -> eval n hd ^ "\n" ^ eval_seq tl
                    |   [] -> "" in
            eval_seq v1
let rec eval_prog = function
  hd::tl -> "" ^ eval 0 hd ^ "\n" ^ eval_prog tl
|   [] -> ""


let rec gen_c_prog = function
  hd::tl -> "" ^ gen_c 1 hd ^ ";\n" ^ gen_c_prog tl
|   [] -> ""



let rec gen_c_funcs = function
  FuncDec(str, vt, vn, stmt, l)::tl -> "struct tree *" ^ str ^ "(struct tree *" ^ vn ^ "){" ^
  "inc_refcount(" ^ vn ^ ");\n" ^ gen_c 1 stmt ^ "return NULL;}\n" ^ gen_c_funcs tl
|   _::tl -> gen_c_funcs tl
|   [] -> ""

let headers =
  "#include <stdio.h>\n#include <stdlib.h>\n" ^
  "#include <stdio.h>\n" ^
  "#include <stdlib.h>\n" ^
  "#include <string.h>\n" ^
  "#include <stdarg.h>\n"

let string_of_program stmts =
  "int main(int argc, char **argv) {\n" ^
  gen_c_prog stmts ^
  "\treturn 0;\n}"
```

==> execute.ml <==

```
open Sast

type prog_els = { variables: Cast.stmt list;
  functions: Cast.stmt list;
  all_statements: Cast.stmt list; }

(* turns S-ast to C-ast*)
let c_vtype = function s_vt -> (match s_vt with
  Sast.Int -> Cast.Int
|   Sast.Char -> Cast.Char
|   Sast.Double -> Cast.Double
|   Sast.Bool -> Cast.Bool
|   Sast.String -> Cast.String
|   Sast.Any -> Cast.Any)
let transform prog =
```

```
  let rec expr = function
  Tree(e,l) -> let l = List.map (fun e -> expr e) l in Cast.Tree(expr e,l)
  | IntLit(s) -> Cast.IntLit(s)
  | ChrLit(s) -> Cast.ChrLit(s)
  | FltLit(s) -> Cast.FltLit(s)
  | StrLit(s) -> Cast.StrLit(s)
  | GetBranch(e1,e2) -> Cast.GetBranch(expr e1, expr e2)
  | GetWidth(e1) -> Cast.GetWidth(expr e1)
  | Void -> Cast.Void
  | FunCall(s,e) -> Cast.FunCall(s, expr e)
  | Eq(e1,e2) -> Cast.Eq(expr e1,expr e2)
  | Neq(e1,e2) -> Cast.Neq(expr e1,expr e2)
  | Lt(e1,e2) -> Cast.Lt(expr e1,expr e2)
  | Leq(e1,e2) -> Cast.Leq(expr e1,expr e2)
  | Gt(e1,e2) -> Cast.Gt(expr e1,expr e2)
  | Geq(e1,e2) -> Cast.Geq(expr e1,expr e2)
  | Add(e1,e2) -> Cast.Add(expr e1,expr e2)
  | Minus(e1,e2) -> Cast.Minus(expr e1,expr e2)
  | Mul(e1,e2) -> Cast.Mul(expr e1,expr e2)
  | Div(e1,e2) -> Cast.Div(expr e1,expr e2)
  | Mod(e1,e2) -> Cast.Mod(expr e1,expr e2)
  | Cast(t,e) -> Cast.Cast(c_vtype t, expr e)
  | Id(s) -> Cast.Id(s) in

  let rec stmt = function
  While(e,s,l) -> let l = List.map (fun e -> stmt e) l in
  Cast.While(expr e, stmt s, l)
  | If(e,s,l) -> let l = List.map (fun e -> stmt e) l in
  Cast.If(expr e, stmt s, l)
  | IfElse(e,s,s2,l) -> let l = List.map (fun e -> stmt e) l in
  Cast.IfElse(expr e, stmt s, stmt s2, l)
  | FuncDec(s, vt, vn, seq,l) -> let l = List.map (fun e -> stmt e) l in
  Cast.FuncDec(s, (c_vtype vt), vn, stmt seq, l)
  | VarDec(t,s,e) -> Cast.VarDec((c_vtype t),s,expr e)
  | Assn(s,e) -> Cast.Assn(s,expr e)
  | Expr(e) -> Cast.Expr(expr e)
  | Return(e,t) -> Cast.Return(expr e)
  | Seq(l) -> let l = List.map (fun s -> stmt s) l in
  Cast.Seq(l) in

  (* Get all funcs and top-level vars *)
  let vars = Sast.get_vars_list prog in
  let funcs = Sast.get_funcs_list prog in {
  variables = List.map (fun v -> stmt v) vars;
  functions = List.map (fun f -> stmt f) funcs;
  all_statements = List.map    (fun s -> stmt s) prog; }

(* Use the string_of funcs in cast.ml to make C file *)
let execute_prog prog outfile = let pe = transform prog in
  Printf.fprintf outfile "%s"
  (
          Cast.headers ^ "\n" ^
          Stubs.ll_c ^ "\n" ^
          Stubs.tree_c ^ "\n" ^
          Cast.gen_c_funcs pe.functions ^
```

```
        Cast.string_of_program pe.all_statements)
```

==> imports.ml <==

```
open Printf

let stdlibdir = "/usr/local/bin/pltree_std/"

let rec lex_file filename outfile opened_list =
    let lexbuf = Lexing.from_channel (
        try open_in filename with
            e -> open_in (Filename.concat stdlibdir filename)
  ) in
  let program = Iparser.program Iscanner.token lexbuf in
  let rec print_program =
        function outfile ->
        function l ->
        function
        Iast.Open(s)::tl ->     if List.exists (function a -> s = a) l then (
                                    print_program outfile l tl
                                ) else (
                                    print_program outfile (lex_file s outfile (s::l)) tl
                                )
  |   Iast.Other(c)::tl -> fprintf outfile "%c" c; print_program outfile l tl
  |   Iast.String(s)::tl -> fprintf outfile "%s" ("\"" ^ s ^ "\""); print_program outfile l tl
  |   [] -> l in
  print_program outfile opened_list program


let check_imports infilename outfilename =
  let out = open_out outfilename in
  ignore (lex_file infilename out []); close_out out
```

==> read.ml <==

```
open Ast

let rec eval = function
        Var(var, expr) -> let v1 = eval var and v2 = eval expr in
        print_endline (v1 ^ v2)
| Dec(func_dec, expr) -> let v1 = eval func_dec and v2 = eval expr in
        print_endline (v1 ^ v2)
| Call(func_call, expr) -> let v1 = eval func_call and v2 = eval expr in
        print_endline (v1 ^ v2)
| Asn(asn, expr) -> let v1 = eval asn and v2 = eval expr in
        print_endline (v1 ^ v2)
| Lit(literal) -> literal
| Binop(e1, op, e2) ->
        let v1 = eval e1 and v2 = eval e2 in
        match op with
  Add -> v1 + v2
        | Sub -> v1 - v2
        | Mul -> v1 * v2
        | Div -> v1 / v2

let _ =
```

```
 let lexbuf = Lexing.from_channel stdin in
 let expr = Parser.expr Scanner.token lexbuf in
 let result = eval expr in
 print_endline (string_of_int result ^ " ")
 (* concatenate with space to remove ^D output *)
```

==> sast.ml <==

```
type vtype =
  Int | Char | Double | Bool | String | Any

type expr =
  Tree of expr * expr list
|   IntLit of string
|   ChrLit of string
|   FltLit of string
|   StrLit of string
|   GetBranch of expr * expr
|   GetWidth of expr
|   Void
|   FunCall of string * expr
|   Eq of expr * expr
|   Neq of expr * expr
|   Lt of expr * expr
|   Leq of expr * expr
|   Gt of expr * expr
|   Geq of expr * expr
|   Add of expr * expr
|   Minus of expr * expr
|   Mul of expr * expr
|   Div of expr * expr
|   Mod of expr * expr
|   Cast of vtype * expr
|   Id of string

type stmt =
  While of expr * stmt * stmt list
|   If of expr * stmt * stmt list
|   IfElse of expr * stmt * stmt * stmt list
|   FuncDec of string * vtype * string * stmt * stmt list
|   VarDec of vtype * string * expr
|   Assn of string * expr
|   Expr of expr
|   Return of expr * vtype
|   Seq of stmt list

type program = stmt list

let rec get_vars_list = function
  [] -> []
  | hd::tl -> match hd with VarDec(_,_,_) -> hd::get_vars_list tl
            | _ -> get_vars_list tl

let rec get_funcs_list = function
  [] -> []
```

```
  | hd::tl -> match hd with FuncDec(_,_,_,Seq(l),_) -> List.concat
            [ get_funcs_list l; hd::get_funcs_list tl ]
            | While(_,Seq(l),_) -> List.concat
            [get_funcs_list l; get_funcs_list tl ]
            | If(_,Seq(l),_) -> List.concat
            [get_funcs_list l; get_funcs_list tl ]
            | IfElse(_,Seq(l),Seq(l2),_) -> List.concat
            [get_funcs_list l; get_funcs_list l2; get_funcs_list tl ]
            |_ -> get_funcs_list tl

let string_of_vtype = function
  Int -> "int"
| Char -> "char"
| Double -> "double"
| Bool -> "bool"
| String -> "string"
| Any -> "any"

let rec string_of_expr = function
  Tree(e,l) -> string_of_expr e ^ if List.length l > 0 then" {{ " ^
           String.concat ", " (List.map string_of_expr l) ^ "}}" else "{{}}"
| IntLit(s) -> s
| ChrLit(s) -> s
| FltLit(s) -> s
| StrLit(s) -> s
| GetBranch(e1,e2) -> string_of_expr e1 ^ "." ^ string_of_expr e2
| GetWidth(e1) -> string_of_expr e1
| Void -> "void"
| FunCall(s,e) -> s ^ "(" ^ string_of_expr e ^ ")"
| Eq(e1, e2) -> string_of_expr e1 ^ " == " ^ string_of_expr e2
| Neq(e1, e2) -> string_of_expr e1 ^ " != " ^ string_of_expr e2
| Lt(e1, e2) -> string_of_expr e1 ^ " < " ^ string_of_expr e2
| Leq(e1, e2) -> string_of_expr e1 ^ " <= " ^ string_of_expr e2
| Gt(e1, e2) -> string_of_expr e1 ^ " > " ^ string_of_expr e2
| Geq(e1, e2) -> string_of_expr e1 ^ " >= " ^ string_of_expr e2
| Add(e1, e2) -> string_of_expr e1 ^ " + " ^ string_of_expr e2
| Minus(e1, e2) -> string_of_expr e1 ^ " - " ^ string_of_expr e2
| Mul(e1, e2) -> string_of_expr e1 ^ " * " ^ string_of_expr e2
| Div(e1, e2) -> string_of_expr e1 ^ " / " ^ string_of_expr e2
| Mod(e1, e2) -> string_of_expr e1 ^ " % " ^ string_of_expr e2
| Cast(t1,e) -> string_of_vtype t1 ^ " from " ^ string_of_expr e
| Id(s) -> s

let rec string_of_stmt = function
  While(e,s,l) -> string_of_expr e ^ " " ^ string_of_stmt s
| If(e,s,l) -> string_of_expr e ^ " " ^ string_of_stmt s
| IfElse(e,s,s2,l) -> string_of_expr e ^ " " ^ string_of_stmt s ^ " " ^ string_of_stmt s2
| FuncDec(str,vt,vn,stmt,l) -> str ^ "[ " ^ string_of_stmt stmt ^ "] "
| VarDec(t,s,e) -> s ^ " " ^ string_of_expr e
| Assn(s,e) -> s ^ " = " ^ string_of_expr e
| Expr(e) -> string_of_expr e
| Return(e,t) -> "return:(" ^ string_of_expr e ^ ")";
| Seq(l) ->  String.concat ", " (List.map string_of_stmt l)

let string_of_program stmts =
```

```
String.concat " " (List.map string_of_stmt stmts)
```