Ripple

make waves

`https://github.com/RipplePLT/ripple`

Final Report

Project Manager:	Amar Dhingra	**asd2157**
System Architect:	Alexander Roth	**air2112**
System Integrator:	Artur Renault	**aur2103**
Language Guru:	Spencer Brown	**stb2132**
Tester & Verifier:	Thomas Segarra	**tas2172**

2015 – 05 – 09

Contents

Chapter 1

Introduction

1.1 Motivation

With the proliferation of devices that are constantly connected today, information becomes outdated faster than ever. Keeping up with rapidly changing data is therefore increasingly problematic in software development. Ripple aims to fulfill this need by enabling programmers to work natively with streams of external data, using a simple and intuitive syntax. Ripple will be a valuable tool in any field, but we expect that it will have especially important applications in statistics, scientific research, and finance.

1.2 Why Ripple?

Ripple is designed to make working with dynamic data easier. Data comes into a Ripple program through "streams" from external sources. Dealing with this data is straightforward in Ripple because the language is reactive: changes in a variable propagate, or "ripple," through the program to affect other variables linked to it. Variables that depend on external data are then as simple to deal with as variables defined within the program itself.

Consider a program that displays the current weather based on periodic updates from the internet. In most existing languages, the programmer would repeatedly need to:

1. create a socket,
2. connect to a server,
3. make an HTTP request,
4. receive a response,
5. parse the payload,
6. update the display to include the new information.

In Ripple, one only needs to open a stream to the web server, and then link the stream to a variable; the variable will automatically reflect all relevant changes. Ripple's value becomes clearer when we consider data that must be processed after it enters the program. In most cases, the data we receive is not exactly in the form we need, and we have to perform additional operations to derive real value from it. If, for example, the weather stream produced temperatures in Fahrenheit, we would need an additional derivation if we wanted to display it in Celsius. In Ripple, this is as simple as *linking* a Celsius variable to the weather stream, and applying the Fahrenheit-Celsius conversion formula. The direct correspondence between the coded variable definition and the mathematical formula makes the language eminently readable.

1.3 Hello, world!

This simple program begins by printing "Hello world!" Then it assigns `w` a `file_stream`, which takes the name of a file, an interval, and a delimiter. Every interval, it will read a string up to the next delimiter and

set the value of `w` to that string. On every change, it passes `w` as an argument to the auxiliary function, `say_hello`. Therefore, the program says hello to the world, and to every line in the file.

The `stop` keyword at the bottom simply halts the execution of the program until the user terminates it by pressing CTRL+C. This ensures that the program will not terminate while the file is being read.

```
1  void say_hello(string x) {
2      print("Hello", x);
3  }
4
5  void main() {
6      string filename = "hello.txt";
7
8      print("Hello, world");
9
10     string w;
11     link(w <- file_stream(filename, 1, "\n")) then say_hello;
12
13     stop;
14 }
```

hello.rpl

1.4 What is Ripple?

1.4.1 Ripple is reactive

In a typical imperative language, the statement `x = y + z` immediately sets x to the current sum of y and z; if `y` or `z` changes afterwards, `x` is not updated unless the programmer explicitly specifies when to do so. In a reactive language like Ripple, if the variables are *linked*, `x` updates every time either `y` or `z` changes. Thus, the programmer is not required to manually and repeatedly update related variables.

1.4.2 Ripple is connected

From our previous example, the values of `y` and `z` do not need to live within the program: both can be dynamically retrieved from a file, a device, or the Internet in the form of streams. Ripple abstracts away the socket and file I/O involved in these operations, eliminating a substantial amount of boilerplate. This encapsulation is achieved by providing a library of pre-defined `streams` for common input types, and a syntax that allows users to parse data read from the `stream` via . Developers will thus have the flexibility to handle a variety of input formats, turning data of any kind into a Ripple-compatible stream.

1.4.3 Ripple is simple

It only takes a single line of Ripple code to *link* a data stream to a variable, which would often be a long and cumbersome process in other languages. This syntax makes it easy to understand how changes in a single variable affect the state of the entire program, and dramatically simplifies what might otherwise be a prohibitively complex control flow. In addition, Ripple is statically-typed, ensuring compatibility between variables and the streams they are *linked* to, thereby minimizing unexpected behavior.

1.5 What else is out there?

Existing implementations of the reactive programming paradigm are platform- dependent. For example, Bacon.js and ELM rely on JavaScript, and were written to simplify the construction of user interfaces on the Web. ReactiveCocoa, to name another, is targeted only at iOS and OSX. Ripple, by contrast, is platform-independent. Any machine that can compile ISO C++11 code can compile Ripple. Similarly, most existing

reactive languages are designed specifically to react to a user's input; Ripple is built to react to anything for which a `stream` can be defined. This generalization significantly broadens the class of problems that can be solved by a reactive program.

1.6 Target Audience

Ripple targets developers with knowledge of a C-like programming language who work with dynamic information. We can see Ripple being used in fields ranging from data science, statistics, natural language processing, machine learning or any other field where analyzing data over time is important.

1.7 Syntax

Ripple is meant to be simple to pick up to anyone who has experience programming in any C-like language. Thus, Ripple retains the familiar control flow statements from C, such as `if-else` statements, `for`-loops, and `while`-loops. Similarly, it retains a subset of all the standard data primitives from C, with the addition of `string` type for basic string manipulation and to avoid the complexity that comes with pointers. The **link** keyword creates relationships between variables and either:

1. data streams, or
2. other variables through *chaining*

Chaining is the process in which we link one variable to n other variables, such that there cannot exist a loop between any pair of variables. That is, the program creates a minimum-spanning tree of connections for the variables, where the root variable is the first linked variable, and the leafs are the last linked variables.

To illustrate this point, suppose we have the program:

```
1  void main(){
2  ...
3      int x = 5;
4      int y;
5      int z;
6      int q;
7      int v;
8
9      link (y -> x + 2);
10     link (z -> y + 10);
11     link (v -> y + 9);
12     link (q -> x - 5);
13 ...
14 }
```

sample.rpl

In this program, we have initialized the integer variable `x` with a value of 5. Afterwards, we define four more integer variables: `y`, `z`, `q`, and `v`. We begin linking varaibles on line 9. The compiler automatically links the variable `y` to the expression `x + 2`. Similar to line 9, the next three line statements will create a dependency graph referred to as a `link tree` which we use to update the value of variables when the value of any intermediate node changes. The **link tree** for this code is shown in Figure 1.1.

All links are hierarchical and uni-directional in nature. For example, if the value of `x` from our previous example were to change, all variables that are "descendents" of `x` will be updated.

Furthermore, link statements with any type of `stream` need not initialize a root node for the link tree. Since all `stream`s read in volatile external data, the compiler is able to infer that the variable linked to the `stream` will be the child of the specified `stream`.

Link statements are syntactically similar to `if`-statements; thus, allowing programmers to link a specific variable to a data stream or to another variable. Developers can also provide functions to be executed every time the left hand side variable is updated.

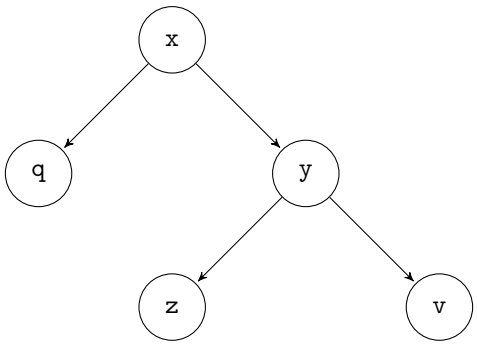


Figure 1.1: The link tree for *sample.rpl*

Ripple implements three basic `stream`s: the `keyboard_stream`, which reads data from the keyboard, the `file_stream`, which parses a file, and the `web_stream`, which fetches data from the Internet. Furthermore, Ripple allows a user to use a filter function to parse the input before it is linked to a variable, or an auxiliary function, which is executed on the input anytime it is updated.

Functions are defined like standard C-like functions, using typical C-like notation. Every function can take a variable number of arguments, and uses the keyword `return` with a data type to represent what the function returns. If there is no returns statement present, then the function returns void.

Chapter 2

Tutorial

2.1 Introduction

Our introduction to Ripple is focused on getting you, the developer, quickly up to speed with the structure and usage of Ripple. While this introduction is not as thorough as the reference manual, we hope that after reading this you will be prepared to write clean reactive programs.

Before we begin, we need to answer the question: What is Ripple? Ripple is an imperative language that implements the reactive programming paradigm. A discussion on reactive programming, along with example use cases, can be found later in this tutorial.

Ripple closely resembles code from languages in the C family, but is more focused on one specific problem than its peers; Ripple focuses on the problem of creating programs that intuitively interact with dynamic data. As such, the most important aspect of Ripple to note from this tutorial is that:

- Ripple includes `link` statements for linking variables, or "streams of variables" together.

Please consult the language reference manual for a more in-depth explanation of Ripple and an exhaustive discussion of its syntax.

2.2 Getting Started

The first program we build in Ripple will be a simple one: it prints

`hello, world`

and then terminates.

So let's get started! In broad pattern, you must first write the program in text (using exclusively ASCII characters), compile it successfully, and run it.

In Ripple, the program to print "hello, world" is:

```
1 #~ A simple program to print hello world ~#
2 void main() {
3     print(``hello, world''); # prints 'hello, world'
4 }
```

hello.rpl

A Ripple program must have a ".rpl" file extension (for example, `hello.rpl`) so that it can then be compiled with the command

`./rpl hello.rpl`

As long as there are no compilation errors, the compiler will produce an executable file with the default name `output`. Programmers can choose the name of their output file by compiling with

`./rpl hello.rpl hello`

If you run `hello` by typing the command

```
./hello
```

This will print

```
hello, world
```

Now for some explanations about the program itself. A Ripple program, whatever its size, consists of *functions*, *variables*, and *whitespace*. Whitespace consists of spaces, tabs, newlines, and comments. Note the unique commenting style in Ripple: a block comment begins with "`#~`" and ends with "`~#`", while single line comments begin with "`#`" and continue until the end of the line. Comments allow your program to be read by humans, but do not affect the execution of the program. In our example, we have a function named *main*, which does not return a value, as specified by the use of the `void` type in the function declaration. As in C, the "`main`" function is a unique function. Every Ripple program will begin execution with the `main` function. Thus, every program must include a `main`.

`main` often calls functions written elsewhere, either within the same file or in external files. One such function is used in our `main` function:

```
print("hello, world");
```

This `print` function prints the given argument to standard output. A function is called by naming it and giving it a parenthesized list of arguments. Thus, this program calls `output` with the argument `‘‘hello, world\n’’`. `output` is a built-in function, so it is not necessary to `import` any outside library to use it.

A sequence of characters in double quotes, like `‘‘hello, world’’`, is called a *string*. In our example, the string `‘‘hello, world’’` is an argument that is passed into the function `output`.

Unlike the standard `C printf` function which does not automatically add a newline delimiter to the end of text being printed, Ripples `print` function appends a `\n` (newline) character to the end of the arguments provided. Any text placed to the right of a newline character will appear on the next line of the standard output.

2.3 Variables and Types

2.3.1 Variables

If a Ripple program is to perform any amount of substantial work, it is necessary to have a construct that stores and accesses values at different times in the lifetime of the program: these are called variables. Like most languages, all variables are declared before use. Each declaration consists of a type name and variable name.

```
string word;
```

In this example, a variable is being declared as a `string` type with the variable name `word`.

The variable name can be as short as one letter or as long as you wish. Variable names cannot start with a number. Type names are predefined by the compiler (See 2.3.2 Types). A variable is *initialized* when you give it a type and give it a value.

Initialization is done with the `=` operator, and occurs after or with declaration.

```
string name = "PLT";
```

Here, a variable is being declared as a `string` type with the variable name `name` and it is being assigned (`=`) to the value of the *string literal* `PLT`.

2.3.2 Types

Types are paramount to the structure of Ripple programs. A type sets the behavior for any given variable within the lifetime of a program. Types are also used to ensure that a function returns the proper value on successful completion. As such, it is important that every variable and function within a program is declared with a type. Any function or variable without a type name in its declaration will throw a compiler error.

There are two distinct classes of types in Ripple:

The Basic (Built-in) Types

1. `void`
2. `bool`
3. `int`
4. `float`
5. `string`

The Derived Types

1. `dataset`
2. `array`

`void`

The `void` type is similar to C's or C++'s `void` type. Any function that is defined as a `void` function does not return any value. Note that `void` uniquely only be used with functions; it cannot be used with a variable name.

```
void func() {...}
```

The above function has the function name `func` and returns nothing, since its type is `void`.

`bool`

The `bool` type is a boolean value that can only take two values: `true` and `false`. A function can return a `bool` value and a variable can be set to a `bool` value. Unlike in some other languages, there are no other "truthy" or "falsy" values. A `bool` is explicitly only the value `true` or `false`, and only `bool`s are true or false.

```
bool always_true() { bool t = true; return t;}
```

Here the function named `always_true` states that its return type will be a `bool` value. Within the function body the `bool` value `t` is set to `true`. It is then returned to the function caller with the statement `return t`. `bool`s can be used in evaluations of logical expressions as well, such as equivalency or range checking. There is no good reason to write a function that returns `true` or `false` without some form of evaluation.

`int`

An int in Ripple is effectively C++'s long type, and thus its size is machine- dependent, but maintains the guarantee that it is at least 8 bytes. `int`s can thus represent numbers within the range $-2^{31}+1$ to $+2^{31}-1$, but this range may differ from machine to machine. Despite this, it is important to note that in Ripple, like most other languages, the int type floors. `int`s handle all the standard mathematical operations with the respective operator.

`float`

Floats are effectively equivalent to C++'s double type, as in Ripple float variables generally support a range of numbers from $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$, and are 64 bits in size. However, a `float` is ultimately machine dependent for their actual size and numbers they can represent. `float`s handle all the standard mathematical operations with the respective operator.

`string`

The `string` type is similar to Java's `String` object. It is used to store literal values and is is delimited by double quotations (“ ”). A `string` can be of any length, with the smallest length being 0 (whose string is the empty string: “ ”). `string`s that are delimited with quotation marks are known as *string literals*, while variables declared with the type `string` are known as `string`s. Strings can be concatenated together using the `+` operator.

```
string prof = ‘‘Aho’’;
```

The variable `prof` is declared as a `string` with the string literal value of `‘‘Aho’’`.

`dataset`

The Ripple designers realize that it is not possible to provide every imaginable data type that the designer might require. Thus, the `dataset` type allows a Ripple developer to easily define their own data type, which are constructed from the other built-in data types. The `dataset` construct must be declared and implemented outside of any function implementation.

```
dataset team {
    string prof,
    string mentor,
    int team_sz,
}
```

This `dataset` named `team` which contains two string variables and one `int` variable named which could be assigned the values `‘‘Aho"`, `‘‘Chae"` and `5` respectively.

Arrays

An array is a data structure that allows you to hold multiple values of the same type. An array can hold any number of the same data types; the smallest array is the empty array – an array initialized with nothing.

Arrays have two built-in operations.

1. Length of an array, which is supported through the the use of the **@** operator.
2. An indexing operation, represented through the bracket (`[]`) operators. This operation retrieves the variable at the given address.

```
string[] mentors = { ‘‘Aho’’, ‘‘Chae’’ };
```

This statement creates a `string` array of size 2, named `mentors` with the initial values of `Aho` and `Chae`.

The statement

```
@mentors
```

will return the value of 2, which is the size of the `mentors` array.

The statement

```
mentors[1]
```

returns the `string` at index 1, which will return `Chae`. Arrays are ordinal when accessing; that is, the first index is represented by 0 and the last index is $n - 1$ for an n length array.

The statement

```
mentors[0] + ‘‘and ’’ + mentors[1]
```

returns a concatenation of three strings, which is `Aho and Chae`.

2.3.3 Example Program

Diving in, we will examine at a program that averages a set of grades and prints the result.

```
1  #~ A simple calculation of averages ~#
2  void main() {
3
4      float average;
5      int floored_average;
6      int total;
7      int i;
8
9      int[10] grades = {25, 30, 55, 75, 80, 80, 82, 85, 97, 100};
10     total = 0;
11
12     for(i = 0; i < 10; i = i + 1){ # loop to total grades
13         total = total + grades[i];
14     }
15     average = total / 10;
16     floored_average = total // 10;
17     print("Regular Average:\t", average);
18     print("Integer Division Average:\t", floored_average);
19 }
```

avg.rpl

After compiling *avg.rpl*, we can run the program's executable and see the following result:

```
$ rpl avg.rpl avg
$ ./output
Regular Average:    70.9
Integer Division Average:    70
```

2.4 Control Flow

Similar to its predecessors, Ripple provides the developer with the standard `if`, `else`, `for` and `while` control flow statements. To make the transition to writing Ripple code easier these statements remain relatively unchanged from the predecessor languages.

2.4.1 `if` and `else` statements

`if` and `else` statements are used for events where the developer wants to test some condition and perform one stream of execution if the condition is true and some other stream of execution if the condition is false. The condition to be tested must be a boolean or an expression that evaluates to a boolean. The `else` statement after each `if` statement is optional.

An example is

```
if (x and y) {
    # do something if both x and y are true
    } else {
    # do something if either x or y is false
}
```

2.4.2 `while` loop

In the event that a developer wants to execute a block of code repeatedly until some condition is met they would use a `while` loop. Developers provide the `while` loop with a boolean variable or statement to be evaluated. The boolean or statement is evaluated once at the start and once at the end of every loop and the loop is executed if it are true.

An example is

```
while ( x ) {
    # do something while x remains true
}
```

2.4.3 `for` loop

Equivalent in power to `while` loops, `for` loops provide a concise syntax to set a variable, test a condition and execute an expression in the statement declaration itself. The variable but be initialization before being used in the for loop. The variable is set once, before the start of the loop. Next, the conditional expression is tested at the start of the loop and for every subsequent iteration. Finally, after each iteration, the end expression is executed.

An example is

```
int x;
for (x = 0; x < 10; x = x + 1 ) {
    # do something while x is less than 10
}
```

2.4.4 `break` keyword

The `break` statement can be used to terminate a loop at any time during its execution. The program will continue execution at the instruction following the loop body.

An example is

```
while ( true ) {
    # breaking out of the infinite loop
    break;
}
```

2.4.5 `continue` keyword

The `continue` keyword causes the program to continue on to the next iteration of the loop. It starts by going back to the start of the loop, reevaluating the loop condition.

An example is

```
while ( true ) {
    # goes back to the start of the loop
    continue;
}
```

2.4.6 `stop` keyword

The `stop` keyword causes the program to stop at the current point of execution. The main thread will progress no further. This is to allow `streams` to continue running in the background when some long-running operation is not happening.

An example is

```
...
stop;
print("this should never print")
...
```

2.5 Functions and Arguments

Functions and their arguments are the bread and butter of any programming language, and are key to writing clean, concise code. Modularizing your code into functions is extremely important in Ripple in light of the nature of link statements, and the subsequent need to conceptualize the reactive nature of Ripple programs. In order for you to begin writing clean Ripple code, we will now discuss the nature of functions and their arguments in Ripple.

2.5.1 Functions

Mentioned at the start of this tutorial, but important to repeat again, is that all Ripple programs must contain a `main()` function of the type void, meaning that it returns nothing. This function will always be executed first within a Ripple program.

Moving on to a more general overview of functions, similar to `C`, functions must be declared before they can be used. Continuing, most functions in Ripple behave the same as C functions; however, several key differences exist. First and foremost, if a function is declared with the type void, it does not mean that it returns the type void, but merely that it returns nothing.

2.5.2 Arguments

Arguments are used to transfer data between functions. The parentheses after a function name can contain a list of arguments. The list of arguments within a function definition can be a variadic size (i.e. the list of arguments can be of an arbitrary length).

2.6 Scope

Scoping within Ripple is handled much like C's version of scoping; that is, *automatic* or *local* variables are created within a block and are removed after the block in which they were declared ends. This scoping rule also applies to any variables that are linked within a function.

Only a single variable with a particular name can be defined within a single scope block.

For example,

```
...
{
    int x = 4;
    int x = 5;
}
```

Within this code snippet, `x` is initialized twice: first to 4, then to 5. This programmer probably meant to change the value of the initial `x` to 5. However, this code will throw a compilation error, since the compiler cannot have two unique references to the same variable.

This can be accomplished with the following code:

```
...
{
    int x = 4;
    x = 5;
}
```

Here, `x` is only declared once and its value is changed to 5. This does not throw any errors since there is only one variable named `x` in this block.

Likewise, we cannot reference variables created in an inner block from an outer block.

2.7 Character I/O

Character input and output is an incredibly important piece of any programming language. You have already seen hints at how these concepts work in Ripple, but we will now go into a more in-depth discussion of Ripple's input and output. We will begin with character output, which has already been covered to some degree. Outtputting strings is handled by the previously seen `print()` function. This function accepts any of the fundamental datatypes, excluding datasets, and will be convert these types to strings and print them. To output a dataset, one can print its constituent parts, or define a function that returns a string describing a given dataset. Additionally, the `print()` function will accept any number of arguments and concatenate them. Thus, the `output` function can either look like `print(arg0, arg1, arg2, ...` . This can also be accomplished by using the `+` operator, which also concatenates two strings: `print(arg0 + arg1 + arg2)`.

The `print()` function will return void. This is a distinct difference from the `input()` function which will return a string. `input()` takes newline delimited lines from standard input; if a file has been piped into a Ripple program, the `input()` function will return newline delimited strings from this file. Another aspect of the input function is that it accepts a string argument that acts as a prompt and will be printed to Standard Out.

An interesting alternative to the input function is the aforementioned `keyboard_stream`; a `keyboard_stream` reads one line at a time from standard in and sets the linked variable to the last string that was input. The following program prints every string put in by the user.

```
1 void main() {
2     string line;
3     link (line <- keyboard_stream()) then print_line;
4     # echoes every line to the user
5     stop;
6 }
```

echo.rpl

2.8 Reactive Programming

The defining feature of Ripple is its implementation of the reactive programming paradigm. **Note:** it must be emphasized that Ripple's reactive programming is *not* functional. Reactive programming, in its most abstract definition, is programming with asynchronous data streams. Another explanation could also be, it is a paradigm that relies on the propagation of change throughout variables of a program. These, however, are somewhat abstruse statements, especially for those who have never programmed before. The simplest way to express this paradigm is to think of an Excel spreadsheet – a relatively painless visualization for propagation of changes exhibited in reactive programming.

In other words, within a spreadsheet you might have a cell, `A1`, set to the sum of the values of cells `B1` and `B2`, such as

```
A1 = B1 + B2
```

Furthermore, `B1` could be set to the values of cells `C1` and `C2`,

```
B1 = C1 + C2
```

In a spreadsheet, changing the value of `C2` would propagate through the rest of the sheet, changing the value of `B1`, and subsequently `A1`.

In a typical imperative language might have the statement `x = y + z` which sets `x` to the summation of the current values of `y` and `z`; if `y` or `z` change value after this assignment, `x` is not subsequently updated unless explicitly specified by the programmer.

Combining both the imperative and reactive programming paradigms, then, is where Ripple comes in. In Ripple, you are able to use the imperative paradigm to explicitly *link* variables that will then exhibit the reactive paradigm by propagating changes. The link statement, and all of its intricacies, is explored in depth in the following section, but hopefully this gives you a small idea of reactive programming.

2.9 The `link` Statement

A defining feature in Ripple that is not found in most programming languages is the functionality provided by the `link` keyword. The `link` keyword is what implements the reactive programming paradigm in Ripple. Let's look at the this keyword in action.

```
1  #~ prints Fahrenheit-Celsius temperature ~#
2  void output_temperature(string temp){
3      print(''Temp in C: ", temp);
4  }
5
6  void main() {
7
8      float TEMP_CONV = 5//9;
9
10     int deg_f = 50;
11     link(int deg_c <- (deg_f - 32) * TEMP_CONV) then output_temperature
12     deg_f = 32;
13 }
```

temp1.rpl

After compiling *temp1.rpl*, we can run the program's executable and see the following results:

```
$ rpl temp1.rpl
$ ./output
  Temp in F: 0
  Temp in C: 10
  Temp in F: 32
  Temp in C: 0
```

On line 6, the program uses the `link` statement to connect the variable `deg_c` to the expression `(deg_f - 32) * TEMP_CONV`. By *linking* a variable to another variable, the program creates a dependency between the two variables, which is added to the program's `dependency tree`. This dependency tree is used by the compiler to update variables connected along the tree based upon the time they were linked to the previous variable in the tree. The dependency tree (Figure 2.1) depicts the flow of data from `deg_f` to `deg_c`.

Note how the arrow is unidirectional; that is, the updates do not flow in both directions. This decision was made intentionally to avoid the issue of cyclical dependencies. Dependency cycles should be avoided in Ripple at all costs, meaning the ordering of link statements is important.

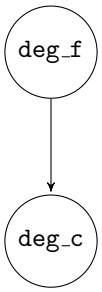


Figure 2.1: The dependency tree for *temp.rpl*

Suppose we have the following piece of code:

```
1 ...
2 int x = 10;
3 link(int y <- x + 2)
4 link(int z <- y - 1)
5
6 link(int q <- x - 3)
7 y = x + 7;
8 ...
```

link.rpl

The dependency tree for *link.rpl* is shown in Figure 2.2. As shown on line 3 through 4, there is a nested `link` statement within the initial `link` statement. This nesting is illustrated in the dependency tree by the addition of a new layer of nodes.

Let's walk through the program, starting on line 2, to see what's happening here.

Line 2 The variable `x` is initialized with the value of 10.

Line 3 The variable `y` is initialized and linked to `x` through the statement `x + 2`; the value of `y` becomes 12. This creates the root node `x` and the child node `y` and connects them on the dependency tree.

Line 4 The nested link statement initializes the integer variable `z` to the variable `y`; the value of `z` is 11. The node `z` is added to the dependency tree as a child of `y` and a link is made between them.

Line 7 The variable `q` is initialized as an integer linked to `x`; the value of `q` is 7. The dependency graph creates the node `q` as a child of `x` and creates a link between them.

Line 8 `y`'s value is changed to 17. This change will cause a cascade down to `z`, registering `z`'s new value as 16. However, the values of `x` and `q` remains at 10 and 7 respectively because both `variables` are not linked to `y`.

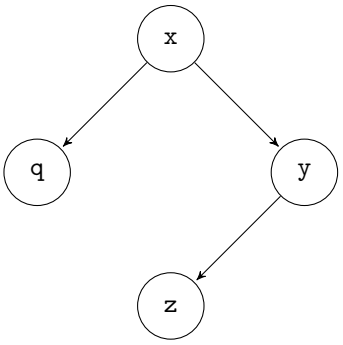


Figure 2.2: The dependency tree for *link.rpl*

Moving back to the `temp.rpl` example; on line 11, when `deg_f` is updated to the new value of 32, `deg_c` is notified of the change and updates its value accordingly. This update causes the change in value seen on the fourth line of output.

`link`s have the interesting property of being many-to-one; that is, we can have a variable link to a myriad of previously declared variables. When one of those parent variables is updated, the child will be updated accordingly. Again, the programmer must be careful in these cases not to create cycles in the dependency tree.

2.10 Streams

`stream`s are interesting consequences related to the `link` keyword within the Ripple language. The intuition behind their genesis is as follows: If the language is to be truly reactive, it would be necessary to have a construct that could handle live data coming from a variety of diverse sources. Thus, the `stream` was born.

A `stream` is fairly simple to implement within code itself, and has the ability to be extended for other types of media. Before we delve further into the implementation of `stream`, let's look at a sample program.

A simple implementation of a `stream` is a line counting program.

```
1 #~ Reads in a file and prints out the line with the line number. ~#
2 void print_line(string line){
3     print(line);
4 }
5
6 void main() {
7     string filename = input("Please input a file name.");
8     link (string line <- file_stream(filename)) then print_line;
9 }
```

count.rpl

Let's walk through the program to see how the `file_stream` works. In this program, we are using the `file_stream`, which provides a way to read through a file asynchronously line by line. This tool is an extension of of the generic `stream` construct, and one of the predefined standard `StreamReader`s.

Line 3 The function `print_line` is declared here; it takes in one arguement: a string whose formal parameter is known as `line`.

Line 7 The program reads in a string from standard in (via the `input` function) and stores it in the string variable `filename`.

Line 8 The `line` variable is initialized and linked to an `file_stream` which is created by providing it with a filename to open and iterate through. After the file is opened, the compiler looks at the auxiliary function (`print_line`) and runs it accordingly. Thus, this program will print out every line that it reads in.

Here is another program that takes in input from a `keyboard_stream()`. It reads a string in from the keyboard, converts it to an integer, and sets it to `deg_f`. This variable is then passed into the `what_to_wear` function. The `stop` keyword puts the program into an infinte loop so it can continue to listen to the input source.

```
1  #~
2  Tells you what to wear!
3  ~#
4
5  void what_to_wear(int x) {
6      if (x < 30) {
7          print("Bundle up, it's cold outside!");
8      } else {
9          if (x < 70) {
10             print("Maybe put on a sweater?");
11         } else {
12             print("Put on shorts and flip flops!");
13         }
14     }
15 }
16
17 void main() {
18     int deg_f;
19     link(deg_f <- str_to_int <- keyboard_stream()) then what_to_wear;
20     stop;
21 }
```

whattowear.rpl

If we run the program after compilation, we have the following output.

```
$ rpl temp2.rpl
  ./temp2
$ 20
 Bundle up, it's cold outside!
$ 80
 Put on shorts and flip flops!
```

2.11 Conclusion

This, hopefully, has given a general overview of the basics of Ripple. Unfortunately, this tutorial cannot give a full and in-depth discussion of the subtler intricacies of Ripple, especially when dealing with `link` statements and `StreamReaders`. However, we believe that we have provided enough tools and a general understanding of the core aspects of Ripple that you may begin experimenting and writing your own code, and we encourage you to go out and do so.

Chapter 3

Language Reference Manual

3.1 Introduction

This manual serves as a reference for the syntactic elements of Ripple. Much of it will be intuitive to programmers familiar with C. In it, we cover lexical conventions of Ripple, as well as syntactic and semantic elements. It can serve as an authority whenever doubts arise with Ripple.

This document covers Ripple in a bottom-up fashion. We begin by introducing our lexical conventions and types, and then use these structures to create compound expressions which combine to form more complex grammatical constructs. By explaining these increasingly complex constructs and their combinations, we will demonstrate how a working Ripple program is ultimately formed.

3.1.1 Notation

For clarity, this guide will follow a notational convention. Ripple reserved words will be represented in `monospace`, and syntactic categories will be written in *italics*. Within the latter, non-reserved regular definitions will be written as capital, italicized words. An OPT subscript indicates that a given element may be omitted.

3.2 Lexical Conventions

The first step in the compilation of a program is lexical analysis. At this time, the characters of the `.rpl` file are separated into discrete groups, commonly referred to as "tokens", which will eventually be parsed by a syntactic analyzer to verify that the program conforms to the context-free grammar defined later in this manual.

Ripple programs are written exclusively using the ASCII character set. Ripple files share the same naming conventions as any UNIX file, that is, the file name may contain any character except "`/`".

3.2.1 Tokens

Each lexical token available in Ripple falls into one of the following five categories:

1. Identifiers (names of variables, functions and datasets)
2. Keywords (e.g. `if`, `return`, `link`, etc.)
3. Constants (particular `int`s, `float`s, `string`s, and `bool`s; all are specified inline)
4. Operators (`+`, `-`, etc.)
5. Miscellaneous separators (`()`, `;`, etc.)

3.2.2 Identifiers

An identifier is a sequence of characters that uniquely identifies a variable or function. Identifiers in Ripple can contain letters, numbers, and underscores, and have any length of at least 1, but may not start with a number. All identifiers are case-sensitive.

Identifiers match the following regular expression:

```
{letter}({letter}|{digit})*
```

where `letter` includes the uppercase and lowercase English letters and the underscore character.

The only exceptions are *reserved words*; these cannot be used as identifiers, because they always denote particular features of Ripple. In Ripple, the set of reserved words is coextensive with the set of keywords, listed in the next section.

3.2.3 Keywords

Keywords are sequences of characters reserved by the Ripple language for special purposes, and therefore cannot be used as identifiers. Keywords specify control flow statements, data types, boolean variables and boolean operators. Ripple has twenty-two reserved words, which are listed in the table below.

The semantics for each keyword will be specified as we cover the related language constructs in this reference manual.

if	else	while	for	link
return	continue	break	then	dataset
void	bool	byte	int	float
string	true	false	not	or
and	stop	file_stream	web_stream	keyboard_stream

3.2.4 Constants

Constants refer to representations of particular values whose literal value is passed from the lexical analyzer to the syntactic analyzer. They include, for example, the number `42`, or the string "`hello, world`".

Constants may be of any of four types.

1. A `bool` either has the value `true` or `false`

2. An `int` is a sequence of digits (0 - 9) without a decimal point (".")

3. `A` float is a sequence of digits containing a decimal point.

4. A `string` is any sequence of characters enclosed in single or double quotation marks.

3.2.5 Operators

An operator is a member of a closed set of symbols, each between one and three characters long, that act as a function on one or two values (called "operands"). These functions produce a resultant value determined by performing some action, represented by the operator, on those operands. Unary operators act on a single operand, while binary operators act on two operands. They are used for logical, relational, and arithmetic actions. The particular semantics of an operator are tied to the type(s) of its operand(s).

The following is an exhaustive list of Ripple's operators:

Unary operators	Binary operators
`not`	`+`
`-`	`-`
`@`	`*`
	`/`
	`//`
	`%`
	∧
	`and`
	`or`
	`<`
	`<=`
	`==`
	`!=`
	`=>`
	`>`
	`.`

3.2.6 Miscellaneous Separators

The eight remaining special symbols used to separate code in Ripple are:

- Parentheses ("`(`" and "`)`") are used to define function parameters and arguments and assign precedence in expressions.
- Square brackets ("`[`" and "`]`") are used for declaring arrays and acessing elements within arrays
- Curly braces ("{" and "}") are used to define variable scope within Ripple, demarcate control flow statement bodies and initialize values in an array.
- Commas ("`,`") are used to separate function arguments in a function call, function declaration arguments in a function declaration and items in an array initialization
- Semicolons ("`;`") are used to separate declarative statements
- The arrow ("`<-`") is used within `link` statements to establish a link between a variable and either an expression or a stream.

3.2.7 Whitespace

Whitespace consists of any sequence of newlines, tabs, spaces, and comments. Whitespace demarcates where one token stops and another begins – for example, the statement `foo bar` forms two tokens, but `foobar` forms one. This is necessary in cases where another separator (e.g. "`;`") is inapplicable. Otherwise, whitespace is completely ignored; thus, additional or excess whitespace has no effect on the execution of the program.

3.2.8 Comments

Comments, like whitespace, are discarded by the compiler and do not bear on the execution of the program. Single-line comments start with `#` and end with a newline, while multiline comments start with `#`∼ and end with ∼`#`. Multiline comments can contain any sequence of characters, except the end sequence marker (namely "∼`#`").

3.3 Types

The five fundamental types in order of increasing size in Ripple are `byte`, `bool`, `int`, `float`, and `string`. All number types can be implicitly up-converted without loss of value, meaning that `bytes` can become `int`s,

`int`s can become `float`s, etc. Additionally, all types have `string` representations. In certain circumstances, the number types can also be down-converted, with their values truncated to represent the limitation of the lower types. In addition to the primitive types, Ripple also provides two derived types, `array` and `dataset`. Ripple is statically typed, which means that all variable types and all returning function values are known at compile time.

3.3.1 `void`

The `void` type is the only type that cannot be used in a variable declaration; instead, it is solely used for setting the return value of a function. A function declared with the `void` type will not return a value. That is, the function does not require a `return` statement, and the function cannot return any expression.

3.3.2 `bool`

`bool`s can take on one of two values: `true` or `false`. These can be produced by either one of the literal constants `true` or `false`, or alternatively by a boolean expression or relational expression as will be covered later in this manual. They are primarily used by `if` statements and `while` or `for` loops to evaluate execution conditions.

3.3.3 `int`

Integers are sequences of digits. Every integer is represented as eight bytes; there are no distinctions between `short`s, `int`s and `long`s. All integers are signed two's complement. The value range of an `int` is $[-2^{31},\ 2^{31}-1]$.

3.3.4 `float`

Floats are made of an integer part, a decimal point, and a fractional part. They all use double precision and are signed. They are represented as 8 bytes each (64 bits), and the value range of a `float` is $[\pm 2.23 \times 10^{-308},\ \pm 1.80 \times 10^{308}]$.

3.3.5 `string`

A `string` is a sequence of 0 or more ASCII characters. Memory is dynamically allocated to match the size of the `string` – i.e., `string`s are represented by one byte per character, plus one null terminating byte. Strings are mutable and can be indexed like `array`s.

String literals are enclosed by either double quotation marks: either `"hello"`. They can contain any character except for newline or quote characters; these must be escaped. The sequence "\n" escapes a newline, and a backslash preceding a quotation mark (of either sort) escapes that quotation mark.

3.3.6 Derived Types

The basic types can be used together to create two more complex data structures, namely `array`s and `dataset`s. Arrays simply store a predefined number of elements of a certain type, while `dataset`s can group together elements of different types and provide a name by which to access each element.

3.3.7 Arrays

Arrays enable the programmer to give one name to many variables of the same type. Arrays in Ripple are dynamically allocated, meaning their sizes can change.

To declare an `array`, one must simply add square brackets to a type declaration, optionally specifying the initial size of the array.

Arrays can also be defined through array literals. Literals follow the following production:

$$\begin{aligned} \textit{array\_initialization} \rightarrow\ & \textit{array\_initialization}\ \texttt{,}\ \textit{expression} \\ \mid\ & \textit{expression} \\ \mid\ & \epsilon \end{aligned}$$

Array elements can be accessed and modified using square brackets as well, through the following production:

$$\textit{array\_access} \rightarrow\ \textit{ID}\ \texttt{[}\ \textit{expression}\ \texttt{]}$$

In Ripple, `arrays` use the ordinal convention, meaning the first element is at index 0 and the last element is at index (length - 1).

3.3.8 `dataset`

It is difficult to realize the wide variety of types a programmer may require when designing a language. However, almost every complex data type can be expressed as some combination of `byte`s, `bool`s, `int`s, `float`s and `string`s. To allow programmers to specify their own, more complex data types, Ripple provides the `dataset` type. `dataset`s are a contiguous block of primitive variables in some pre- determined order. All `dataset`s must be declared outside of functions and can be used either within a function or as part of a `final` declaration.

To construct a `dataset` the grammar rule applied is

$$\begin{aligned} \textit{dataset} \rightarrow\ & \texttt{dataset}\ \textit{ID}\ \{\ \textit{declaration\_argst}\ \};\\ \textit{declaration\_args} \rightarrow\ & \textit{declaration\_args}\texttt{,}\ \textit{dtype}\ \textit{ID} \\ \mid\ & \textit{dtype}\ \textit{ID} \\ \mid\ & \epsilon \end{aligned}$$

which specifies that a dataset consists of the keyword `dataset` followed by an identifier that names the dataset being created. Finally between curly braces the programmer specifies a list of declarations separated by semicolons. Datasets are basically identical to C's structs.

3.4 Expressions

An expression is some combination of operators and operands that will be evaluated to one of the built-in types. Our grammar defines that expressions are parsed in a specific order, thereby providing precedence to certain operators. The matriarch of all expressions is the *expression* nonterminal, with the following production:

$$\begin{aligned} \textit{expression} \rightarrow\ & \textit{expression}\ \texttt{or}\ \textit{and\_expression} \\ \mid\ & \textit{and\_expression} \end{aligned}$$

From this, all other expressions will be derived.

3.4.1 Or Expressions

This first production also establishes the left-associative `or` operator, which has the lowest precedence within Ripple. This expression is a binary boolean operator that takes a boolean on both sides and returns true if either operand is true.

3.4.2 And Expressions

$$\begin{aligned} \textit{and\_expression} \rightarrow\ & \textit{and\_expression}\ \texttt{and}\ \textit{eq\_expression} \\ |\ & \textit{eq\_expression} \end{aligned}$$

`and` has the next highest precedence. The expression takes two boolean operands and returns true only if both operands are true. Similar to the `or` operator it is also left-associative.

3.4.3 Equality Expressions

$$\begin{aligned} \textit{eq\_expression} \rightarrow\ & \textit{eq\_expression}\ \texttt{==}\ \textit{rel\_expression} \\ |\ & \textit{eq\_expression}\ \texttt{!=}\ \textit{rel\_expression} \\ |\ & \textit{rel\_expression} \end{aligned}$$

`==` and `!=` are relational operators that compare any two types by value. `==` returns `true` if both are equal, false otherwise, while `!=` does the opposite. Both are left-associative, and are separated from the other relational operators due to their lower precedence.

3.4.4 Relational Expressions

$$\begin{aligned} \textit{rel\_expression} \rightarrow\ & \textit{rel\_expression}\ \texttt{>=}\ \textit{plus\_expression} \\ |\ & \textit{rel\_expression}\ \texttt{<=}\ \textit{plus\_expression} \\ |\ & \textit{rel\_expression}\ \texttt{>}\ \textit{plus\_expression} \\ |\ & \textit{rel\_expression}\ \texttt{<}\ \textit{plus\_expression} \\ |\ & \textit{plus\_expression} \end{aligned}$$

Ripple offers the usual relational operators as well, all of which are left- associative. Relational operators, however, can only operate on number types (`int`s, `byte`s, and `float`s), and return boolean types.

3.4.5 Plus and Minus Expressions

$$\begin{aligned} \textit{plus\_expression} \rightarrow\ & \textit{plus\_expression}\ \texttt{+}\ \textit{mult\_expression} \\ |\ & \textit{plus\_expression}\ \texttt{-}\ \textit{mult\_expression} \\ |\ & \textit{mult\_expression} \end{aligned}$$

Plus and minus expressions represent the next step in our operator hierarchy.

The minus operator `-` shares a lexeme with unary negation, but serves a different function. It can take any number type and it will return the difference from the second to the first operand. If given two different number types, it will return the larger of the two types. For example, if given at least one `float` it will return a `float`.

The `+` operator is more complicated. Depending on its use, `+` can either represent addition or concatenation. It is always left-associative.

When applied to two number types (i.e. `int`, `byte`, or `float`), it will return the sum of the two values. The sum will be of the larger of the two operand types (e.g., if you are adding an `int` and a `byte`, the result will be an `int`). If one of the two operands is a `float`, the result will also be a `float`.

When at least one of the operands is a string, `+` becomes concatenation. The non-string argument is converted into its literal string representation. For example, ``` ``Number of repetitions: '' + 10 ``` returns ``` ``Number of repetitions: 10'' ```.

3.4.6 Multiplicative Expressions

$$\begin{aligned} \textit{mult\_expression} \rightarrow\ & \textit{mult\_expression}\ \texttt{*}\ \textit{unary\_expression} \\ \mid\ & \textit{mult\_expression}\ \texttt{/}\ \textit{unary\_expression} \\ \mid\ & \textit{mult\_expression}\ \texttt{//}\ \textit{unary\_expression} \\ \mid\ & \textit{mult\_expression}\ \texttt{\%}\ \textit{unary\_expression} \\ \mid\ & \textit{unary\_expression} \end{aligned}$$

The next level of precedence is the multiplicative expression. Multiplicative operators are left-associative and operate only on the numeric types.

`*` is the multiplication operator. It follows the same conversion rules as the subtraction operator.

`/` is the standard division operator. If either of the operands is a `float`, it returns a `float`. Otherwise, it will return an integer or byte rounded down to the nearest integer.

Unlike in other, less sensible languages, `//` is floating point division. It behaves identically to `/`, except that even if both of the arguments are `int`s or `byte`s, the result will be a `float`.

`%` is the modulus operator. It can only take `int`s or bytes as its left argument, and returns the remainder when the first value is divided by the second.

3.4.7 Unary Expressions

$$\begin{aligned} \textit{unary\_expression} \rightarrow\ & \texttt{not}\ \textit{unary\_expression} \\ \mid\ & \texttt{-}\ \textit{unary\_expression} \\ \mid\ & \texttt{@}\ \textit{unary\_expression} \\ \mid\ & \texttt{(}\ \textit{TYPE}\ \texttt{)}\ \textit{unary\_expression} \\ \mid\ & \textit{exp\_expression} \end{aligned}$$

Unary expressions are right-associative in Ripple. There are four different unary operators in Ripple: they are `-`, `not`, `@` and casts.

`not` represents boolean negation. It takes one boolean argument, returning `false` if the argument is `true` and `true` if the argument is `false`.

`-` uses the same symbol as a subtraction, but here it represents unary arithmetic negation. It takes a number type and returns its negation.

`@` is Ripple size operator. It returns the size of its operand. This size is the size of bytes used to represent the value of the expression, no matter the type of expression. However, in the case of an array, it returns the length of the array. For example, `@int[10]` returns 10, since the length of the array is 10. This is the usual case for arrays, but ultimately all results of this operator will be machine-dependent.

Casts can only be performed between primitive types, in some predefined ways. All number types can be cast to one another. When `float`s are cast to `int`s, the fractional part will be dropped. Similarly, if we are down- casting from an `int` to a `byte`, only the 8 least significant bits will be preserved. If we were to up-cast from a `byte` to an `int`, the value of the `byte` would be maintained. This implementation holds across any form of up-casting between the numerical types. Any type can be cast to a `string`, but a `string` can only be cast to other types if its format is compatible (i.e., "123" can be converted into an integer but "This string cannot be converted to an integer" cannot).

3.4.8 Exponential Expressions

$$\begin{aligned} \textit{exp\_expression} \rightarrow\ & \textit{exp\_expression} \wedge \textit{exp\_expression} \\ \mid\ & \textit{var} \mid \texttt{(}\ \textit{expression}\ \texttt{)} \end{aligned}$$

Exponential expressions refer to the $\wedge$ operator, which performs exponentiation. It is left-associative and can take any combination of number types, performing the same changes to types as previous operations.

The exponential operator has the highest precedence of all operators in Ripple. To perform operations of lower precedence before those of higher precedence, exponentiation expressions provides a production that goes to a parenthesized expression.

3.5 Statements

A *statement* is the full specification of an action to perform. Since Ripple is an imperative language, statements are very important. Ripple *statement*s are separated into five categories, as follows:

$$\begin{aligned} \textit{statement} \rightarrow\ & \textit{conditional\_statement} \\ \mid\ & \textit{loop\_statement} \\ \mid\ & \textit{link\_statement} \\ \mid\ & \textit{declarative\_statement} \\ \mid\ & \textit{jump\_statement} \end{aligned}$$

3.5.1 Conditional Statements

A conditional statement is one method of specifying flow control. It has the following grammar:

$$\textit{conditional\_statement} \rightarrow \texttt{if (}\ \textit{expression}\ \texttt{,)}\ \textit{statement\_block}\ \textit{else\_statement}$$

When a conditional statement is encountered, the *statement_block* code is executed if and only if the boolean expression *expression* evaluates to `true`. If *expression* evaluates to `false`, then *statement_block* is not executed, and the *else_statement*, if one exists, is executed instead.

$$\begin{aligned} \textit{else\_statement} \rightarrow\ & \texttt{else}\ \textit{statement\_block} \\ \mid\ & \epsilon \end{aligned}$$

3.5.2 Loop Statements

Loop statements are another common method of specifying flow control. Ripple includes both `while` and `for` loops, which are syntactically identical to their counterparts in C.

$$\begin{aligned} \textit{loop\_statement} \rightarrow\ & \texttt{while (}\ \textit{expression}\ \texttt{)}\ \textit{statement\_block} \\ \mid\ & \texttt{for (}\ \textit{declarative\_statement}\ \textit{expression}_{OPT};\ \textit{expression}_{OPT}\ \texttt{)}\ \textit{statement\_block} \\ \mid\ & \texttt{for (;}\ \textit{expression}_{OPT};\ \textit{expression}_{OPT}\ \texttt{)}\ \textit{statement\_block} \end{aligned}$$

A `while` loop checks whether the boolean expression *expression* evaluates to `true`. If it does, then it executes *statement_block*. It performs these two steps repeatedly, until *expression* evaluates to `false`. After this change, the program proceeds to execute the code after the *loop_statement*.

A `for` loop first executes the *declarative_statement* between the opening parenthesis and the first semicolon, if there is one. This is typically used to initialize an iteration variable. Then the following steps are performed repeatedly: check whether the boolean expression *expression* in between two semicolons evaluates to `true`; if it does, execute *statement_block* and then evaluate the expression after the second semicolon; if it doesn't, proceed to the next instruction after the loop. The second expression is the increment statement for the for-loop, as long as the evaluation statement evaluates to `true`, the *statement_block* will be executed and the increment statement will be evaluated. Once the evaluation statement evaluates to `false`, the increment statement will be ignored.

3.5.3 Link Statements

Link statements are unique to Ripple; they achieve a combination of flow control and reactive state manipulation. The syntax of a *link_statement* is specified as follows:

link_statement → `link (` *var* `<-` *expression*`);`
| `link (` *var* `<-` *expression*`) then ID;`
| `link (` *var* `<-` *stream_reader* `);`
| `link (` *var* `<-` *stream_reader* `) then ID;`
| `link (` *var* `<-` *var* `<-` *stream_reader* `);`
| `link (` *var* `<-` *var* `<-` *stream_reader* `) then ID;`

Inside the parentheses is a statement syntactically similar to an assignment, except that instead of an equals sign, there is a left-facing arrow ("`<-`"). Concretely, the conjunction *declaration link_stream* might take the form:

```
x <- y + 5
```

This specifies that the variable x should be *linked* to the expression $y + 5$ – that is, whenever the value of y changes, the value of x will automatically update to equal $y + 5$.

The ID in this production represents a *function_call*, if provided, it will be called every time this update occurs. Additionally, the middle `var` variable in the productions with `stream_reader`s means that the value that the `stream_reader` reads in will be passed to this variable, with the return value being what the linked variable is updates with.

Programmers should be careful not to create cycles with link statements. For example, if `x` was linked to `y`, and later in the program `y` was linked to `x`, this would create a cycle. Changes in `x` would cause changes in `y`, which would cause changes in `x`, creating an infinite loop of changes. Behavior in these situations is undefined and should be avoided at all costs.

3.5.4 Streams

An important aspect of Ripple is the ability to treat external variables as local to the program. To this end Ripple provides the `stream` construct which is used in conjunction with `link` statements to "link" a variable to a changing external stream of data. Streams are equivalent to stream readers. The name stream reader is used in the backend code and grammar, and the ripple user uses the keyword stream.

The `link` statement creates a dependency between the `stream` and the variable; the linked variable will be updated by the stream. Hence the local variable represents a link to external data. Additionally, the data read into the stream will pass through a filter function which will be what ultimately sets the value of the linked variable. The filter function must return the same type as the linked variable, or else the program will crash.

There are three types of streams: web streams, file streams, and keyboard streams. All three are produced with the same production function, and we differentiate by their names. The production syntax is defined below:

stream_reader → *stream_reader_name* `(` *args* `)`
stream_reader_name → `STREAM_READER_CODE`

The simplest stream is the keyboard_stream, which takes no arguments, will continuously read from stdin, and is newline delimited.

The other two streams are slightly more complicated. They both take three arguments, and all are required.

The web stream takes a URL or IP address, port number, and interval. The stream will attempt to connect to the designated address and port, and if interval is specified will sleep for that time after writing to the linked variable. After, it will read from the same address and port again.

The file stream takes a file path, an interval, and a delimiter. It will read from this file on the delimiter and set it to the linked variable. After reading through the file, it will sleep if interval is set, and afterwords will read from the file again.

3.5.5 The Standard Library

Accompanying Ripple's `Streams` are a series of standard functions that make up the standard library of auxiliary functions. These functions are called within link statements after a value has been read in from some type of stream. Generally, these functions are acted upon by the incoming data from the stream. Thus, all functions in the Standard Library return void and accept a single argument that is of the same type as the variable that is being linked to the `stream`.

Similarly, the Standard Library contains some functions that can be used within the link statement. These functions return values that are the same type as those variables that are linked together. Like the auxiliary functions, the argument taken in by this function must be of the same type as the linked variable.

3.5.6 Declarative Statements

Declarative statements enable the programmer to create and modify variables in Ripple. Their general syntax is given below:

$$\begin{aligned} \textit{declarative\_statement} \rightarrow\ & \textit{declaration} \texttt{ = } \textit{expression}; \\ \mid\ & \textit{expression}; \\ \mid\ & \epsilon \end{aligned}$$

If a variable has not been previously declared in a given scope, its declaration must specify its type. If it has already been declared, the programmer may change its value without declaring the type.

Variables may also be declared `final` in order to specify that they will be constant. Once declared, a final variable cannot be changed. These variables must be declared at the top of the file, and so their declarations have an extra production:

$$\textit{final\_declaration} \rightarrow \texttt{final}\ \textit{declaration\_statement}$$

3.5.7 Jump Statements

A jump statement is the statement that specifies control flow most directly. When the program reaches one of these statements, instead of executing the statement that immediately follows, it is redirected to some other instruction – which instruction is performed next depends on the kind of jump statement. They are divided syntactically as follows:

$$\begin{aligned} \textit{jump\_statement} \rightarrow\ & \texttt{return}\ \textit{expression}_{OPT}; \\ \mid\ & \texttt{continue}; \\ \mid\ & \texttt{break}; \\ \mid\ & \texttt{stop}; \end{aligned}$$

The `return` statement resides within a function. It causes the function call to evaluate to the value of *expression*, and proceeds to execute the instruction following where the function was called. If the function was called due to an update from a link statement the `return` statement only causes the function to end; there is no next instruction in this case.

The `continue` statement is used within a loop. It skips the remainder of the loop and proceeds to execute the statement at the top of the loop block.

The `break` statement is also used within a loop. It skips the remainder of the loop and proceeds to execute the next statement after the loop block.

The `stop` statement can be used anywhere. It simply halts the execution of the program at a given point until the user terminates it with CTRL+C. This is useful because `streams` will continue flowing only until the main function ends.

3.6 Top Level Declarations

Top level declarations are the translation units of Ripple. These declarations can only occur in the file body; they cannot be declared within functions, `dataset`s or other declarations that are not top level. Ripple allows two top level declarations based on the following grammar

$$
\begin{aligned}
\textit{program\_declarations} &\rightarrow \textit{program\_declarations}\ \textit{program\_declaration} \\
&\mid \epsilon \\
\textit{program\_declaration} &\rightarrow \textit{dataset} \\
&\mid \textit{function} \\
&\mid \textit{final\_declaration}
\end{aligned}
$$

3.6.1 `dataset` Declaration

$$\textit{dataset} \rightarrow \texttt{dataset}\ ID\ \{\ \textit{declaration\_args}\ \};$$

As explained in the `dataset` section, a `dataset` declaration consists of the keyword `dataset`, an identifier and a list of variable declarations.

The `final` keyword creates a read-only variable that sets a specific type and value to a specific identifier. The variable created must be initialized on creation. Mutating a `final` variable is not permitted. You should use the `final` keyword in scenarios where predefined constants are in use in order to avoid issues such as magic numbers and to clarify code.

3.6.2 `function` Declarations

$$\textit{function} \rightarrow \textit{dtype}\ ID\ (\ \textit{declaration\_args}\ )\ \textit{statement\_block}$$

Function declarations are used to name and specify the arguments, return type, and code to be executed of a function.

3.7 Scope

Ripple's variables use block scope. This means that variables exist only within the block in which they are defined; this is true for both internal variables defined within a program, and for variables that depend on a stream.

All functions from within the same file can be used within any scope as can functions from another file which has been imported. When a file is imported the programmer also has access to all `final` variables declared in the imported file.

3.8 Functions

Functions are named sets of instructions; as such, they encourage modularity and easy reuse. A function in Ripple takes *expression*s as inputs (i.e. arguments) and returns a single value.

A function definition specifies the number and types of arguments, the return type, and the set of statements to execute. Functions need not be defined before they are called. Every Ripple function must specify its return type before its identifier. Ripple functions conform to the following syntax.

$$\textit{function} \rightarrow \textit{dtype}\ ID\ (\ \textit{declaration\_args}\ )\ \textit{statement\_block}$$

To use a function, one must simply call it and provide any required arguments. The syntax of a function call is:

$$
\begin{aligned}
\mathit{function\_call} &\rightarrow \mathit{ID}\ \texttt{(}\ \mathit{args}\ \texttt{)} \\
\mathit{args} &\rightarrow \mathit{args}\texttt{,}\mathit{expression} \\
&\mid \mathit{expression} \\
&\mid \epsilon
\end{aligned}
$$

Here, the programmer provides actual parameters for the previously declared formal parameters, and the code for the function executes as normal.

3.9 Grammar

Below we reproduce the full grammar that has been described throughout this manual. It follows the same notation as before.

The start symbol is *program*. To reiterate, terms in *italics* are syntactic constructs, while terms in `monospaced font` are literal strings. Terms in all-caps italics are tokens.

$$
\begin{aligned}
\mathit{program} &\rightarrow \mathit{program}\ \mathit{program\_section} \\
&\mid \epsilon \\
\mathit{program\_section} &\rightarrow \mathit{dataset\_declaration} \\
&\mid \mathit{function} \\
&\mid \texttt{FINAL}\ \mathit{declaration\_statementdataset\_declaration} \rightarrow \quad \texttt{dataset}\ \mathit{ID}\ \{\ \mathit{declaration\_args}\ \};\\
\mathit{function} &\rightarrow \mathit{dtype}\ \mathit{ID}\ \texttt{(}\ \mathit{declaration\_args}\ \texttt{)}\ \mathit{statement\_block} \\
\mathit{declaration\_args} &\rightarrow \mathit{declaration\_args}\texttt{,}\ \mathit{dtype}\ \mathit{var} \\
&\mid \mathit{dtype}\ \mathit{var} \\
&\mid \epsilon \\
\mathit{statement\_block} &\rightarrow \{\ \mathit{statement\_list}\ \} \\
&\mid \{\ \} \\
&\mid \epsilon \\
\mathit{statement\_list} &\rightarrow \mathit{statement\_list}\ \mathit{statement} \\
&\mid \mathit{statement} \\
&\mid \epsilon
\end{aligned}
$$

3.9.1 Statements

$$
\begin{aligned}
\textit{statement} &\rightarrow \textit{conditional\_statement} \\
&\mid \textit{loop\_statement} \\
&\mid \textit{link\_statement} \\
&\mid \textit{declarative\_statement} \\
&\mid \textit{jump\_statement} \\
&\mid \textit{link\_statement} \\
\textit{conditional\_statement} &\rightarrow \texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement\_block}\ \textit{else\_statement} \\
\textit{else\_statement} &\rightarrow \texttt{else}\ \textit{statement\_block} \\
&\mid \epsilon \\
\textit{loop\_statement} &\rightarrow \texttt{while (}\ \textit{expression}\ \texttt{)}\ \textit{statement\_block} \\
&\mid \texttt{for (}\ \textit{expression}_{OPT}\texttt{;}\ \textit{expression}_{OPT}\texttt{;}\ \textit{expression}_{OPT}\ \texttt{)}\ \textit{statement\_block} \\
\textit{link\_statement} &\rightarrow \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{expression}\ \texttt{);} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{expression}\ \texttt{) then ID ;} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{var}\ \texttt{<-}\ \textit{expression}\ \texttt{);} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{var}\ \texttt{<-}\ \textit{expression}\ \texttt{) then ID;} \\
\textit{declarative\_statement} &\rightarrow \textit{dtype}\ \textit{expression}\texttt{;} \\
&\mid \textit{expression}\texttt{;} \\
\textit{jump\_statement} &\rightarrow \texttt{return}\ \textit{expression}\texttt{;} \\
&\mid \texttt{continue;} \\
&\mid \texttt{break;} \\
&\mid \texttt{stop;} \\
\textit{link\_statement} &\rightarrow \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{expression}\texttt{);} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{expression}\texttt{) then ID;} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{stream\_reader}\ \texttt{);} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{stream\_reader}\ \texttt{) then ID;} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{var}\ \texttt{<-}\ \textit{stream\_reader}\ \texttt{);} \\
&\mid \texttt{link (}\ \textit{var}\ \texttt{<-}\ \textit{var}\ \texttt{<-}\ \textit{stream\_reader}\ \texttt{) then ID;} \\
\textit{stream\_reader} &\rightarrow \textit{stream\_reader\_name}\ \texttt{(}\ \textit{args}\ \texttt{)} \\
&\mid \textit{stream\_reader\_name} \rightarrow \texttt{STREAM\_READER\_CODE}
\end{aligned}
$$

3.9.2 Expressions

$$
\begin{aligned}
\mathit{expression} &\rightarrow \mathit{or\_expression} \\
&\mid \mathit{value}\ \texttt{=}\ \mathit{or\_expression} \\
\mathit{or\_expression} &\rightarrow \mathit{or\_expression}\ \texttt{or}\ \mathit{and\_expression} \\
&\mid \mathit{and\_expression} \\
\mathit{and\_expression} &\rightarrow \mathit{and\_expression}\ \texttt{and}\ \mathit{eq\_expression} \\
&\mid \mathit{eq\_expression} \\
\mathit{eq\_expression} &\rightarrow \mathit{eq\_expression}\ \texttt{==}\ \mathit{rel\_expression} \\
&\mid \mathit{eq\_expression}\ \texttt{!=}\ \mathit{rel\_expression} \\
&\mid \mathit{rel\_expression} \\
\mathit{rel\_expression} &\rightarrow \mathit{rel\_expression}\ \texttt{>=}\ \mathit{plus\_expression} \\
&\mid \mathit{rel\_expression}\ \texttt{<=}\ \mathit{plus\_expression} \\
&\mid \mathit{rel\_expression}\ \texttt{>}\ \mathit{plus\_expression} \\
&\mid \mathit{rel\_expression}\ \texttt{<}\ \mathit{plus\_expression} \\
&\mid \mathit{plus\_expression} \\
\mathit{plus\_expression} &\rightarrow \mathit{plus\_expression}\ \texttt{+}\ \mathit{mult\_expression} \\
&\mid \mathit{plus\_expression}\ \texttt{-}\ \mathit{mult\_expression} \\
&\mid \mathit{mult\_expression} \\
\mathit{mult\_expression} &\rightarrow \mathit{mult\_expression}\ \texttt{*}\ \mathit{exp\_expression} \\
&\mid \mathit{mult\_expression}\ \texttt{/}\ \mathit{exp\_expression} \\
&\mid \mathit{mult\_expression}\ \texttt{//}\ \mathit{exp\_expression} \\
&\mid \mathit{mult\_expression}\ \texttt{\%}\ \mathit{exp\_expression} \\
&\mid \mathit{exp\_expression} \\
\mathit{exp\_expression} &\rightarrow \mathit{exp\_expression} \wedge \mathit{exp\_expression} \\
&\mid \mathit{unary\_expression} \\
\mathit{unary\_expression} &\rightarrow \texttt{not}\ \mathit{unary\_expression} \\
&\mid \texttt{-}\ \mathit{unary\_expression} \\
&\mid \texttt{@}\ \mathit{unary\_expression} \\
&\mid \texttt{(}\ \mathit{TYPE}\ \texttt{)}\mathit{unary}_e\mathit{xpression} \\
&\mid \mathit{value}
\end{aligned}
$$

3.9.3 Variables

$$
\begin{aligned}
value &\rightarrow literal \\
&\mid function\_call \\
&\mid array\_access \\
&\mid dataset\_access \\
&\mid var \\
&\mid \texttt{(}\ expression\ \texttt{)} \\
&\mid \texttt{\{}\ array\_initialization\ \texttt{\}} \\
array\_initialization &\rightarrow array\_initialization\ \texttt{,}\ expression \\
&\mid expression \\
&\mid \epsilon \\
args &\rightarrow args\texttt{,}expression \\
&\mid expression \\
&\mid \epsilon \\
array\_opt &\rightarrow \texttt{[}\ value\ \texttt{]} \\
&\mid \texttt{[ ]} \\
&\mid \epsilon \\
function\_call &\rightarrow ID\ \texttt{(}\ args\ \texttt{)} \\
dataset\_access &\rightarrow ID\texttt{.}var \\
array\_access &\rightarrow ID\ \texttt{[}\ expression\ \texttt{]} \\
var &\rightarrow ID \\
dtype &\rightarrow \texttt{TYPE}\ array\_opt \\
&\mid codeDATASET\ codeID \\
literal &\rightarrow \texttt{INTEGER} \\
&\mid \texttt{FLOAT\_LIT} \\
&\mid \texttt{STRING\_LITERAL} \\
&\mid \texttt{TRUE} \\
&\mid \texttt{FALSE}
\end{aligned}
$$

Chapter 4

Project Plan

Written by Amar Dhingra (Project Manager)

4.1 The Development Process

Ripple is the second language idea we came up with for this class. We began the semester by coming up with the idea for Whiskey, a web development language for systems programmers. After a month of working on Whiskey and beginning to work on the grammar we discovered ELM, a language that did everything we had planned for Whiskey. With just a few days remaining to write a language white paper we spent an entire day coming up with our current idea, Ripple. After switching paths a 6 weeks into the semester, our first goal was to divide the ideas we had for our language into a few categories; those that were absolutely essential to our language, those that would be interesting additions to our language and those that would be fun to implement but were not absolutely essential to Ripple. We next settled on the development environment we would use to work on our language so that we would not have compatibility issues. While brainstorming our language we had decided to compile to C++ so we decided to write our compiler in C++ to get more familiar with the language. Our team would meet once a week during the semester to make sure we were keeping on schedule and work collaboratively on parts of the language. In order to make sure that everone showed up to every meeting one team member was responsible for organizing food and drinks (usually pizza and coke) on a rotating basis. At our meetings we would assign tasks to be completed by our next meeting and choose a time for our next meeting.

4.2 Roles and Responsibilities

Having settled on what we would be developing and how we would be developing it we divided our team into two sub-teams. Each team was wholly responsible for their part of the project.

4.2.1 Code generation

The first sub-team consisted of Artur and I and was responsible for developing the lexer and parser for our language. This included generating all the lex and yacc code and converting the source code into the target code. We began by working on the structure of the Abstract Syntax Tree together including compiling simple expressions. Once we both had a clear vision of how our tree would be constructed we divided sections of the tree between us based on who had a better understanding of that section of our language and worked on them independently. We would then integrate the parts we had built at our weekly team meetings.

4.2.2 Language specific features

The second sub-team consisted of Alex, Spencer and Tom who were responsible for writing the language specific features of Ripple, namely code for `link` statements, `StreamReaders` and auxiliary functions. We

decided this sub-team should have more team members as figuring out how to implement these features seemed to be a more complex task than creating the code generator. Within the team each group member took responsibility for one of the three components, with Alex responsible for auxiliary functions, Spencer responsible for `StreamReaders` and Tom responsible for `link` statements. Since each component only tangentially depended on each other each team member defined an API by which the others could access their code. This API was also given to Artur and I as the code we had to generate for our language to function.

4.3 Team Ripple Style Guide

This style guide is the guide followed by team Ripple for all code written in C++ during the course of designing and developing our language. All code written for Ripple uses the C++11 standard.

4.3.1 File Names

All C++ code should be written and saved in `.cpp` and `.h` files. File names should be in lower case characters with words separated with an underscore ("_"). Class definitions, function prototypes, and macros should be placed in the `.h` file. Functions that consist solely of assignment operations or return statements can be defined in the class definition in the `.h` file. All `.h` files should be surrounded with guards of the form:

```
1 #ifndef __<filename>_H__
2 #define __<filename>_H__
3  ...
4 #endif
```

All other code should be placed in the `.cpp` files. Functions should be defined before class methods in the file. Class methods should be defined by explicitly declaring the namespace of the function before the method name. Class definitions should be placed in their own `.cpp` file except in the case where classes are directly related to one another. In this case, classes should be defined in some logical order to make understanding their interactions easy.

4.3.2 Include Statements

Whenever possible, a `.cpp` file should only `#include` its own `.h` file. All other dependencies should be placed in the `.h` file. `#include` statements should appear at the top of the `.h` file, and should be sepearted into two sections by a single newline.

The first section should contain all C++ library `#include` statements, while the second section should contain all user-defined `.h` files.

4.3.3 Classes

Class names should be written in `CamelCase` with the first letter of every word capitalized. The words `public`, `private`, and `protected`, which are used to declare the scope of variables and methods of a class, should be aligned with the class name. That is, the declarations should be as far left as possible. All variables and methods should be indented by four spaces.

4.3.4 Variables

Variable names should be written in lowercase with words being separated by an underscore ("_"). Variable name length should be kept as short as possible as long as the purpose of the variable can still be easily discerned. Whenever possible, variables that are used in a function or method should be declared at the top of the function/method before use. If possible, the variable should be initialized on declaration.

For all pointer variables, the asterisk (*) should be attached to the variable name. If a pointer type is being used in a template, it should be separated from the templated type.

4.3.5 Spacing

Line Length

Lines should be no longer than 120 characters. Lines that are more than 120 characters should be split into multiple, shorter lines. Due to the structure of lex and yacc files, lines within those files can be up to 120 characters long.

Whitespace

Code within functions and methods should be divided into logical, easy-to-read sections using a single line of whitespace. Function and method bodies should be separated by two new lines.

4.3.6 Structs and Enums

Structs and enums should only be used to store static data. Any types which are used to build data structures or perform complex operations should be classes.

4.3.7 Function

Functions should be as short as possible to maintain readability. All variable declarations should appear prior to use.

4.3.8 `NULL` vs `nullptr`

Whenever possible, use `std::nullptr`. If using legacy C code, use `NULL` or 0.

4.3.9 Curly Braces

Functions and Methods

For functions and methods the opening curly brace should be on the same line as the funciton/method definition. The closing curly brace should go on the line following the last line of the function.

Control Flow

`for` and `while` loops should have the opening curly brace following the closing parenthesis, and the closing curly brace should be placed on the line following the last line of the loop. If the loop body consists of a single line, the curly braces may be omitted.

`if` statements should have the opening curly brace following the parenthesis, and the closing curly brace should be placed on the line following the last line of the loop. `else` statements should start on the same line as the closing curly brace of the matching `if` statement. In the even that the body of an `if` statement consists of a single line, the curly braces may be omitted only if there is no matching `else` statement.

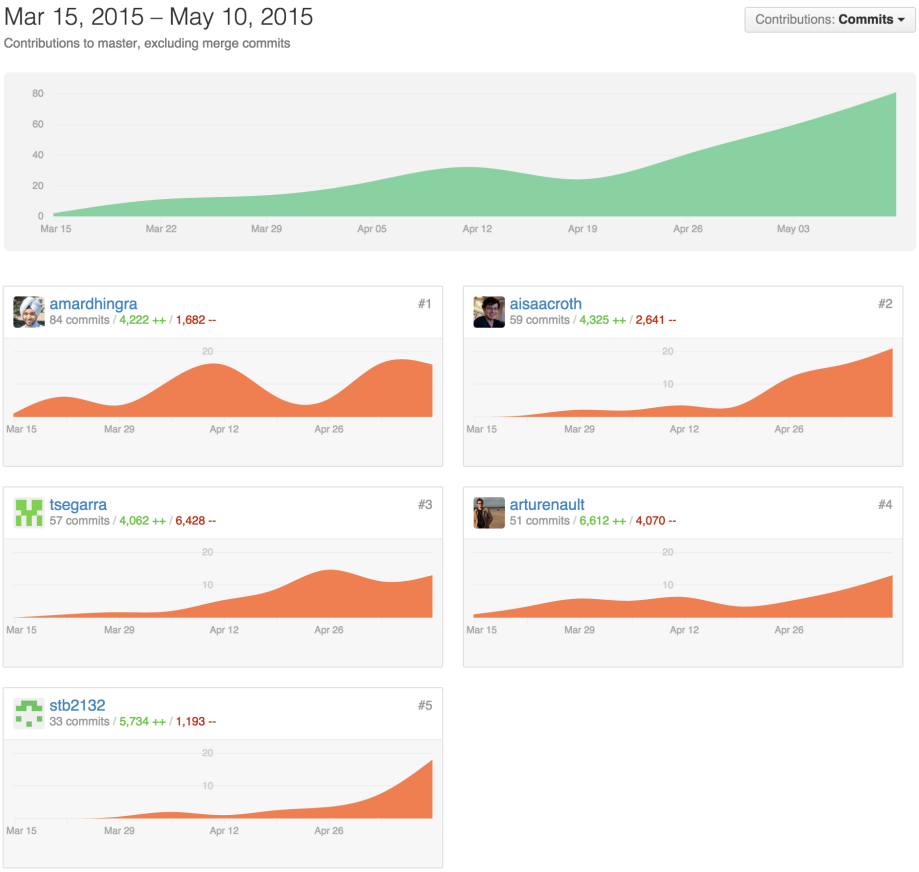
4.4 Ripple Timeline

Work on Ripple progressed at a farily regular pace through the semesters apart from a few days when we help all day code meetings. On those days we would implement large features of the language which required multiple team members to work collaboratively on. Meeting weekly allowed us to make sure we maintained a steady pace through the semester and were always aware of the next goal we needed to complete. Code generation, which required finding data structures with which to efficiently represent our grammar, was completed in small steps with a small part usually being added every week. Creating language specific features proved to be more difficult as the team not only had to find ways to efficiently create a reactive language, but also find ways to perform those tasks in C++. This required gaining more familiarity with the language than most of our team members had. However, once we had settled on a way to implement an idea Alex, Spencer or Tom could implement it very rapidly and integrate it with the rest of our language. This

led to our language growing in phases with a single feature taking weeks to figure out but being implemented in a matter of days.

4.5 The Ripple Commit Log

As you can see from the commit log below, the work we performed increased continuously through the semester. The spikes in commits come from the all-day code sessions we held twice in the semester. This graph is indicative of how the work increased towards the end of the semester. Members of the team tried to work on their own branches so that we would not modify the same files concurrently.



Chapter 5

Language Evolution

Written by Spencer Brown (Language Guru)

5.1 Language Evolution

Ripple was conceived as an imperative programming language that implemented the reactive programming paradigm. To achieve this reactivity, we identified several important ideas from which to work from. From there, we decided decided those elements that were most important to achieving this paradigm, in addition to being feasible, and then set a priority on our implementation of them. Ultimately, we had two key ideas: easy interaction with streams of data and progogation of change through linked variables.

Once we had decided on these ideas, we focused our attention on achieving them in the language, and added other ideas as we could. We achieved these two concepts with Ripple, but the actual implementation suffered some compromises. We provide interaction with streams of data through StreamReaders, and propogation of change through the link statement. Additionally, you can link StreamReaders with variables to have an asynchronously updating variable and subsequent propagation tree. However, as we implemented the language, we realized it made sense to abstract away much of the boiler plate of grabbing data. As such, we provide the ability for users to interact with the data streamreader grabs through functions they provide. These functions can parse the input and the return will be the updated linked variable.

Ultimately, we achieved the reactivity of Ripple that we desired, but the usage of the language saw some changes throughout the design and implementation process.

5.2 The Compiler Tools

Like most groups, we used a lex-yacc combination for our compiler tools. In our case, we used flex and bison to code our compiler in C/C++ and generate our intermediate C/C++ code.

5.3 Unusual Libraries

The only truly unusual library used within the compiler is libcurl. This library is used by the WebStream-Reader to provide well-supported functionality for scraping addresses. We also used pthread, something slightly strange for C++, as we were most familiar with this style of threading and libcurl did not support c++ std::threads.

5.4 LRM vs. Compiler

The LRM served as the basis for our compiler. In constructing the compiler, however, we discovered many problems. These, we would evaluated on a case-by-base basis in which we decided if changes were necessary. Small elements sometimes needed changing from the LRM to actually work in lex/yacc, as well as sometimes

major changes in the structure of a statement, like that of link and StreamReader, forced us to return to the LRM and update it.

Chapter 6

Translator Architecture

Written by Alex Roth (System Architect)

6.1 The Ripple Translator

6.1.1 `flex` and `bison`

A Ripple program begins its translation by first being read into a series of tokens through `flex`, and is converted into an abstract syntax tree through the use of `bison`. During construction of the abstract syntax tree, each node on the tree becomes decorated with the intermediate C++ code. Once the tree is compiled via bottom-up parsing, the nodes are traversed and generate the intermediate C++ code. This intermediate code is stored in a file known as `output.cpp`, which is linked to a number of standard C++ headers within the `link_files/ripple_header.h` file.

The `flex` file (`ripple.l`) contains the tokens that we create when we read in the standard ripple file. The output is a token stream that is sent to `bison`.

The `bison` file (`ripple.ypp`) contains the whole grammar, and is used to construct the AST. The classes for the abstract syntax tree are defined in the `frontend/ast.h` file and implemented in the `frontend/ast.cpp`.

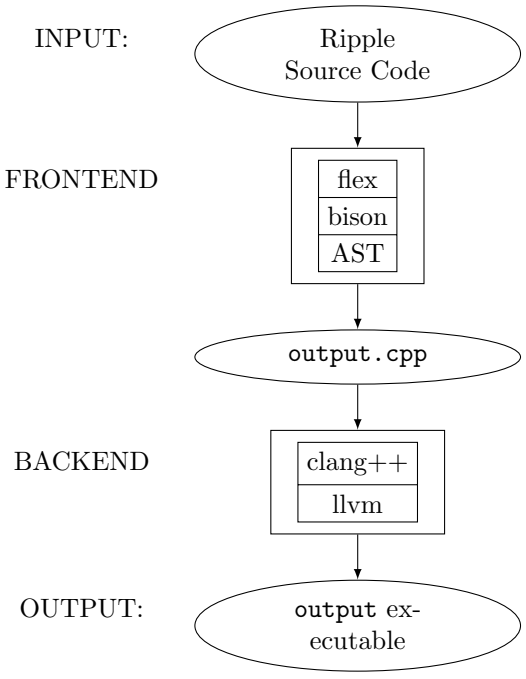


Figure 6.1: Ripple's Compiler Architecture

6.1.2 clang++ and llvm

Once the frontend generates the `output.cpp` file, the backend takes over. Since our intermediary language is C++; we use clang++ and llvm to generate the `output` executable.

6.2 Interfaces Between Modules

Ripple's linking and `StreamReader` features are generated at compile time in the form of special C++ intermediate code. Once a link statement is called, a dependency tree is created within the runtime environment for that variable that is being linked. This dependency tree is actually a vector of dependencies that can be traversed when a value is updated. These dependices are actually references to the linked variables within the symbol table. Thus, when an update occurs on a linked variable, say the value of the linked variable is changed, the `update()` function is called. The `update()` function checks the dependents of the variable being updated, and traverse down the vector of dependents. At each dependent, their dependents are updated as well. Thus, the `update` function acts as an iterative depth-first search.

`StreamReader`s are the second language specific feature for Ripple. A `StreamReader` is converted into two C++ threads: the main thread of the program and the listening thread on the source. The listening thread will listen on the source for any incoming data. In the case of a web reader, this listening is implemented through a refresh rate, which is initially set to 0. The user can define a different integer value for the refresh rate. For the file reader, the user can define a delimiter value that the reader will parse. Thus, a file reader will read up to the delimiter, take in the value, and then continue onwards throughout the file.

The parser libraries included with the `StreamReader`s are C++ functions that are wrapped in Ripple functions. Thus, when a call is made to these standard functions in a Ripple program, the compiler translates the wrapper function into the correct C++ function.

6.3 Who Wrote What?

The Ripple team divided work on the compiler into two distinct groups:

1. the Frontend team, and
2. the Backend team

Both teams were responsible for writing test suites for their individual sections.

6.3.1 The Frontend Team

The Frontend team (Amar and Artur) was in charge of writing the lexical analyzer, syntax analyzer, semantic analyzer, and the intermediate code generation. They wrote the files located within the `frontend` directory. That is, they built the abstract syntax tree, the lex parser, and the yacc parser.

6.3.2 The Backend Team

The Backend team, consisting of Spencer, Tom, and Alex, was tasked with writing the language specific features of Ripple. That is, they were given the task of designing and implementing the linking semantics, designing and implementing the `StreamReader`s, and creating the parsing libraries that accompany the `StreamReader`s.

Chapter 7

Development and Run-time Environment

Written by Artur Renault (System Integrator).

7.1 Development and Run-time Environment

7.1.1 Development Environment

Vagrant

Vagrant was used to unify the development environment among our team members, which was especially important considering two members of our team have Windows machines and two have Macs. Vagrant enabled us to quickly set up a virtual machine we could all develop on with the inclusion in our repository of a Vagrantfile that specfiied the operating system to be installed and all of the provisions we needed.

We used the 64-bit version of Ubuntu 14.04 (trusty) in our development. The provisions included correspond, for the most part, to the tools described below.

GNU Make

We used GNU Make to integrate the development of this project. The main Makefile is in the root directory of our project, and it compiles the entire compiler when "make" is run. Other available targets are "clean", which deletes the files generated by "make", "all", which runs "clean" followed by "make", and "debug", which creates an executable that prints additional debug information

Flex

Flex was the lexical analyzer generator we used, being an easier-to-obtain alternative to lex that is compatible with C++. It functions exactly like lex in all other respects.

GNU Bison

We used bison as our LALR parser generator, since it functions exactly like yacc but can compile C++. It built our parse tree, and due to the object-oriented design of our code, also performed semantic analysis and intermediate code generation.

Clang

Clang++ was our compiler of choice for compiling both our source code and the intermediate code. It appealed to us due to its friendly user interface and ease of adherence to the C++11 standard.

Git

We used git as the version control system of our code. We kept a functioning version of the source in the branch "master" at all times, creating other branches for features we were implementing. Merges were an occasional challenge, but we were able to resolve them without complications.

Github

Our remote repository was stored on Github, chosen because it is a familiar, free service that displays useful statistics about our development. The repository is stored at `https://www.github.com/rippleplt/ripple/`.

ShareLaTeX

ShareLaTeX is a useful online tool for writing and compiling LaTeX code. Thanks to it, we were able to collaborate on our documentation live and create beautiful LaTeX documents.

7.1.2 Run-time environment

Since C++, our target language, is machine-dependent and produces a binary executable, the run-time environment of the Ripple compiler must be the same as the one it was compiled on. This is to say, any computer that has make, flex, bison, and clang installed can run `make` and proceed to execute `rpl` directly.

The only qualification to this statement is that to use certain streams, one must have a few external libraries installed. To construct any thread, the pthread library is necessary. `web_streams`, one must have libcurl installed. In Ubuntu, for example, this is available in the apt-get package "libcurl4-gnutls-dev".

Chapter 8

Test Plan

Written by Tom Segarra (Tester & Verifier)

8.1 Test Methodology

Although testing was of crucial importance in directing the development of the compiler, our team didn't utilize any especially novel approaches to testing, and there was much room for improvement in the test methods we did implement. Our overriding eagerness to understand, plan and implement the language features, along with our relative inexperience in such large-scale technical undertakings, unfortunately meant that testing only became a central part of our process late in the semester. At that time, however – roughly halfway through the project – we started giving tests their due attention, and collaborated to define a coherent, shared method of verifying that the parts of our program worked well, and worked together. What was perhaps unusual was how organically our processes of testing came about. We found, and discussed at our meetings, that we were frequently rewriting the same pieces of code: we would write a small test program for Feature A, then alter the same test slightly to instead test Feature B. This would require us to rewrite Feature A's test, or even more unfortunately, not to realize when a further change broke Feature A. So we started saving each test, writing scripts to run them sequentially, and thereby implementing increasingly comprehensive automated test suites. Each member was responsible for writing tests for his own contributions to the program, and verifying that they ran properly before and after combining them with the common codebase (i.e. the master Git branch).

8.1.1 Backend Tests

For the backend team, we decided that our tests should function generally as proofs-of-concept. They demonstrated the possible and intended use cases, and showed how the code would interact with other parts of the program. This was effective because of the modular nature of most of our language-specific features; the data structures for "link" statements, for instance, nearly constitute a full programming project of their own. Thus small-scale unit tests were required to ensure that all of its parts continued to function properly (e.g. that integer addition still worked after implementing floating point addition), and these grew in importance as the program grew in size. This modularity also exacerbated a dire need for integration tests. Since the intermediate code-generation was performed by one team, and the infrastructure for properly processing that code was performed by another, members of the back-end team had to exercise special (and indeed systematic) care to verify that these all worked together as expected.

All of our tests were written in C++, and most made heavy use of the standard `assert()` checks.

8.1.2 Frontend Tests

For the frontend team, tests were used to check for correctness of our grammar and in our implementation of that grammar. Thus, we have tests for every type of declaration and every type of built-in operation that is possible within our language. These tests helped immensely in the debugging process and during the

building up of the compiler, as they were able to alert the frontend team early and precisely to when the build was broken or when there was a bug in an implementation of a language feature.

8.1.3 Merge Tests

Before any merge from a branch to master (or any larger branch), we ran a series of tests throughout the system.

The smaller branch tests were used to make sure that the incoming branch would not break the build it was entering. We created a series of small tests to check the functionality of the smaller branch. Next, we ran the larger branch tests to ensure that the main branch was stable enough for the merge. After the merge, we would run the test suite again to confirm that the branch was stable again. If any of these tests failed, the team would have to do some debugging and analyze what was going wrong.

Chapter 9

Conclusions

9.1 Lessons Learned As A Team

- Collaborative debugging is very useful We found that discussing bugs and describing the code out loud made it much easier to locate errors in the code and make it more clear and organized. Often it is the case that because of different schedules or perspectives, one person will be able to solve a problem that another teammate has been stuck on.

- Make sure everyone has clear goals and is aware of where everyone else is at all times. When goals were not strict, people would often miss their deadlines or discover new problems that delayed their completion. When we coded together, or simply communicated to each other what our challenges and goals were, we all felt more motivated and capable of solving our problems.

- Trust each others' talents We delegated a lot of tasks throughout our group. Often this resulted in us not knowing what was getting done and what wasn't. While this could be partially solved by addressing the previous issue, we also found it helpful to remember that our teammates are talented and that we can trust each other to think thoroughly about problems and find the best possible solution.

- Weeks of coding can save you hours of planning Lots of times we simply started coding without really knowing what our code needed to do. This unsurprisingly was often unsuccessful; many of our most productive team meetings were ones where we did not write a single line of code. When we finally did get to the code-writing, we found that our code was much better.

- Don't be scared to dive in On the other hand, there were times when we were so hung up on building a complete, functional piece of our project that we never actually sat down and started writing it. A good example was the grammar, which was initially completely dysfunctional. At this point, we decided to start writing a Yacc file for it from the bottom up, parsing values and expressions and only then having a full parser. When we took these baby steps, after adequate planning of course, our design was much more successful.

9.2 Lessons Learned Individually

9.2.1 Amar

The most important lesson I learned from this project was having clearly defined roles and deadlines to make sure every member in the team is aware of the current state of the project, what is left to be done and by when it needs to be finished. Though we set deadlines for most of the semester we did not begin to stick to those deadlines until less than a month before the final project was due.

The second important lesson I learned was to make sure everyone was clear what other members of the team were working on so that interconnected components would work when they were merged together.

Making sure there is communication and understanding among members of the team is crucial when working with a large code base.

The final major lesson I learned from this project was how to effectively use tools designed for teams to work together such as Git, Google Drive and Trello. By making sure these platforms were always up-to-date team members could be aware of what others were working on.

9.2.2 Alex

The biggest lesson I learned from building this language was that "talk is cheap." The backend team, myself included, spent a good amount of our time planning and designing the backend language features, without building larger proof-of-concept tests. However, we would often spend most of our time theorizing, and very little time implementing. Thus, when it came time to add some language features, we realized that we should have more rigorous code samples. With the leadership of Amar, we were able to build small samples, and then we extended the functionality of these samples.

The second lesson for me was to learn when to ask questions, especially when you are confused or concerned with how a section of the language was implemented. Oftentimes, someone who had not been working on a section of the code asked how it functioned, and through the explanation of the code, we would find errors or concepts that needed expansion. This type of code review helped expand the core language features and helped clear up some misconceptions with our language.

Finally, I learned that, when building a compiler, it is very important for every member to take some time and relax. Building a language is a complex task, and every member should not be distracted from other life events. Thus, it is ok to tell your team that you need some time for yourself or that you need a day off.

9.2.3 Artur

This project has taught me a ton about development and teamwork. In terms of development, I have learned a lot about how compilers work, about the intricacies of Makefile design, about Git version control, and the importance of unit testing. I have also gained useful practice with C++, a language to which I had only been previously exposed during the last month of the Advanced Programming class.

In terms of teamwork, this project has also reminded me that I am a bit of a control freak. I noticed that whenever I did not directly see the code for a part of the compiler, I would assume that it wasn't being written and panic. The past few months have been a friendly reminder to trust my teammates a little more.

9.2.4 Spencer

The greatest lesson of this project for me has been to begin writing code sooner rather than later. Thinking about how one might write code is great, but unless you start, it is difficult to conceive of all the difficulties of implementation. I also found that collaborating in the creation and debugging of code to be supremely important and productive for creating mroe fully formed ideas.

In more concrete terms, I expanded my knowledge of c++, threading, and git. I learned how to code in a collaborative environment, and interact with a project in which I did not necessarily know all parts of the code.

9.2.5 Tom

I would divide the things I learned from this project into two broad categories: the technical, which I expected, and the non-technical, which turned out to be much more important. Since it's the largest software project I've ever contributed to, the most rewarding takeaways were generally principles of effectively building a complex system with a small team. Aside from an enhanced familiarity with C++, GNU Make, and language theory, I learned the crucial importance of modularity – that not all members of a team must plan all components of the system. Specialization and careful attention to the interface between parts is a wiser method for coordinating work.

Something I would count as a "lesson," though it isn't easily communicated, is the "know-how" or experience that we accumulated rapidly. No member of the team had previously worked on a single project

for such a long term, and that can be fatal to planning. Each of us now has a more realistic idea of how much work we can expect to achieve in a given amount of time.

The pressure of having to fulfill obligations to my teammates, in conjunction with the sheer complexity of the task, forced me to expect more of myself. Faced with seemingly impossible-to-understand error messages, seemingly impenetrable blocks of error messages, and seemingly unattainable combinations of deadlines, I learned that self-doubt accomplishes nothing, and that instead setting an unreasonably high bar for oneself (or for others) can substantically improve productivity.

9.3 Advice for Future Teams

- Make sure everybody understands the larger picture for the language. Once everyone understands how the language works and how it needs to be implemented, you will be much more successful at implementing the language. Do not leave any loose ends or implementation details to the very end; make sure you all know what you're building before you try to build it. Obviously new issues arise while building a compiler, but try to predict these within reason.

- Set clear goals and deadlines. People are much more likely to complete a task if they are held accountable for what they sign up for. For this to be effective, goals need to be clear, responsibilities need to be set, and deadlines need to be honored.

- Ask questions and understand the details. Don't be scared of asking the professor, your mentor, or other students about things that you might be stuck on. Chances are, somebody has run into a similar issue before and may be able to provide you some guidance on how to proceed or where to look.

- Have fun and horse around. Writing a compiler is hard, but it is also an exciting learning experience. By the end of the semester you'll be able to brag that you have come up with a brand new programming language, written thousands of lines of code, and watched your programs compile and execute. You will also spend a lot of time with your team, so you might as well get along, tell jokes, and laugh throughout your long hack sessions that will soon dominate your friday nights.

- Start early, and don't be afraid to put a lot of time into the language. A compiler is a lot of work, so yes, you should start early. You should also put a lot of work in; the earlier you do this, the better, because it will expose more issues in your code. Since you started early you'll be able to solve them; you don't want to be stuck debugging at 5PM on the Sunday the project is due.

- Don't take PLT and OS concurrently. Seriously. You may think you can do it, and you may be right, but you don't want to. You will not be able to put as much thought into your language as you should, and you will not have enough time to finish all your OS assignments. Our entire group was silly enough to take both classes at once, so take it from us. You have been warned.

9.4 Suggestions for the Instructor

- A clearer definition of what the roles are. The entire time we were a little confused about what exactly the responsibilities of each role are. To some extent, it sounds like the system architect is responsible for writing the majority of the compiler, while the system integrator just writes Makefiles, the tester just writes tests, and the language guru handles theoretical things. A clearer description of each person's responsibilities, perhaps from the previous year, might have made them easier to understand.

- Perhaps a flipped classroom model would work for this class. Learning the material at home and having class time reserved for meetings with the group and the professor could be an interesting way to revamp this course. We felt like we learned much more from actually writing code than from going to class, and while the class is already very focused on this project, we felt like some of the class material and work distracted us from its actual development.

- Leave the CS theory in CS theory. The best example of the above is the excessive review of CS theory. We felt like the time we used to review the theory of DFAs and CFGs in the beginning of the class could have been better used if Lex and Yacc had been introduced earlier. Those were expected knowledge from having taken CS Theory and we feel like we did not need the review, since it only delayed our ability to start the project, since we were waiting on a lot of crucial information.

- Have a good team from the previous semester present. Seeing a presentation early in the semester would have been very welcome to clarify exactly what's expected from the project. There's only so much one can learn from a project report, and the opportunity to ask questions to former PLTers in person would have been very helpful.

- Emphasize important things like testing and style from the beginning. Testing and style, among other important programming techniques, became important in the latter half of the class. However, we feel like some coding techniques to which we had not necessarily been exposed beforehand were not introduced or really emphasized until we already had produced a lot of bad code. We recommend introducing these techniques very early in the classroom.

Appendices

Appendix A

Source Code

```
1       Ripple
2  ~*~make waves~*~
3
4  Authors:
5  Alex Roth
6  Amar Dhingra
7  Artur Renault
8  Spencer Brown
9  Tom Segarra
10
11 To compile the Ripple compiler:
12 1) Make sure you have flex, bison, and clang++ installed.
13 2) Run "make"
14
15 To compile a Ripple program:
16 ./rpl <input_filename> [output_filename]
17
18 If you don't provide the output_filename, the output file will be
19 simply called output.
20
21 Enjoy!
```

../source-code/README.md

```
1  .PHONY: default
2  default: rpl
3
4  VPATH=.:frontend:misc
5
6  debug: MODE=-DDEBUG
7  debug: rpl
8
9  MODE=
10 CC=gcc-4.9
11 CXX=clang++
12 LEX=flex
13 YACC=bison
14 CXXFLAGS= -std=c++11 -w $(INCLUDES) $(MODE)
15 LDLIBS= -L./frontend/symbol_table/ -L./backend/lib/
16 INCLUDES= -I./link_files/
```

```
17 YFLAGS= -Wnone
18 LFLAGS=
19 MISCFLAGS=
20 OBJS=ast.o ripple.tab.o lex.yy.o frontend/symbol_table/symbol_table.o \
21      frontend/symbol_table/hashmap.o debug_tools.o
22 BACKEND_OBJS=backend/linked_var.o backend/expression_tree.o backend/link_val.o
23
24 rpl: ast.o ripple.tab.o lex.yy.o debug_tools.o libsym.a libbackend.a libfile.a
25         $(CXX) -o rpl $(OBJS) $(LDLIBS) -lfl -lfile -lxml -lhtml
26         rm -f *.o *.hpp *.cpp *.c *.cc
27
28 ast.o: ast.cpp ast.h
29         $(CXX) -c frontend/ast.cpp $(CXXFLAGS)
30
31 debug_tools.o: debug_tools.cpp debug_tools.h
32         $(CXX) -c misc/debug_tools.cpp $(CXXFLAGS)
33
34 lex.yy.o: lex.yy.c ripple.tab.h ast.h debug_tools.h
35         $(CXX) -c lex.yy.c $(CXXFLAGS)
36
37 lex.yy.c: ripple.l ripple.tab.h ast.h
38         $(LEX) frontend/ripple.l $(LFLAGS)
39
40 ripple.tab.o: ripple.tab.cpp ripple.tab.h ast.h
41         $(CXX) -c ripple.tab.cpp $(CXXFLAGS)
42
43 ripple.tab.cpp ripple.tab.h: ripple.ypp ast.h
44         $(YACC) -d frontend/ripple.ypp $(YFLAGS)
45
46 libsym.a:
47         $(MAKE) -C frontend/symbol_table
48
49 libbackend.a: backend/linked_var.o backend/expression_tree.o backend/link_val.
   ↪ o
50         ar rcs libbackend.a $(BACKEND_OBJS)
51         ranlib libbackend.a
52
53 libfile.a:
54         $(MAKE) -C backend all
55
56 .PHONY: clean
57 clean:
58         rm -f *.o *.hpp *.cpp *.c *.cc *.a rpl output
59         $(MAKE) -C frontend/symbol_table clean
60         $(MAKE) -C backend mrproper
61
62 .PHONY: all
63 all: clean default
```

../source-code/Makefile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure(2) do |config|
  # The most common configuration options are documented and commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://atlas.hashicorp.com/search.
  config.vm.box = "ubuntu/trusty64"

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  # config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # config.vm.network "forwarded_port", guest: 80, host: 8080

  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  config.vm.network "private_network", ip: "192.168.33.10"

  # Create a public network, which generally matched to bridged network.
  # Bridged networks make the machine appear as another physical device on
  # your network.
  # config.vm.network "public_network"

  # Share an additional folder to the guest VM. The first argument is
  # the path on the host to the actual folder. The second argument is
  # the path on the guest to mount the folder. And the optional third
  # argument is a set of non-required options.
  # config.vm.synced_folder "../data", "/vagrant_data"

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  # Example for VirtualBox:
  #
  # config.vm.provider "virtualbox" do |vb|
  #   # Display the VirtualBox GUI when booting the machine
  #   vb.gui = true
  #
  #   # Customize the amount of memory on the VM:
  #   vb.memory = "1024"
  # end
  #
  # View the documentation for the provider you are using for more
```

```
  # information on available options.

  # Define a Vagrant Push strategy for pushing to Atlas. Other push strategies
  # such as FTP and Heroku are also available. See the documentation at
  # https://docs.vagrantup.com/v2/push/atlas.html for more information.
  # config.push.define "atlas" do |push|
  #   push.app = "YOUR_ATLAS_USERNAME/YOUR_APPLICATION_NAME"
  # end

  # Enable provisioning with a shell script. Additional provisioners such as
  # Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the
  # documentation for more information about their specific syntax and use.
  config.vm.provision "shell", inline: <<-SHELL
     sudo add-apt-repository ppa:ubuntu-toolchain-r/test
     sudo apt-get update -y
     sudo apt-get install -y clang
     sudo apt-get install -y git
     sudo apt-get install -y cmake
     sudo apt-get install -y valgrind
     sudo apt-get install -y flex
     sudo apt-get install -y bison
     sudo apt-get upgrade -y
     locale-gen
     echo "ALEX SUCKS"
  SHELL
end
```

../source-code/Vagrantfile

```
CC       = g++-4.9
CXX      = clang++
CCFLAGS = -pthread -std=c++11
OBJS     = linked_var.o link_val.o expression_tree.o

.PHONY: objects
objects: $(OBJS)

linked_var.o: linked_var.cpp linked_var.h
        $(CXX) -c linked_var.cpp $(CCFLAGS)

expression_tree.o: expression_tree.cpp expression_tree.h
        $(CXX) -c expression_tree.cpp $(CCFLAGS)

link_val.o: link_val.cpp link_val.h
        $(CXX) -c link_val.cpp $(CCFLAGS)

.PHONY: clean-subdirs
clean-subdirs:
        $(MAKE) -C parse_lib clean
        $(MAKE) -C tests clean

.PHONY: clean
clean:
        rm -rf $(OBJS) streamreader/*.gch

```

```
27 .PHONY: clean-all
28 clean-all: clean clean-subdirs
29
30 .PHONY: subdirs
31 subdirs:
32         $(MAKE) -C parse_lib all
33         $(MAKE) -C tests all
34
35 .PHONY: all
36 all: clean objects subdirs
37
38 .PHONY: mrproper
39 mrproper:
40         $(MAKE) clean
41         $(MAKE) clean-subdirs
42         rm -rf lib
```

../source-code/backend/Makefile

```
1  #ifndef __EXPRESSION_TREE_H__
2  #define __EXPRESSION_TREE_H__
3
4  #include <algorithm>
5  #include <iostream>
6  #include <string>
7  #include <string.h>
8  #include <vector>
9
10 #include "../structures/enum.h"
11 #include "link_val.h"
12
13 /*
14  * The nodes defined in this file are used at runtime to construct
15  * an ExpressionNode, which represents the expression to which a
16  * linked variable is linked.
17  *
18  * All node classes implement the evaluate() method, which returns a
19  * link_val struct representing the value of a given node.
20  */
21
22 using namespace std;
23
24
25 class ExpressionNode;
26 class BinaryExpressionNode;
27 class UnaryExpressionNode;
28 class ValueNode;
29 class LiteralNode;
30 class VariableNode;
31
32
33 union operand {
34     BinaryExpressionNode *b_exp;
35     UnaryExpressionNode *u_exp;
36     ValueNode *v_node;
```

```
37 };
38
39 class LiteralNode {
40 public:
41     link_val val;
42     vector<void*> *dependencies;
43
44     LiteralNode (int i);
45     LiteralNode (double f);
46     LiteralNode (bool b);
47     LiteralNode (const char *s);
48     link_val evaluate();
49 };
50
51
52 /*
53  * Used to represent regular variables (e.g. "int x") in a link
54  * expression.
55  * Ex: If x is a Ripple integer, then "link (y <- x)" should create a
56  * VariableNode for x.
57  *
58  */
59 class VariableNode {
60 public:
61     link_val val;
62     void *var;
63     vector<void*> *dependencies;
64
65     VariableNode (int *var);
66     VariableNode (double *var);
67     VariableNode (bool *var);
68     VariableNode (string **var);
69     link_val evaluate();
70 };
71
72
73 /*
74 <<<<<<< HEAD
75  * Represents a call to a function in a link expression.
76  */
77 class FunctionCallNode {
78 public:
79     link_val val;
80     void *fn;
81     vector<void*> *dependencies;
82
83     FunctionCallNode (void *);
84     link_val evaluate ();
85 };
86
87
88 /*
89 =======
90 >>>>>>> streamreader
```

```
91  * Contains a LiteralNode or a VariableNode.
92  */
93 class ValueNode {
94 public:
95         bool is_literal;
96         bool is_expression;
97         LiteralNode *lit_node;
98         VariableNode *var_node;
99         ExpressionNode *expr_node;
100        vector<void*> *dependencies;
101
102        ValueNode (LiteralNode *l);
103        ValueNode (VariableNode *v);
104        ValueNode (ExpressionNode *e);
105        link_val evaluate();
106 };
107
108
109 class UnaryExpressionNode {
110 public:
111     enum e_op op;
112     union operand right_operand;
113
114     vector<void*> *dependencies;
115
116     UnaryExpressionNode(UnaryExpressionNode *u, string _op);
117     UnaryExpressionNode(ValueNode *v);
118     link_val evaluate();
119 };
120
121
122 class BinaryExpressionNode {
123 public:
124     union operand left_operand;
125     union operand right_operand;
126     enum e_op op;
127     bool left_is_binary;
128     bool right_is_binary;
129     vector<void*> *dependencies;
130
131     BinaryExpressionNode(BinaryExpressionNode *bl, string _op,
        ↪ BinaryExpressionNode *br);
132     BinaryExpressionNode(BinaryExpressionNode *bl, string _op,
        ↪ UnaryExpressionNode *ur);
133     BinaryExpressionNode(UnaryExpressionNode *ul);
134     link_val evaluate();
135 };
136
137
138 class ExpressionNode {
139 public:
140     BinaryExpressionNode *bin_exp;
141     ValueNode *value;
142     vector<void*> *dependencies;
```

```
143
144     ExpressionNode();
145     ~ExpressionNode();
146     ExpressionNode(BinaryExpressionNode *b);
147     link_val evaluate();
148     static vector<void*> *dep_union(vector<void*> *r1, vector<void*> *r2);
149 };
150
151 #endif
```

../source-code/backend/expression_tree.h

```
1  #include "expression_tree.h"
2
3  enum e_op str_to_op(const std::string op_string) {
4      if(op_string.compare("+") == 0)
5          return PLUS;
6      else if (op_string.compare("-") == 0)
7          return MINUS;
8      else if (op_string.compare("*") == 0)
9          return TIMES;
10     else if (op_string.compare("/") == 0)
11         return DIV;
12     else if (op_string.compare("//") == 0)
13         return FLDIV;
14     else if (op_string.compare("^") == 0)
15         return EXP;
16     else if (op_string.compare("and") == 0)
17         return bAND;
18     else if (op_string.compare("or") == 0)
19         return bOR;
20     else if (op_string.compare("not") == 0)
21         return bNOT;
22     else if (op_string.compare("==") == 0)
23         return EQ;
24     else if (op_string.compare("!=") == 0)
25         return NE;
26     else if (op_string.compare(">") == 0)
27         return GT;
28     else if (op_string.compare("<") == 0)
29         return LT;
30     else if (op_string.compare(">=") == 0)
31         return GE;
32     else if (op_string.compare("<=") == 0)
33         return LE;
34     else if (op_string.compare("@") == 0)
35         return SIZE;
36         else
37                 return NONE;
38 };
39
40
41 /* UnaryExpressionNode */
42 UnaryExpressionNode::UnaryExpressionNode(UnaryExpressionNode *u, string _op)
43 {
```

```
44      op = str_to_op(_op);
45      right_operand.u_exp = u;
46          dependencies = u->dependencies;
47  }
48
49
50  UnaryExpressionNode::UnaryExpressionNode(ValueNode *v)
51  {
52      op = NONE;
53      right_operand.v_node = v;
54          dependencies = v->dependencies;
55  }
56
57
58  link_val UnaryExpressionNode::evaluate() {
59          link_val result = (op == NONE) ? this->right_operand.v_node->evaluate
                ↪ () :
60                  (op == bNOT) ? !this->right_operand.u_exp->evaluate() :
61                  (op == MINUS) ? -this->right_operand.u_exp->evaluate() :
62                  (op == SIZE) ? this->right_operand.u_exp->evaluate().size_node
                        ↪ () :
63                  this->right_operand.u_exp->evaluate();
64          return result;
65  }
66
67
68  /* BinaryExpressionNode */
69  BinaryExpressionNode::BinaryExpressionNode(BinaryExpressionNode *bl, string
        ↪ _op,BinaryExpressionNode *br) {
70      left_operand.b_exp = bl;
71      right_operand.b_exp = br;
72      op = str_to_op(_op);
73      left_is_binary = right_is_binary = true;
74          dependencies = ExpressionNode::dep_union(bl->dependencies, br->
                ↪ dependencies);
75  }
76
77
78  BinaryExpressionNode::BinaryExpressionNode(BinaryExpressionNode *bl, string
        ↪ _op, UnaryExpressionNode *ur) {
79      left_operand.b_exp = bl;
80      right_operand.u_exp = ur;
81      op = str_to_op(_op);
82      left_is_binary = true;
83      right_is_binary = false;
84          if (bl->dependencies == NULL && ur->dependencies == NULL)
85                  dependencies = NULL;
86          else if (bl->dependencies == NULL)
87                  dependencies = ur->dependencies;
88          else if (ur->dependencies == NULL)
89                  dependencies = bl->dependencies;
90          else
91                  dependencies = ExpressionNode::dep_union(bl->dependencies, ur
                        ↪ ->dependencies);
```

```
92 }
93
94
95 BinaryExpressionNode::BinaryExpressionNode(UnaryExpressionNode *ul) {
96     left_operand.u_exp = ul;
97     op = NONE;
98         dependencies = ul->dependencies;
99 }
100
101
102 link_val BinaryExpressionNode::evaluate() {
103         link_val result, left_value, right_value;
104
105         if (this->op == NONE)
106                 return this->left_operand.u_exp->evaluate();
107
108         left_value = left_is_binary ? this->left_operand.b_exp->evaluate() :
109                 this->left_operand.u_exp->evaluate();
110         right_value = right_is_binary ? this->right_operand.b_exp->evaluate()
            ↪ :
111                 this->right_operand.u_exp->evaluate();
112
113         switch(op) {
114         case (PLUS):
115                 return left_value + right_value;
116                 break;
117         case (TIMES):
118                 return left_value * right_value;
119                 break;
120         case (MINUS):
121                 return left_value - right_value;
122                 break;
123         case (DIV):
124                 return left_value / right_value;
125                 break;
126         case (EXP):
127                 return left_value ^ right_value;
128                 break;
129         case (GT):
130                 return left_value > right_value;
131                 break;
132         case (LT):
133                 return left_value < right_value;
134                 break;
135         case (GE):
136                 return left_value >= right_value;
137                 break;
138         case (LE):
139                 return left_value <= right_value;
140                 break;
141         case (EQ):
142                 return left_value == right_value;
143                 break;
144         case (NE):
```

```
145                 return left_value != right_value;
146                 break;
147         case (bAND):
148                 return left_value && right_value;
149                 break;
150         case (bOR):
151                 return left_value || right_value;
152                 break;
153         default:
154                 result.type = ltNONE;
155                 result.value.ptr = NULL;
156         }
157         return result;
158 }
159
160
161 /* ExpressionNode */
162 ExpressionNode::ExpressionNode() { }
163 ExpressionNode::~ExpressionNode() { }
164 ExpressionNode::ExpressionNode(BinaryExpressionNode *b) {
165     bin_exp = b;
166     value = NULL;
167         dependencies = b->dependencies;
168 }
169
170
171 link_val ExpressionNode::evaluate() {
172         return this->bin_exp->evaluate();
173 }
174
175
176 vector<void *> *ExpressionNode::dep_union(vector<void *> *r1, vector<void *> *
    ↪ r2) {
177         int i;
178         if (r1 == NULL && r2 == NULL)
179                 return NULL;
180         else if (r1 == NULL)
181                 return r2;
182         else if (r2 == NULL)
183                 return r1;
184
185         // Exclude duplicates from union
186         int j;
187         bool is_duplicate;
188         for (i = 0; i < r2->size(); i++) {
189                 is_duplicate = false;
190                 for (j = 0; j < r1->size(); j++)
191                         if ((*r1)[j] == (*r2)[i])
192                                 is_duplicate = true;
193                 if (!is_duplicate)
194                         r1->push_back( (*r2)[i] );
195         }
196
197         free(r2);
```

```
198         return r1;
199 }
200
201
202 /* ValueNode */
203 ValueNode::ValueNode(LiteralNode *l) {
204         this->is_literal = true;
205         this->is_expression = false;
206         this->lit_node = l;
207
208         this->dependencies = NULL;
209 }
210
211
212 ValueNode::ValueNode(VariableNode *v) {
213         this->is_literal = false;
214         this->is_expression = false;
215         this->var_node = v;
216
217         this->dependencies = v->dependencies;
218 }
219
220
221 ValueNode::ValueNode(ExpressionNode *e) {
222         this->is_literal = false;
223         this->is_expression = true;
224         this->expr_node = e;
225
226         this->dependencies = e->dependencies;
227 }
228 link_val ValueNode::evaluate() {
229         return is_literal ? lit_node->evaluate() :
230                 is_expression? expr_node->evaluate() :
231                 var_node->evaluate();
232 }
233
234
235 /* LiteralNode */
236 LiteralNode::LiteralNode(int i) {
237         this->val.type = ltINT;
238         this->val.value.intval = i;
239         this->dependencies = NULL;
240 }
241
242
243 LiteralNode::LiteralNode(double d) {
244         this->val.type = ltDOUBLE;
245         this->val.value.doubleval = d;
246         this->dependencies = NULL;
247 }
248
249
250 LiteralNode::LiteralNode(bool b) {
251         this->val.type = ltBOOL;
```

```
252         this->val.value.boolval = b;
253         this->dependencies = NULL;
254 }
255
256
257 LiteralNode::LiteralNode(const char *s) {
258         this->val.type = ltSTR;
259         this->val.value.strval = new string(s);
260         this->dependencies = NULL;
261 }
262
263
264 link_val LiteralNode::evaluate() {
265         return this->val;
266 }
267
268 /* VariableNode */
269 VariableNode::VariableNode(int *var) {
270         this->var = var;
271         this->val.type = ltINT_PTR;
272         this->val.value.ptr = (void *)var;
273
274         this->dependencies = new vector<void *>();
275         this->dependencies->push_back(this->val.value.ptr);
276 }
277
278
279 VariableNode::VariableNode(double *var) {
280         this->var = var;
281         this->val.type = ltDOUBLE_PTR;
282         this->val.value.ptr = (void *)var;
283
284         this->dependencies = new vector<void *>();
285         this->dependencies->push_back(this->val.value.ptr);
286 }
287
288
289 VariableNode::VariableNode(bool *var) {
290         this->var = var;
291         this->val.type = ltBOOL_PTR;
292         this->val.value.ptr = (void *)var;
293
294         this->dependencies = new vector<void *>();
295         this->dependencies->push_back(this->val.value.ptr);
296 }
297
298
299 VariableNode::VariableNode(string **s) {
300         this->var = s;
301         this->val.type = ltSTR_PTR;
302         this->val.value.ptr = (void *)s;
303
304         this->dependencies = new vector<void *>();
305         this->dependencies->push_back(this->val.value.ptr);
```

```
306 }
307
308
309 link_val VariableNode::evaluate() {
310         return this->val;
311 }
```

../source-code/backend/expression_tree.cpp

```
 1 #ifndef __LINK_VAL_H__
 2 #define __LINK_VAL_H__
 3
 4 #include <cmath>
 5 #include <iostream>
 6 #include <stdio.h>
 7 #include <string.h>
 8
 9 /*
10  * The link_val class represents a linked variable's value and its
11  * type, so that we can deal with dynamic types.
12  */
13
14 using namespace std;
15
16 enum link_val_type {
17         ltINT,
18         ltINT_PTR,
19         ltDOUBLE,
20         ltDOUBLE_PTR,
21         ltBOOL,
22         ltBOOL_PTR,
23         ltSTR,
24         ltSTR_PTR,
25         ltNONE
26 };
27
28 class link_val {
29 public:
30         enum link_val_type type;
31         union {
32                 int intval;
33                 double doubleval;
34                 bool boolval;
35                 string *strval;
36                 void *ptr;
37         } value;
38
39         static bool is_bool_op(const char *op);
40         static link_val integer_op(link_val a, link_val b, const char *op);
41         static link_val integer_op(link_val a, const char *op);
42         static link_val double_op(link_val a, link_val b, const char *op);
43         static link_val double_op(link_val a, const char *op);
44         static link_val bool_op(link_val a, link_val b, const char *op);
45         static link_val bool_op(link_val a, const char *op);
46         static link_val str_op(link_val a, link_val b, const char *op);
```

```
47          int get_int_val() const;
48          double get_double_val() const;
49          bool get_bool_val() const;
50          string get_str_val() const;
51          link_val size_node();
52          link_val operator+(const link_val &other) const;
53          link_val operator-(const link_val &other) const;
54          link_val operator*(const link_val &other) const;
55          link_val operator/(const link_val &other) const;
56          link_val operator^(const link_val &other) const;
57          link_val operator>(const link_val &other) const;
58          link_val operator<(const link_val &other) const;
59          link_val operator>=(const link_val &other) const;
60          link_val operator<=(const link_val &other) const;
61          link_val operator==(const link_val &other) const;
62          link_val operator!=(const link_val &other) const;
63          link_val operator&&(const link_val &other) const;
64          link_val operator||(const link_val &other) const;
65          link_val operator!() const;
66          link_val operator-() const;
67
68  private:
69          // static int integer_op(int a, int b, const char *op);
70          template <typename T>
71                  static T generic_op(T a, T b, const char *op);
72          template <typename T> static T generic_op(T a, const char *op);
73          static link_val link_val_op(link_val a, link_val b, const char *op);
74          static link_val link_val_op(link_val a, const char *op);
75  };
76
77  #endif
```

../source–code/backend/link_val.h

```
1   #include "link_val.h"
2
3   bool link_val::is_bool_op(const char *op) {
4           return (!strcmp(op, ">") ||
5                   !strcmp(op, "<") ||
6                   !strcmp(op, ">=") ||
7                   !strcmp(op, "<=") ||
8                   !strcmp(op, "==") ||
9                   !strcmp(op, "!="));
10  }
11
12  int link_val::get_int_val() const {
13          return (type == ltINT) ? value.intval :
14                  (type == ltINT_PTR) ? *(int *)value.ptr:
15                  0; // should never happen
16  }
17
18  double link_val::get_double_val() const {
19          return (type == ltDOUBLE) ? value.doubleval :
20                  (type == ltINT) ? value.intval :
21                  (type == ltINT_PTR) ? *(int *)value.ptr :
```

```
22                  (type == ltDOUBLE_PTR) ? *(double *)value.ptr:
23                  0;
24  }
25
26  bool link_val::get_bool_val() const {
27          return (type == ltDOUBLE) ? !!value.doubleval :
28                  (type == ltINT) ? !!value.intval :
29                  (type == ltBOOL) ? value.boolval :
30                  (type == ltINT_PTR) ? !!*(int *)value.ptr :
31                  (type == ltDOUBLE_PTR) ? !!*(double *)value.ptr :
32                  (type == ltBOOL_PTR) ? *(bool *)value.ptr:
33                  0;
34  }
35
36  string link_val::get_str_val() const {
37          switch(type) {
38          case (ltINT) :
39                  return to_string(value.intval);
40                  break;
41          case (ltINT_PTR) :
42                  return to_string(*(int *)value.ptr);
43                  break;
44          case (ltSTR) :
45                  return *value.strval;
46                  break;
47          case (ltSTR_PTR) :
48                  return *(string *)value.ptr;
49                  break;
50          case (ltDOUBLE) :
51                  return to_string(value.doubleval);
52                  break;
53          case (ltDOUBLE_PTR) :
54                  return to_string(*(double *)value.ptr);
55                  break;
56          case (ltBOOL) :
57                  return to_string(value.boolval);
58                  break;
59          case (ltBOOL_PTR) :
60                  return to_string(*(bool *)value.ptr);
61                  break;
62          default:
63                  return NULL;
64          }
65  }
66
67  link_val link_val::size_node() {
68          link_val *result = new link_val();
69          result->type = ltINT;
70          result->value.intval =
71                  (type == ltINT) ? sizeof(int) :
72                  (type == ltDOUBLE) ? sizeof(double) :
73                  (type == ltBOOL) ? sizeof(bool) :
74                  (type == ltINT_PTR) ? sizeof(void *) :
75                  (type == ltDOUBLE_PTR) ? sizeof(void *) :
```

```
76                  (type == ltBOOL_PTR) ? sizeof(void *) :
77                  0;
78
79          return *result;
80  }
81
82  template <typename T>
83  T link_val::generic_op(T a, const char *op) {
84          if (!strcmp(op, "!"))
85                  return !a;
86          else
87                  return 0;
88  }
89
90  template <typename T>
91  T link_val::generic_op(T a, T b, const char *op) {
92          if (!strcmp(op, "+"))
93                  return a + b;
94          else if (!strcmp(op, "-"))
95                  return a - b;
96          else if (!strcmp(op, "*"))
97                  return a * b;
98          else if (!strcmp(op, "/"))
99                  return a / b;
100         else if (!strcmp(op, "//"))
101                 return a / b; // @TODO
102         else if (!strcmp(op, "^"))
103                 return (T)pow(a, b);
104         else if (!strcmp(op, ">"))
105                 return a > b;
106         else if (!strcmp(op, "<"))
107                 return a < b;
108         else if (!strcmp(op, ">="))
109                 return a >= b;
110         else if (!strcmp(op, "<="))
111                 return a <= b;
112         else if (!strcmp(op, "=="))
113                 return a == b;
114         else if (!strcmp(op, "!="))
115                 return a != b;
116         else if (!strcmp(op, "&&"))
117                 return a && b;
118         else if (!strcmp(op, "||"))
119                 return a || b;
120         else
121                 return 0;
122 }
123
124 link_val link_val::link_val_op(link_val a, link_val b, const char *op) {
125         if (a.type == ltSTR || b.type == ltSTR ||
126                         a.type == ltSTR_PTR || b.type == ltSTR_PTR) {
127                 return str_op(a, b, op);
128         }
129         if (a.type == ltBOOL || b.type == ltBOOL ||
```

```
130                         a.type == ltBOOL_PTR || b.type == ltBOOL_PTR)
131                 return bool_op(a, b, op);
132         if (a.type == ltDOUBLE || b.type == ltDOUBLE ||
133                         a.type == ltDOUBLE_PTR || b.type == ltDOUBLE_PTR)
134                 return double_op(a, b, op);
135         return integer_op(a, b, op);
136 }
137
138 link_val link_val::link_val_op(link_val a, const char *op) {
139         if (a.type == ltINT || a.type == ltINT_PTR)
140                 return integer_op(a, op);
141         else if (a.type == ltDOUBLE || a.type == ltDOUBLE_PTR)
142                 return double_op(a, op);
143         return bool_op(a, op); // logical not
144 }
145
146 link_val link_val::str_op(link_val a, link_val b, const char *op) {
147         string a_str, b_str;
148         link_val *result = new link_val();
149
150         a_str = a.get_str_val();
151         b_str = b.get_str_val();
152
153         result->type = ltSTR;
154
155         if (!strcmp(op, "+")) {
156                 string *temp = new string(a_str + b_str);
157                 result->value.strval = temp;
158         }
159
160         return *result;
161 }
162
163 link_val link_val::integer_op(link_val a, link_val b, const char *op) {
164         int a_int, b_int;
165         link_val *result = new link_val();
166
167         a_int = a.get_int_val();
168         b_int = b.get_int_val();
169
170         if (is_bool_op(op)) {
171                 result->type = ltBOOL;
172                 result->value.boolval = generic_op<int>(a_int, b_int, op);
173         } else {
174                 result->type = ltINT;
175                 result->value.intval = generic_op<int>(a_int, b_int, op);
176         }
177
178         return *result;
179 }
180
181 link_val link_val::integer_op(link_val a, const char *op) {
182         // Currently only includes negation
183         int a_int;
```

```
184         link_val *result = new link_val();
185
186         a_int = a.get_int_val();
187
188         result->type = ltINT;
189         result->value.intval = -a_int;
190
191         return *result;
192 }
193
194 link_val link_val::double_op(link_val a, link_val b, const char *op) {
195         double a_double, b_double;
196         link_val *result = new link_val();
197
198         a_double = a.get_double_val();
199         b_double = b.get_double_val();
200
201         if (is_bool_op(op)) {
202                 result->type = ltBOOL;
203                 result->value.boolval = generic_op<double>(a_double, b_double,
                        ↪ op);
204         } else {
205                 result->type = ltDOUBLE;
206                 result->value.doubleval = generic_op<double>(a_double,
                        ↪ b_double, op);
207         }
208
209         return *result;
210 }
211
212 link_val link_val::double_op(link_val a, const char *op) {
213         // Currently only negation
214         double a_double;
215         link_val *result = new link_val();
216
217         a_double = a.get_double_val();
218
219         result->type = ltDOUBLE;
220         result->value.doubleval = -a_double;
221
222         return *result;
223 }
224
225 link_val link_val::bool_op(link_val a, link_val b, const char *op) {
226         bool a_bool, b_bool;
227         link_val *result = new link_val();
228
229         a_bool = a.get_bool_val();
230         b_bool = b.get_bool_val();
231
232         result->type = ltBOOL;
233         result->value.boolval = generic_op<bool>(a_bool, b_bool, op);
234
235         return *result;
```

```
236 }
237
238 link_val link_val::bool_op(link_val a, const char *op) {
239         bool a_bool;
240         link_val *result = new link_val();
241
242         a_bool = a.get_bool_val();
243
244         result->type = ltBOOL;
245         result->value.boolval = generic_op<bool>(a_bool, op);
246
247         return *result;
248 }
249
250 link_val link_val::operator+(const link_val &other) const {
251         return link_val_op(*this, other, "+");
252 }
253
254 link_val link_val::operator-(const link_val &other) const {
255         return link_val_op(*this, other, "-");
256 }
257
258 link_val link_val::operator*(const link_val &other) const {
259         return link_val_op(*this, other, "*");
260 }
261
262 link_val link_val::operator/(const link_val &other) const {
263         return link_val_op(*this, other, "/");
264 }
265
266 link_val link_val::operator^(const link_val &other) const {
267         return link_val_op(*this, other, "^");
268 }
269
270 link_val link_val::operator>(const link_val &other) const {
271         return link_val_op(*this, other, ">");
272 }
273
274 link_val link_val::operator<(const link_val &other) const {
275         return link_val_op(*this, other, "<");
276 }
277
278 link_val link_val::operator>=(const link_val &other) const {
279         return link_val_op(*this, other, ">=");
280 }
281
282 link_val link_val::operator<=(const link_val &other) const {
283         return link_val_op(*this, other, "<=");
284 }
285
286 link_val link_val::operator==(const link_val &other) const {
287         return link_val_op(*this, other, "==");
288 }
289
```

```
290 link_val link_val::operator!=(const link_val &other) const {
291         return link_val_op(*this, other, "!=");
292 }
293
294 link_val link_val::operator&&(const link_val &other) const {
295         return link_val_op(*this, other, "&&");
296 }
297
298 link_val link_val::operator||(const link_val &other) const {
299         return link_val_op(*this, other, "||");
300 }
301
302 link_val link_val::operator!() const {
303         return link_val_op(*this, "!");
304 }
305
306 link_val link_val::operator-() const {
307         return link_val_op(*this, "-");
308 }
```

../source-code/backend/link_val.cpp

```
 1 #ifndef __LINKED_VAR_H__
 2 #define __LINKED_VAR_H__
 3
 4 #include <iostream>
 5 #include <unordered_map>
 6 #include <vector>
 7
 8 #include "expression_tree.h"
 9 #include "link_val.h"
10 #include "linked_var.h"
11
12 using namespace std;
13
14 union aux_fn {
15     void (*int_fn)(int);
16     void (*double_fn)(double);
17     void (*bool_fn)(bool);
18     void (*str_fn)(string *);
19 };
20
21
22 /*
23  * The linked_var class represents a single linked variable.
24  */
25 class linked_var {
26 private:
27     link_val value; // Current value
28     void *address; // Address of corresponding C++ variable
29     ExpressionNode expression; // Linked expression
30     union aux_fn aux; // Auxiliary function
31
32     void update_cpp_var();
33     void call_aux(void *arg);
```

```
34 public:
35     // Hash map from a memory address to a list of linked_vars
36     //   which depend on that memory address.
37     static unordered_map<void *, vector<linked_var *>*> references;
38
39     bool has_aux;
40     static void register_cpp_var (void *var);
41     static void update_nonlinked_var (void *var);
42     static void reset_refs ();
43     linked_var(void *var, ExpressionNode *exp); // Ctor
44     link_val get_value();
45     void update(link_val new_value); // Assignment
46     void update(); // For testing only
47     void assign_aux_fn(void *fn);
48 };
49
50 #endif
```

../source–code/backend/linked_var.h

```
1  #include "linked_var.h"
2
3  /*
4   * Global hash map from memory address to list of linked_vars
5   * which reference that memory address.
6   */
7  unordered_map<void *, vector<linked_var *>*> linked_var::references;
8
9  /*
10  * Creates a linked_var object, which represents a linked variable.
11  * It takes a pointer to the corresponding C++ variable, and the
12  * link expression represented as an ExpressionNode.
13  */
14 linked_var::linked_var(void *var, ExpressionNode *exp) {
15         int i;
16
17         // Assign member values
18         this->address = var;
19         this->expression = *exp;
20         this->value = exp->evaluate();
21         this->has_aux = false;
22
23         // Put references into dependency tree
24         if (exp->dependencies != NULL)
25                 for (i = 0; i < exp->dependencies->size(); i++)
26                         references[(*exp->dependencies)[i]]->push_back(this);
27
28         // Set the corresponding C++ variable to the proper value.
29         this->update_cpp_var();
30 }
31
32
33 link_val linked_var::get_value() {
34     return this->value;
35 }
```

```
36
37
38 void linked_var::update_cpp_var() {
39         switch (this->value.type) {
40         case (ltINT):
41                 *(int *)(this->address) = this->value.value.intval;
42                 break;
43         case (ltINT_PTR):
44                 *(int *)(this->address) = *(int *)this->value.value.ptr;
45                 break;
46         case (ltDOUBLE):
47                 *(double *)(this->address) = this->value.value.doubleval;
48                 break;
49         case (ltDOUBLE_PTR):
50                 *(double *)(this->address) = *(double *)this->value.value.ptr;
51                 break;
52         case (ltSTR):
53                  *(string *)this->address = *(new string(*(this->value.value.
                       ↪ strval)));
54                 break;
55         case (ltSTR_PTR):
56                 *(string *)(this->address) = *(string *)this->value.value.ptr;
57                 break;
58         default:
59                 break;
60         }
61 }
62
63
64 /*
65  * Update the value of a linked variable, and all the linked
66  * variables that depend on it, based on the present values  of the
67  * relevant C++ variables, at least one of which presumably has
68  * changed.
69  */
70 void linked_var::update() {
71         int i;
72         this->value = this->expression.evaluate();
73         this->update_cpp_var();
74
75         if (this->has_aux)
76                 this->call_aux(&this->value.value);
77
78         // Recursively update children
79         if (references[this->address] != NULL)
80                 for (i = 0; i < references[this->address]->size(); i++)
81                         if ((*references[this->address])[i] != this)
82                                 (*references[this->address])[i]->update();
83 }
84
85
86 /*
87  * Set the present linked_var to the given value, then accordingly
88  * update all linked_vars which depend on this one.
```

```
89  */
90 void linked_var::update(link_val new_value) {
91         int i;
92         this->value = new_value;
93         this->update_cpp_var();
94 
95         // Recursively update children
96         for (i = 0; i < references[this->address]->size(); i++)
97                 (*references[this->address])[i]->update();
98 }
99 
100 
101 /*
102  * This function takes a pointer to the C++ variable to be used in a
103  * linked_var. (Each linked_var must have one.) It creates an
104  * entry for that var in the dependency tree, if one doesn't already
105  * exist.
106  */
107 void linked_var::register_cpp_var (void *var) {
108         if (references[var] == NULL)
109                 references[var] = new vector<linked_var*>();
110 }
111 
112 
113 /*
114  * Call this function immediately after directly updating a nonlinked
115  * variable. It will propagate the changes to all vars linked to it,
116  * directly or indirectly.
117  */
118 void linked_var::update_nonlinked_var (void *var) {
119         int i;
120         if (references[var] != NULL) {
121                 for (i = 0; i < references[var]->size(); i++) {
122                         (*references[var])[i]->update();
123                 }
124         }
125 }
126 
127 
128 /*
129  * Unregister all cpp vars. Used for testing.
130  */
131 void linked_var::reset_refs () {
132         references.clear();
133 }
134 
135 
136 void linked_var::call_aux (void *arg) {
137         switch(this->value.type) {
138         case (ltINT):
139                 this->aux.int_fn(*(int *)arg);
140                 break;
141         case (ltINT_PTR):
142                 this->aux.int_fn(**(int **)arg);
```

```
143                 break;
144         case (ltDOUBLE) :
145                 this->aux.double_fn(*(double *)arg);
146                 break;
147         case (ltDOUBLE_PTR):
148                 this->aux.double_fn(**(double **)arg);
149                 break;
150         case (ltBOOL) :
151                 this->aux.bool_fn(*(bool *)arg);
152                 break;
153         case (ltBOOL_PTR):
154                 this->aux.bool_fn(**(bool **)arg);
155                 break;
156         case (ltSTR) :
157                 this->aux.str_fn(*(string **)arg);
158                 break;
159         case (ltSTR_PTR) :
160                 this->aux.str_fn(*(string **)arg);
161                 break;
162         default :
163                 return;
164         }
165 }
166
167
168 void linked_var::assign_aux_fn (void *fn) {
169         this->has_aux = true;
170
171         switch(this->value.type) {
172         case (ltINT) :
173                 this->aux.int_fn = (void (*)(int))fn;
174                 break;
175         case (ltINT_PTR) :
176                 this->aux.int_fn = (void (*)(int))fn;
177                 break;
178         case (ltDOUBLE) :
179                 this->aux.double_fn = (void (*)(double))fn;
180                 break;
181         case (ltDOUBLE_PTR) :
182                 this->aux.double_fn = (void (*)(double))fn;
183                 break;
184         case (ltBOOL) :
185                 this->aux.bool_fn = (void (*)(bool))fn;
186                 break;
187         case (ltBOOL_PTR) :
188                 this->aux.bool_fn = (void (*)(bool))fn;
189                 break;
190         case (ltSTR) :
191                 this->aux.str_fn = (void (*)(string *))fn;
192                 break;
193         case (ltSTR_PTR) :
194                 this->aux.str_fn = (void (*)(string *))fn;
195                 break;
196         default :
```

```
197                     return;
198             }
199     }
```

../source–code/backend/linked_var.cpp

```
1  CC           = g++4.9
2  CXX          = clang++
3  CCFLAGS      = -pthread -std=c++11
4  OBJS         = file_lib.o html_lib.o xml_lib.o
5  LIBS         = libfile.a libhtml.a libxml.a
6  LIB_LOCATION = ../lib/libfile.a ../lib/libhtml.a ../lib/libxml.a
7
8  .PHONY: default
9  default: objects libs
10
11 .PHONY: objects
12 objects: $(OBJS)
13
14 file_lib.o: file_lib.h file_lib.cpp
15         $(CXX) -c file_lib.cpp $(CCFLAGS)
16
17 html_lib.o: html_lib.h html_lib.cpp
18         $(CXX) -c html_lib.cpp $(CCFLAGS)
19
20 xml_lib.o: xml_lib.h xml_lib.cpp
21         $(CXX) -c xml_lib.cpp $(CCFLAGS)
22
23 .PHONY: libs
24 libs: $(LIBS)
25
26 libfile.a: file_lib.o
27         if [ ! -d ../lib ]; then mkdir ../lib; fi;
28         ar rcs ../lib/libfile.a file_lib.o
29         ranlib ../lib/libfile.a
30
31 libhtml.a: html_lib.o
32         ar rcs ../lib/libhtml.a html_lib.o
33         ranlib ../lib/libhtml.a
34
35 libxml.a: xml_lib.o
36         ar rcs ../lib/libxml.a xml_lib.o
37         ranlib ../lib/libxml.a
38
39 .PHONY: clean
40 clean:
41         rm -rf $(OBJS) $(LIBS_LOCATION)
42
43 .PHONY: all
44 all: clean objects libs
```

../source–code/backend/parse_lib/Makefile

```
1  #ifndef __FILE_LIB_H__
2  #define __FILE_LIB_H__
```

```
3
4  #include <iostream>
5  #include <stack>
6  #include <string>
7  #include <vector>
8
9  namespace ripple {
10
11     bool contains_word(std::string line, std::string word);
12
13     int length(std::string line);
14     int locate_word(std::string line, std::string word);
15
16     void print_line(std::string line);
17
18 }
19
20 #endif
```

../source–code/backend/parse_lib/file_lib.h

```
1  #include "file_lib.h"
2
3  using namespace std;
4
5
6  bool ripple::contains_word(string line, string word) {
7      return line.find(word) != string::npos ? true : false;
8  }
9
10
11 int ripple::length(string line) {
12     return line.size();
13 }
14
15
16 int ripple::locate_word(string line, string word) {
17     return line.find(word);
18 }
19
20
21 void ripple::print_line(string line) {
22     cout << line << endl;
23 }
```

../source–code/backend/parse_lib/file_lib.cpp

```
1  #ifndef __HTML_LIB_H__
2  #define __HTML_LIB_H__
3
4  #include <iostream>
5  #include <sstream>
6  #include <stack>
7  #include <string>
8  #include <vector>
```

```
9
10 static void empty_stack();
11
12 namespace ripple {
13
14     bool contains_tag(std::string line, std::string tag);
15
16     int get_num_tags(std::string line, std::string tag);
17     int size(std::string line);
18
19     std::string get_body(std::string line);
20     std::string get_head(std::string line);
21     std::string get_tag(std::string line, std::string tag);
22
23     std::string trim(std::string line);
24 }
25
26 #endif
```

../source–code/backend/parse_lib/html_lib.h

```
1 #include "html_lib.h"
2
3 using namespace std;
4
5 static stack<string> parse_stack;
6
7
8 static void empty_stack() {
9
10     while (!parse_stack.empty()) {
11         parse_stack.pop();
12     }
13 }
14
15
16 string ripple::trim(string line) {
17     int index = line.find_first_not_of(" ");
18     return line.substr(index);
19 }
20
21
22 bool ripple::contains_tag(string line, string tag) {
23     return line.find(tag) != string::npos ? true : false;
24 }
25
26
27 int ripple::get_num_tags(string line, string tag) {
28     empty_stack();
29
30     int count = 0;
31
32     string start_tag = "<" + tag + ">";
33     string end_tag = "</" + tag + ">";
34
```

```
35      stringstream node_stream(line);
36      string word;
37
38      for (; node_stream >> word; ) {
39          if (word == start_tag) {
40              parse_stack.push(word);
41          } else if (word == end_tag && parse_stack.top() == start_tag) {
42              parse_stack.pop();
43              ++count;
44          }
45      }
46
47      return count;
48  }
49
50
51  int ripple::size(string line) {
52      return line.size();
53  }
54
55
56  string ripple::get_body(string line) {
57      string body_open_tag = "<body>";
58      string body_close_tag = "</body>";
59
60      int start_index = line.find(body_open_tag);
61      int end_index = line.find(body_close_tag) + body_close_tag.size();
62      int body_size = end_index - start_index;
63
64      string result = line.substr(start_index, body_size);
65      return result;
66  }
67
68
69  string ripple::get_head(string line) {
70      string head_open_tag = "<head>";
71      string head_close_tag = "</head>";
72
73      int start_index = line.find(head_open_tag);
74      int end_index = line.find(head_close_tag) + head_close_tag.size();
75      int head_size = end_index - start_index;
76
77      string result = line.substr(start_index, head_size);
78      return result;
79  }
80
81
82  string ripple::get_tag(string line, string tag) {
83      empty_stack();
84
85      string start_tag = "<" + tag + ">";
86      string end_tag = "</" + tag + ">";
87
88      stringstream node_stream(line);
```

```
89
90      string word;
91      string result;
92
93      for (; node_stream >> word; ) {
94          if (word == end_tag && !parse_stack.empty()) {
95              while (parse_stack.top() != start_tag) {
96                  result = result.insert(0, " " + parse_stack.top());
97                  parse_stack.pop();
98              }
99
100             result = result.insert(0, " " + parse_stack.top());
101             parse_stack.pop();
102         } else {
103             parse_stack.push(word);
104         }
105     }
106
107     return trim(result + " " + end_tag);
108 }
```

../source–code/backend/parse_lib/html_lib.cpp

```
1  #ifndef __XML_LIB_H__
2  #define __XML_LIB_H__
3
4  #include <iostream>
5  #include <sstream>
6  #include <stack>
7  #include <string>
8  #include <vector>
9
10 using namespace std;
11
12 static void empty_stack();
13
14 namespace ripple {
15
16     int get_num_nodes(string line, string tag);
17
18     string get_node(string line, string tag);
19     string get_node_text(string line, string tag);
20
21 }
22
23 #endif
```

../source–code/backend/parse_lib/xml_lib.h

```
1  #include "xml_lib.h"
2  #include "file_lib.h"
3  #include "html_lib.h"
4
5  using namespace std;
6
```

```
7  static stack<string> parse_stack;
8
9
10 static void empty_stack() {
11
12     while (!parse_stack.empty()) {
13         parse_stack.pop();
14     }
15 }
16
17
18 int ripple::get_num_nodes(string line, string tag) {
19     empty_stack();
20
21     int count = 0;
22
23     string start_tag = "<" + tag + ">";
24     string end_tag = "</" + tag + ">";
25
26     stringstream node_stream(line);
27     string word;
28
29     for (; node_stream >> word; ) {
30         if (word == start_tag) {
31             parse_stack.push(word);
32         } else if (word == end_tag && parse_stack.top() == start_tag) {
33             parse_stack.pop();
34             ++count;
35         }
36     }
37
38     return count;
39 }
40
41
42 int locate_word(string line, string word) {
43     return line.find(word);
44 }
45
46
47 string ripple::get_node(string line, string tag) {
48     empty_stack();
49
50     string start_tag = "<" + tag + ">";
51     string end_tag = "</" + tag + ">";
52
53     stringstream node_stream(line);
54
55     string word;
56     string result;
57
58     for (; node_stream >> word; ) {
59         if (word == end_tag && !parse_stack.empty()) {
60             while (parse_stack.top() != start_tag) {
```

```
61                  result = result.insert(0, " " + parse_stack.top());
62                  parse_stack.pop();
63              }
64              result = result.insert(0, " " + parse_stack.top());
65              parse_stack.pop();
66          } else {
67              parse_stack.push(word);
68          }
69
70      }
71
72      return ripple::trim(result + " " + end_tag);
73  }
74
75
76  string ripple::get_node_text(string line, string tag) {
77      empty_stack();
78
79      string start_tag = "<" + tag + ">";
80      string end_tag = "</" + tag + ">";
81
82      stringstream node_stream(line);
83
84      string word;
85      string result;
86
87      for (; node_stream >> word; ) {
88          if (word == end_tag && !parse_stack.empty()) {
89
90              while (parse_stack.top() != start_tag) {
91                  result = result.insert(0, " " + parse_stack.top());
92                  parse_stack.pop();
93              }
94
95              parse_stack.pop();
96          } else {
97              parse_stack.push(word);
98          }
99      }
100
101     return trim(result);
102 }
```

../source-code/backend/parse_lib/xml_lib.cpp

```
1  #ifndef __FILE_STREAM_READER_H__
2  #define __FILE_STREAM_READER_H__
3
4  #include <fstream>
5  #include <sstream>
6  #include "stream_reader.h"
7
8  template <typename T>
9  class FileStreamReader : StreamReader<T>{
10
```

```
11 private:
12     ifstream file_stream;
13     string file_path;
14     string delimiter;
15
16 public:
17     FileStreamReader<T>(void *to_update, typename FuncPtr<T>::f_ptr f =
           ↪ nullptr, string file_path="", int interval=0, const string delim = "
           ↪ \0") {
18         this->to_update = (T *)to_update;
19         this->filter_func_ptr = f;
20         this->file_path = file_path;
21         this->interval = interval;
22         this->delimiter = delim;
23     }
24
25    ~FileStreamReader<T>() {};
26
27     /*
28      *Public accessor function used to begin running the instantiated
           ↪ WebStreamReader.
29      */
30     void start_thread() {
31         if (pthread_create(&(this->stream_thread), NULL, this->
               ↪ run_stream_thread_proxy, this)) {
32             cerr << "Could not create StreamReader" << endl;
33             exit(1);
34         }
35     }
36
37 protected:
38
39     /*
40      *This function is called from start_thred(), as p_threads in c++
41      *cannot be called directly on member functions due to their implicit
42      *this-> accessor. Declaring this function as static acts as workaround
43      *that enables the ability to call the non-static function
           ↪ run_stream_thread.
44      */
45     static void* run_stream_thread_proxy(void *p) {
46         static_cast<FileStreamReader*>(p)->run_stream_thread();
47         return NULL;
48     }
49
50     /*
51      *Function that does work of the thread. Work done in a permanent while
           ↪ loop that will
52      *continuously update a linked variable.
53      *
54      *File and file i/o errors cause the process to exit. Depending on
           ↪ arguments of function,
55      *either entire file or character-delimited strings will be read from a
           ↪ file.
56      */
```

```
57      void run_stream_thread() {
58
59          string read_buffer;
60
61          while(1) {
62              if (this->stop_stream)
63                  break;
64
65              file_stream.open(file_path, ios::in | ios::binary);
66
67              //File failed to open, die
68              if (!file_stream.is_open()) {
69                  cerr << "File not opened properly" << endl;
70                  exit(1);
71              }
72
73              if (delimiter == "\0") {
74                  file_stream.seekg(0, ios::end);
75                  read_buffer.resize(file_stream.tellg());
76                  file_stream.seekg(0, ios::beg);
77                  file_stream.read(&read_buffer[0], read_buffer.size());
78
79                  if (file_stream.fail()) {
80                      cerr << "File I/O error" << endl;
81                      exit(1);
82                  }
83
84                  *this->to_update = this->filter_func_ptr(read_buffer);
85                  linked_var::update_nonlinked_var(this->to_update);
86
87                  if (this->interval)
88                      sleep(this->interval);
89
90              } else {
91                  while(getline(file_stream, read_buffer, *delimiter.c_str())) {
92
93                      if (file_stream.fail()) {
94                          cerr << "File I/O error" << endl;
95                          exit(1);
96                      }
97                      *this->to_update = this->filter_func_ptr(read_buffer);
98                      linked_var::update_nonlinked_var(this->to_update);
99                      if(this->interval)
100                         sleep(this->interval);
101                 }
102             }
103             file_stream.close();
104         }
105     }
106 };
107 #endif
```

../source-code/backend/streamreader/file_stream_reader.h

```
1 #ifndef __KEYBOARD_STREAM_READER_H__
```

```
2  #define __KEYBOARD_STREAM_READER_H__
3
4  #include <fstream>
5  #include <sstream>
6  #include "stream_reader.h"
7
8
9  template <typename T>
10 class KeyboardStreamReader : StreamReader<T> {
11 public:
12
13     KeyboardStreamReader<T> (void *to_update = nullptr, typename FuncPtr<T>::
       ↪ f_ptr f = nullptr) {
14         this->to_update = (T *)to_update;
15         this->filter_func_ptr = f;
16     }
17
18     ~KeyboardStreamReader<T>() {};
19
20     /*
21      * Public accessor function used to begin running the instantiated
           ↪ WebStreamReader.
22      */
23     void start_thread() {
24         if (pthread_create(&(this->stream_thread), NULL, this->
               ↪ run_stream_thread_proxy, this)) {
25             cerr << "Could not create StreamReader" << endl;
26             exit(1);
27         }
28     }
29
30 protected:
31
32     /*
33      * This function is called from start_thred(), as p_threads in c++
34      * cannot be called directly on member functions due to their implicit
35      * this-> accessor. Declaring this function as static acts as workaround
36      * that enables the ability to call the non-static function
           ↪ run_stream_thread.
37      */
38     static void* run_stream_thread_proxy(void *p) {
39         static_cast<KeyboardStreamReader*>(p)->run_stream_thread();
40         return NULL;
41     }
42
43     /*
44      * Function that does work of the thread. Work done in a permanent while
           ↪ loop that will
45      * continuously update a linked variable.
46      *
47      * File and file i/o errors cause the process to exit. Depending on
           ↪ arguments of function,
48      * either entire file or character-delimited strings will be read from a
           ↪ file.
```

```
49      */
50     void run_stream_thread() {
51 
52         string read_buffer;
53 
54         while(1) {
55             if (this->stop_stream)
56                 break;
57 
58             string line;
59             getline(cin, line);
60 
61             *this->to_update = this->filter_func_ptr(line);
62             linked_var::update_nonlinked_var(this->to_update);
63         }
64     }
65 
66 private:
67     ifstream in_stream;
68 };
69 #endif
```

../source-code/backend/streamreader/keyboard_stream_reader.h

```
1  #ifndef __STREAM_READER_H__
2  #define __STREAM_READER_H__
3  
4  #include <pthread.h>
5  #include <iostream>
6  #include <unistd.h>
7  #include <string>
8  
9  #include "../linked_var.h"
10 
11 using namespace std;
12 
13 /*
14  * Template used to define a variable type function pointer.
15  * Used to pass what stream reads to be parsed
16  */
17 template <typename T>
18 struct FuncPtr {
19     typedef T (*f_ptr)(string argument);
20 };
21 
22 /*
23  * Base class for StreamReaders that declares member variables that are
24  * used in the derived classes
25  */
26 template <typename T>
27 class StreamReader {
28 public:
29     StreamReader(){
30         this->stop_stream = false;
31     }
```

```
32
33      ~StreamReader(){
34          this->stop_stream = true;
35          pthread_join(stream_thread, NULL);
36      }
37
38      virtual void start_thread()=0;
39
40  protected:
41      T* to_update;
42      typename FuncPtr<T>::f_ptr filter_func_ptr;
43      pthread_t stream_thread;
44      bool stop_stream;
45      int interval;
46      virtual void run_stream_thread()=0;
47  };
48
49  #endif
```

../source-code/backend/streamreader/stream_reader.h

```
1  #ifndef __WEB_STREAM_READER_H__
2  #define __WEB_STREAM_READER_H__
3
4  #include <vector>
5  #include "stream_reader.h"
6  #include <curl/curl.h>
7
8  #define ERROR_BUF_SIZE 1024
9
10 template <typename T>
11 class WebStreamReader : StreamReader<T>{
12
13 private:
14     unsigned int port;
15     string URL;
16     CURL *curl;
17     CURLcode curl_result;
18
19 public:
20     WebStreamReader(void *to_update, typename FuncPtr<T>::f_ptr f = nullptr,
21                     string URL = nullptr, unsigned int port = 80, int interval =
                            ↪ 0) {
22         this->to_update = (T *)to_update;
23         this->filter_func_ptr = f;
24         this->URL = URL;
25         this->port = port;
26         this->interval = interval;
27     }
28
29     ~WebStreamReader() {}
30
31     /*
32      * Public accessor function used to begin running the instantiated
            ↪ WebStreamReader.
```

```
33          */
34          void start_thread() {
35              if (pthread_create(&(this->stream_thread), NULL, this->
                    ↪ run_stream_thread_proxy, this)) {
36                  cerr << "Could not create StreamReader" << endl;
37                  exit(1);
38              }
39          }
40
41      protected:
42          /*
43           * This function is called from start_thred(), as p_threads in c++
44           * cannot be called directly on member functions due to their implicit
45           * this-> accessor. Declaring this function as static acts as workaround
46           * that enables the ability to call the non-static function
                 ↪ run_stream_thread.
47           */
48          static void* run_stream_thread_proxy(void *p) {
49              static_cast<WebStreamReader*>(p)->run_stream_thread();
50              return NULL;
51          }
52
53          /*
54           * Auxiliary function to be used by cURL library. Determines size of the
                 ↪ data read
55           * in by curl, which allows this data to be written to a string.
56           */
57          static size_t WriteCallback(void *contents, size_t size, size_t nmemb,
                ↪ void *userp) {
58              ((string*)userp)->append((char*)contents, size * nmemb);
59              return size * nmemb;
60          }
61
62          //pthread runs this function to continuously get and pass webpage to
                ↪ either the function or something else
63          void run_stream_thread() {
64              string read_buffer;
65              vector<char> error_buffer(ERROR_BUF_SIZE);
66              int count = 0;
67
68              while(1) {
69                  if (this->stop_stream)
70                      break;
71
72                  curl = curl_easy_init();
73
74                  //Set cURL options
75                  if (curl) {
76                      curl_easy_setopt(curl, CURLOPT_ERRORBUFFER, &error_buffer[0]);
77                      curl_easy_setopt(curl, CURLOPT_URL, URL.c_str());
78
79                      if (port)
80                          curl_easy_setopt(curl, CURLOPT_PORT, port);
81
```

```
82                  curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, &WebStreamReader
                        ↪ ::WriteCallback);
83                  curl_easy_setopt(curl, CURLOPT_WRITEDATA, &read_buffer);
84                  curl_result = curl_easy_perform(curl);
85              }
86
87              curl_easy_cleanup(curl);
88
89              //Result is set to something other than 0 - there was an error so
                    ↪ print and exit program
90              if (curl_result) {
91                  cerr<<"Error from StreamReader " << URL << ": " <<
                        ↪ curl_easy_strerror(curl_result) << endl;
92                  exit(1);
93              }
94
95              *this->to_update = this->filter_func_ptr(read_buffer);
96              linked_var::update_nonlinked_var(this->to_update);
97
98              if (this->interval)
99                  sleep(this->interval);
100         }
101     }
102 };
103 #endif
```

../source-code/backend/streamreader/web_stream_reader.h

```
1  #include "web_stream_reader.h"
2  #include <curl/curl.h>
3
4  int main(){
5      WebStreamReader *stream = new WebStreamReader("www.cs.columbia.edu");
6      stream->run_curl();
7      stream->start_thread();
8
9
10     int count = 0;
11     int count2 = 0;
12     while(1){
13         if(count2 > 10)
14             break;
15         if(count < 10000000){
16             count++;
17         }else{
18             count = 0;
19             count2++;
20         }
21     }
22
23     cout << "ended while" << endl;
24
25     return 0;
```

```
}
```

../source–code/backend/streamreader/web_stream_reader_test.cpp

```
CC          = g++-4.9
CXX         = clang++
CCFLAGS     = -I ../parse_lib -pthread -std=c++11
PARENT_OBJS = ../expression_tree.o ../link_val.o ../linked_var.o
CUR_OBJS    = file_parse_test.o html_parse_test.o tree_test.o \
              vartest.o xml_parse_test.o
EXES        = file_parse_test html_parse_test tree_test vartest \
              xml_parse_test

.PHONY: tests
tests: $(EXES)

file_parse_test: file_parse_test.o
	$(CXX) file_parse_test.o -L ../lib -lfile -lhtml -lxml -o
	    ↪ file_parse_test

file_parse_test.o: file_parse_test.cpp file_parse_test.h
	$(CXX) -c -static file_parse_test.cpp $(CCFLAGS)

html_parse_test: html_parse_test.o
	$(CXX) html_parse_test.o -L ../lib -lhtml -lfile -lxml -o
	    ↪ html_parse_test

html_parse_test.o: html_parse_test.cpp html_parse_test.h
	$(CXX) -c -static html_parse_test.cpp $(CCFLAGS)

tree_test: tree_test.o $(PARENT_OBJS)
	$(CXX) tree_test.o $(PARENT_OBJS) -o tree_test $(CCFLAGS) -lcurl

tree_test.o: tree_test.cpp ../streamreader/keyboard_stream_reader.h
	$(CXX) -c tree_test.cpp $(CCFLAGS)

vartest: vartest.o $(PARENT_OBJS)
	$(CXX) vartest.o $(PARENT_OBJS) -o vartest $(CCFLAGS)

vartest.o: vartest.cpp ../linked_var.o
	$(CXX) -c vartest.cpp $(CCFLAGS)

xml_parse_test: xml_parse_test.o
	$(CXX) xml_parse_test.o -L ../lib -lfile -lhtml -lxml -o
	    ↪ xml_parse_test

xml_parse_test.o: xml_parse_test.cpp xml_parse_test.h
	$(CXX) -c -static xml_parse_test.cpp $(CCFLAGS)

.PHONY: clean
clean:
	rm -rf $(CUR_OBJS) $(EXES)

.PHONY: all
```

```
48 all: clean tests
```

../source-code/backend/tests/Makefile

```
1  #ifndef __FILE_PARSE_TEST_H__
2  #define __FILE_PARSE_TEST_H__
3
4  #include <assert.h>
5  #include <iostream>
6  #include <string>
7
8  #include "file_lib.h"
9
10 using namespace std;
11
12 void test_contains_word(string line);
13
14 void test_does_not_contain_word(string line);
15
16 void test_length(string line);
17 void test_locate_word(string line);
18
19 #endif
```

../source-code/backend/tests/file_parse_test.h

```
1  #include "file_parse_test.h"
2
3  /**
4   * Simple test suite for file parse library.
5   *
6   * Author: Alexander Roth
7   **/
8  int main(int argc, char **argv) {
9      // Set up
10     string test_line = "Here is a sample string";
11
12     // Test contains_word method
13     cout << "ripple::contains_word() Test" << endl;
14     cout << "==========" << endl;
15
16     test_contains_word(test_line);
17     cout << "test_contains_word passed!" << endl;
18
19     test_does_not_contain_word(test_line);
20     cout << "test_does_not_contain_word passed!" << endl;
21     cout << endl;
22
23     // Test length method
24     cout << "ripple::length() Test" << endl;
25     cout << "==========" << endl;
26
27     test_length(test_line);
28     cout << "test_length passed!" << endl;
29     cout << endl;
```

```
30
31      // Test locate_word method
32      // assert(check == local);
33      // assert(check == local);
34      cout << "ripple::locate_word() Test" << endl;
35      cout << "==========================================================" << endl;
36
37      test_locate_word(test_line);
38      cout << "test_locate_word passed!" << endl;
39      cout << endl;
40
41      return 0;
42  }
43
44  void test_contains_word(string line) {
45      bool contains = ripple::contains_word(line, "sample");
46      assert(contains == true);
47  }
48
49  void test_does_not_contain_word(string line) {
50      bool contains = ripple::contains_word(line, "dog");
51      assert(contains == false);
52  }
53
54  void test_length(string line) {
55      int check = line.size();
56      int len = ripple::length(line);
57      assert(len == check);
58  }
59
60  void test_locate_word(string line) {
61      int local = ripple::locate_word(line, "sample");
62      int check = line.find("sample");
63      assert(local != string::npos);
64      assert(check == local);
65  }
```

../source-code/backend/tests/file_parse_test.cpp

```
1  #ifndef __HTML_PARSE_TEST_H__
2  #define __HTML_PARSE_TEST_H__
3
4  #include <assert.h>
5  #include <iostream>
6  #include <string>
7
8  #include "html_lib.h"
9  #include "file_lib.h"
10 #include "xml_lib.h"
11
12 using namespace std;
13
14 void test_contains_tag(string line);
15 void test_contains_word(string line);
16
```

```
17 void test_does_not_contain_tag(string line);
18 void test_does_not_contain_word(string line);
19
20 void test_get_body(string line);
21 void test_get_collection(string line);
22 void test_get_head(string line);
23 void test_get_num_tags(string line);
24 void test_get_tag(string line);
25
26 void test_size(string line);
27
28 #endif
```

../source-code/backend/tests/html_parse_test.h

```
1 #include "html_parse_test.h"
2
3 /**
4  * Small test suite to check the functionality of the html parse library.
5  *
6  * Author: Alexander Roth
7  **/
8 int main() {
9
10     // Setup
11     string test_line = "<html>\n\t<head> Some text </head>\n\t<body> Hello
       ↪ World!"\
12                        " </body>\n</html>";
13
14     // Test contains_tag method
15     cout << "ripple::contains_tag() Tests" << endl;
16     cout << "=============================================================" << endl;
17
18     test_contains_tag(test_line);
19     cout << "test_contains_tag passed!" << endl;
20
21     test_does_not_contain_tag(test_line);
22     cout << "test_does_not_contain_tag passed!" << endl;
23     cout << endl;
24
25     // Test contains_word method
26     cout << "ripple::contains_word() Tests" << endl;
27     cout << "=============================================================" << endl;
28
29     test_contains_word(test_line);
30     cout << "test_contains_word passed!" << endl;
31
32     test_does_not_contain_word(test_line);
33     cout << "test_does_not_contain_word passed!" << endl;
34     cout << endl;
35
36     // Test get_body method
37     cout << "ripple::get_body() Tests" << endl;
38     cout << "=============================================================" << endl;
39
```

```
40      test_get_body(test_line);
41      cout << "test_get_body passed!" << endl;
42      cout << endl;
43
44      // Test get_head method
45      cout << "ripple::get_head() Tests" << endl;
46      cout << "========================================================" << endl;
47
48      test_get_head(test_line);
49      cout << "test_get_head passed!" << endl;
50      cout << endl;
51
52      // Test get_num_tags method
53      cout << "ripple::get_num_tags() Test" << endl;
54      cout << "========================================================" << endl;
55
56      test_get_num_tags(test_line);
57      cout << "test_get_num_tags passed!" << endl;
58      cout << endl;
59
60      // Test size method
61      cout << "ripple::size() Tests" << endl;
62      cout << "========================================================" << endl;
63
64      test_size(test_line);
65      cout << "test_size passed!" << endl;
66      cout << endl;
67
68      return 0;
69  }
70
71  void test_contains_tag(string line) {
72      bool contains = ripple::contains_tag(line, "<head>");
73      assert(contains == true);
74  }
75
76  void test_does_not_contain_tag(string line) {
77      bool contains = ripple::contains_tag(line, "<test>");
78      assert(contains == false);
79  }
80
81  void test_contains_word(string line) {
82      bool contains = ripple::contains_word(line, "text");
83      assert(contains == true);
84  }
85
86  void test_does_not_contain_word(string line) {
87      bool contains = ripple::contains_word(line, "dog");
88      assert(contains == false);
89  }
90
91  void test_get_body(string line) {
92      string body = ripple::get_body(line);
93      assert(body == "<body> Hello World! </body>");
```

```
94 }
95
96 void test_get_head(string line) {
97     string head = ripple::get_head(line);
98     assert(head == "<head> Some text </head>");
99 }
100
101 void test_get_num_tags(string line) {
102     int num_tag = ripple::get_num_tags(line, "body");
103     assert(num_tag == 1);
104 }
105
106 void test_get_tag(string line) {
107     string tag = ripple::get_tag(line, "head");
108     assert(tag == "<head> Some text </head>");
109 }
110
111 void test_size(string line) {
112     int check = line.size();
113     int count = ripple::size(line);
114     assert(check == count);
115 }
```

../source–code/backend/tests/html_parse_test.cpp

```
1  #include <thread>
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5
6  #include "web_stream_reader.h"
7  #include "file_stream_reader.h"
8  #include "keyboard_stream_reader.h"
9
10 using namespace std;
11
12 int int_test_func(string msg){
13     cout << "Test func message: " << msg << endl;
14     return 1;
15 }
16
17 /*float float_test_func(string msg){}
18
19 bool bool_test_func(string msg){}
20
21 string string_test_func(string msg){}
22 */
23 /*int test_1(){
24     string tmp;
25
26     int count = 0;
27     int count2 = 0;
28     int to_update = 0;
29
30
```

```
31      FuncPtr<int>::f_ptr f;
32      f = &test_func;
33      WebStreamReader<int> *streamy = new WebStreamReader<int>("asopidgjewoiprj
            ↪ ", 8001, f);
34
35      streamy->start_thread();
36
37      while(1){
38          if(count ==  1000000){
39              count = 0;
40              count2++;
41          }else{
42              count++;
43          }
44          if(count2 > 10000)
45              break;
46
47      }
48
49      return 0;
50
51  }*/
52
53  //This method is not to be run, we want to see if all
54  //Combinations of declarations are possible
55  void compilation_test(){
56      unsigned int port = 2;
57      FuncPtr<int>::f_ptr f;
58      f = &int_test_func;
59
60
61  }
62
63  int main(){
64      string tmp;
65
66      int count = 0;
67      int count2 = 0;
68      int to_update = 0;
69
70      FuncPtr<int>::f_ptr f;
71      f = &int_test_func;
72
73      FuncPtr<int>::f_ptr g;
74      g = NULL;
75
76
77
78      KeyboardStreamReader<int> *streamy = new KeyboardStreamReader<int>(f);
79
80      streamy->start_thread();
81
82      FileStreamReader<int> *streamy1 = new FileStreamReader<int>("test1.txt", g
            ↪ , 2);
```

```
83
84      FileStreamReader<int> *streamy2 = new FileStreamReader<int>("test2.txt", g
            ↪ , 2);
85
86      streamy1->start_thread();
87
88      streamy2->start_thread();
89
90
91
92      while(1){
93          if(count ==  1000000){
94              count = 0;
95              count2++;
96          }else{
97              count++;
98          }
99          if(count2 > 10000)
100             break;
101
102     }
103
104
105 }
```

../source-code/backend/tests/stream_test.cpp

```
1  #include <iostream>
2  #include <cmath>
3  #include <assert.h>
4
5  #include "../expression_tree.h"
6  #include "../link_val.h"
7  #include "../linked_var.h"
8  #include "../streamreader/keyboard_stream_reader.h"
9  #include "../streamreader/file_stream_reader.h"
10 #include "../streamreader/web_stream_reader.h"
11
12 /* Spaghetti */
13
14 class TreeTest {
15 public:
16         static VariableNode *create_var_node(int *i);
17         static VariableNode *create_var_node(double *d);
18         static LiteralNode *create_literal_node(int i);
19         static LiteralNode *create_literal_node(double d);
20         static linked_var *link_int_var(int *i);
21         static linked_var *link_double_var(double *d);
22         static linked_var *link_bool_lit_op_bool_var
23                 (void *linked, bool b, bool *d, const char *op);
24         static linked_var *link_int_lit_op_double_var
25                 (void *linked, int i, double *d, const char *op);
26         static void test_unary_ops();
27         static void test_nested_expressions();
28         static void test_string_concatenation();
```

```
29          static void test_string_int_concatenation();
30          static void test_aux_fn_expressions();
31          static void test_streamreader_int_expressions();
32          static void test_streamreader_string_expressions();
33          static void test_filestream_reader();
34          static void test_webstream_reader();
35          static void run_all_unit_tests();
36          static void run_all_integration_tests();
37  };
38
39  void test_aux_fn (int n) {
40          cerr << "[AUX_FN] " << n << endl;
41  }
42
43  void test_aux_str_fn (string n) {
44          cerr << "[AUX_FN] " << n << endl;
45  }
46
47  int rpl_str_to_int (string msg) {
48          return atoi(msg.c_str());
49  }
50
51  string default_rpl_str_str (string msg) {
52          return msg;
53  }
54
55  void TreeTest::test_filestream_reader() {
56          string *prefix = new string("Ripple is ");
57          string *suffix = new string();
58          string *sentence = new string();
59
60          linked_var::register_cpp_var(prefix);
61          linked_var::register_cpp_var(suffix);
62          linked_var::register_cpp_var(sentence);
63
64          // link (sentence <- prefix + suffix)
65          //              test_aux_str_fn(sentence);
66          linked_var *sentence_var = new linked_var(sentence, new ExpressionNode
                ↪ (
67                  new BinaryExpressionNode(
68                          new BinaryExpressionNode(
69                                  new UnaryExpressionNode(
70                                  new ValueNode(
71                                  new VariableNode((string **)prefix)))), "+",
72                          new BinaryExpressionNode(
73                                  new UnaryExpressionNode(
74                                  new ValueNode(
75                                  new VariableNode((string **)suffix)))))));
76          sentence_var->assign_aux_fn((void *)&test_aux_str_fn);
77
78          // link (suffix <- str_to_str <- KSR());
79          string *stream = new string();
80          linked_var::register_cpp_var(stream);
81          FuncPtr<string>::f_ptr f = &default_rpl_str_str;
```

```
82          FileStreamReader<string> *sr = new FileStreamReader<string>(suffix, f,
                ↪ "buzzwords.txt", 5, ",");
83          linked_var *suffix_var = new linked_var(suffix, new ExpressionNode (
84                  new BinaryExpressionNode(
85                  new UnaryExpressionNode(
86                  new ValueNode(
87                  new VariableNode((string **)stream))))));
88
89          sr->start_thread();
90          while (1) {}
91  }
92
93  void TreeTest::test_webstream_reader() {
94          string *prefix = new string("Ripple is ");
95          string *suffix = new string();
96          string *sentence = new string();
97
98          linked_var::register_cpp_var(prefix);
99          linked_var::register_cpp_var(suffix);
100         linked_var::register_cpp_var(sentence);
101
102         // link (sentence <- prefix + suffix)
103         //              test_aux_str_fn(sentence);
104         linked_var *sentence_var = new linked_var(sentence, new ExpressionNode
                ↪ (
105                 new BinaryExpressionNode(
106                         new BinaryExpressionNode(
107                                 new UnaryExpressionNode(
108                                 new ValueNode(
109                                 new VariableNode((string **)prefix)))), "+",
110                         new BinaryExpressionNode(
111                                 new UnaryExpressionNode(
112                                 new ValueNode(
113                                 new VariableNode((string **)suffix)))))));
114         sentence_var->assign_aux_fn((void *)&test_aux_str_fn);
115
116         // link (suffix <- str_to_str <- KSR());
117         string *stream = new string();
118         linked_var::register_cpp_var(stream);
119         FuncPtr<string>::f_ptr f = &default_rpl_str_str;
120         WebStreamReader<string> *sr = new WebStreamReader<string>(suffix, f, "
                ↪ www.reddit.com", 80, 5);
121         linked_var *suffix_var = new linked_var(suffix, new ExpressionNode (
122                 new BinaryExpressionNode(
123                 new UnaryExpressionNode(
124                 new ValueNode(
125                 new VariableNode((string **)stream))))));
126
127         sr->start_thread();
128         while (1) {}
129 }
130
131 void TreeTest::test_streamreader_int_expressions() {
132         cerr << "y <- x + 2" << endl;
```

```
133         cerr << "Enter x." << endl;
134
135         int y, x = 2;
136         linked_var::register_cpp_var(&x);
137         linked_var::register_cpp_var(&y);
138
139         // link (y <- x + 2)
140         //      text_aux_fn(y);
141         linked_var *y_var = new linked_var(&y, new ExpressionNode (
142                 new BinaryExpressionNode(
143                         new BinaryExpressionNode(
144                                 new UnaryExpressionNode(
145                                 new ValueNode(
146                                 new VariableNode(&x)))), "+",
147                         new BinaryExpressionNode(
148                                 new UnaryExpressionNode(
149                                 new ValueNode(
150                                 new LiteralNode(2)))))));
151         y_var->assign_aux_fn((void *)&test_aux_fn);
152
153         // link (x <- str_to_int <- KSR());
154         int stream;
155         linked_var::register_cpp_var(&stream);
156         FuncPtr<int>::f_ptr f = &rpl_str_to_int;
157         KeyboardStreamReader<int> *sr = new KeyboardStreamReader<int>(&stream,
              ↪ f);
158
159         linked_var *sr_var = new linked_var(&x, new ExpressionNode (
160                 new BinaryExpressionNode(
161                 new UnaryExpressionNode(
162                 new ValueNode(
163                 new VariableNode(&stream))))));
164
165         sr->start_thread();
166         while (1) {}
167 }
168
169 void TreeTest::test_streamreader_string_expressions() {
170         string *prefix = new string("Ripple is ");
171         string *suffix = new string();
172         string *sentence = new string();
173
174         linked_var::register_cpp_var(prefix);
175         linked_var::register_cpp_var(suffix);
176         linked_var::register_cpp_var(sentence);
177
178         // link (sentence <- prefix + suffix)
179         //              test_aux_str_fn(sentence);
180         linked_var *sentence_var = new linked_var(sentence, new ExpressionNode
              ↪ (
181                 new BinaryExpressionNode(
182                         new BinaryExpressionNode(
183                                 new UnaryExpressionNode(
184                                 new ValueNode(
```

```
185                                     new VariableNode((string **)prefix)))), "+",
186                             new BinaryExpressionNode(
187                                     new UnaryExpressionNode(
188                                     new ValueNode(
189                                     new VariableNode((string **)suffix)))))));
190         sentence_var->assign_aux_fn((void *)&test_aux_str_fn);
191
192         // link (suffix <- str_to_str <- KSR());
193         string *stream = new string();
194         linked_var::register_cpp_var(stream);
195         FuncPtr<string>::f_ptr f = &default_rpl_str_str;  // @TODO !!! default
196         KeyboardStreamReader<string> *sr = new KeyboardStreamReader<string>(
            ↪ suffix, f);
197         linked_var *suffix_var = new linked_var(suffix, new ExpressionNode (
198                 new BinaryExpressionNode(
199                 new UnaryExpressionNode(
200                 new ValueNode(
201                 new VariableNode((string **)stream))))));
202
203         sr->start_thread();
204         while (1) {}
205 }
206
207 void TreeTest::test_aux_fn_expressions() {
208         int y, x = 2;
209         linked_var::register_cpp_var(&x);
210         linked_var::register_cpp_var(&y);
211
212         // link (y <- x + 2)
213         //      text_aux_fn(y);
214         linked_var *y_var = new linked_var(&y, new ExpressionNode (
215                 new BinaryExpressionNode(
216                         new BinaryExpressionNode(
217                                 new UnaryExpressionNode(
218                                 new ValueNode(
219                                 new VariableNode(&x)))), "+",
220                         new BinaryExpressionNode(
221                                 new UnaryExpressionNode(
222                                 new ValueNode(
223                                 new LiteralNode(2)))))));
224         y_var->assign_aux_fn((void *)&test_aux_fn);
225
226         // x = 100;
227         x = 100;
228         linked_var::update_nonlinked_var(&x);
229
230         assert(y == 102);
231 }
232
233 void TreeTest::test_nested_expressions() {
234         int z, x = 1;
235         linked_var::register_cpp_var(&x);
236         linked_var::register_cpp_var(&z);
237
```

```
238         // link (z <- (2 + x) + 2)
239         linked_var *z_var = new linked_var(&z, new ExpressionNode (
240                 new BinaryExpressionNode(
241                         new BinaryExpressionNode(
242                         new UnaryExpressionNode(
243                         new ValueNode(
244                         new ExpressionNode(
245                                 new BinaryExpressionNode(
246                                         new BinaryExpressionNode(
247                                                 new UnaryExpressionNode(
248                                                 new ValueNode(
249                                                 new LiteralNode(2)))), "+",
250                                         new BinaryExpressionNode(
251                                                 new UnaryExpressionNode(
252                                                 new ValueNode(
253                                                 new VariableNode(&x)))))))))),
                                                     ↪ "+",
254                         new BinaryExpressionNode(
255                         new UnaryExpressionNode(
256                         new ValueNode(
257                         new LiteralNode(2))))))));
258
259         assert(z == 5);
260
261         x = 100;
262         linked_var::update_nonlinked_var(&x);
263
264         assert(z == 104);
265         linked_var::reset_refs();
266 }
267
268 void TreeTest::test_string_int_concatenation() {
269         LiteralNode *prefix_litnode = new LiteralNode("We have ");
270         LiteralNode *suffix_litnode =
271                 new LiteralNode(" days left to satisfy Aho.");
272
273         int num = 4;
274         linked_var::register_cpp_var(&num);
275         VariableNode *num_varnode = new VariableNode(&num);
276
277         ValueNode *prefix_valnode = new ValueNode(prefix_litnode);
278         ValueNode *suffix_valnode = new ValueNode(suffix_litnode);
279         ValueNode *num_valnode = new ValueNode(num_varnode);
280
281         UnaryExpressionNode *prefix_unode =
282                 new UnaryExpressionNode(prefix_valnode);
283         UnaryExpressionNode *suffix_unode =
284                 new UnaryExpressionNode(suffix_valnode);
285         UnaryExpressionNode *num_unode =
286                 new UnaryExpressionNode(num_valnode);
287
288         BinaryExpressionNode *prefix_bnode =
289                 new BinaryExpressionNode(prefix_unode);
290         BinaryExpressionNode *suffix_bnode =
```

```
291                 new BinaryExpressionNode(suffix_unode);
292         BinaryExpressionNode *num_bnode =
293                 new BinaryExpressionNode(num_unode);
294
295         ExpressionNode *num_enode =
296                 new ExpressionNode(num_bnode);
297         linked_var *num_var = new linked_var (&num, num_enode);
298
299         BinaryExpressionNode *low_link =
300                 new BinaryExpressionNode(prefix_bnode, "+", num_bnode);
301         BinaryExpressionNode *high_link =
302                 new BinaryExpressionNode(low_link, "+", suffix_bnode);
303
304         string *despair = new string();
305         linked_var::register_cpp_var(&despair);
306         ExpressionNode *despair_enode =
307                 new ExpressionNode(high_link);
308         linked_var *despair_var =
309                 new linked_var (despair, despair_enode);
310
311         for (num = 4; num >= 0; num--) {
312                 linked_var::update_nonlinked_var(&num);
313                 cerr << despair_var->get_value().value.strval->c_str() << endl
                        ↪ ;
314                 assert (!strcmp(despair_var->get_value().value.strval->c_str()
                        ↪ ,
315                         despair->c_str()));
316         }
317
318         cerr << "[TREE_TEST] String-int concatenation passed, but satisfying
                ↪ Aho failed." << endl;
319 }
320
321 void TreeTest::test_string_concatenation() {
322         // Make variable node
323         // [rpl] string predicate = "a problem.";
324         string *predicate = new string ("a problem.");
325         linked_var::register_cpp_var(predicate);
326         VariableNode *predicate_varnode = new VariableNode((string **)
                ↪ predicate);
327         ValueNode *predicate_valnode = new ValueNode (predicate_varnode);
328         UnaryExpressionNode *predicate_unode =
329                 new UnaryExpressionNode (predicate_valnode);
330         BinaryExpressionNode *predicate_bnode =
331                 new BinaryExpressionNode (predicate_unode);
332         ExpressionNode *predicate_enode =
333                 new ExpressionNode (predicate_bnode);
334         linked_var *predicate_var =
335                 new linked_var (predicate, predicate_enode);
336
337         assert(predicate_var->get_value().type == ltSTR_PTR);
338         assert(!strcmp((*(string *)predicate_var->get_value().value.ptr).c_str
                ↪ (), "a problem."));
339
```

```
340         // [rpl] link (string sentence <- "Amar says: we have " + predicate);
341         string *sentence = new string();
342         linked_var::register_cpp_var(sentence);
343 
344         LiteralNode *prefix_litnode = new LiteralNode("Amar says: we have ");
345         ValueNode *prefix_valnode = new ValueNode(prefix_litnode);
346         UnaryExpressionNode *prefix_unode =
347                 new UnaryExpressionNode (prefix_valnode);
348         BinaryExpressionNode *prefix_bnode =
349                 new BinaryExpressionNode (prefix_unode);
350 
351         BinaryExpressionNode *sentence_bnode =
352                 new BinaryExpressionNode (prefix_bnode, "+", predicate_bnode);
353         ExpressionNode *sentence_enode =
354                 new ExpressionNode (sentence_bnode);
355         linked_var *sentence_var =
356                 new linked_var (sentence, sentence_enode);
357 
358         assert(sentence_var->get_value().type == ltSTR);
359         assert(!strcmp((*(string *)(sentence_var->get_value().value.ptr)).
            ↪ c_str(), "Amar says: we have a problem."));
360 
361         // Change suffix
362         // [rpl] predicate = "no problem.";
363         predicate = new string ("no problem.");
364         linked_var::update_nonlinked_var(predicate);
365 
366         assert(!strcmp((*(string *)(sentence_var->get_value().value.ptr)).
            ↪ c_str(), "Amar says: we have a problem."));
367         assert(strcmp((*(string *)(sentence_var->get_value().value.ptr)).c_str
            ↪ (), "Amar says: we have no problem."));
368 
369         linked_var::reset_refs();
370 }
371 
372 void TreeTest::test_unary_ops() {
373         bool root;
374         LiteralNode *bLit = new LiteralNode(true);
375 
376         assert(bLit->evaluate().size_node().value.intval == sizeof(bool));
377 
378         ValueNode *v = new ValueNode(bLit);
379         UnaryExpressionNode *u = new UnaryExpressionNode(v);
380         UnaryExpressionNode *u2 = new UnaryExpressionNode(u, "not");
381         BinaryExpressionNode *b = new BinaryExpressionNode(u2);
382         ExpressionNode *e = new ExpressionNode(b);
383         linked_var *root_var = new linked_var(&root, e);
384 
385         bool root2 = false;
386         VariableNode *bVar = new VariableNode(&root2);
387         linked_var::references[&root2] = new vector<linked_var*>();
388 
389         assert(bVar->evaluate().size_node().value.intval == sizeof(bool *));
390 
```

```
391         ValueNode *v2 = new ValueNode(bVar);
392         UnaryExpressionNode *u3 = new UnaryExpressionNode(v2);
393         UnaryExpressionNode *u4 = new UnaryExpressionNode(u3, "not");
394         BinaryExpressionNode *b2 = new BinaryExpressionNode(u4);
395         ExpressionNode *e2 = new ExpressionNode(b2);
396         linked_var *root2_var = new linked_var(&root2, e2);
397
398         assert(root_var->get_value().type == ltBOOL);
399         assert(root_var->get_value().value.boolval == false);
400         assert(root2_var->get_value().type == ltBOOL);
401         assert(root2_var->get_value().value.boolval == true);
402
403         linked_var::reset_refs();
404 }
405
406 VariableNode *TreeTest::create_var_node(int *i) {
407     // Code
408     VariableNode *v = new VariableNode (i);
409
410     // Test
411     assert(v->val.type == ltINT_PTR);
412     assert(v->val.value.ptr == (void *)i);
413     assert(v->var == i);
414     assert(v->dependencies->size() == 1);
415     assert((*v->dependencies)[0] == i);
416
417     return v;
418 }
419
420 VariableNode *TreeTest::create_var_node(double *d) {
421     // Code
422     VariableNode *v = new VariableNode (d);
423
424     // Test
425     assert(v->val.type == ltDOUBLE_PTR);
426     assert(v->val.value.ptr == (void *)d);
427     assert(v->var == d);
428     assert(v->dependencies->size() == 1);
429     assert((*v->dependencies)[0] == d);
430
431     return v;
432 }
433
434 LiteralNode *TreeTest::create_literal_node(int i) {
435     LiteralNode *l = new LiteralNode(i);
436
437     assert(l->val.type == ltINT);
438     assert(l->val.value.intval == i);
439
440     return l;
441 }
442
443 LiteralNode *TreeTest::create_literal_node(double d) {
444     LiteralNode *l = new LiteralNode(d);
```

```
445
446     assert(l->val.type == ltDOUBLE);
447     assert(l->val.value.doubleval == d);
448
449     return l;
450 }
451
452 void TreeTest::run_all_unit_tests() {
453     int a = 5;
454     int b = 0;
455     double c = 5.5;
456     double d = 5;
457     double e = 0;
458
459     /* VariableNode */
460     create_var_node(&a);
461     create_var_node(&b);
462     create_var_node(&c);
463     create_var_node(&d);
464     create_var_node(&e);
465
466     /* LiteralNode */
467     create_literal_node(a);
468     create_literal_node(b);
469     create_literal_node(c);
470     create_literal_node(d);
471     create_literal_node(e);
472
473         linked_var::reset_refs();
474 }
475
476 /* Link to one integer variable */
477 linked_var *TreeTest::link_int_var(int *i) {
478     // Code
479     VariableNode *v = new VariableNode (i);
480
481     // Test
482     assert(v->val.type == ltINT_PTR);
483     assert(v->val.value.ptr == (void *)i);
484     assert(v->var == i);
485     assert(v->dependencies->size() == 1);
486     assert((*v->dependencies)[0] == i);
487
488     // Code
489     linked_var::references[i] = new vector<linked_var*>();
490     ValueNode *valNode = new ValueNode (v);
491
492     // Test
493     assert(valNode->is_literal == false);
494     assert(valNode->var_node == v);
495     assert(valNode->dependencies->size() == 1);
496     assert((*valNode->dependencies)[0] == i);
497     assert(v->evaluate().value.ptr == i);
498     assert(v->evaluate().type == ltINT_PTR);
```

```
499
500     // Code
501     UnaryExpressionNode *u = new UnaryExpressionNode (valNode);
502
503     // Test
504     assert(u->op == NONE);
505     assert(u->right_operand.v_node == valNode);
506     assert(u->dependencies == valNode->dependencies);
507     assert(u->evaluate().value.ptr == i);
508     assert(u->evaluate().type == ltINT_PTR);
509
510     // Code
511     BinaryExpressionNode *b_x = new BinaryExpressionNode (u);
512     assert(b_x->left_operand.u_exp == u);
513     assert(b_x->op == NONE);
514     assert(b_x->dependencies == u->dependencies);
515     assert(b_x->evaluate().value.ptr == i);
516     assert(b_x->evaluate().type == ltINT_PTR);
517
518     // Code
519     ExpressionNode *e_x = new ExpressionNode (b_x);
520
521     // Test
522     assert(e_x->bin_exp == b_x);
523     assert(e_x->value == NULL);
524     assert(e_x->dependencies == b_x->dependencies);
525     assert(e_x->evaluate().value.ptr == i);
526     assert(e_x->evaluate().type == ltINT_PTR);
527
528     // Code
529     linked_var *var_x = new linked_var (i, e_x);
530     assert(var_x->get_value().value.ptr == i);
531     assert(var_x->get_value().type == ltINT_PTR);
532
533     return var_x;
534 }
535
536 /* Link to one double variable */
537 linked_var *TreeTest::link_double_var(double *d) {
538     // Code
539     VariableNode *v = new VariableNode (d);
540
541     // Test
542     assert(v->val.type == ltDOUBLE_PTR);
543     assert(v->val.value.ptr == (void *)d);
544     assert(v->var == d);
545     assert(v->dependencies->size() == 1);
546     assert((*v->dependencies)[0] == d);
547
548     // Code
549     linked_var::references[d] = new vector<linked_var*>();
550     ValueNode *valNode = new ValueNode (v);
551
552     // Test
```

```
553     assert(valNode->is_literal == false);
554     assert(valNode->var_node == v);
555     assert(valNode->dependencies->size() == 1);
556     assert((*valNode->dependencies)[0] == d);
557     assert(v->evaluate().value.ptr == d);
558     assert(v->evaluate().type == ltDOUBLE_PTR);
559
560     // Code
561     UnaryExpressionNode *u = new UnaryExpressionNode (valNode);
562
563     // Test
564     assert(u->op == NONE);
565     assert(u->right_operand.v_node == valNode);
566     assert(u->dependencies == valNode->dependencies);
567     assert(u->evaluate().value.ptr == d);
568     assert(u->evaluate().type == ltDOUBLE_PTR);
569
570     // Code
571     BinaryExpressionNode *b_x = new BinaryExpressionNode (u);
572     assert(b_x->left_operand.u_exp == u);
573     assert(b_x->op == NONE);
574     assert(b_x->dependencies == u->dependencies);
575     assert(b_x->evaluate().value.ptr == d);
576     assert(b_x->evaluate().type == ltDOUBLE_PTR);
577
578     // Code
579     ExpressionNode *e_x = new ExpressionNode (b_x);
580
581     // Test
582     assert(e_x->bin_exp == b_x);
583     assert(e_x->value == NULL);
584     assert(e_x->dependencies == b_x->dependencies);
585     assert(e_x->evaluate().value.ptr == d);
586     assert(e_x->evaluate().type == ltDOUBLE_PTR);
587
588     // Code
589     linked_var *var_x = new linked_var (d, e_x);
590     assert(var_x->get_value().value.ptr == d);
591     assert(var_x->get_value().type == ltDOUBLE_PTR);
592
593     return var_x;
594 }
595
596 /* Link to int literal (op) double var */
597 linked_var *TreeTest::link_int_lit_op_double_var(void *linked, int i,
598         double *d, const char *op) {
599     // Determine target value
600     double target;
601     if (!strcmp(op, "+"))
602         target = i + *d;
603     else if (!strcmp(op, "-"))
604         target = i - *d;
605     else if (!strcmp(op, "*"))
606         target = i * *d;
```

```
607     else if (!strcmp(op, "/"))
608         target = i / *d;
609     else if (!strcmp(op, "^"))
610         target = pow(i, *d);
611     else
612         target = 0;
613 
614     // Left operand
615     LiteralNode *l1 = new LiteralNode(i);
616     ValueNode *v1 = new ValueNode(l1);
617     UnaryExpressionNode *u1 = new UnaryExpressionNode(v1);
618     BinaryExpressionNode *b1 = new BinaryExpressionNode(u1);
619     assert(b1->evaluate().value.intval == i);
620     assert(b1->evaluate().type == ltINT);
621 
622     // Right operand
623     VariableNode *l2 = new VariableNode(d);
624     ValueNode *v2 = new ValueNode (l2);
625     UnaryExpressionNode *u2 = new UnaryExpressionNode(v2);
626     BinaryExpressionNode *b2 = new BinaryExpressionNode(u2);
627     assert(b2->evaluate().value.ptr == d);
628     assert(b2->evaluate().type == ltDOUBLE_PTR);
629 
630     // Operation
631     BinaryExpressionNode *bin = new BinaryExpressionNode (b1, op, b2);
632     if (link_val::is_bool_op(op)) {
633         assert(bin->evaluate().type == ltBOOL);
634     } else {
635         assert(bin->evaluate().type == ltDOUBLE);
636         assert(bin->evaluate().value.doubleval == target);
637     }
638 
639     ExpressionNode *exp = new ExpressionNode(bin);
640     linked_var *var = new linked_var(linked, exp);
641 
642     return var;
643 }
644 
645 /* Link to bool literal (op) bool var */
646 linked_var *TreeTest::link_bool_lit_op_bool_var(void *linked, bool i,
647         bool *d, const char *op) {
648     // Determine target value
649     bool target;
650     if (!strcmp(op, "and"))
651         target = i && *d;
652     else if (!strcmp(op, "or"))
653         target = i || *d;
654     else
655         target = false;
656 
657     // Left operand
658     LiteralNode *l1 = new LiteralNode(i);
659     ValueNode *v1 = new ValueNode(l1);
660     UnaryExpressionNode *u1 = new UnaryExpressionNode(v1);
```

```
661     BinaryExpressionNode *b1 = new BinaryExpressionNode(u1);
662     assert(b1->evaluate().value.boolval == i);
663     assert(b1->evaluate().type == ltBOOL);
664
665     // Right operand
666     VariableNode *l2 = new VariableNode(d);
667     ValueNode *v2 = new ValueNode (l2);
668     UnaryExpressionNode *u2 = new UnaryExpressionNode(v2);
669     BinaryExpressionNode *b2 = new BinaryExpressionNode(u2);
670     assert(b2->evaluate().value.ptr == d);
671     assert(b2->evaluate().type == ltBOOL_PTR);
672
673     // Operation
674     BinaryExpressionNode *bin = new BinaryExpressionNode (b1, op, b2);
675     assert(bin->evaluate().type == ltBOOL);
676     assert(bin->evaluate().value.boolval == target);
677
678     ExpressionNode *exp = new ExpressionNode(bin);
679     linked_var *var = new linked_var(linked, exp);
680
681     return var;
682 }
683
684 void TreeTest::run_all_integration_tests() {
685         assert(linked_var::references.size() == 0);
686
687         int a = 5;
688         link_int_var(&a);
689
690         double b = 2.5;
691         link_double_var(&b);
692
693         double z;
694         linked_var *z_link;
695         z_link = link_int_lit_op_double_var(&z, a, &b, "+");
696         assert(z_link->get_value().type == ltDOUBLE);
697         assert(z_link->get_value().value.doubleval == 7.5);
698
699         z_link = link_int_lit_op_double_var(&z, a, &b, "-");
700         assert(z_link->get_value().value.doubleval == 2.5);
701
702         z_link = link_int_lit_op_double_var(&z, a, &b, "*");
703         assert(z_link->get_value().value.doubleval == 12.5);
704
705         z_link = link_int_lit_op_double_var(&z, a, &b, "/");
706         assert(z_link->get_value().value.doubleval == 2);
707
708         bool q;
709         z_link = link_int_lit_op_double_var(&q, a, &b, ">");
710         assert(z_link->get_value().value.boolval == true);
711         z_link = link_int_lit_op_double_var(&q, a, &b, ">=");
712         assert(z_link->get_value().value.boolval == true);
713
714         z_link = link_int_lit_op_double_var(&q, a, &b, "<");
```

```
715         assert(z_link->get_value().value.boolval == false);
716         z_link = link_int_lit_op_double_var(&q, a, &b, "<=");
717         assert(z_link->get_value().value.boolval == false);
718 
719         b = 5;
720         z_link = link_int_lit_op_double_var(&q, a, &b, ">=");
721         assert(z_link->get_value().value.boolval == true);
722         z_link = link_int_lit_op_double_var(&q, a, &b, "<=");
723         assert(z_link->get_value().value.boolval == true);
724         z_link = link_int_lit_op_double_var(&q, a, &b, "==");
725         assert(z_link->get_value().value.boolval == true);
726         z_link = link_int_lit_op_double_var(&q, a, &b, "!=");
727         assert(z_link->get_value().value.boolval == false);
728 
729         a = 5, b = 2;
730         z_link = link_int_lit_op_double_var(&z, a, &b, "^");
731         assert(z_link->get_value().value.doubleval == 25);
732 
733         bool r = true;
734         linked_var *r_link;
735         linked_var::references[&r] = new vector<linked_var*>();
736         r_link = link_bool_lit_op_bool_var(&r, (3 > 2), &r, "and");
737         assert(r_link->get_value().value.boolval == true);
738         r_link = link_bool_lit_op_bool_var(&r, (2 > 2), &r, "and");
739         assert(r_link->get_value().value.boolval == false);
740         r = false;
741         r_link = link_bool_lit_op_bool_var(&r, (3 > 2), &r, "or");
742         assert(r_link->get_value().value.boolval == true);
743         r_link = link_bool_lit_op_bool_var(&r, (2 > 2), &r, "or");
744         assert(r_link->get_value().value.boolval == false);
745 
746         test_unary_ops();
747         cerr << "[TREE_TEST] All unary operation tests passed." << endl;
748         linked_var::reset_refs();
749 
750         test_string_concatenation();
751         cerr << "[TREE_TEST] String-string concatenation test passed." << endl
                ↪ ;
752         linked_var::reset_refs();
753         test_string_int_concatenation();
754         cerr << "[TREE_TEST] All string operation tests passed." << endl;
755         linked_var::reset_refs();
756         test_nested_expressions();
757         cerr << "[TREE_TEST] Nested expression tests passed." << endl;
758         linked_var::reset_refs();
759         test_aux_fn_expressions();
760         cerr << "[TREE_TEST] Auxiliary link function tests passed." << endl;
761         linked_var::reset_refs();
762         //test_streamreader_int_expressions();
763         //test_streamreader_string_expressions();
764         test_webstream_reader();
765         linked_var::reset_refs();
766 }
767 
```

```
768 int main(int c, char **argv) {
769     TreeTest::run_all_unit_tests();
770         cerr << "[TREE_TEST] All unit tests passed." << endl;
771     TreeTest::run_all_integration_tests();
772 
773     cerr << "[TREE_TEST] All tests passed." << endl;
774 }
```

../source-code/backend/tests/tree_test.cpp

```
 1 #include <iostream>
 2 
 3 #include "../linked_var.h"
 4 #include "../expression_tree.h"
 5 
 6 /*
 7  * This test is an example of the intermediate code that link
 8  * statements will compile to.
 9  */
10 int main()
11 {
12     /*
13      * Everything is allocated on the heap, since all threads need
14      * access to this data.
15      */
16 
17         // === Assignment ===
18         //                int root = 5;
19     cout << "int root = 5;" << endl;
20     int root = 5;
21         linked_var::register_cpp_var(&root);
22 
23     /* === Link Statement ===
24      *           link (int x <- root);
25      */
26     cout << "link (int x <- root);" << endl;
27         int x;
28         linked_var::register_cpp_var(&x);
29     VariableNode *l = new VariableNode (&root);
30     ValueNode *v = new ValueNode (l);
31     UnaryExpressionNode *u = new UnaryExpressionNode (v);
32     BinaryExpressionNode *b_x = new BinaryExpressionNode (u);
33     ExpressionNode *e_x = new ExpressionNode (b_x);
34     linked_var *var_x = new linked_var (&x, e_x);
35     /* === End of code for link (x <- root) === */
36 
37     /* === Link Statement ===
38      *           link (int y <- x + 2)
39      */
40     cout << "link (int y <- x + 2);" << endl;
41         int y;
42         linked_var::register_cpp_var(&y);
43     VariableNode *l1 = new VariableNode (&x);
44     ValueNode *v1 = new ValueNode (l1);
45     LiteralNode *l2 = new LiteralNode (2);
```

```
46      ValueNode *v2 = new ValueNode (12);
47      UnaryExpressionNode *u1 = new UnaryExpressionNode (v1);
48      UnaryExpressionNode *u2 = new UnaryExpressionNode (v2);
49      BinaryExpressionNode *b1 = new BinaryExpressionNode (u1);
50      BinaryExpressionNode *b2 = new BinaryExpressionNode (u2);
51      BinaryExpressionNode *b_y = new BinaryExpressionNode (b1, "+", b2);
52      ExpressionNode *e_y = new ExpressionNode (b_y);
53      linked_var *var_y = new linked_var (&y, e_y);
54      /* === End of code for "link (y <- x + 2)" === */
55
56      cout << "\tx == " << *(int *)var_x->get_value().value.ptr << endl;
57      cout << "\ty == " << var_y->get_value().value.intval << endl;
58
59      /*
60       * === Assignment ===
61       *           root = 6;
62       */
63      cout << "root = 6;" << endl;
64      root = 6;
65          linked_var::update_nonlinked_var(&root);
66      /* === End of code for "root = 6" === */
67
68      cout << "\tx == " << *(int *)var_x->get_value().value.ptr << endl;
69      cout << "\ty == " << var_y->get_value().value.intval << endl;
70
71      return 0;
72  }
```

../source-code/backend/tests/vartest.cpp

```
1  #ifndef __XML_PARSE_TEST_H__
2  #define __XML_PARSE_TEST_H__
3
4  #include <assert.h>
5  #include <iostream>
6  #include <string>
7  #include <vector>
8
9  #include "file_lib.h"
10 #include "html_lib.h"
11 #include "xml_lib.h"
12
13 using namespace std;
14
15 void test_contains_tag(string line);
16 void test_contains_word(string line);
17
18 void test_does_not_contain_tag(string line);
19 void test_does_not_contain_word(string line);
20
21 void test_get_collection(string line);
22 void test_get_node(string line);
23 void test_get_node_text(string line);
24 void test_get_num_nodes(string line);
25
```

```
26 void test_line_size(string line);
27
28 #endif
```

../source-code/backend/tests/xml_parse_test.h

```
1  #include "xml_parse_test.h"
2
3  using namespace std;
4
5  /**
6   * Small test suite to check the functionality of the xml parsing library.
7   *
8   * Author: Alexander Roth
9   **/
10 int main() {
11
12     // Set up
13     string test_line = "<Book>\n\t<Title> Moby Dick </Title>\n\t<Author>" \
14                        " Herman Melville </Author>\n<Book>";
15     string test_lines = "<Root>\n\t<Books>\n\t\t<Title> Test 1 </Title>\n" \
16                         "\t\t<Title> Test 2 </Title>\n\t</Books>\n</Root>";
17
18     // Test contains_tag method
19     cout << "ripple::contains_tag() Tests" << endl;
20     cout << "====================================================" << endl;
21
22     test_contains_tag(test_line);
23     cout << "test_contains_tag passed!" << endl;
24
25     test_does_not_contain_tag(test_line);
26     cout << "test_does_not_contain_tag passed!" << endl;
27     cout << endl;
28
29     // Test contains_word method
30     cout << "ripple::contains_word() Tests" << endl;
31     cout << "====================================================" << endl;
32
33     test_contains_word(test_line);
34     cout << "test_contains_word passed!" << endl;
35
36     test_does_not_contain_word(test_line);
37     cout << "test_does_not_contain_word passed!" << endl;
38     cout << endl;
39
40     // Test get_node method
41     cout << "ripple::get_node() Test" << endl;
42     cout << "====================================================" << endl;
43
44     test_get_node(test_line);
45     cout << "test_get_node passed!" << endl;
46     cout << endl;
47
48     // Test get_node_text method
49     cout << "ripple::get_node_text() Test" << endl;
```

```
50      cout << "==========================================================" << endl;
51
52      test_get_node_text(test_line);
53      cout << "test_get_node_text passed!" << endl;
54      cout << endl;
55
56
57      // Test get_num_nodes method
58      cout << "ripple::get_num_nodes() Test" << endl;
59      cout << "==========================================================" << endl;
60
61      test_get_num_nodes(test_line);
62      cout << "test_get_num_nodes passed!" << endl;
63      cout << endl;
64
65      // Test size method
66      cout << "ripple::size() Test" << endl;
67      cout << "==========================================================" << endl;
68      test_line_size(test_line);
69      test_line_size(test_lines);
70      cout << "test_line_size passed!" << endl;
71      cout << endl;
72
73      cout << test_lines << endl;
74      return 0;
75  }
76
77  void test_contains_tag(string line) {
78      bool contains = ripple::contains_tag(line, "<Title>");
79      assert(contains == true);
80  }
81
82  void test_contains_word(string line) {
83      bool contains = ripple::contains_word(line, "Herman");
84      assert(contains == true);
85  }
86
87  void test_does_not_contain_tag(string line) {
88      bool contains = ripple::contains_tag(line, "<Dog>");
89      assert(contains == false);
90  }
91
92  void test_does_not_contain_word(string line) {
93      bool contains = ripple::contains_word(line, "Doge");
94      assert(contains == false);
95  }
96
97  void test_get_node(string line) {
98      string author = ripple::get_node(line, "Author");
99      assert(author == "<Author> Herman Melville </Author>");
100 }
101
102 void test_get_node_text(string line) {
103     string text = ripple::get_node_text(line, "Title");
```

```
104     assert(text == "Moby Dick");
105 }
106
107 void test_get_num_nodes(string line) {
108     int count = ripple::get_num_nodes(line, "Author");
109     assert(count == 1);
110 }
111
112 void test_line_size(string line) {
113     int check = line.size();
114     int len = ripple::size(line);
115     assert(check == len);
116 }
```

../source-code/backend/tests/xml_parse_test.cpp

```
1  #ifndef __AST_H__
2  #define __AST_H__
3
4  #include <string>
5  #include <string.h>
6  #include <iostream>
7  #include <vector>
8  #include <algorithm>
9  #include <fstream>
10
11 #include "symbol_table/hashmap.h"
12 #include "symbol_table/symbol_table.h"
13 #include "../structures/enum.h"
14 #include "../structures/union.h"
15 #include "../misc/debug_tools.h"
16
17 #define LINE_ERR "Error on line number " + to_string(line_no) + ": " <<
18
19 #define RPL_STD_OUTPUT_FUNCTION "print"
20 #define RPL_STD_INPUT_FUNCTION "input"
21 #define RPL_STD_OPEN_FUNCTION "open"
22 #define RPL_STD_CLOSE_FUNCTION "close"
23 #define RPL_STD_READ_FUNCTION "read"
24
25 #define RPL_STD_INPUT_FUNCTION_ERR LINE_ERR "input() takes a single string
       ↪ input"
26
27 #define VARIABE_NODE_NAME "VARIABLE_NODE_NAME"
28 #define VALUE_NODE_NAME "VALUE_NODE_NAME"
29 #define UNARY_EXPRESSION_NODE_NAME "UNARY_EXPRESSION_NAME"
30 #define BINARY_EXPRESSION_NODE_NAME "BINARY_EXPRESSION_NAME"
31 #define EXPRESSION_NODE_NAME "EXPRESSION_NODE_NAME"
32
33 #define UNARY_STRING_CAST_ERR LINE_ERR "cannot cast string to bool, int or
       ↪ float"
34 #define INVAL_UNARY_NOT_ERR LINE_ERR "unary - not supported between provided
       ↪ operands"
35 #define INVAL_UNARY_MINUS_ERR LINE_ERR "unary - not supported between provided
       ↪ operands"
```

```
36 #define INVAL_BINARY_PLUS_ERR LINE_ERR "binary + not supported between
       ↪ provided operands"
37 #define INVAL_BINARY_MINUS_ERR LINE_ERR "binary - not supported between
       ↪ provided operands"
38 #define INVAL_BINARY_TIMES_ERR LINE_ERR "binary * not supported between
       ↪ provided operands"
39 #define INVAL_BINARY_DIV_ERR LINE_ERR "binary // not supported between
       ↪ provided operands"
40 #define INVAL_BINARY_EXP_ERR LINE_ERR "binary ^ not supported between provided
       ↪ operands"
41 #define INVAL_BINARY_FLDIV_ERR LINE_ERR "binary / not supported between
       ↪ provided operands"
42 #define INVAL_BINARY_MOD_ERR LINE_ERR "binary % not supported between provided
       ↪ operands"
43 #define INVAL_BINARY_EQ_ERR LINE_ERR "binary == not supported between provided
       ↪ operands"
44 #define INVAL_BINARY_NE_ERR LINE_ERR "binary != not supported between provided
       ↪ operands"
45 #define INVAL_BINARY_GT_ERR LINE_ERR "binary > not supported between provided
       ↪ operands"
46 #define INVAL_BINARY_LT_ERR LINE_ERR "binary < not supported between provided
       ↪ operands"
47 #define INVAL_BINARY_GE_ERR LINE_ERR "binary >= not supported between provided
       ↪ operands"
48 #define INVAL_BINARY_LE_ERR LINE_ERR "binary <= not supported between provided
       ↪ operands"
49 #define INVAL_BINARY_AND_ERR LINE_ERR "binary and not supported between
       ↪ provided operands"
50 #define INVAL_BINARY_OR_ERR LINE_ERR "binary or not supported between provided
       ↪ operands"
51 #define INVAL_FUNC_CALL_ERR LINE_ERR "function call error"
52 #define LOOP_CONDITION_ERR LINE_ERR "loop condition variable must be of type
       ↪ bool"
53 #define UNKNOWN_TYPE_ERR LINE_ERR "unknown type error"
54 #define FUNCTION_BASIC_TYPE_ERR LINE_ERR "functions can only return primitive
       ↪ types"
55 #define RETURN_TYPE_ERROR LINE_ERR "return type does not match function type"
56
57 #define INVALID_DECL_ERR LINE_ERR "all declarations must have have an
       ↪ associated variable"
58 #define VARIABLE_REDECL_ERR LINE_ERR "variable being declared is a
       ↪ redeclaration of a previously declared variable"
59 #define UNDECLARED_ERROR LINE_ERR "use of undeclared identifier"
60
61 #define INVALID_FILE_SR_TYPES_ERR LINE_ERR "invlaid types for file_stream"
62 #define INVALID_KEYBOARD_SR_ERR LINE_ERR "incorrect number of arguments for
       ↪ keyboard_stream error"
63 #define INVALID_FILE_SR_ERR LINE_ERR "incorrect number of arguments for
       ↪ file_stream"
64 #define INVALID_WEB_SR_ERR LINE_ERR "incorrect number of arguments for
       ↪ web_stream"
65 #define INVALID_WEB_SR_TYPES_ERR LINE_ERR "invalid types for web_stream"
66
67 #define ARR_ELEMENT_TYPE_ERR LINE_ERR "all elements in an array initialization
```

```
    ↪ must have the same type"
#define ARR_UNARY_MINUS_ERR LINE_ERR "cannot perform negation on arrays"
#define ARR_UNARY_NOT_ERR LINE_ERR "cannot perform boolean not on arrays"
#define ARR_UNARY_CAST_ERR LINE_ERR "cannot cast array to any other type type"
#define ARR_BINEXP_ERR LINE_ERR "cannot perform binary operations on arrays"
#define ARR_VAR_ASSIGN_ERR LINE_ERR "cannot assign array to non-array variable
    ↪ "
#define ARR_INT_SIZE_ERR LINE_ERR "array initialization size must be int"
#define ARR_UNKNOWN_SIZE_ERR LINE_ERR "variable size array cannot be
    ↪ initialized"
#define ARR_UNKNOWN_INIT_ERR LINE_ERR "cannot create an array of unknown size
    ↪ without initialization"
#define ARR_SMALL_SIZE_ERR LINE_ERR "size of array declared is too small"
#define ARR_ASSIGN_ERR LINE_ERR "can't assign array to non-array variable"

#define FINAL_MUST_INITIALIZE LINE_ERR "must initialize a final variable"
#define FINAL_REDECL_ERR LINE_ERR "cannot change a final variable"
#define UNLINKABLE_NO_VAR_ERR LINE_ERR "linked expression must have variables"
#define UNLINKABLE_EXPRESSION_ERR LINE_ERR "expression provided cannot be
    ↪ linked"
#define INVAL_FUNC_ARGS_ERR LINE_ERR \
                                "an auxiliary function may only have one argument
                                    ↪ of the same type as the linked variable"
#define NOT_A_FUNC_ERR LINE_ERR "provided identifier is not callable"
#define COND_STMT_ERR LINE_ERR "expression in if statement must be of type
    ↪ boolean"
#define LOOP_CONDITION_ERR LINE_ERR "condition expression in loop must be of
    ↪ type boolean"

#define ASSIGN_ERR LINE_ERR "left operand of assignment expression must be a
    ↪ variable"

#define ERROR "error"
#define MAIN_FUNC_ERROR "All ripple programs need a main function."
#define COMPILE_ERR "Unable to complete compilation due to errors in code. Get
    ↪ good."

inline string VARIABLE_NODE(string arg){ return "new VariableNode(&" + arg +
    ↪ ")"; }
inline string LITERAL_NODE(string arg){ return "new LiteralNode(" + arg + ")
    ↪ "; }
inline string VALUE_NODE(string arg){ return  "new ValueNode(" + arg + ")";
    ↪ }
inline string UNARY_EXPRESSION(string arg){ return  "new UnaryExpressionNode(
    ↪ " + arg + ")"; }
inline string UNARY_EXPRESSION(string arg, string op)
            { return  "new UnaryExpressionNode(" + arg + ",\"" + op + "\")"
                ↪ ; }
inline string BINARY_EXPRESSION(string arg1){ return  "new
    ↪ BinaryExpressionNode(" + arg1 + ")"; }
inline string BINARY_EXPRESSION(string arg1, string op, string arg2)
            { return  "new BinaryExpressionNode(" + arg1 + ",\"" + op + "\",
                ↪ " + arg2 + ")"; }
inline string EXPRESSION_NODE(string arg){ return  "new ExpressionNode(" +
```

```
    ↪ arg + "␣)"; }
105
106 using namespace std;
107
108 /* Abstract Syntax Tree node classes */
109 class Node;
110 class ArrayInitNode;
111 class ValueNode;
112 class ExpressionNode;
113 class BinaryExpressionNode;
114 class UnaryExpressionNode;
115 class LiteralNode;
116 class IDNode;
117 class FunctionCallNode;
118 class ArrayAccessNode;
119 class DatasetAccessNode;
120 class ExpressionNode;
121 class ValueNode;
122 class ArgsNode;
123 class DeclArgsNode;
124 class StreamReaderNode;
125 class ConditionalStatementNode;
126 class DeclarativeStatementNode;
127 class JumpStatementNode;
128 class LoopStatementNode;
129 class LinkStatementNode;
130 class StatementListNode;
131 class DatasetNode;
132 class FunctionNode;
133
134 /* Helper functions that translate between enums and strings */
135 enum e_op str_to_op(string op_string);
136 enum e_type str_to_type(string type);
137 enum e_jump str_to_jump(string type);
138 string type_to_str(e_type type);
139
140 void write_to_file(string filename, string code);
141
142 extern SymbolTable sym_table;
143 extern e_type func_type;
144
145 #define IS_STD_RPL_FUNCTION(f_name) (f_name).compare(RPL_STD_INPUT_FUNCTION)
    ↪ == 0    || \
146                                     (f_name).compare(RPL_STD_OUTPUT_FUNCTION)
                                        ↪ == 0    || \
147                                     (f_name).compare(RPL_STD_OPEN_FUNCTION) ==
                                        ↪  0      || \
148                                     (f_name).compare(RPL_STD_READ_FUNCTION) ==
                                        ↪  0      || \
149                                     (f_name).compare(RPL_STD_CLOSE_FUNCTION)
                                        ↪ == 0    || \
150                                     (f_name).compare("contains_word") == 0
                                        ↪             || \
151                                     (f_name).compare("length") == 0
```

```
                                    ↪                    || \
152                                 (f_name).compare("print_line") == 0
                                    ↪                  || \
153                                 (f_name).compare("locate_word") == 0
                                    ↪                || \
154                                 (f_name).compare("contains_tag") == 0
                                    ↪              || \
155                                 (f_name).compare("contains_word") == 0
                                    ↪            || \
156                                 (f_name).compare("get_num_tags") == 0
                                    ↪              || \
157                                 (f_name).compare("size") == 0
                                    ↪                      || \
158                                 (f_name).compare("get_body") == 0
                                    ↪                    || \
159                                 (f_name).compare("get_head") == 0
                                    ↪                    || \
160                                 (f_name).compare("get_tag") == 0
                                    ↪                    || \
161                                 (f_name).compare("get_num_nodes") == 0
                                    ↪              || \
162                                 (f_name).compare("get_node") == 0
                                    ↪                    || \
163                                 (f_name).compare("get_node_text") == 0
164
165 #define INVAL_ASSIGN_ERR(val_type, expression_type) { cout << LINE_ERR \
166     "invalid assignment between operands of type " <<  \
167     val_type << " and " << expression_type << endl; }
168
169 extern int line_no;
170 extern bool error;
171 extern string filename_cpp;
172
173 union operand {
174     BinaryExpressionNode *b_exp;
175     UnaryExpressionNode *u_exp;
176     ValueNode *v_node;
177 };
178
179 union value {
180     IDNode *id_val;
181     LiteralNode *lit_val;
182     FunctionCallNode *function_call_val;
183     ArrayAccessNode *array_access_val;
184     DatasetAccessNode *dataset_access_val;
185     ExpressionNode *expression_val;
186     ArrayInitNode *a_init;
187 };
188
189 union statements {
190     DeclarativeStatementNode *decl;
191     ConditionalStatementNode *cond;
192     JumpStatementNode *jump;
193     LoopStatementNode *loop;
```

```
194     LinkStatementNode *link;
195 };
196
197 union program_section {
198     FunctionNode *function;
199     DatasetNode *dataset;
200     DeclarativeStatementNode *decl;
201 };
202
203
204 /* This is the generic Node class.
205  * All other Nodes inherit from it, so it contains
206  * members that will be used by multiple classes. */
207 class Node {
208     public:
209          string code;
210         string ds_name = "";
211         int array_length;
212         string link_code;
213         std::vector<string *> linked_vars;
214         bool is_linkable = false;
215         e_type type = tNOTYPE;
216         e_type get_type();
217         e_symbol_type sym = tNOSTYPE;
218         bool is_number();
219         bool is_bool();
220         bool is_string();
221         bool returns_value = false;
222 };
223
224 /* The names of Node classes are better understood
225  * when compared to the grammar */
226 class ValueNode: public Node {
227     public:
228         union value val;
229         enum e_value_type val_type;
230
231         ValueNode(IDNode *i);
232         ValueNode(LiteralNode *l);
233         ValueNode(FunctionCallNode *f);
234         ValueNode(ArrayAccessNode *a);
235         ValueNode(DatasetAccessNode *d);
236         ValueNode(ExpressionNode *e);
237         ValueNode(ArrayInitNode *a);
238         void seppuku();
239 };
240
241
242 class IDNode: public Node {
243     public:
244         Entry *entry;
245         IDNode(Entry *ent);
246         string get_name();
247         e_type get_type();
```

```
248         void seppuku();
249 };
250
251
252 class FunctionCallNode: public Node {
253     ArgsNode *args_list;
254     string func_id;
255
256     public:
257     FunctionCallNode(string f, ArgsNode *a);
258     FunctionCallNode(string f);
259     void typecheck();
260     string generate_std_rpl_function();
261     void seppuku();
262 };
263
264
265 class ArrayInitNode: public Node{
266     public:
267         int array_length;
268         std::vector<ExpressionNode *> *args_list;
269         bool has_elements;
270         ArrayInitNode();
271         ArrayInitNode(ExpressionNode *arg);
272         void add_arg(ExpressionNode *arg);
273         void seppuku();
274 };
275
276
277 class ArgsNode: public Node {
278     public:
279         std::vector<ExpressionNode *> *args_list;
280
281         ArgsNode();
282         ArgsNode(ExpressionNode *arg);
283         list<e_type> *to_enum_list();
284         void add_arg(ExpressionNode *arg);
285         void seppuku();
286 };
287
288
289 class TypeNode: public Node {
290     public:
291         ValueNode *value;
292         TypeNode(e_type t, string name);
293         TypeNode(e_type t, ValueNode *val);
294 };
295
296
297 class DeclArgsNode: public Node {
298     std::vector<IDNode *> decl_args_list;
299
300     public:
301     DeclArgsNode();
```

```
302     DeclArgsNode(TypeNode *type, IDNode* arg);
303     void add_arg(TypeNode *type,IDNode* arg);
304     list<e_type> *to_enum_list();
305     vector<IDNode*>::iterator begin();
306     vector<IDNode*>::iterator end();
307     void seppuku();
308 };
309
310
311 class LiteralNode: public Node {
312     public:
313         union literal val;
314         enum e_type type;
315
316         // Constructors for different types
317         LiteralNode(int i);
318         LiteralNode(double d);
319         LiteralNode(string *s);
320         LiteralNode(bool b);
321         void seppuku();
322 };
323
324
325 class ArrayAccessNode: public Node {
326     public:
327         ValueNode *value_node;
328         ExpressionNode *en;
329
330         ArrayAccessNode(ValueNode *v, ExpressionNode *e);
331         void seppuku();
332 };
333
334
335 class DatasetAccessNode: public Node {
336     public:
337         ValueNode *value_node;
338         string id;
339
340         DatasetAccessNode(string c, string i);
341         void seppuku();
342 };
343
344
345 class UnaryExpressionNode: public Node {
346     public:
347         enum e_op op;
348         union operand right_operand;
349
350         UnaryExpressionNode(UnaryExpressionNode *u, string _op);
351         UnaryExpressionNode(UnaryExpressionNode *u, TypeNode *t);
352         UnaryExpressionNode(ValueNode *v);
353         void seppuku();
354
355     private:
```

```
356         void typecheck(e_op op);
357 };
358
359
360 class BinaryExpressionNode: public Node {
361     public:
362         union operand left_operand;
363         union operand right_operand;
364         enum e_op op;
365         bool left_is_binary;
366         bool right_is_binary;
367
368         ValueNode *get_value_node();
369
370         BinaryExpressionNode(BinaryExpressionNode *bl, string _op,
                ↪ BinaryExpressionNode *br);
371         BinaryExpressionNode(BinaryExpressionNode *bl, string _op,
                ↪ UnaryExpressionNode *ur);
372         BinaryExpressionNode(UnaryExpressionNode *ul);
373         void seppuku();
374
375     private:
376         void typecheck(Node *left, Node *right, e_op op);
377         string gen_binary_code(string l_code, enum e_op op, string r_code,
                ↪ e_type l_type, e_type r_type);
378 };
379
380
381 class ExpressionNode: public Node {
382     public:
383         BinaryExpressionNode *bin_exp;
384         ValueNode *value;
385         std::vector<string *> linked_vars;
386
387         ExpressionNode();
388         ExpressionNode(BinaryExpressionNode *b);
389         ExpressionNode(BinaryExpressionNode *b, ValueNode *v);
390         ~ExpressionNode();
391         void seppuku();
392
393     private:
394         void typecheck(BinaryExpressionNode *expression, ValueNode *value);
395
396 };
397
398
399 class DeclarativeStatementNode: public Node {
400     public:
401         e_type type;
402         ExpressionNode *en;
403         ValueNode *a_size;
404
405         DeclarativeStatementNode(TypeNode *t, ExpressionNode *expression_node)
                ↪ ;
```

```
406         DeclarativeStatementNode(ExpressionNode *expression_node);
407         void typecheck();
408         void seppuku();
409 };
410
411
412 class ConditionalStatementNode: public Node {
413     public:
414         ExpressionNode *condition;
415         StatementListNode *consequent;
416         StatementListNode *alternative;
417
418         ConditionalStatementNode(ExpressionNode *e, StatementListNode *s,
                ↪ StatementListNode *a);
419         void seppuku();
420 };
421
422
423 class JumpStatementNode: public Node {
424     public:
425         e_jump type;
426         ExpressionNode *en;
427
428         JumpStatementNode(string _type, ExpressionNode *expression_node);
429         JumpStatementNode(string _type);
430         void seppuku();
431 };
432
433
434 class LoopStatementNode: public Node {
435     public:
436         ExpressionNode *initializer;
437         ExpressionNode *condition;
438         ExpressionNode *next;
439         StatementListNode *statements;
440
441         LoopStatementNode(ExpressionNode *init, ExpressionNode *cond,
                ↪ ExpressionNode *n, StatementListNode *stmts);
442         void seppuku();
443 };
444
445 class StreamReaderNode: public Node {
446 public:
447     string name;
448     ArgsNode *arg_list;
449
450     StreamReaderNode(string n, ArgsNode *args_list);
451     string generate_code(e_type type);
452 };
453
454
455 class LinkStatementNode: public Node {
456     public:
457         IDNode *id_node, *filter;
```

```
458         ExpressionNode *expression_node;
459         StreamReaderNode *stream_reader_node;
460         string auxiliary = "";
461 
462         LinkStatementNode(IDNode *idn, ExpressionNode *expn);
463         LinkStatementNode(IDNode *idn, ExpressionNode *expn, string func);
464         LinkStatementNode(IDNode *idn, StreamReaderNode *srn);
465         LinkStatementNode(IDNode *idn, StreamReaderNode *srn, string func);
466         LinkStatementNode(IDNode *idn, IDNode *filt, StreamReaderNode *srn);
467         LinkStatementNode(IDNode *idn, IDNode *filt, StreamReaderNode *srn,
                ↪ string func);
468 
469         void sepukku();
470 };
471 
472 
473 class StatementNode: public Node {
474     public:
475         union statements stmts;
476 
477         StatementNode(DeclarativeStatementNode *d);
478         StatementNode(ConditionalStatementNode *c);
479         StatementNode(JumpStatementNode *j);
480         StatementNode(LoopStatementNode *l);
481         StatementNode(LinkStatementNode *l);
482         void seppuku();
483 };
484 
485 
486 class StatementListNode: public Node {
487     public:
488         vector<StatementNode *> *stmt_list;
489         SymbolTableNode *st_node;
490 
491         StatementListNode();
492         StatementListNode(SymbolTableNode *s);
493         void push_statement(StatementNode *s);
494         void seppuku();
495 };
496 
497 
498 class DatasetNode: public Node {
499     public:
500         string name;
501         DeclArgsNode *decl_args;
502 
503         DatasetNode(string s, DeclArgsNode *d);
504         void seppuku();
505 };
506 
507 
508 class FunctionNode: public Node {
509     public:
510         enum e_type type;
```

```
511         string id;
512         DeclArgsNode *decl_args;
513         StatementListNode *stmt_list;
514
515         FunctionNode(TypeNode *_type, string id_node, DeclArgsNode *
                ↪ decl_args_list, StatementListNode *stmt_list_n);
516         void seppuku();
517 };
518
519
520 class ProgramSectionNode: public Node {
521     union program_section contents;
522
523     public:
524     ProgramSectionNode(FunctionNode *f);
525     ProgramSectionNode(DatasetNode *d);
526     ProgramSectionNode(DeclarativeStatementNode *dsn);
527     void seppuku();
528 };
529
530
531 class ProgramNode: public Node {
532     public:
533         ProgramNode();
534         void add_section(ProgramSectionNode *);
535         FunctionNode *func;
536         ProgramNode(FunctionNode *f);
537         void seppuku();
538 };
539 #endif
```

../source–code/frontend/ast.h

```
1  #include "ast.h"
2
3  string stream = "stream_name";
4  string function_pointer = "func_pointer";
5  string stream_reader_name = "stream_reader_name";
6  int num = 0;
7
8  /* Takes in a string corresponding to an operator,
9   * returns the corresponding enum e_op */
10 enum e_op str_to_op(const std::string op_string) {
11     if(op_string.compare("+") == 0)
12         return PLUS;
13     else if (op_string.compare("-") == 0)
14         return MINUS;
15     else if (op_string.compare("*") == 0)
16         return TIMES;
17     else if (op_string.compare("/") == 0)
18         return DIV;
19     else if (op_string.compare("%") == 0)
20         return MOD;
21     else if (op_string.compare("//") == 0)
22         return FLDIV;
```

```
23      else if (op_string.compare("^") == 0)
24          return EXP;
25      else if (op_string.compare("and") == 0)
26          return bAND;
27      else if (op_string.compare("or") == 0)
28          return bOR;
29      else if (op_string.compare("not") == 0)
30          return bNOT;
31      else if (op_string.compare("==") == 0)
32          return EQ;
33      else if (op_string.compare("!=") == 0)
34          return NE;
35      else if (op_string.compare(">") == 0)
36          return GT;
37      else if (op_string.compare("<") == 0)
38          return LT;
39      else if (op_string.compare(">=") == 0)
40          return GE;
41      else if (op_string.compare("<=") == 0)
42          return LE;
43      else if (op_string.compare("@") == 0)
44          return SIZE;
45  };
46
47
48  /* Takes in a type in string form, returns an
49   * enum e_type */
50  enum e_type str_to_type(const std::string type){
51      cout << type << endl;
52      if (type.compare("int") == 0)
53          return tINT;
54      else if (type.compare("float") == 0)
55          return tFLOAT;
56      else if (type.compare("void") == 0)
57          return tVOID;
58      else if (type.compare("bool") == 0)
59          return tBOOL;
60      else if (type.compare("string") == 0)
61          return tSTRING;
62      else if (type.compare("byte") == 0)
63          return tBYTE;
64  }
65
66
67  /* Takes in a jump keyword in string form, returns
68   * an enum e_jump */
69  enum e_jump str_to_jump(const std::string type){
70      if (type.compare("return") == 0)
71          return tRETURN;
72      else if (type.compare("break") == 0)
73          return tBREAK;
74      else if (type.compare("continue") == 0)
75          return tCONTINUE;
76      else if (type.compare("stop") == 0)
```

```
77          return tSTOP;
78  }
79
80  /* Does the inverse of str_to_type */
81  string type_to_str(e_type type){
82      switch(type){
83          case tBYTE:
84              return "byte";
85          case tBOOL:
86              return "bool";
87          case tINT:
88              return "int";
89          case tFLOAT:
90              return "float";
91          case tSTRING:
92              return "string";
93          case tVOID:
94              return "void";
95          default:
96              return "undefined type";
97      }
98  }
99
100
101 /* Writes the intermediate code to a file */
102 void write_to_file(string filename, string code){
103     ofstream file;
104     file.open(filename);
105     file << "#include \"link_files/ripple_header.h\"\n\n";
106     file << code;
107     file.close();
108 }
109
110
111 /* Node */
112 e_type Node::get_type() {
113     return this->type;
114 }
115
116
117 bool Node::is_bool() {
118     return (type == tBOOL);
119 }
120
121
122 bool Node::is_number() {
123     return (type == tINT || type == tFLOAT);
124 }
125
126
127 bool Node::is_string() {
128     return (type == tSTRING);
129 }
130
```

```
131
132 /* ValueNode */
133 ValueNode::ValueNode(IDNode *i) {
134     val_type = IDENT;
135     val.id_val = i;
136     type = i->type;
137     sym = i->sym;
138     code = i->code;
139     link_code = VALUE_NODE(VARIABLE_NODE(code));
140     linked_vars.push_back(new string(code));
141     is_linkable = true;
142 }
143
144
145 ValueNode::ValueNode(LiteralNode *l) {
146     val_type = LIT;
147     val.lit_val = l;
148     type = l->type;
149     sym = l->sym;
150     code = l->code;
151     link_code = VALUE_NODE(LITERAL_NODE(code));
152     is_linkable = true;
153     array_length = l->array_length;
154 }
155
156
157 ValueNode::ValueNode(FunctionCallNode *f) {
158     val_type = FUNC_CALL;
159     val.function_call_val = f;
160     type = f->type;
161     sym = f->sym;
162     code = f->code;
163     is_linkable = false;
164 }
165
166
167 ValueNode::ValueNode(ArrayAccessNode *a) {
168     val_type = ARR_ACC;
169     val.array_access_val = a;
170     type = a->type;
171     sym = a->sym;
172     code = a->code;
173     is_linkable = false;
174 }
175
176
177 ValueNode::ValueNode(DatasetAccessNode *d) {
178     val_type = DS_ACC;
179     val.dataset_access_val = d;
180     type = d->type;
181     sym = d->sym;
182     code = d->code;
183     is_linkable = false;
184 }
```

```
185
186
187 ValueNode::ValueNode(ExpressionNode *e) {
188     val_type = EXPR;
189     val.expression_val = e;
190     type = e->type;
191     sym = e->sym;
192     code = "( " + e->code + " )";
193     linked_vars.insert(linked_vars.end(), e->linked_vars.begin(), e->
        ↪ linked_vars.end());
194     link_code = VALUE_NODE(e->link_code);
195     is_linkable = true;
196 }
197
198
199 ValueNode::ValueNode(ArrayInitNode *a) {
200     val.a_init = a;
201     type = a->type;
202     sym = tARR;
203     code = "{ " + a->code + " }";
204     array_length = a->has_elements ? a->array_length : -1;
205     is_linkable = false;
206 }
207
208
209 void ValueNode::seppuku(){
210     val.a_init->seppuku();
211     delete this;
212 }
213
214
215 /* DatasetNode */
216 DatasetNode::DatasetNode(string s, DeclArgsNode *d) {
217     name = s;
218     decl_args = d;
219     replace( decl_args->code.begin(), decl_args->code.end(), ',', ';');
220     code = "struct " + s + "{\n" + decl_args->code + ";\n};";
221 }
222
223
224 void DatasetNode::seppuku(){
225     decl_args->seppuku();
226     delete this;
227 }
228
229
230 /* IDNode */
231 IDNode::IDNode(Entry *ent) {
232     entry = ent;
233     if (entry) {
234         type = ent->type;
235         code = ent->name;
236         sym = ent->symbol_type;
237     }
```

```
238 }
239
240
241 e_type IDNode::get_type() {
242     return type;
243 }
244
245
246 string IDNode::get_name() {
247     return entry->name;
248 }
249
250
251 void IDNode::seppuku(){
252     delete this;
253 }
254
255
256 /* FunctionCallNode */
257 FunctionCallNode::FunctionCallNode(string f, ArgsNode *a) {
258     func_id = f;
259     args_list = a;
260     sym = tFUNC;
261
262     if(IS_STD_RPL_FUNCTION(func_id)){
263         code = generate_std_rpl_function();
264         typecheck();
265     } else {
266         typecheck();
267         code = f + "( " + a->code + " )";
268     }
269 }
270
271
272 FunctionCallNode::FunctionCallNode(string f) {
273     func_id = f;
274     args_list = new ArgsNode();
275     sym = tFUNC;
276
277     if(IS_STD_RPL_FUNCTION(func_id)){
278         code = generate_std_rpl_function();
279         typecheck();
280     } else {
281         typecheck();
282         code = f + "()";
283     }
284 }
285
286
287 void FunctionCallNode::seppuku(){
288     args_list->seppuku();
289     delete this;
290 }
291
```

```
292
293 void FunctionCallNode::typecheck() {
294     Entry *entry = sym_table.get(func_id);
295     if (entry) {
296         /* This is an attempt to call a variable that isn't a function */
297         if (entry->symbol_type != tFUNC) {
298             error = true;
299             cout << NOT_A_FUNC_ERR << endl;
300         } else if (entry->args) {
301             /* The args are incorrect in this example */
302             if (*entry->args != *args_list->to_enum_list()) {
303                 error = true;
304                 cout << INVAL_FUNC_CALL_ERR << endl;
305             }
306         }
307         type = entry->type;
308     } else {
309         error = true;
310         cout << "use of undeclared function " << func_id << endl;
311     }
312 }
313
314
315 /* Standard functions are specially generated to allow for
316  * things that normal functions can't have, like variable args*/
317 string FunctionCallNode::generate_std_rpl_function(){
318     string func_name = func_id;
319     string code;
320     if (func_name.compare(RPL_STD_OUTPUT_FUNCTION) == 0){
321         code = "std::cout << ";
322         for(std::vector<ExpressionNode *>::iterator it = args_list->args_list
            ↪ ->begin();
323             it != args_list->args_list->end(); ++it) {
324             code += (*it)->code + " << \" \" << ";
325         }
326         code += "std::endl";
327     } else if (func_name.compare(RPL_STD_INPUT_FUNCTION) == 0){
328         if(args_list->args_list->size() != 1 || args_list->args_list->at(0)->
            ↪ type != tSTRING){
329             error = true;
330             cout << RPL_STD_INPUT_FUNCTION_ERR << endl;
331         } else {
332             type = tSTRING;
333             code = "ripple::input(" + args_list->args_list->at(0)->code + ")";
334         }
335     } else if (func_name.compare(RPL_STD_OPEN_FUNCTION) == 0){
336
337     } else if (func_name.compare(RPL_STD_CLOSE_FUNCTION) == 0){
338
339     } else if (func_name.compare(RPL_STD_READ_FUNCTION) == 0){
340
341     } else {
342         code = func_id + "(" + args_list->code + ")";
343     }
```

```
344     return code;
345 }
346
347
348 /* ArrayInitNode */
349 ArrayInitNode::ArrayInitNode() {
350     args_list = new vector<ExpressionNode *>();
351     sym = tARR;
352     has_elements = false;
353     code = "";
354     array_length = 0;
355 }
356
357
358 ArrayInitNode::ArrayInitNode(ExpressionNode *arg) {
359     type = arg->type;
360     sym = tARR;
361     has_elements = true;
362     args_list->push_back(arg);
363     code = arg->code;
364     array_length = 1;
365 }
366
367
368 void ArrayInitNode::add_arg(ExpressionNode *arg) {
369     args_list->push_back(arg);
370     sym = tARR;
371     has_elements = true;
372     if(type == 0 || (type == tINT && arg->type == tFLOAT)){
373         type = arg->type;
374     } else if(type == tFLOAT && arg->type == tINT) {
375     } else if(arg->type != type){
376         error = true;
377         cout << ARR_ELEMENT_TYPE_ERR << endl;
378     }
379     if(code.compare("") == 0)
380         code += arg->code;
381     else
382         code += ", " + arg->code;
383
384     array_length++;
385 }
386
387
388 void ArrayInitNode::seppuku(){
389     for(std::vector<ExpressionNode *>::iterator it = args_list->begin();
390             it != args_list->end(); ++it)
391         (*it)->seppuku();
392
393     delete this;
394 }
395
396
397 /* ArgsNode */
```

```
398 ArgsNode::ArgsNode() {
399     args_list = new vector<ExpressionNode *>();
400     code = "";
401 }
402
403
404 ArgsNode::ArgsNode(ExpressionNode *arg) {
405     if (arg->type == tNOTYPE) {
406         error = true;
407         cout << UNDECLARED_ERROR << endl;
408     }
409     args_list->push_back(arg);
410     code = arg->code;
411 }
412
413
414 void ArgsNode::add_arg(ExpressionNode *arg) {
415     args_list->push_back(arg);
416
417     if (arg->type == tNOTYPE) {
418         error = true;
419         cout << UNDECLARED_ERROR << endl;
420     }
421
422     if(code.compare("") == 0)
423         code += arg->code;
424     else
425         code += ", " + arg->code;
426 }
427
428
429 list<e_type> *ArgsNode::to_enum_list() {
430     list<e_type> *ret = new list<e_type>();
431     for (vector<ExpressionNode *>::iterator i = args_list->begin(); i !=
        ↪ args_list->end(); i++) {
432         ret->push_back((*i)->type);
433     }
434     return ret;
435 }
436
437
438 void ArgsNode::seppuku(){
439     for(std::vector<ExpressionNode *>::iterator it = args_list->begin();
440             it != args_list->end(); ++it)
441         (*it)->seppuku();
442     delete this;
443 }
444
445
446 /* DeclArgsNode */
447 DeclArgsNode::DeclArgsNode() {
448     code = "";
449 }
450
```

```
451
452 DeclArgsNode::DeclArgsNode(TypeNode *t, IDNode* arg) {
453     decl_args_list.push_back(arg);
454     arg->type = t->type;
455     arg->entry->type = t->type;
456     type = arg->type;
457     code = type_to_str(type) + " " + arg->code;
458 }
459
460
461 void DeclArgsNode::add_arg(TypeNode *t, IDNode* arg) {
462     decl_args_list.push_back(arg);
463     arg->type = t->type;
464     arg->entry->type = t->type;
465     type = arg->type;
466     code += ", " + type_to_str(type) + " " + arg->code;
467 }
468
469
470 vector<IDNode *>::iterator DeclArgsNode::begin() {
471     return decl_args_list.begin();
472 }
473
474
475 vector<IDNode *>::iterator DeclArgsNode::end() {
476     return decl_args_list.end();
477 }
478
479
480 list<e_type> *DeclArgsNode::to_enum_list() {
481     list<e_type> *ret = new list<e_type>();
482     for (vector<IDNode *>::iterator i = decl_args_list.begin(); i !=
        ↪ decl_args_list.end(); i++) {
483         ret->push_back((*i)->get_type());
484     }
485     return ret;
486 }
487
488
489 void DeclArgsNode::seppuku(){
490     for(std::vector<IDNode *>::iterator it = decl_args_list.begin();
491             it != decl_args_list.end(); ++it)
492         (*it)->seppuku();
493     delete this;
494 }
495
496
497 /* LiteralNode */
498 LiteralNode::LiteralNode(int i) {
499     sym = tVAR;
500     val.int_lit = i;
501     array_length = i;
502     type = tINT;
503 }
```

```
504
505
506 LiteralNode::LiteralNode(double d) {
507     sym = tVAR;
508     val.float_lit = d;
509     type = tFLOAT;
510 }
511
512
513 LiteralNode::LiteralNode(string *s) {
514     sym = tVAR;
515     val.string_lit = s;
516     type = tSTRING;
517 }
518
519
520 LiteralNode::LiteralNode(bool b) {
521     sym = tVAR;
522     val.bool_lit = b;
523     type = tBOOL;
524 }
525
526
527 void LiteralNode::seppuku(){
528     if(type == tSTRING)
529         delete val.string_lit;
530     delete this;
531 }
532
533
534 /* ArrayAccessNode */
535 ArrayAccessNode::ArrayAccessNode(ValueNode *val, ExpressionNode *exp) {
536     value_node = val;
537     en = exp;
538     type = val->type;
539     sym = val->sym;
540     code = val->code + "[" + exp->code + "]";
541 }
542
543
544 void ArrayAccessNode::seppuku(){
545     value_node->seppuku();
546     en->seppuku();
547     delete this;
548 }
549
550
551 /* DatasetAccessNode */
552 DatasetAccessNode::DatasetAccessNode(string c, string i) {
553     Entry *entry = sym_table.get(c);
554     if (!entry) {
555         error = true;
556         cout << LINE_ERR "use of undeclared variable" << c << endl;
557     }
```

```
558
559     Entry *member_entry = sym_table.get_dataset_member(entry->ds_name, i);
560     if (!member_entry) {
561         error = true;
562         cout << LINE_ERR "dataset " << c << " of type " << entry->ds_name <<
563                                 " does not contain a member named " << i << endl;
564         type = tDERIV;
565         sym = tVAR;
566     } else {
567         entry = member_entry;
568         type = member_entry->type;
569         sym = member_entry->symbol_type;
570         array_length = member_entry->array_length;
571     }
572     code += c + "." + i ;
573 }
574
575
576 void DatasetAccessNode::seppuku(){
577     value_node->seppuku();
578     delete this;
579 }
580
581
582 /* UnaryExpressionNode */
583 UnaryExpressionNode::UnaryExpressionNode(UnaryExpressionNode *u, string _op) {
584     right_operand.u_exp = u;
585     array_length = u->array_length;
586     op = str_to_op(_op);
587     sym = u->sym;
588     typecheck(op);
589     switch(op) {
590         case MINUS:
591             code = "-" + u->code;
592             break;
593         case SIZE:
594             if(sym == tARR){
595                 code = "sizeof( " + u->code + " ) / sizeof( " +
596                     type_to_str(u->type) + " )";
597             } else {
598                 code = "sizeof( " + u->code + " )";
599             }
600             break;
601         case bNOT:
602             code = "!" + u->code;
603             break;
604         default:
605             break;
606     }
607     link_code = UNARY_EXPRESSION(u->link_code, _op);
608     linked_vars.insert(linked_vars.end(), u->linked_vars.begin(), u->
        ↪ linked_vars.end());
609     is_linkable = u->is_linkable;
610 }
```

```
611
612
613 UnaryExpressionNode::UnaryExpressionNode(UnaryExpressionNode *u, TypeNode *t){
614     right_operand.u_exp = u;
615     sym = u->sym;
616     op = CAST;
617     type = t->type;
618     typecheck(CAST);
619     is_linkable = false;
620
621     if(type_to_str(type).compare("string") == 0){
622         code = "(" + type_to_str(type) + ")" + u->code;
623     } else {
624         code = "to_string(" + u->code + ")";
625     }
626 }
627
628
629 UnaryExpressionNode::UnaryExpressionNode(ValueNode *v){
630     op = NONE;
631     right_operand.v_node = v;
632     array_length = v->array_length;
633     type = v->type;
634     sym = v->sym;
635     code = v->code;
636     link_code = UNARY_EXPRESSION(v->link_code);
637     linked_vars.insert(linked_vars.end(), v->linked_vars.begin(), v->
        ↪ linked_vars.end());;
638     is_linkable = v->is_linkable;
639 }
640
641
642 void UnaryExpressionNode::typecheck(e_op op){
643     e_type child_type = right_operand.u_exp->type;
644
645     if(sym == tARR && op != SIZE){
646         error = true;
647         if(op == bNOT)
648             cout << ARR_UNARY_NOT_ERR << endl;
649         else if(op == MINUS)
650             cout << ARR_UNARY_MINUS_ERR << endl;
651         else if(op == CAST)
652             cout << ARR_UNARY_CAST_ERR << endl;
653         return;
654     }
655
656     switch(op){
657         case bNOT:
658             if(child_type != tBOOL){
659                 cout << INVAL_UNARY_NOT_ERR << endl;
660                 error = true;
661             }
662             else
663                 type = tBOOL;
```

```
664             break;
665         case MINUS:
666             if(child_type != tINT && child_type != tFLOAT){
667                 cout << INVAL_UNARY_MINUS_ERR << endl;
668                 error = true;
669             }
670             else
671                 type = child_type;
672             break;
673         case CAST:
674             if(child_type == tSTRING && (type == tBOOL || type == tINT || type
                    == tFLOAT)){
675                 error = true;
676                 cout << UNARY_STRING_CAST_ERR << endl;
677             }
678         default:
679             type = child_type;
680     }
681 }
682
683
684 void UnaryExpressionNode::seppuku(){
685     if(op == NONE)
686         right_operand.v_node->seppuku();
687     else
688         right_operand.u_exp->seppuku();
689     delete this;
690 }
691
692
693 /* BinaryExpressionNode */
694 BinaryExpressionNode::BinaryExpressionNode(BinaryExpressionNode *bl, string
        _op, BinaryExpressionNode *br) {
695     left_operand.b_exp = bl;
696     right_operand.b_exp = br;
697     left_is_binary = right_is_binary = true;
698
699     op = str_to_op(_op);
700
701     typecheck(bl, br, op);
702
703     code = gen_binary_code(bl->code, op, br->code, bl->type, br->type);
704     link_code = BINARY_EXPRESSION(bl->link_code, _op, br->link_code);
705     linked_vars.insert(linked_vars.end(), bl->linked_vars.begin(), bl->
            linked_vars.end());
706     linked_vars.insert(linked_vars.end(), br->linked_vars.begin(), br->
            linked_vars.end());
707     is_linkable = bl->is_linkable && br->is_linkable;
708 }
709
710
711 BinaryExpressionNode::BinaryExpressionNode(BinaryExpressionNode *bl, string
        _op, UnaryExpressionNode *ur) {
712     left_operand.b_exp = bl;
```

```
713     right_operand.u_exp = ur;
714     left_is_binary = true;
715     right_is_binary = false;
716 
717     op = str_to_op(_op);
718     typecheck(bl, ur, op);
719 
720     code = gen_binary_code(bl->code, op, ur->code, bl->type, ur->type);
721     link_code = BINARY_EXPRESSION(bl->link_code, _op, ur->link_code);
722     linked_vars.insert(linked_vars.end(), bl->linked_vars.begin(), bl->
        ↪ linked_vars.end());
723     linked_vars.insert(linked_vars.end(), ur->linked_vars.begin(), ur->
        ↪ linked_vars.end());
724     is_linkable = bl->is_linkable && ur->is_linkable;
725 }
726 
727 
728 BinaryExpressionNode::BinaryExpressionNode(UnaryExpressionNode *ul) {
729     left_operand.u_exp = ul;
730     left_is_binary = false;
731     right_operand.b_exp = nullptr;
732     type = ul->type;
733     sym = ul->sym;
734     code = ul->code;
735     array_length = ul->array_length;
736     op = NONE;
737 
738     link_code = BINARY_EXPRESSION(ul->link_code);
739     is_linkable = ul->is_linkable;
740     linked_vars.insert(linked_vars.end(), ul->linked_vars.begin(), ul->
        ↪ linked_vars.end());
741 }
742 
743 
744 void BinaryExpressionNode::typecheck(Node *left, Node *right, e_op op){
745     if(left->sym == tARR || right->sym == tARR){
746         error = true;
747         cout << ARR_BINEXP_ERR << endl;
748     }
749 
750     if((left->is_string() || right->is_string()) && op == PLUS) {
751         type = tSTRING;
752         return;
753     }
754 
755     if (op == PLUS || op == MINUS || op == TIMES || op == DIV || op == EXP) {
756         if (left->is_number() && right->is_number())
757             if (left->type == tFLOAT || right->type == tFLOAT)
758                 type = tFLOAT;
759             else
760                 type = tINT;
761         else
762             switch(op){
763                 case PLUS:
```

```
764                     cout << INVAL_BINARY_PLUS_ERR << endl;
765                     error = true;
766                     break;
767                 case MINUS:
768                     cout << INVAL_BINARY_MINUS_ERR << endl;
769                     error = true;
770                     break;
771                 case TIMES:
772                     cout << INVAL_BINARY_TIMES_ERR << endl;
773                     error = true;
774                     break;
775                 case DIV:
776                     cout << INVAL_BINARY_DIV_ERR << endl;
777                     error = true;
778                     break;
779                 case EXP:
780                     cout << INVAL_BINARY_EXP_ERR << endl;
781                     error = true;
782                     break;
783                 default:
784                     cout << ERROR << endl;
785                     error = true;
786             }
787
788     } else if (op == FLDIV) {
789
790         if (left->is_number() && right->is_number())
791             type = tFLOAT;
792         else{
793             cout << INVAL_BINARY_FLDIV_ERR << endl;
794             error = true;
795         }
796
797     } else if (op == MOD) {
798
799         if (left->is_number() && right->is_number()
800                 && left->type == tINT && right->type == tINT){
801             type = tINT;
802         } else {
803             error = true;
804             cout << INVAL_BINARY_MOD_ERR << endl;
805         }
806
807     } else if (op == EQ || op == NE || op == GT ||
808             op == LT || op == GE || op == LE) {
809
810         if (left->is_number() && right->is_number())
811             type = tBOOL;
812         else
813             switch(op){
814                 case EQ:
815                     cout << INVAL_BINARY_EQ_ERR << endl;
816                     error = true;
817                     break;
```

```
818                 case NE:
819                     cout << INVAL_BINARY_NE_ERR << endl;
820                     error = true;
821                     break;
822                 case GT:
823                     cout << INVAL_BINARY_GT_ERR << endl;
824                     error = true;
825                     break;
826                 case LT:
827                     cout << INVAL_BINARY_LT_ERR << endl;
828                     error = true;
829                     break;
830                 case GE:
831                     cout << INVAL_BINARY_GE_ERR << endl;
832                     error = true;
833                     break;
834                 case LE:
835                     cout << INVAL_BINARY_LE_ERR << endl;
836                     error = true;
837                     break;
838                 default:
839                     cout << ERROR << endl;
840                     error = true;
841             }
842 
843     } else if (op == bAND || bOR) {
844         if (left->is_bool() && right->is_bool())
845             type = tBOOL;
846         else
847             switch(op){
848                 case bAND:
849                     cout << INVAL_BINARY_AND_ERR << endl;
850                     error = true;
851                     break;
852                 case bOR:
853                     cout << INVAL_BINARY_OR_ERR << endl;
854                     error = true;
855                     break;
856                 default:
857                     cout << ERROR << endl;
858                     error = true;
859             }
860     }
861 }
862 
863 
864 string BinaryExpressionNode::gen_binary_code(string l_code, enum e_op op,
    ↪ string r_code, e_type l_type, e_type r_type){
865     string code;
866 
867     if(r_type == tSTRING && l_type != tSTRING)
868         l_code = "std::to_string(" + l_code + ")";
869     else if(l_type == tSTRING && r_type != tSTRING)
870         r_code = "std::to_string(" + r_code + ")";
```

```
871
872     switch(op) {
873         case PLUS:
874             code = l_code + " + " + r_code;
875             break;
876         case MINUS:
877             code = l_code + " - " + r_code;
878             break;
879         case TIMES:
880             code = l_code + " * " + r_code;
881             break;
882         case DIV:
883             code = l_code + " / " + r_code;
884             break;
885         case FLDIV:
886             code = "(double)" + l_code + "/ (double)" + r_code;
887             break;
888         case MOD:
889             code = l_code + " % " + r_code;
890             break;
891         case EXP:
892             code = "pow(" + l_code + ", " + r_code + ")";
893             break;
894         case bAND:
895             code = l_code + " && " + r_code;
896             break;
897         case bOR:
898             code = l_code + " || " + r_code;
899             break;
900         case EQ:
901             code = l_code + " == " + r_code;
902             break;
903         case NE:
904             code = l_code + " != " + r_code;
905             break;
906         case GT:
907             code = l_code + " > " + r_code;
908             break;
909         case LT:
910             code = l_code + " < " + r_code;
911             break;
912         case GE:
913             code = l_code + " >= " + r_code;
914             break;
915         case LE:
916             code = l_code + " <= " + r_code;
917             break;
918         default:
919             code = "";
920             break;
921     }
922
923     return code;
924 }
```

```
925
926
927 ValueNode *BinaryExpressionNode::get_value_node() {
928     if (op == NONE && left_operand.u_exp && left_operand.u_exp->op == NONE)
929         return left_operand.u_exp->right_operand.v_node;
930     return nullptr;
931 }
932
933
934 void BinaryExpressionNode::seppuku(){
935     if(left_is_binary){
936         left_operand.b_exp->seppuku();
937     }
938     else{
939         left_operand.u_exp->seppuku();
940     }
941
942     if(op != NONE){
943         if(right_is_binary){
944             right_operand.b_exp->seppuku();
945         }
946         else {
947             right_operand.u_exp->seppuku();
948         }
949     }
950
951     delete this;
952 }
953
954
955 /* ExpressionNode */
956 ExpressionNode::ExpressionNode(BinaryExpressionNode *b) {
957     ValueNode *v = b->get_value_node();
958     if (v) {
959         value = v;
960         bin_exp = nullptr;
961     } else {
962         value = nullptr;
963         bin_exp = b;
964     }
965
966     type = b->type;
967     code = b->code;
968     array_length = b->array_length;
969     link_code = EXPRESSION_NODE(b->link_code);
970     linked_vars = b->linked_vars;
971     is_linkable = b->is_linkable;
972 }
973
974
975 ExpressionNode::ExpressionNode(BinaryExpressionNode *b, ValueNode *v) {
976     bin_exp = b;
977     value = v;
978     typecheck(b, v);
```

```
979     sym = b->sym;
980     array_length = b->array_length;
981     code = v->code + " = " + b->code;
982 }
983
984
985 ExpressionNode::~ExpressionNode() {}
986
987
988 void ExpressionNode::typecheck(BinaryExpressionNode *expression, ValueNode *
        ↪ value){
989     if(value->sym != tVAR){
990         error = true;
991         cout << ASSIGN_ERR << endl;
992     }
993     Entry *ent = sym_table.get(value->code);
994     if(ent && ent->is_final){
995         error = true;
996         cout << FINAL_REDECL_ERR << endl;
997     }
998
999     type = expression->type;
1000 }
1001
1002
1003 void ExpressionNode::seppuku(){
1004     if(bin_exp != nullptr)
1005         bin_exp->seppuku();
1006     delete this;
1007 }
1008
1009
1010 /* DeclarativeStatementNode */
1011 DeclarativeStatementNode::DeclarativeStatementNode(TypeNode *t, ExpressionNode
        ↪  *expression_node){
1012     type = t->type;
1013     sym = t->sym;
1014     ds_name = t->ds_name;
1015     array_length = t->array_length;
1016     en = expression_node;
1017
1018     if (!expression_node->value || expression_node->value->val_type != IDENT)
        ↪ {
1019         error = true;
1020         cout << INVALID_DECL_ERR << endl;
1021     }
1022     typecheck();
1023     Entry *entry = sym_table.get(expression_node->value->code);
1024     switch(sym) {
1025         case tVAR:
1026             if (entry->type != tNOTYPE) {
1027                 error = true;
1028                 cout << VARIABLE_REDECL_ERR << endl;
1029             } else {
```

```
1030                    entry->type = type;
1031                }
1032                code += type_to_str(type) + " " + expression_node->code + ";\n";
1033                break;
1034            case tARR:
1035                if (entry->type != tNOTYPE) {
1036                    error = true;
1037                    cout << VARIABLE_REDECL_ERR << endl;
1038                } else {
1039                    entry->type = type;
1040                }
1041                if(expression_node->sym != tARR && expression_node->sym !=
                    ↪ tNOSTYPE) {
1042                    error = true;
1043                    cout << ARR_ASSIGN_ERR << endl;
1044                }
1045
1046                if( array_length == -1 && !expression_node->bin_exp) {
1047                    error = true;
1048                    cout << ARR_UNKNOWN_INIT_ERR << endl;
1049                }
1050
1051                if(array_length != -1 && array_length < expression_node->
                    ↪ array_length) {
1052                    error = true;
1053                    cout << ARR_SMALL_SIZE_ERR << endl;
1054                }
1055
1056                if(!sym_table.add_array(expression_node->value->code, type,
                    ↪ line_no, array_length)) {
1057                    error = true;
1058                    cout <<  VARIABLE_REDECL_ERR << endl;
1059                }
1060
1061                code += type_to_str(type) + " " + expression_node->value->code + "
                    ↪ [" + t->value->code + "]";
1062                if (expression_node->bin_exp) {
1063                    code += "=" + expression_node->bin_exp->code;
1064                }
1065                code += ";\n";
1066                break;
1067            case tDSET:
1068                if (sym_table.instantiate_dataset(expression_node->value->code,
                    ↪ ds_name, line_no)) {
1069                    error = true;
1070                    cout << VARIABLE_REDECL_ERR << endl;
1071                }
1072                code += "struct " + ds_name + " " + expression_node->value->code +
                    ↪  ";\n";
1073                break;
1074        }
1075    }
1076
1077
```

```
1078 DeclarativeStatementNode::DeclarativeStatementNode(ExpressionNode *
     ↪ expression_node){
1079     type = expression_node->type;
1080     en = expression_node;
1081     a_size = nullptr;
1082     code = expression_node->code + ";\n";
1083 
1084     ValueNode *val_node = expression_node->value;
1085     if (val_node) {
1086         if (val_node->val_type == IDENT) {
1087             Entry *entry = sym_table.get(val_node->code);
1088             if (entry) {
1089                 if (entry->type == tNOTYPE) {
1090                     error = true;
1091                     cout << UNDECLARED_ERROR << endl;
1092                 }
1093                 if (entry->has_dependents) {
1094                     code += "linked_var::update_nonlinked_var(&" + entry->name
                         ↪  + ");\n";
1095                 }
1096             }
1097         }
1098     }
1099 }
1100 
1101 
1102 void DeclarativeStatementNode::typecheck() {
1103     if (!en->bin_exp || type == tFLOAT && en->type == tINT){
1104     } else if (en->type != type){
1105         if(en->sym == tARR || sym == tARR){
1106             INVAL_ASSIGN_ERR(type_to_str(type) + "[]", type_to_str(en->type) +
                     ↪  "[]");
1107         }
1108         else{
1109             INVAL_ASSIGN_ERR(type_to_str(type), type_to_str(en->type));
1110         }
1111         error = true;
1112     }
1113 }
1114 
1115 
1116 void DeclarativeStatementNode::seppuku(){
1117     if(a_size != nullptr)
1118         a_size->seppuku();
1119     en->seppuku();
1120     delete this;
1121 }
1122 
1123 
1124 /* TypeNode */
1125 TypeNode::TypeNode(e_type t, string name){
1126     type = t;
1127     sym = tDSET;
1128     ds_name = name;
```

```
1129
1130        value = nullptr;
1131
1132    }
1133
1134
1135    TypeNode::TypeNode(e_type t, ValueNode *val) {
1136        type = t;
1137        value = val;
1138
1139        if (type == tNOTYPE) {
1140            error = true;
1141            cout << UNKNOWN_TYPE_ERR << endl;
1142        }
1143
1144        if(val){
1145            sym = tARR;
1146            array_length = val->array_length;
1147            if(val->type != tINT){
1148                cout << ARR_INT_SIZE_ERR << endl;
1149                error = true;
1150            }
1151        } else {
1152            sym = tVAR;
1153        }
1154    }
1155
1156
1157    /* ConditionalStatementNode */
1158    ConditionalStatementNode::ConditionalStatementNode(ExpressionNode *e,
        ↪ StatementListNode *s, StatementListNode *a) {
1159        condition = e;
1160        consequent = s;
1161        alternative = a;
1162
1163        if(e->type != tBOOL){
1164            error = true;
1165            cout << COND_STMT_ERR << endl;
1166        }
1167
1168        code = "if (" + e->code + ")" + s->code;
1169
1170        if(a != nullptr)
1171            code += "else " + a->code;
1172    }
1173
1174
1175    void ConditionalStatementNode::seppuku(){
1176        condition->seppuku();
1177        consequent->seppuku();
1178        if(alternative != nullptr)
1179            alternative->seppuku();
1180        delete this;
1181    }
```

```
1182
1183
1184 /* JumpStatementNode */
1185 JumpStatementNode::JumpStatementNode(string _type, ExpressionNode *
     ↪ expression_node){
1186     type = str_to_jump(_type);
1187     en = expression_node;
1188     if (expression_node->type != func_type) {
1189         //error = true;
1190         //mcout << RETURN_TYPE_ERROR << endl;
1191     }
1192     code = _type + "␣" + expression_node->code + ";\n";
1193 }
1194
1195
1196 JumpStatementNode::JumpStatementNode(string _type){
1197     type = str_to_jump(_type);
1198     if (type == tRETURN && func_type != tVOID) {
1199         cout << RETURN_TYPE_ERROR << endl;
1200     }
1201     en = nullptr;
1202     if (type == tSTOP) {
1203         code = "while(1);\n";
1204     } else {
1205         code = _type + ";";
1206     }
1207 }
1208
1209
1210 void JumpStatementNode::seppuku(){
1211     if(en != nullptr)
1212         en->seppuku();
1213     delete this;
1214 }
1215
1216
1217 /* LoopStatementNode */
1218 LoopStatementNode::LoopStatementNode(ExpressionNode *init, ExpressionNode *
     ↪ cond,
1219                                      ExpressionNode *n, StatementListNode *
                                            ↪ stmts){
1220     string init_code, cond_code, n_code;
1221
1222     initializer = init;
1223     condition = cond;
1224     next = n;
1225     statements = stmts;
1226
1227     if(cond->type != tBOOL){
1228         cout << LOOP_CONDITION_ERR << endl;
1229         error = true;
1230     }
1231
1232     if(init != nullptr){
```

```
1233            init_code = init->code;
1234        } else {
1235            init_code = "";
1236        }
1237
1238        if(cond != nullptr) {
1239            cond_code = cond->code;
1240        } else {
1241            cond_code = "";
1242        }
1243
1244        if(n != nullptr) {
1245            n_code = n->code;
1246        } else {
1247            n_code = "";
1248        }
1249
1250        code = "for (" + init_code + "; " + cond_code + "; " + n_code + ")" +
                ↪ stmts->code;
1251    }
1252
1253
1254    void LoopStatementNode::seppuku(){
1255        if(initializer != nullptr)
1256            initializer->seppuku();
1257        if(condition != nullptr)
1258            condition->seppuku();
1259        if(next != nullptr)
1260            next->seppuku();
1261
1262        delete this;
1263    }
1264
1265
1266    /* LinkStatementNode */
1267    LinkStatementNode::LinkStatementNode(IDNode *idn, ExpressionNode *expn){
1268        id_node = idn;
1269        expression_node = expn;
1270        if(!expression_node->is_linkable){
1271            error = true;
1272            cout << UNLINKABLE_EXPRESSION_ERR << endl;
1273        }
1274
1275        if(expression_node->linked_vars.size() == 0){
1276            error = true;
1277            cout << UNLINKABLE_NO_VAR_ERR << endl;
1278        }
1279
1280        code = "linked_var::register_cpp_var(&" + idn->code + ");\n";
1281        for(vector<string *>::iterator it = expression_node->linked_vars.begin();
1282                it != expression_node->linked_vars.end(); it++) {
1283            code += "linked_var::register_cpp_var(&" + **it + ");\n";
1284            Entry *linked_entry = sym_table.get(**it);
1285            if (linked_entry) {
```

```
1286            linked_entry->has_dependents = true;
1287        } else {
1288            error = true;
1289            cout << UNDECLARED_ERROR << endl;
1290        }
1291    }
1292    code += "universal_linked_var_ptr = new_linked_var(&" + idn->code + ", "
        ↪ + expression_node->link_code + ");\n";
1293 }
1294
1295
1296 LinkStatementNode::LinkStatementNode(IDNode *idn, ExpressionNode *expn, string
    ↪  func){
1297    id_node = idn;
1298    expression_node = expn;
1299    auxiliary = func;
1300
1301    if(!expression_node->is_linkable){
1302        error = true;
1303        cout << UNLINKABLE_EXPRESSION_ERR << endl;
1304    }
1305
1306    if(expression_node->linked_vars.size() == 0){
1307        error = true;
1308        cout << UNLINKABLE_NO_VAR_ERR << endl;
1309    }
1310
1311    code = "linked_var::register_cpp_var(&" + idn->code + ");\n";
1312    for(vector<string *>::iterator it = expression_node->linked_vars.begin();
1313                it != expression_node->linked_vars.end(); it++) {
1314        code += "linked_var::register_cpp_var(&" + **it + ");\n";
1315        Entry *linked_entry = sym_table.get(**it);
1316        if (linked_entry) {
1317            linked_entry->has_dependents = true;
1318        } else {
1319            error = true;
1320            cout << UNDECLARED_ERROR << endl;
1321        }
1322    }
1323
1324    code += "universal_linked_var_ptr = new_linked_var(&" + idn->code + ", "
        ↪ + expression_node->link_code + ");\n";
1325    Entry *entry = sym_table.get(auxiliary);
1326    if (!entry) {
1327        cout << UNDECLARED_ERROR << endl;
1328    } else {
1329        if (entry->symbol_type != tFUNC || entry->type != tVOID) {
1330            error = true;
1331            cout << INVAL_FUNC_CALL_ERR << endl;
1332        } else  {
1333            if (entry->args->size() != 1 || entry->args->front() != id_node->
                ↪ type) {
1334                error = true;
1335                cout << INVAL_FUNC_ARGS_ERR << endl;
```

```
1336                } else {
1337                    code += "universal_linked_var_ptr->assign_aux_fn((void *)" +
                            ↪ auxiliary + ");";
1338                }
1339            }
1340        }
1341    }
1342
1343    LinkStatementNode::LinkStatementNode(IDNode *idn, StreamReaderNode *srn){
1344
1345        id_node = idn;
1346        stream_reader_node = srn;
1347
1348
1349        Entry *ent = sym_table.get(idn->code);
1350        if(ent && ent->type == tNOTYPE){
1351            error = true;
1352            cout << UNDECLARED_ERROR << endl;
1353        }
1354
1355            if (idn->type == tSTRING)
1356                    code = type_to_str(idn->type) + " *" + stream + to_string(num)
                            ↪  + " = new string();\n";
1357            else
1358                    code = type_to_str(idn->type) + " " + stream + to_string(num)
                            ↪ + ";\n";
1359            code += "linked_var::register_cpp_var(&" + stream + to_string(num) + "
                    ↪ );\n";
1360        code += "FuncPtr<" + type_to_str(idn->type) + ">::f_ptr " +
                ↪ function_pointer + to_string(num) + " = &default_rpl_str_str;\n";
1361        code += srn->generate_code(idn->type);
1362            if (idn->type == tSTRING) {
1363                    code += "universal_linked_var_ptr = new linked_var(&" + idn->
                            ↪ code + ", " +
1364                                    EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE("new
                                        ↪ VariableNode((string **)&" + stream +
                                        ↪ to_string(num))))) + "));\n";
1365            } else {
1366                    code += "universal_linked_var_ptr = new linked_var(&" + idn->
                            ↪ code + ", " +
1367                                    EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE(
                                        ↪ VARIABLE_NODE(stream + to_string(num))))
                                        ↪ )) + ");\n";
1368            }
1369        code += stream_reader_name + to_string(num) + "->start_thread();\n";
1370
1371        num++;
1372    }
1373
1374    LinkStatementNode::LinkStatementNode(IDNode *idn, StreamReaderNode *srn,
            ↪ string func){
1375
```

```
1376      id_node = idn;
1377      stream_reader_node = srn;
1378      auxiliary = func;
1379
1380      Entry *ent = sym_table.get(idn->code);
1381      if(ent && ent->type == tNOTYPE){
1382          error = true;
1383          cout << UNDECLARED_ERROR << endl;
1384      }
1385
1386          if (idn->type == tSTRING)
1387                  code = type_to_str(idn->type) + " *" + stream + to_string(num)
                        ↪  + " = new string();\n";
1388          else
1389                  code = type_to_str(idn->type) + " " + stream + to_string(num)
                        ↪ + ";\n";
1390          code += "linked_var::register_cpp_var(&" + stream + to_string(num) + "
                ↪ );\n";
1391      code += "FuncPtr<" + type_to_str(idn->type) + ">::f_ptr " +
            ↪ function_pointer + to_string(num) + " = &default_rpl_str_str;\n";
1392      code += srn->generate_code(idn->type);
1393          if (idn->type == tSTRING) {
1394                  code += "universal_linked_var_ptr = new linked_var(&" + idn->
                        ↪ code + ", " +
1395                                  EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE("new
                                        ↪ VariableNode((string **)&" + stream +
                                        ↪ to_string(num))))) + "));\n";
1396          } else {
1397                  code += "universal_linked_var_ptr = new linked_var(&" + idn->
                        ↪ code + ", " +
1398                                  EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE(
                                        ↪ VARIABLE_NODE(stream + to_string(num))))
                                        ↪ )) + ");\n";
1399          }
1400      code += "universal_linked_var_ptr->assign_aux_fn((void *)" + auxiliary + "
            ↪ );\n";
1401      code += stream_reader_name + to_string(num) + "->start_thread();\n";
1402
1403      num++;
1404
1405  }
1406
1407  LinkStatementNode::LinkStatementNode(IDNode *idn, IDNode *filt,
        ↪ StreamReaderNode *srn){
1408
1409      id_node = idn;
1410      stream_reader_node = srn;
1411          filter = filt;
1412
1413      Entry *ent = sym_table.get(idn->code);
1414      if(ent && ent->type == tNOTYPE){
1415          error = true;
```

```
1416            cout << UNDECLARED_ERROR << endl;
1417        }
1418
1419            if (idn->type == tSTRING)
1420                    code = type_to_str(idn->type) + " *" + stream + to_string(num)
                            ↪  + " = new string();\n";
1421            else
1422                    code = type_to_str(idn->type) + " " + stream + to_string(num)
                            ↪ + ";\n";
1423            code += "linked_var::register_cpp_var(&" + stream + to_string(num) + "
                    ↪ );\n";
1424        code += "FuncPtr<" + type_to_str(idn->type) + ">::f_ptr " +
                ↪ function_pointer + to_string(num) + " = &" + filt->code + ";\n";
1425        code += srn->generate_code(idn->type);
1426            if (idn->type == tSTRING) {
1427                    code += "universal_linked_var_ptr = new linked_var(&" + idn->
                            ↪ code + ", " +
1428                                    EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE("new
                                        ↪ VariableNode((string **)&" + stream +
                                        ↪ to_string(num))))) + "));\n";
1429            } else {
1430                    code += "universal_linked_var_ptr = new linked_var(&" + idn->
                            ↪ code + ", " +
1431                                    EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE(
                                        ↪ VARIABLE_NODE(stream + to_string(num))))
                                        ↪ )) + ");\n";
1432            }
1433        code += stream_reader_name + to_string(num) + "->start_thread();\n";
1434
1435        num++;
1436
1437 }
1438
1439 LinkStatementNode::LinkStatementNode(IDNode *idn, IDNode *filt,
        ↪ StreamReaderNode *srn, string func){
1440
1441        id_node = idn;
1442        filter = filt;
1443        stream_reader_node = srn;
1444        auxiliary = func;
1445
1446        Entry *ent = sym_table.get(idn->code);
1447        if(ent && ent->type == tNOTYPE){
1448            error = true;
1449            cout << UNDECLARED_ERROR << endl;
1450        }
1451
1452            if (idn->type == tSTRING)
1453                    code = type_to_str(idn->type) + " *" + stream + to_string(num)
                            ↪  + " = new string();\n";
1454            else
1455                    code = type_to_str(idn->type) + " " + stream + to_string(num)
```

```
                    ↪ + ";\n";
1456            code += "linked_var::register_cpp_var(&" + stream + to_string(num) + "
                    ↪ );\n";
1457        code += "FuncPtr<" + type_to_str(idn->type) + ">::f_ptr " +
                ↪ function_pointer + to_string(num) + " = &" + filt->code + ";\n";
1458        code += srn->generate_code(idn->type);
1459            if (idn->type == tSTRING) {
1460                    code += "universal_linked_var_ptr = new linked_var(&" + idn->
                            ↪ code + ", " +
1461                                    EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE("new
                                        ↪ VariableNode((string **)&" + stream +
                                        ↪ to_string(num))))) + "));\n";
1462            } else {
1463                    code += "universal_linked_var_ptr = new linked_var(&" + idn->
                            ↪ code + ", " +
1464                                    EXPRESSION_NODE(BINARY_EXPRESSION(
                                        ↪ UNARY_EXPRESSION(VALUE_NODE(
                                        ↪ VARIABLE_NODE(stream + to_string(num))))
                                        ↪ )) + ");\n";
1465            }
1466        code += "universal_linked_var_ptr->assign_aux_fn((void *)" + auxiliary + "
                ↪ );\n";
1467        code += stream_reader_name + to_string(num) + "->start_thread();\n";
1468
1469        num++;
1470    }
1471
1472    StreamReaderNode::StreamReaderNode(string n, ArgsNode *args){
1473        name = n;
1474        arg_list = args;
1475
1476        if(name.compare("FileStreamReader") == 0){
1477            if(arg_list->args_list->size() != 3){
1478                error = true;
1479                cout << INVALID_FILE_SR_ERR << endl;
1480            } else {
1481                if(arg_list->args_list->at(0)->type != tSTRING &&
1482                    arg_list->args_list->at(1)->type != tINT &&
1483                    arg_list->args_list->at(2)->type != tSTRING){
1484                    error = true;
1485                    cout << INVALID_FILE_SR_TYPES_ERR << endl;
1486                }
1487            }
1488        } else if (name.compare("KeyboardStreamReader") == 0){
1489            if(arg_list->args_list->size() != 0) {
1490                error = true;
1491                cout << INVALID_KEYBOARD_SR_ERR << cout;
1492            }
1493        } else if (name.compare("WebStreamReader") == 0){
1494            if(arg_list->args_list->size() != 3){
1495                error = true;
1496                cout << INVALID_WEB_SR_ERR << endl;
1497            } else {
```

```
1498                if(arg_list->args_list->at(0)->type != tSTRING &&
1499                    arg_list->args_list->at(1)->type != tINT &&
1500                    arg_list->args_list->at(2)->type != tINT){
1501                    error = true;
1502                    cout << INVALID_WEB_SR_TYPES_ERR << endl;
1503                }
1504            }
1505        }
1506    }
1507
1508    string StreamReaderNode::generate_code(e_type type){
1509            string s;
1510            if (type != tSTRING) {
1511                    if (arg_list->args_list->size() == 0) {
1512                            s = name + "<" + type_to_str(type) + ">*" +
                                ↪ stream_reader_name + to_string(num) +
1513                                    "=new" + name + "<" + type_to_str(type) + "
                                        ↪ >(&" + stream + to_string(num) + ",
                                        ↪ function_pointer + to_string(num) + ");\
                                        ↪ n";
1514                    } else {
1515                            s = name + "<" + type_to_str(type) + ">*" +
                                ↪ stream_reader_name + to_string(num) +
1516                                    "=new" + name + "<" + type_to_str(type) + "
                                        ↪ >(&" + stream + to_string(num) + ",
                                        ↪ function_pointer + to_string(num) + ",
                                        ↪  + arg_list->code + ");\n";
1517                    }
1518            } else {
1519                    if (arg_list->args_list->size() == 0) {
1520                            s = name + "<" + type_to_str(type) + ">*" +
                                ↪ stream_reader_name + to_string(num) +
1521                                    "=new" + name + "<" + type_to_str(type) + "
                                        ↪ >(&" + stream + to_string(num) + ",
                                        ↪ function_pointer + to_string(num) + ");\
                                        ↪ n";
1522                    } else {
1523                            s = name + "<" + type_to_str(type) + ">*" +
                                ↪ stream_reader_name + to_string(num) +
1524                                    "=new" + name + "<" + type_to_str(type) + "
                                        ↪ >(&" + stream + to_string(num) + ",
                                        ↪ function_pointer + to_string(num) + ",
                                        ↪  + arg_list->code + ");\n";
1525                    }
1526            }
1527        return s;
1528    }
1529
1530
1531    /* StatementNode */
1532    StatementNode::StatementNode(DeclarativeStatementNode *d){
1533        stmts.decl = d;
1534        code = d->code;
1535    }
```

```
1536
1537
1538 StatementNode::StatementNode(ConditionalStatementNode *c) {
1539     stmts.cond = c;
1540     code = c->code;
1541 }
1542
1543
1544 StatementNode::StatementNode(JumpStatementNode *j) {
1545     stmts.jump = j;
1546     code = j->code;
1547     returns_value = j->returns_value;
1548 }
1549
1550
1551 StatementNode::StatementNode(LoopStatementNode *l) {
1552     stmts.loop = l;
1553     code = l->code;
1554 }
1555
1556
1557 void StatementNode::seppuku(){
1558     stmts.decl->seppuku();
1559     delete this;
1560 }
1561
1562
1563 StatementNode::StatementNode(LinkStatementNode *l){
1564     stmts.link = l;
1565     code = l->code;
1566 }
1567
1568
1569 /* StatementListNode */
1570 StatementListNode::StatementListNode() {
1571     stmt_list = new vector<StatementNode *>();
1572     st_node = nullptr;
1573 }
1574
1575
1576 StatementListNode::StatementListNode(SymbolTableNode *s) {
1577     stmt_list = new vector<StatementNode *>();
1578     st_node = s;
1579 }
1580
1581
1582 void StatementListNode::push_statement(StatementNode *s) {
1583     stmt_list->push_back(s);
1584     if (s->returns_value) {
1585         returns_value = true;
1586     }
1587     code = code + s->code;
1588 }
1589
```

```
1590
1591 void StatementListNode::seppuku(){
1592     for(std::vector<StatementNode *>::iterator it = stmt_list->begin();
1593             it != stmt_list->end(); ++it)
1594         (*it)->seppuku();
1595
1596     delete this;
1597 }
1598
1599
1600 /* FunctionNode */
1601 FunctionNode::FunctionNode(TypeNode *_type, string id_node,
1602             DeclArgsNode *decl_args_list, StatementListNode *stmt_list_n){
1603     if (_type->sym != tVAR) {
1604         error = true;
1605         cout << FUNCTION_BASIC_TYPE_ERR << endl;
1606     }
1607     id = id_node;
1608     decl_args = decl_args_list;
1609     stmt_list = stmt_list_n;
1610     type = _type->type;
1611
1612     if (type != tVOID && !stmt_list->returns_value) {
1613         error = true;
1614         cout << RETURN_TYPE_ERROR << endl;
1615     }
1616
1617     if (id.compare("main") == 0) {
1618         code = "int " + id + "(" + decl_args_list->code + ")" + stmt_list_n->
                ↪ code;
1619     } else {
1620         code = type_to_str(type) + " " + id + "(" + decl_args_list->code + ")"
                ↪  + stmt_list_n->code;
1621     }
1622 }
1623
1624
1625 void FunctionNode::seppuku(){
1626     decl_args->seppuku();
1627     stmt_list->seppuku();
1628     delete this;
1629 }
1630
1631
1632 /* ProgramSectionNode */
1633 ProgramSectionNode::ProgramSectionNode(FunctionNode *f) {
1634     contents.function = f;
1635     code = f->code;
1636 }
1637
1638
1639 ProgramSectionNode::ProgramSectionNode(DatasetNode *d) {
1640     contents.dataset = d;
1641     code = d->code;
```

```
1642 }
1643
1644 ProgramSectionNode::ProgramSectionNode(DeclarativeStatementNode *d){
1645     contents.decl = d;
1646     if(d->en->value){
1647         Entry *ent = sym_table.get(d->en->value->code);
1648         if(ent)
1649             ent->is_final = true;
1650     }
1651
1652     if(d->en->bin_exp == nullptr){
1653         error = true;
1654         cout << FINAL_MUST_INITIALIZE << endl;
1655     }
1656     code = "const " + d->code;
1657 }
1658
1659 void ProgramSectionNode::seppuku(){
1660     contents.function->seppuku();
1661     delete this;
1662 }
1663
1664
1665 /* ProgramNode */
1666 ProgramNode::ProgramNode() {
1667     code = "";
1668 }
1669
1670
1671 void ProgramNode::add_section(ProgramSectionNode *p) {
1672     code += p->code;
1673 }
```

../source-code/frontend/ast.cpp

```
 1 %{
 2 #include <string>
 3 #include "frontend/ast.h"
 4 #include "structures/enum.h"
 5 #include "ripple.tab.hpp"
 6 #include "frontend/symbol_table/symbol_table.h"
 7
 8 #define CXX "clang++"
 9 #define CPP_STANDARD "c++11"
10 #define LIBS "-L./backend/lib/ -lbackend -lfile -lhtml -lxml -pthread"
11
12 void install_id();
13
14 extern "C" int yywrap();
15
16 int line_no = 1;          // line number counter
17 bool error = false;       // program will not compile if this is true
18
19 string filename_cpp;      // intermediate code file
20 string filename;          // executable file
```

```
21
22 string cpp_code;
23
24
25 bool saw_id;
26 string last_id;
27 e_type last_type;
28 void db(std::string m){
29 //       cout << m << endl;
30 }
31
32 SymbolTable sym_table;
33
34 %}
35
36 /* regular definitions */
37 delim        [ \t]
38 nl                        \n
39 comment      ("#~"[^(~#)]*"~#")|(#[^(~|\n)]*\n)
40 ws           ({comment}|{delim})+
41 letter       [A-Za-z_]
42 digit        [0-9]
43 literal      \"([^\"]|(\\\"))*\"
44 id          {letter}({letter}|{digit})*
45 integer     {digit}+
46 float       {digit}+\.{digit}*
47
48 %%
49
50 {nl}          {line_no++;}
51 {ws}          {string str = yytext; size_t s = count(str.begin(), str.end(),
       ↪ '\n'); line_no += s;       }
52 if            {db(yytext); saw_id = false; return IF;  }
53 else          {db(yytext); saw_id = false; return ELSE;          }
54 while         {db(yytext); saw_id = false; return WHILE;        }
55 for           {db(yytext); saw_id = false; return FOR; }
56 link          {db(yytext); saw_id = false; return LINK;          }
57 dataset       {db(yytext); saw_id = false; return DATASET;      }
58 return        {db(yytext); saw_id = false; return RETURN;       }
59 continue      {db(yytext); saw_id = false; return CONTINUE;     }
60 break         {db(yytext); saw_id = false; return BREAK;  }
61 stop          {db(yytext); saw_id = false; return STOP;   }
62 int           {db(yytext); saw_id = false; last_type = tINT;
63                yylval.string = new std::string("INT"); return TYPE;     }
64 float         {db(yytext); saw_id = false; last_type = tFLOAT;
65                yylval.string = new std::string("FLOAT"); return TYPE;   }
66 void          {db(yytext); saw_id = false; last_type = tVOID;
67                yylval.string = new std::string("VOID"); return TYPE;    }
68 string        {db(yytext); saw_id = false; last_type = tSTRING;
69                yylval.string = new std::string("STRING"); return TYPE;  }
70 bool          {db(yytext); saw_id = false; last_type = tBOOL;
71                yylval.string = new std::string("BOOL"); return TYPE;    }
72 true          {db(yytext); saw_id = false; yylval.boolean = true; return TRUE
       ↪ ;}
```

```
false␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.boolean␣=␣false;␣return␣
    ↪ FALSE;}
not␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣NOT;␣}
and␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣AND;␣}
or␣␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣OR;␣}
then␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣THEN;␣}
final␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣FINAL;␣}
web_stream␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.string␣=␣new␣string("
    ↪ WebStreamReader");␣return␣tSTREAM_READER;␣}
keyboard_stream␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.string␣=␣new␣string("
    ↪ KeyboardStreamReader");␣return␣tSTREAM_READER;␣}
file_stream␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.string␣=␣new␣string("
    ↪ FileStreamReader");␣return␣tSTREAM_READER;␣}


{id}␣␣␣␣␣␣␣{␣␣␣db(yytext);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣yylval.string␣=␣new␣std::string(yytext);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣saw_id␣=␣true;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣last_id␣=␣std::string(yytext);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣ID;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}

{integer}␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.integer␣=␣atoi(yytext);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣INTEGER;␣}
{float}␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.decimal␣=␣atof(yytext);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣FLOAT_LIT;␣}
{literal}␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣yylval.string␣=␣new␣string(yytext)
    ↪ ;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣STRING_LITERAL;␣}

"="␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣tASSIGN;␣}
"+"␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣tPLUS;␣␣␣}
"-"␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣tMINUS;␣}
"*"␣␣␣␣␣␣␣␣␣{␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣tTIMES;␣}
"/"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{␣␣␣␣␣␣␣db(yytext);␣saw_id␣=␣false;␣return␣tDIV;␣}
"//"              {       db(yytext); saw_id = false; return tFLDIV; }
"%"           {   db(yytext); saw_id = false; return tMOD; }
"^"                       {       db(yytext); saw_id = false; return tEXP;   }
"@"                       {       db(yytext); saw_id = false; return tSIZE;  }
">"                       {       db(yytext); saw_id = false; return tGT;    }
"<"                       {       db(yytext); saw_id = false; return tLT;    }
">="              {       db(yytext); saw_id = false; return tGE;  }
"<="              {       db(yytext); saw_id = false; return tLE;  }
"=="              {       db(yytext); saw_id = false; return tEQ;  }
"!="              {       db(yytext); saw_id = false; return tNE;  }
","                       {       db(yytext); saw_id = false; return tCOMMA; }
"("                       {       db(yytext); saw_id = false; return tL_PAREN; }
")"                       {       db(yytext); saw_id = false; return tR_PAREN; }
"["                       {       db(yytext); saw_id = false; return tL_BRACKET;
    ↪ }
"]"                       {       db(yytext); saw_id = false; return tR_BRACKET;
    ↪ }
"{"                       {       db(yytext); if(saw_id)
                                  sym_table.new_dataset(line_no, last_id);
```

```
120                             else
121                                 sym_table.scope_in(line_no);
122                             saw_id = false;
123                             return tL_CURLY; }
124 "}"                     {       db(yytext); sym_table.scope_out(line_no);
    ↪ return tR_CURLY; }
125 ";"             {   db(yytext); saw_id = false; return tSEMI;   }
126 "\."            {   db(yytext); saw_id = false; return tACCESS; }
127 "<-"            {   db(yytext); saw_id = false; return tARROW; }
128 %%
129
130
131 int main(int argc, char **argv) {
132
133     /* Handle improper input */
134         if(argc < 2){
135                 cout << "usage ./rpl <input_file> [output_file]" << endl;
136                 exit(1);
137         } else if (argc < 3){
138                 filename = "output";
139         filename_cpp = "output.cpp";
140         }
141         else {
142                 filename = argv[2];
143         filename_cpp = filename + ".cpp";
144         }
145
146     yyin = fopen(argv[1], "r");
147     yyout = fopen(filename_cpp.c_str(), "w");
148     yyparse();
149
150     /* Detect if a main function has been specified.
151      * Program will not compile if it hasn't. */
152     Entry *main_func = sym_table.get("main");
153     if (!main_func || main_func->symbol_type != tFUNC) {
154         cout << MAIN_FUNC_ERROR << endl;
155         error = true;
156     }
157
158
159     if(error){
160         cout << COMPILE_ERR << endl;
161         return 1;
162     }
163
164     write_to_file(filename_cpp, cpp_code);
165     string compile_cpp = string(CXX) + " " + filename_cpp + " -L. -I./backend/
        ↪ streamreader/ -o " + filename
166                         + " --std=" + CPP_STANDARD + " " + LIBS;
167     system(compile_cpp.c_str());
168 #ifndef DEBUG
169     remove(filename_cpp.c_str());
170 #endif
171     return 0;
```

```
172 }
```

../source–code/frontend/ripple.l

```
1  %{
2  #include <cctype>
3  #include <cstdio>
4  #include <string>
5  #include <iostream>
6  #include "frontend/symbol_table/symbol_table.h"
7  #include "frontend/ast.h"
8  #include "misc/debug_tools.h"
9  using namespace std;
10
11 extern int yylex();
12 void yyerror(const char *s) { printf("%s\n", s); }
13
14 extern SymbolTable sym_table;
15
16 e_type func_type;
17
18 extern int line_no;
19 extern string cpp_code;
20 extern e_type last_type;
21 %}
22
23
24 %union {
25     ProgramNode *prog;
26     ProgramSectionNode *progsec;
27     FunctionNode *func;
28     SymbolTableNode *st_node;
29     DatasetNode *dataset;
30
31     StatementListNode *stmt_list;
32     StatementNode *stmt;
33     DeclarativeStatementNode *dec_stmt;
34     ConditionalStatementNode *cond_stmt;
35     JumpStatementNode *jump_stmt;
36     LoopStatementNode *loop_stmt;
37     LinkStatementNode *link_stmt;
38
39     ExpressionNode *expr;
40     BinaryExpressionNode *bin_expr;
41     UnaryExpressionNode *un_expr;
42     StreamReaderNode *str_read;
43     ArrayAccessNode *arr_acc;
44     DatasetAccessNode *ds_acc;
45     FunctionCallNode *fn_call;
46     LiteralNode *lit;
47     ValueNode *val;
48     ArgsNode *args;
49     ArrayInitNode *ainit;
50     DeclArgsNode *decl_args;
51     IDNode *idn;
```

```
52      TypeNode *type;
53
54      int integer;
55      string *string;
56      double decimal;
57      bool boolean;
58  }
59
60  /***************
61      KEYWORDS
62   ***************/
63  %token <integer> IF
64  %token <integer> ELSE
65  %token <integer> FOR
66  %token <integer> WHILE
67  %token <integer> LINK
68  %token <integer> IMPORT
69  %token <integer> FINAL
70  %token <integer> RETURN
71  %token <integer> CONTINUE
72  %token <integer> BREAK
73  %token <integer> THEN
74  %token <integer> STOP
75
76  /***************
77          TYPES
78   ***************/
79  %token <integer> TYPE
80  %token <integer> BOOL
81  %token <integer> INT
82  %token <integer> FLOAT
83  %token <integer> STRING
84  %token <integer> DATASET
85  %token <integer> VOID
86  %token <integer> tSTREAM_READER
87
88  /***************
89      IDIABLES
90   ***************/
91  %token <string> ID
92  %token <integer> INTEGER
93  %token <decimal> FLOAT_LIT
94  %token <string>  STRING_LITERAL
95  %token <boolean> TRUE
96  %token <boolean> FALSE
97
98  /*****************
99      SEPARATORS
100  *****************/
101 %token tSEMI
102 %token tARROW
103 %token tCOMMA
104 %token tL_PAREN
105 %token tR_PAREN
```

```
106 %token tL_BRACKET
107 %token tR_BRACKET
108 %token tL_CURLY
109 %token tR_CURLY
110
111 /************************
112     TYPE DECLARATIONS
113  ************************/
114 %type <prog> program
115 %type <progsec> program_section
116 %type <dataset> dataset_declaration
117 %type <idn> var
118 %type <type> dtype
119 %type <lit> literal
120 %type <un_expr> unary_expression
121 %type <bin_expr> exp_expression mult_expression add_expression rel_expression
    ↪ eq_expression and_expression or_expression
122 %type <expr> expression opt_expression
123 %type <arr_acc> array_access
124 %type <fn_call> function_call
125 %type <ds_acc> dataset_access
126 %type <val> value array_opt
127 %type <args> args
128 %type <ainit> array_initialization
129 %type <dec_stmt> declarative_statement
130 %type <jump_stmt> jump_statement
131 %type <cond_stmt> conditional_statement
132 %type <loop_stmt> loop_statement
133 %type <link_stmt> link_statement
134 %type <stmt> statement
135 %type <stmt_list> statement_list statement_block else_statement
136 %type <func> function
137 %type <decl_args> decl_args
138 %type <string> stream_reader_name
139 %type <str_read> stream_reader
140
141 /*********************
142     ASSOCIATIVITY
143  *********************/
144 %left tEQ
145 %left tNE
146
147 %left tLT
148 %left tLE
149 %left tGT
150 %left tGE
151
152 %left AND
153 %left OR
154
155 %left tPLUS
156 %left tMINUS
157 %left tTIMES
158 %left tDIV
```

```
%left tMOD
%left tFLDIV

%right tEXP
%right tSIZE
%right NOT

%right tASSIGN
%right tACCESS

%%

program : program program_section      { $$->add_section($2);
                                         cpp_code = $$->code;
                                         d("program");   }
        |                                { $$ = new ProgramNode(); }
        ;

program_section : function { $$ = new ProgramSectionNode($1); d("program_section"); }
                | dataset_declaration { $$ = new ProgramSectionNode($1); d("program_section"); }
                | FINAL declarative_statement { $$ = new ProgramSectionNode($2); }
                ;

function : dtype ID tL_PAREN decl_args tR_PAREN statement_block { func_type = $1->type;
                                                                   $$ = new FunctionNode($1, *$2, $4, $6);
                                                                   sym_table.add_function(*($2), ($1)->type, line_no,
                                                                                                                      $4
                                                                   d("function");
                                                                 }
          ;

dataset_declaration : DATASET ID tL_CURLY decl_args tR_CURLY  { $$ = new
```

```
    ↪ DatasetNode(*$2, $4);
191                                                                 d("dataset␣
                                                                        ↪ declaration
                                                                        ↪ "); }
192                     ;
193
194 statement_block : tL_CURLY statement_list tR_CURLY { $$ = $2; $$->code = "{\n"
    ↪  + $2->code + "}\n";
195                                                         d("statement_block"); }
196                 | tL_CURLY tR_CURLY { $$ = new StatementListNode(); $$->code =
                        ↪  "{}\n"; d("empty␣statement_block");}
197                 ;
198
199 statement_list : statement_list statement       { $1->push_statement($2); d("
    ↪ statement_list␣-␣stmt-list"); }
200                | statement                      { $$ = new StatementListNode()
                        ↪ ; $$->push_statement($1);
201                                                   d("statement_list␣-␣stmt");
                                                        ↪ }
202                |
203                ;
204
205 statement : declarative_statement               { $$ = new StatementNode($1);
    ↪ d("statement␣-␣declarative"); }
206           | conditional_statement               { $$ = new StatementNode($1);
                  ↪ d("statement␣-␣conditional"); }
207           | jump_statement                      { $$ = new StatementNode($1);
                  ↪ d("statement␣-␣jump"); }
208           | loop_statement                      { $$ = new StatementNode($1);
                  ↪ d("statement␣-␣loop"); }
209           | link_statement                      { $$ = new StatementNode($1);
                  ↪ d("statement␣-␣link"); }
210           ;
211
212 jump_statement : RETURN expression tSEMI        { $$ = new JumpStatementNode("
    ↪ return", $2); $$->returns_value = true; d("jump␣-␣return"); }
213                | RETURN tSEMI                   { $$ = new JumpStatementNode("
                        ↪ return"); d("jump␣-return"); }
214                | CONTINUE tSEMI                 { $$ = new JumpStatementNode("
                        ↪ continue"); d("jump␣-␣continue"); }
215                | BREAK tSEMI                    { $$ = new JumpStatementNode("
                        ↪ break"); d("jump␣-␣break"); }
216                | STOP tSEMI                     { $$ = new JumpStatementNode("
                        ↪ stop"); d("jump␣-␣break"); }
217                ;
218
219
220 conditional_statement : IF tL_PAREN expression tR_PAREN statement_block
    ↪ else_statement  { $$ =
221                                                                     new
                                                                        ↪
                                                                        ↪ Condition
                                                                        ↪ (
                                                                        ↪ $3
```

```
↪ ,
↪
↪ $5
↪ ,
↪
↪ $6
↪ )
↪ ;
↪
                                                                                      c

                    ;

else_statement : ELSE statement_block    { $$ = $2; d("else_statement"); }
               |                          { $$ = NULL; }
               ;

loop_statement : FOR tL_PAREN opt_expression tSEMI opt_expression tSEMI
    ↪ opt_expression tR_PAREN statement_block
                                                        { $$ = new
                                                          ↪ LoopStatementNode($3
                                                          ↪ , $5, $7, $9); d("
                                                          ↪ loop - for"); }
               | WHILE tL_PAREN expression tR_PAREN statement_block
                                                { $$ = new LoopStatementNode(
                                                  ↪ nullptr ,$3, nullptr, $5); d
                                                  ↪ ("loop - while");}
               ;

declarative_statement : dtype expression tSEMI    { $$ = new
    ↪ DeclarativeStatementNode($1, $2);
                                                       d("declarative_statement - 
                                                         ↪  type expr"); delete
                                                         ↪ $1; }
                      | expression tSEMI          { $$ = new
                        ↪ DeclarativeStatementNode($1);
                                                        d("declarative_statement -
                                                          ↪  expr"); }
                      ;

link_statement : LINK tL_PAREN var tARROW expression tR_PAREN tSEMI          {
    ↪ $$ = new LinkStatementNode($3, $5);
                                                                               d
                                                                                 ↪
```

```
                    | LINK tL_PAREN var tARROW expression tR_PAREN THEN ID tSEMI { $$ = new LinkStatementNode($3, $5, *$8);
                    d("link statement with then!"); }
                    | LINK tL_PAREN var tARROW stream_reader tR_PAREN tSEMI      { $$ = new LinkStatementNode($3, $5); d("link stream"); }
                    | LINK tL_PAREN var tARROW stream_reader tR_PAREN THEN ID tSEMI  { $$ = new LinkStatementNode($3, $5, *$8); }
                    | LINK tL_PAREN var tARROW var tARROW stream_reader tR_PAREN tSEMI      { $$ = new LinkStatementNode($3, $5, $7); }
                    | LINK tL_PAREN var tARROW var tARROW stream_reader tR_PAREN THEN ID tSEMI      { $$ = new LinkStatementNode($3, $5, $7, *$10); }
                    ;

stream_reader : stream_reader_name tL_PAREN args tR_PAREN { $$ = new StreamReaderNode(*$1, $3); d("stream_reader"); }

stream_reader_name : tSTREAM_READER { $$ = yylval.string; }

opt_expression : expression              { $$ = $1; d("opt_expression"); }
                    |                                { $$ = nullptr; d("no opt_expression");}
                    ;

expression : or_expression                 { $$ = new ExpressionNode($1);  d("expression"); }
               | value tASSIGN or_expression { $$ = new ExpressionNode($3, $1); d(
```

```
                ↪ "expression assign"); }
261             ;
262
263 or_expression : and_expression                    { $$ = $1; d("or_expression");
    ↪  }
264                | or_expression OR and_expression { $$ = new
                     ↪ BinaryExpressionNode($1, "or", $3); d("or_expression OR");
                     ↪  }
265                ;
266
267 and_expression : eq_expression                     { $$ = $1; d("
    ↪ and_expression"); }
268                | and_expression AND eq_expression  { $$ = new
                     ↪ BinaryExpressionNode($1, "and", $3);
269                                                      d("and_expression AND");
                                                         ↪  }
270                ;
271
272 eq_expression : rel_expression                     { $$ = $1; d("
    ↪ eq_expression"); }
273                | eq_expression tEQ rel_expression  { $$ = new
                     ↪ BinaryExpressionNode($1, "==", $3);
274                                                      d("eq_expression ==");  }
275                | eq_expression tNE rel_expression  { $$ = new
                     ↪ BinaryExpressionNode($1, "!=", $3);
276                                                      d("eq_expression !=");  }
277                ;
278
279 rel_expression : add_expression                    { $$ = $1; d("
    ↪ rel_expression"); }
280                | rel_expression tGT add_expression { $$ = new
                     ↪ BinaryExpressionNode($1, ">", $3);
281                                                      d("rel_expression >");  }
282                | rel_expression tLT add_expression { $$ = new
                     ↪ BinaryExpressionNode($1, "<", $3);
283                                                      d("rel_expression <");  }
284                | rel_expression tGE add_expression { $$ = new
                     ↪ BinaryExpressionNode($1, ">=", $3);
285                                                      d("rel_expression >=");
                                                         ↪ }
286                | rel_expression tLE add_expression { $$ = new
                     ↪ BinaryExpressionNode($1, "<=", $3);
287                                                      d("rel_expression <=");
                                                         ↪ }
288                ;
289
290 add_expression : mult_expression                    { $$ = $1; d("
    ↪ add_expression");}
291                | add_expression tPLUS mult_expression  { $$ = new
                     ↪ BinaryExpressionNode($1, "+", $3);
292                                                        d("add_expression +")
                                                           ↪ ;}
293                | add_expression tMINUS mult_expression { $$ = new
                     ↪ BinaryExpressionNode($1, "-", $3);
```

```
294                                                                 d("add_expression -")
                                                                        ↪ ; }
295                 ;
296
297 mult_expression : exp_expression                                   { $$ = $1; d("
        ↪ mult_expression");}
298                 | mult_expression tTIMES exp_expression         { $$ = new
                        ↪ BinaryExpressionNode($1, "*", $3);
299                                                                   d("mult_expression
                                                                        ↪ *"); }
300                 | mult_expression tDIV exp_expression           { $$ = new
                        ↪ BinaryExpressionNode($1, "/", $3);
301                                                                   d("mult_expression
                                                                        ↪ /"); }
302                 | mult_expression tFLDIV exp_expression         { $$ = new
                        ↪ BinaryExpressionNode($1, "//", $3);
303                                                                   d("mult_expression
                                                                        ↪ //"); }
304                 | mult_expression tMOD exp_expression           { $$ = new
                        ↪ BinaryExpressionNode($1, "%", $3);
305                                                                   d("mult_expression
                                                                        ↪ %"); }
306                 ;
307
308 exp_expression : unary_expression                                   { $$ = new
        ↪ BinaryExpressionNode($1); d("exp_expression");}
309                 | exp_expression tEXP unary_expression          { $$ = new
                        ↪ BinaryExpressionNode($1, "^", $3);
310                                                                   d("exp expression ^
                                                                        ↪ "); }
311                 ;
312
313 unary_expression : value                                { $$ = new UnaryExpressionNode(
        ↪ $1); d("unary_expression");};
314                  | tMINUS unary_expression              { $$ = new UnaryExpressionNode(
                        ↪ $2, "-"); d("unary_expression -"); }
315                  | tSIZE unary_expression               { $$ = new UnaryExpressionNode(
                        ↪ $2, "@"); d("unary_expression @"); }
316                  | tL_PAREN dtype tR_PAREN unary_expression { $$ = new
                        ↪ UnaryExpressionNode($4, $2); d("unary cast"); }
317                  | NOT unary_expression                 { $$ = new UnaryExpressionNode(
                        ↪ $2, "not"); d("unary_expression not"); }
318                  ;
319
320 value : literal             { $$ = new ValueNode($1); d("value - literal"); }
321       | function_call        { $$ = new ValueNode($1); d("value - function_call"
            ↪ ); }
322       | array_access         { $$ = new ValueNode($1); d("value - array_access")
            ↪ ; }
323       | dataset_access       { $$ = new ValueNode($1); d("value - dataset_access
            ↪ "); }
324       | var                   { $$ = new ValueNode($1); d("value - id");}
325       | tL_PAREN expression tR_PAREN { $$ = new ValueNode($2); d("value -
            ↪ expression"); }
```

```
326       | tL_CURLY array_initialization tR_CURLY  { $$ = new ValueNode($2); d("
             ↪ value - array_initialization"); }
327       ;
328
329 array_initialization : array_initialization tCOMMA expression  { $1->add_arg(
       ↪ $3); d("array_initialization added - 1");}
330                      | expression                               { $$ = new
                           ↪ ArrayInitNode(); $$->add_arg($1);
331                                                                   d("arg added
                                                                      ↪ "); }
332                      |                                          { $$ = new
                           ↪ ArrayInitNode(); d("empty arg");}
333                      ;
334
335 args : args tCOMMA expression   { $1->add_arg($3); d("arg added - 1"); }
336      | expression               { $$ = new ArgsNode(); $$->add_arg($1); d("arg
          ↪ added"); }
337      |                          { $$ = new ArgsNode(); d("empty arg");}
338      ;
339
340 decl_args : decl_args tCOMMA dtype var   { $1->add_arg($3, $4); d("decl arg
       ↪ added - 1"); }
341           | dtype var                    { $$ = new DeclArgsNode($1, $2); d("
                ↪ decl arg added"); }
342           |                              { $$ = new DeclArgsNode(); d("empty
                ↪ decl arg"); }
343           ;
344
345 array_opt : tL_BRACKET value tR_BRACKET { $$ = $2; d("array_opt"); }
346           | tL_BRACKET tR_BRACKET        { $$ = new ValueNode(new ArrayInitNode
                ↪ ()); $$->type = tINT; $$->code = "";
347                                            $$->sym = tVAR; d("empty array opt")
                                                ↪ ;}
348           |                              { $$ = nullptr; }
349           ;
350
351 function_call : ID tL_PAREN args tR_PAREN { $$ = new FunctionCallNode(*($1),
       ↪ $3); d("function call"); }
352                 ;
353
354 array_access : value tL_BRACKET expression tR_BRACKET { $$ = new
       ↪ ArrayAccessNode($1, $3); d("Array access");}
355              ;
356
357 dataset_access : ID tACCESS var { $$ = new DatasetAccessNode(*$1, $3->entry->
       ↪ name); d("Dataset access");}
358                ;
359
360 var : ID { Entry *entry = sym_table.get(*yylval.string);
361          if (entry) {
362               $$ = new IDNode(entry); d("id");
363           } else {
364               sym_table.put(*yylval.string, tNOTYPE, line_no, tVAR);
365               Entry *new_entry = sym_table.get(*yylval.string);
```

```
366                 $$ = new IDNode(new_entry); d("undeclared id");
367             }
368           }
369     ;
370
371 dtype : TYPE array_opt {  $$ = new TypeNode(last_type, $2); d("type: "); }
372       | DATASET ID  { $$ = new TypeNode(last_type, *$2); d("dataset: " + *
            ↪ yylval.string); }
373       ;
374
375 literal : INTEGER             { $$ = new LiteralNode($1); $$->code = to_string
    ↪ (yylval.integer); d("literal - INT");}
376         | FLOAT_LIT           { $$ = new LiteralNode($1); $$->code = to_string
            ↪ (yylval.decimal); d("literal - FLOAT");}
377         | STRING_LITERAL      { $$ = new LiteralNode($1); $$->code = *yylval.
            ↪ string; d("literal - STRING"); }
378         | TRUE                          { $$ = new LiteralNode($1); $$->code
            ↪ = "true";  d("literal - true"); }
379         | FALSE                         { $$ = new LiteralNode($1); $$->code
            ↪ = "false"; d("literal - false"); }
380         ;
381
382 %%
```

../source-code/frontend/ripple.ypp

```
1  CC  = gcc
2  CXX = clang++
3
4  INCLUDES =
5  CFLAGS   = -g -Wall $(INCLUDES)
6  CXXFLAGS = -g -Wall -std=c++11 $(INCLUDES)
7
8  LDFLAGS = -g
9  LDLIBS = -lm
10
11 default: libsym.a
12
13 libsym.a: symbol_table.o
14         ar rc libsym.a symbol_table.o
15         ranlib libsym.a
16
17 symbol_table.o: hashmap.o
18
19 hashmap.o: hashmap.h
20
21 .PHONY: clean
22 clean:
23         rm -rf hashmap linkedlist symbol_table *~ *.o *.dSYM a.out *.a
24
25 .PHONY: cleani
26 cleani:
27         rm -rf hashmap linkedlist symbol_table *~ *.o *.dSYM a.out
28
29 .PHONY: all
```

```
30 all: clean default
```

../source–code/frontend/symbol_table/Makefile

```
1  #ifndef __HASHMAP_H__
2  #define __HASHMAP_H__
3
4  #include <array>
5  #include <cmath>
6  #include <iostream>
7  #include <list>
8  #include <string>
9
10 #include "../../structures/enum.h"
11 #include "../../structures/union.h"
12
13 #define HASH_PRIME 31
14 #define TABLE_SIZE 29
15
16 using namespace std;
17
18 class Entry {
19 public:
20
21     string name;
22     enum e_type type;
23     int line_no;
24     string ds_name;
25     union literal val;
26     enum e_symbol_type symbol_type;
27     list<e_type> *args;
28     int array_length;
29     bool has_dependents = false;
30     bool is_final = false;
31
32     Entry(string n, e_type v, int line, e_symbol_type s);
33     Entry(string n, int line);
34
35     void classify (e_symbol_type s);
36     void add_args (list<e_type> *l);
37     void add_length (int i);
38
39     bool operator==(string n);
40
41     ~Entry();
42 };
43
44 typedef array<list<Entry *>*, TABLE_SIZE> table;
45
46 class HashMap{
47     table t;
48
49 public:
50     HashMap();
51     ~HashMap();
```

```
52
53     bool put(string word, e_type v, int line_no, e_symbol_type s);
54
55      bool contains(string word);
56      Entry *get(string word);
57 };
58
59 #endif
```

../source–code/frontend/symbol_table/hashmap.h

```
1  #include "hashmap.h"
2
3  using namespace std;
4
5  bool list_contains(list<Entry *> l, string word) {
6      for (auto element = l.begin(); element != l.end(); ++element) {
7          if ((*element)->name.compare(word) == 0 ) {
8              return true;
9          }
10     }
11     return false;
12 }
13
14
15 int hashCode(const string word) {
16
17     int h = 0;
18
19     if (word.length() > 0) {
20         int i;
21         for (i = 0; i < word.length(); i++) {
22             h = HASH_PRIME * h + word[i];
23         }
24     }
25
26     return (int) std::abs((float) h);
27 }
28
29
30 int isPrime(int num){
31
32     if(num < 4)
33         return 0;
34
35     int max = (int) pow(num, 0.5) + 1;
36     int i;
37     for( i = 2; i < max ; ++i )
38         if(num % i == 0)
39             return 0;
40     return 1;
41 }
42
43
44 /* HashMap */
```

```
45 HashMap::HashMap() {
46     for (int i = 0; i < TABLE_SIZE; i++) {
47         t[i] = new list<Entry *>;
48     }
49 }
50
51
52 HashMap::~HashMap() {
53     for(int i = 0; i < TABLE_SIZE; i++) {
54         delete t[i];
55     }
56 }
57
58
59 bool HashMap::contains(string word) {
60     int pos = hashCode(word) % TABLE_SIZE;
61     return list_contains(*t[pos], word);
62 }
63
64
65 bool HashMap::put(string word, e_type v, int line_no, e_symbol_type s =
       ↪ tNOSTYPE){
66     int pos = hashCode(word) % TABLE_SIZE;
67     if(list_contains(*t[pos], word)) {
68         return false;
69     }
70
71     Entry *new_entry = new Entry(word, v, line_no, s);
72     t[pos]->push_back(new_entry);
73     return true;
74 }
75
76
77 Entry *HashMap::get(string word) {
78     int pos = hashCode(word) % TABLE_SIZE;
79
80     for (auto element = t[pos]->begin(); element != t[pos]->end(); ++element)
       ↪ {
81         if ((*element)->name.compare(word) == 0 ) {
82             return *element;
83         }
84     }
85
86     return nullptr;
87 }
88
89
90 Entry::Entry(string n, e_type v, int line, e_symbol_type s = tNOSTYPE) {
91     name = n;
92     type = v;
93     line_no = line;
94     symbol_type = s;
95 }
96
```

```
97
98  Entry::Entry(string n, int line) {
99      name = n;
100     type = tNOTYPE;
101     line_no = line;
102     symbol_type = tNOSTYPE;
103 }
104
105
106 void Entry::classify(e_symbol_type s) {
107     symbol_type = s;
108 }
109
110
111 void Entry::add_args(list<e_type> *a) {
112     args = a;
113 }
114
115
116 void Entry::add_length(int l) {
117     array_length = l;
118 }
119
120
121 bool Entry::operator==(string n) {
122     return name.compare(n) == 0;
123 }
124
125
126 Entry::~Entry() {}
```

../source–code/frontend/symbol_table/hashmap.cpp

```
1  #ifndef __SYMBOL_TABLE_H__
2  #define __SYMBOL_TABLE_H__
3
4  #include <cstdio>
5  #include <iostream>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string>
9  #include <string.h>
10 #include <vector>
11
12 #include "hashmap.h"
13 #include "../../structures/enum.h"
14
15 #define RPL_STD_OUTPUT_FUNCTION "print"
16 #define RPL_STD_INPUT_FUNCTION "input"
17 #define RPL_STD_OPEN_FUNCTION "open"
18 #define RPL_STD_CLOSE_FUNCTION "close"
19 #define RPL_STD_READ_FUNCTION "read"
20
21 #define NUMBER_RESERVED_WORDS 10
22
```

```
23 #define RPL_STD_OUTPUT_FUNCTION "print"
24 #define RPL_STD_INPUT_FUNCTION "input"
25 #define RPL_STD_OPEN_FUNCTION "open"
26 #define RPL_STD_CLOSE_FUNCTION "close"
27 #define RPL_STD_READ_FUNCTION "read"
28
29 using namespace std;
30
31 class SymbolTableNode {
32 public:
33     HashMap *hashmap;
34     string name;
35     SymbolTableNode *sibling;
36     SymbolTableNode *child;
37     SymbolTableNode *parent;
38
39     SymbolTableNode();
40     SymbolTableNode(string n);
41     e_type get_type(string n);
42     ~SymbolTableNode();
43 };
44
45
46 class SymbolTable {
47     SymbolTableNode *start;
48     SymbolTableNode *current;
49
50     void insert_standard_functions();
51
52 public:
53     SymbolTable();
54     ~SymbolTable();
55     void scope_in(int line_no);
56     void scope_out(int line_no);
57
58     void new_dataset(int line_no, string name);
59     SymbolTableNode *get_dataset(string name);
60     Entry *get_dataset_member(string c, string i);
61
62     bool put(string word, e_type type, int line_no, e_symbol_type s);
63
64     bool contains(string word);
65     bool contains_in_scope(string word);
66
67     bool add_array(string name, e_type type, int line_no, int arr_length);
68     void add_length(string word, int i);
69     bool add_function(string name, e_type type, int line_no, list<e_type> *l);
70     bool instantiate_dataset(string name, string ds_name, int line_no);
71     void add_args(string word, list<e_type> *l);
72     void add_dsname(string word, string ds_name);
73     void classify(string word, e_symbol_type s);
74
75     Entry *get(string word);
76 };
```

```
77
78 #endif
```

../source–code/frontend/symbol_table/symbol_table.h

```
1  #include "symbol_table.h"
2
3  SymbolTableNode::SymbolTableNode() {
4      name = "";
5      hashmap = new HashMap();
6      sibling = nullptr;
7      child = nullptr;
8      parent = nullptr;
9  }
10
11
12 SymbolTableNode::SymbolTableNode(string n) {
13     name = n;
14     hashmap = new HashMap();
15     sibling = nullptr;
16     child = nullptr;
17     parent = nullptr;
18 }
19
20
21 e_type SymbolTableNode::get_type(string n) {
22     Entry *found;
23     if ((found = hashmap->get(n)))
24         return found->type;
25     else
26         return tNOTYPE;
27 }
28
29
30 SymbolTableNode::~SymbolTableNode() {
31     delete child;
32
33     SymbolTableNode *n = sibling;
34     SymbolTableNode *next;
35
36     while(n) {
37         next = n->sibling;
38         delete n;
39         n = next;
40     }
41
42     delete hashmap;
43 }
44
45
46 SymbolTable::SymbolTable(){
47     SymbolTableNode *main = new SymbolTableNode();
48     start = main;
49     current = main;
50     insert_standard_functions();
```

```
51 }
52
53
54 void SymbolTable::insert_standard_functions() {
55     /* Print */
56     add_function(RPL_STD_OUTPUT_FUNCTION, tVOID, 0, nullptr);
57     add_function(RPL_STD_INPUT_FUNCTION, tSTRING, 0, new list<e_type>({
       ↪ tSTRING }));
58
59     /* Files */
60     add_function("contains_word", tBOOL, 0,  new list<e_type>({ tSTRING,
       ↪ tSTRING }));
61     add_function("length", tINT, 0, new list<e_type>({ tSTRING }));
62     add_function("print_line", tVOID, 0, new list<e_type>({ tSTRING }));
63     add_function("locate_word", tINT, 0, new list<e_type>({ tSTRING, tSTRING
       ↪ }));
64
65     /* HTML */
66     add_function("contains_tag", tBOOL, 0, new list<e_type>({ tSTRING, tSTRING
       ↪  }));
67     add_function("get_num_tags", tINT, 0, new list<e_type>({ tSTRING, tSTRING
       ↪ }));
68     add_function("size", tINT, 0, new list<e_type>({ tSTRING }));
69     add_function("get_body", tSTRING, 0, new list<e_type>({ tSTRING }));
70     add_function("get_head", tSTRING, 0, new list<e_type>({ tSTRING }));
71     add_function("get_tag", tSTRING, 0, new list<e_type>({ tSTRING, tSTRING })
       ↪ );
72
73     /* XML */
74     add_function("get_num_nodes", tINT, 0, new list<e_type>({ tSTRING, tSTRING
       ↪  }));
75     add_function("get_node", tINT, 0, new list<e_type>({ tSTRING, tSTRING }));
76     add_function("get_node_text", tINT, 0, new list<e_type>({ tSTRING, tSTRING
       ↪  }));
77
78     /* conversions */
79     add_function("str_to_int", tINT, 0, new list<e_type>({ tSTRING }));
80     add_function("str_to_float", tFLOAT, 0, new list<e_type>({ tSTRING }));
81 }
82
83
84 void SymbolTable::scope_in(int line_no) {
85     SymbolTableNode *node = new SymbolTableNode;
86     if (!start) {
87         start = node;
88         node->sibling = node->parent = nullptr;
89     } else {
90         node->parent = current;
91         node->sibling = current->child;
92         current->child = node;
93     }
94     current = node;
95 }
96
```

```
97
98  void SymbolTable::scope_out(int line_no) {
99      if (current->parent){
100         current = current->parent;
101         return;
102     }
103
104     SymbolTableNode *node = new SymbolTableNode();
105
106     current->sibling = node;
107     current = node;
108 }
109
110
111 bool SymbolTable::add_function(string name, e_type type, int line_no, list<
        ↪ e_type> *args) {
112     bool error = false;
113     if (contains_in_scope(name)) {
114         cout << "Redefinition of function " << name << " previously defined on
                ↪ line " << line_no << endl;
115         error = true;
116     } else {
117         put(name, type, line_no, tFUNC);
118         add_args(name, args);
119     }
120     return error;
121 }
122
123
124 bool SymbolTable::add_array(string name, e_type type, int line_no, int length)
        ↪ {
125     bool error = false;
126     if (contains_in_scope(name)) {
127         error = true;
128     } else {
129         put(name, type, line_no, tARR);
130         add_length(name, length);
131     }
132     return error;
133 }
134
135
136 bool SymbolTable::instantiate_dataset(string instance_name, string ds_name,
        ↪ int line_no) {
137     bool error = false;
138     if (contains_in_scope(instance_name) && get(instance_name)->type !=
            ↪ tNOTYPE) {
139         error = true;
140     } else {
141         add_dsname(instance_name, ds_name);
142     }
143     return error;
144 }
145
```

```
146
147 void SymbolTable::add_dsname(string name, string ds_name) {
148     Entry *entry = get(name);
149     entry->ds_name = ds_name;
150     entry->type = tDERIV;
151 }
152
153
154 void SymbolTable::new_dataset(int line_no, string name) {
155     SymbolTableNode *node = new SymbolTableNode(name);
156
157     node->parent = current;
158     node->sibling = current->child;
159     current->child = node;
160     current = node;
161 }
162
163
164 SymbolTableNode *SymbolTable::get_dataset(string name) {
165     SymbolTableNode *node;
166     for(node = current; node; node = node->sibling) {
167         if (!node->name.empty() && node->name.compare(name) == 0) {
168             return node;
169         }
170     }
171     return nullptr;
172 }
173
174
175 Entry *SymbolTable::get_dataset_member(string name, string member_name) {
176     SymbolTableNode *node = get_dataset(name);
177     return node->hashmap->get(member_name);
178 }
179
180
181 bool SymbolTable::put(string word, e_type v, int line_no, e_symbol_type s =
    ↪ tNOSTYPE) {
182     return current->hashmap->put(word, v, line_no, s);
183 }
184
185
186 void SymbolTable::classify(string word, e_symbol_type s) {
187     get(word)->classify(s);
188 }
189
190
191 void SymbolTable::add_args(string word, list<e_type> *l) {
192     get(word)->add_args(l);
193 }
194
195
196 void SymbolTable::add_length(string word, int l) {
197     get(word)->add_length(l);
198 }
```

```
199
200
201 bool SymbolTable::contains(string word) {
202     SymbolTableNode *n = current;
203     while(n) {
204         if (n->hashmap->contains(word))
205             return true;
206         n = n->parent;
207     }
208     return false;
209 }
210
211
212 bool SymbolTable::contains_in_scope(string word){
213     return current->hashmap->contains(word);
214 }
215
216
217 Entry *SymbolTable::get(string word) {
218     SymbolTableNode *n = current;
219     while(n) {
220         if (n->hashmap->contains(word))
221             return n->hashmap->get(word);
222         n = n->parent;
223     }
224     return nullptr;
225 }
226
227
228 SymbolTable::~SymbolTable() {
229 }
```

../source-code/frontend/symbol_table/symbol_table.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  #include "symbol_table.h"
6
7  using namespace std;
8
9  int main () {
10
11     int line_no = 0;
12     SymbolTable::SymbolTable st;
13
14     string line;
15     ifstream myfile;
16     myfile.open("symbol_tester.txt");
17     if(myfile.is_open()){
18         while (getline(myfile,line)){
19             ++line_no;
20             //cout << line << endl;
21             if(line.compare("{") == 0){
```

```
22                  st.scope_in(line_no);
23              } else if(line.compare("}") == 0) {
24                  myfile.seekg(0);
25                  while (getline(myfile,line)){
26                      cout << st.get(line) << endl;
27                  }
28              } else {
29                  st.put(line);
30              }
31          }
32      }
33      myfile.close();
34      return 0;
35  }
```

../source–code/frontend/symbol_table/tester.cpp

```
1  #ifndef __RIPPLE_HEADER_H__
2  #define __RIPPLE_HEADER_H__
3
4  #include <cmath>
5  #include <iostream>
6  #include <fstream>
7  #include <string>
8
9  #include "expression_tree.h"
10 #include "file_lib.h"
11 #include "html_lib.h"
12 #include "link_val.h"
13 #include "linked_var.h"
14 #include "xml_lib.h"
15 #include "keyboard_stream_reader.h"
16 #include "web_stream_reader.h"
17 #include "file_stream_reader.h"
18
19 using namespace ripple;
20
21 namespace ripple {
22     string input(string p){
23         string x;
24         cout << p;
25         cout.flush();
26         cin >> x;
27         return x;
28     }
29 }
30
31 string default_rpl_str_str(string xyzder){ return xyzder; }
32 int str_to_int(string xyzder) { return stoi(xyzder); }
33 float str_to_float(string xyzder) { return stof(xyzder); }
34 // Used for short-term manipulation of linked vars
35 linked_var *universal_linked_var_ptr;
36
```

```
37 #endif
```

../source–code/link_files/ripple_header.h

```
1 #ifndef __DEBUG_TOOLS_H__
2 #define __DEBUG_TOOLS_H__
3
4 #include <iostream>
5 #include <string>
6
7 using namespace std;
8
9 void d(string m);
10
11 #endif
```

../source–code/misc/debug_tools.h

```
1 #include "debug_tools.h"
2
3 void d(string m) {
4 #ifdef DEBUG
5     cout << m << endl;
6 #endif
7 }
```

../source–code/misc/debug_tools.cpp

```
1 #ifndef __RPL_ENUM_H__
2 #define __RPL_ENUM_H__
3
4 enum e_type {
5     tNOTYPE,
6     tINT,
7     tBOOL,
8     tFLOAT,
9     tSTRING,
10    tBYTE,
11    tVOID,
12    tDERIV,
13 };
14
15 enum e_symbol_type {
16    tNOSTYPE,
17    tVAR,
18    tFUNC,
19    tDSET,
20    tARR,
21    tRES
22 };
23
24 enum e_value_type {
25    LIT,
26    FUNC_CALL,
27    ARR_ACC,
28    DS_ACC,
```

```
29      IDENT,
30      EXPR
31  };
32
33  enum e_op {
34      PLUS,
35      MINUS,
36      TIMES,
37      DIV,
38      FLDIV,
39      MOD,
40      EXP,
41      bAND,
42      bOR,
43      bNOT,
44      EQ,
45      NE,
46      GT,
47      LT,
48      GE,
49      LE,
50      SIZE,
51      CAST,
52      NONE
53  };
54
55  enum e_jump {
56      tRETURN,
57      tCONTINUE,
58      tBREAK,
59      tSTOP
60  };
61
62  #endif
```

../source–code/structures/enum.h

```
1  #ifndef __UNION_H__
2  #define __UNION_H__
3
4  #include <cstring>
5  using namespace std;
6
7  union literal {
8      int int_lit;
9      double float_lit;
10     string *string_lit;
11     bool bool_lit;
12     char byte_lit;
13
14     literal() { memset(this, 0, sizeof(literal)); }
15     ~literal() {}
16 };
17
```

```
18 #endif
```

../source–code/structures/union.h

```
1  void main() {
2
3          int x;
4
5          bool[] b;
6          int[] i;
7          float[] f;
8          string[] s;
9
10         bool[b] b2;
11         int[x] i2;
12         float[f] f2;
13         string[s] s2;
14
15         bool[10] b3;
16         int[10] i3;
17         float[10] f3;
18         string[10] s3;
19
20         bool[] b4 = {1,2,3};
21         int[] i4 = {true};
22         float[] f4 = {"hello"};
23         string[] s4 = {1,2,3};
24
25         bool[] b5 = {true, true, false};
26         int[] i5 = {1,2,3};
27         float[] f5 = {1,2.0,3};
28         string[] s5 = {"hello", "world"};
29
30         bool[1] b6 = {true, true, false};
31         int[1] i6 = {1,2,3};
32         float[1] f6 = {1,2.0,3};
33         string[1] s6 = {"hello", "world"};
34
35 }
```

../source–code/tests/array_test.rpl

```
1  void main(){
2          bool b = true;
3          int x = 3;
4          float y = 2;
5          string s = "hello";
6
7          s = (string) y;
8          s = (string) b;
9          s = (string) x;
10         s = (string) x;
11         s = (string) b;
12         print(s);
```

```
13 }
```

../source–code/tests/cast_test.rpl

```
1  void main(){
2
3          print("Should print true:");
4          if(true){
5                  print("true");
6          }
7
8          print("\nShould print false");
9          if(false){
10                 print("true");
11         } else {
12                 print("false");
13         }
14
15         print("\nShould print true once and exit loop");
16         while(true){
17                 print("true");
18                 break;
19         }
20
21
22         while(false){
23                 print("This should not be printing");
24         }
25
26         print("\nShould print numbers from 0 to 9");
27         int i;
28         for(i = 0; i < 10; i = i + 1){
29                 print(i);
30         }
31
32         for(; false; ){ }
33
34 }
```

../source–code/tests/control_test.rpl

```
1  dataset foo {
2      int x,
3      int y
4  }
5
6  void main() {
7      dataset foo banana;
8      banana.x = 0;
9      int x2 = banana.x;
10
11     int[] x3 = { 1, 2, 3, 4 };
12
13     print(banana.x, banana.y, x3[2]);
```

```
}
```

../source–code/tests/dataset_test.rpl

```
void main(){
        bool b;
        int i;
        float f;
        string s;
        bool b2 = false;
    bool b3 = true;

        int i2 = 0;
        int i3 = -2;
        int i4 = 2;


    float f2 = 0;
        float f3 = -1;
        float f4 = 1;
        float f5 = 0.0;
        float f6 = -1.0;
        float f7 = 1.0;

        string s2 = "hello";
        string s3 = "";

        b = b2 and b3;
        b = b2 or b3;
        b = not b;

    i = i2 + i3;
        i = i3 - i4;
        i = i2 * i3;
        i = i3 / i4;
        f = i3 // i4;
        i = i3 % i4;
        i = i3 ^ i4;
        i = -i2;
        i = @i2;

        f = i2 + i3;
        f = i3 - i4;
        f = i2 * i3;
        f = i3 / i4;
        f = i3 // i4;
        f = i3 ^ i4;
        f = -i2;
        i = @i2;

        f = f2 + f3;
        f = f3 - f4;
        f = f2 * f3;
        f = f3 / f4;
        f = f3 // f4;
```

```
52          f = f3 ^ f4;
53          f = -f2;
54          i = @f2;
55
56          s = s2 + b2;
57          s = s2 + i2;
58          s = s2 + f2;
59          s = s2 + s2;
60          i = @s2;
61
62          int[10] a;
63          int i123 = a[0];
64
65  }
```

../source-code/tests/decl_test.rpl

```
1   int main() {
2
3           int i1 = true;
4           int i2 = false;
5           int i3 = 1.0;
6           int i4 = "hello";
7           int i5 = {1,2,3};
8
9           float f1 = true;
10          float f2 = false;
11          float f3 = "hello";
12          float f4 = {1,2,3};
13
14          bool b1 = 1;
15          bool b2 = 1.0;
16          bool b3 = "true";
17          bool b4 = {true};
18
19          string s1 = 1;
20          string s2 = 1.0;
21          string s3 = true;
22          string s4 = false;
23          string s5 = {"hello", "world"};
24
25  }
```

../source-code/tests/errors.rpl

```
1   void printline(int str){
2           print(str);
3   }
4
5   int str_to_int(string str2){
6           return 0;
7   }
8
9   void main(){
10          int s;
```

```
11          link(s <- str_to_int <- file_stream("hello.txt", 5, ",")) then
                ↪ printline;
12          while(true) {  }
13  }
```

../source–code/tests/filestream0.rpl

```
1   void printline(string str){
2           print(str);
3   }
4
5   int str_to_int(string str2){
6           return 0;
7   }
8
9   void main(){
10          int s;
11          link(s <- str_to_int <- file_stream("hello.txt", 5, ","));
12          while(true) {  }
13  }
```

../source–code/tests/filestream1.rpl

```
1   void printline(string str){
2           print(str);
3   }
4
5   int str_to_int(string str2){
6           return 0;
7   }
8
9   void main(){
10          string s;
11          link(s <- file_stream("hello.txt", 5, ","));
12          while(true) {  }
13  }
```

../source–code/tests/filestream2.rpl

```
1   void printline(string str){
2           print(str);
3   }
4
5   int str_to_int(string str2){
6           return 0;
7   }
8
9   void main(){
10          string s;
11          link(s <- file_stream("hello.txt", 5, ",")) then printline;
12          while(true) {  }
13  }
```

../source–code/tests/filestream3.rpl

```
1   final int x = 6;
```

```
2 void main(){
3         int x22 = x;
4         print(x22);
5 }
```

../source–code/tests/final_test.rpl

```
1 int hello(int x, int y) {
2     print(x, y);
3 }
4
5 void main() {
6     hello(1, 2);
7 }
```

../source–code/tests/function_test.rpl

```
1 #~ A simple program to print hello world ~#
2 # This is the main function
3 void main() {
4     int x = 4;
5     print("hello, world", 5);
6     # prints 'hello, world'
7     stop;
8 }
```

../source–code/tests/hello.rpl

```
1 void say_hello(string x) {
2     print("Hello", x);
3 }
4
5 void main() {
6     string filename = "hello.txt";
7
8     print("Hello, world");
9
10    string w;
11    link(w <- file_stream(filename, 1, "\n" )) then say_hello;
12
13    stop;
14 }
```

../source–code/tests/hello2.rpl

```
1 void main(){
2         string s = input("Enter a string");
3         print(s);
4 }
```

../source–code/tests/input_test.rpl

```
1 void print_int(int a) {
2     print(a);
3 }
4
5 void main(){
```

```
6           int x;
7           int y = 3;
8           int z = 4;
9       link(x <- (y + z) + 2) then print_int;
10          y = 5;
11          z = 5;
12  }
```

../source–code/tests/link_test.rpl

```
1   void printline(string str){
2           print(str);
3   }
4
5   int str_to_int(string str2){
6           return 0;
7   }
8
9   void main(){
10          string s;
11          link(s <- file_stream("hello.txt", 5, ",")) then printline;
12          while(true) { }
13  }
```

../source–code/tests/link_test2.rpl

```
1   void main() {
2       string text = "PLT is awesome and cool!";
3       print(contains_word(text, "PLT"));
4   }
```

../source–code/tests/std_function_test.rpl

```
1   #~
2   Converts temperatures from fahrenheit to celsius as they are typed into the
        ↪ keyboard.
3   ~#
4
5   void what_to_wear(int x) {
6       if (x < 30) {
7           print("Bundle up, it's cold outside!");
8       } else { if (x < 70) {
9            print("Maybe put on a sweater?");
10      } else {
11           print("Put on shorts and flip flops!");
12      }
13      }
14  }
15
16  void main() {
17      int deg_f;
18      link(deg_f <- str_to_int <- keyboard_stream()) then what_to_wear;
19      stop;
20  }
```

../source–code/tests/temp.rpl

```
final float FACTOR = 9//5;

void print_temp_c(int temp_f){
    float tc = (temp_f - 32) // FACTOR;
    print("Temperature in C: ", tc);
}

void main(){

    int tf;
    link(tf <- str_to_int <- keyboard_stream()) then print_temp_c;

    stop;

}
```

../source-code/tests/temp_conv.rpl

```
void test_integer_decl() {
    # Integer Declarations
    print("[INT DECLARATION TEST]");
    int var_1;
    var_1 = 4;
    int var_2 = 7;
    print("\tValue for var_1:\t", var_1);
    print("\tValue for var_2:\t", var_2);
    print("\tInteger Declarations Passed!\n");
}


void test_float_decl() {
    # Float Declarations
    print("[FLOAT DECLARATION TEST]");
    float var_3;
    var_3 = 5.0;
    float var_4 = 10.0;
    print("\tValue for var_3:\t", var_3);
    print("\tValue for var_4:\t", var_4);
    print("\tFloat Declarations Passed!\n");
}


void test_string_decl() {
    # String Declarations
    print("[STRING DECLARATION TEST]");
    string var_5;
    var_5 = "hello";
    string var_6 = "world";
    print("\tValue for var_5:\t", var_5);
    print("\tValue for var_6:\t", var_6);
    print("\tString Declarations Passed!\n");
}


void test_bool_decl() {
```

```
38      # Bool Declarations
39      print("[BOOL DECLARATION TEST]");
40      bool var_7;
41      var_7 = true;
42      bool var_8 = false;
43      print("\tValue for var_7:\t", var_7);
44      print("\tBool Declarations Passed!\n");
45  }
46
47
48  void test_int_arithmetic() {
49     # Integer Arithmetic
50     print("[INTEGER ARITHMETIC TEST]");
51
52     int result_1 = 1 + 2;
53     print("\tLiteral Addition:\t", result_1);
54
55     int result_2 = 4 - 2;
56     print("\tLiteral Substraction:\t", result_2);
57
58     int result_3 = 4 * 2;
59     print("\tLiteral Multiplication:\t", result_3);
60
61     int result_4 = 4 / 2;
62     print("\tLiteral Division:\t", result_4);
63
64     int result_5 = 2 ^ 4;
65     print("\tLiteral Exponent:\t", result_5);
66
67     int result_6 = 10 % 5;
68     print("\tLiteral Modulo:     \t", result_6);
69     print("\tLiteral Tests Passed!\n");
70
71     result_1 = result_2 + 4;  # Should be 6
72     print("\tSingle Var Addition:\t", result_1);
73
74     result_2 = result_1 - 10; # Should be -4
75     print("\tSingle Var Subtraction:\t", result_2);
76
77     result_3 = result_2 * 2;  # Should be -8
78     print("\tSingle Var Multi:\t", result_3);
79
80     result_4 = result_3 / 5;  # Should be -1
81     print("\tSingle Var Division:\t", result_4);
82
83     result_5 = result_4 ^ 2;  # Should be 1
84     print("\tSingle Var Exponent:\t", result_5);
85
86     result_6 = result_1 % 4; # Should be 2
87     print("\tSingle Var Modulo:\t", result_6);
88     print("\tSingle Var Tests Passed!\n");
89
90     result_1 = result_2 + result_3; # Should be -12
91     print("\tTwo Var Addition:\t", result_1);
```

```
92
93      result_2 = result_1 - result_3; # Should be -4
94      print("\tTwo Var Subtraction:\t", result_2);
95
96      result_3 = result_1 * result_2; # Should be 48
97      print("\tTwo Var Multiplication:\t", result_3);
98
99      result_4 = result_3 / result_1; # Should be -4
100     print("\tTwo Var Division:\t", result_4);
101
102     result_5 = result_4 ^ -result_2; # Should be 256
103     print("\tTWo Var Exponentiation:\t", result_5);
104
105     result_6 = result_5 % result_2; # Should be 0
106     print("\tTwo Var Modulo:\t\t", result_6);
107     print("\tInteger Arithmetic Passed!\n");
108 }
109
110
111 void test_float_arithmetic() {
112    # Integer Arithmetic
113     print("[FLOAT ARITHMETIC TEST]");
114
115     float result_1 = 1 + 2;
116     print("\tLiteral Addition:\t", result_1);
117
118     float result_2 = 4 - 2;
119     print("\tLiteral Substraction:\t", result_2);
120
121     float result_3 = 4 * 2;
122     print("\tLiteral Multiplication:\t", result_3);
123
124     float result_4 = 4 / 2;
125     print("\tLiteral Division:\t", result_4);
126
127     float result_6 = 10 // 2;
128     print("\tLiteral Float Division:\t", result_6);
129
130     float result_5 = 2 ^ 4;
131     print("\tLiteral Exponent:\t", result_5);
132     print("\tLiteral Tests Passed!\n");
133
134     result_1 = result_2 + 4;  # Should be 6
135     print("\tSingle Var Addition:\t", result_1);
136
137     result_2 = result_1 - 10; # Should be -4
138     print("\tSingle Var Subtraction:\t", result_2);
139
140     result_3 = result_2 * 2;  # Should be -8
141     print("\tSingle Var Multi:\t", result_3);
142
143     result_4 = result_3 / 5;  # Should be -1
144     print("\tSingle Var Division:\t", result_4);
145
```

```
146     result_6 = result_3 // -2;
147     print("\tSingle Var Float Div:\t", result_6);
148 
149     result_5 = result_4 ^ 2;  # Should be 1
150     print("\tSingle Var Exponent:\t", result_5);
151     print("\tSingle Var Tests Passed!\n");
152 
153     result_1 = result_2 + result_3; # Should be -12
154     print("\tDouble Var Addition:\t", result_1);
155 
156     result_2 = result_1 - result_3; # Should be -4
157     print("\tDouble Var Subtraction:\t", result_2);
158 
159     result_3 = result_1 * result_2; # Should be 48
160     print("\tTwo Var Multiplication:\t", result_3);
161 
162     result_4 = result_3 / result_1; # Should be -4
163     print("\tTwo Var Division:\t", result_4);
164 
165     result_6 = result_1 // result_2;
166     print("\tTwo Var Float Division:\t", result_6);
167 
168     result_5 = result_4 ^ -result_2; # Should be 256
169     print("\tTwo Var Exponent:\t", result_5);
170     print("\tFloat Arithmetic Passed!\n");
171 }
172 
173 
174 void test_string_concatenation() {
175    # String Concatenation
176     print("[STRING CONCATENATION TEST]");
177     string test_1 = "hello";
178     string test_2 = "world";
179 
180     string result_1 = test_1 + test_2;
181     print("\tValue of Result:\t", result_1);
182 
183     result_1 = test_1 + " " + test_2;
184     print("\tValue of Result:\t", result_1);
185 
186     result_1 = test_1 + " " + test_2 + " and Professor!";
187     print("\tValue of Result:\t", result_1);
188 
189     print("\tString Concatenation Test Passed!\n");
190 }
191 
192 void test_negation() {
193      print("[NEGATION TEST]");
194      int test_1 = -4;
195      print("\tLiteral Integer Negation:\t", test_1);
196 
197      float test_3 = -5;
198      print("\tLiteral Float Negation:\t\t", test_3);
199      print("\tLitearl Negation Tests Passed!\n");
```

```
200
201
202     int test_2 = - test_1;
203     print("\tVariable Integer Negation:\t", test_2);
204
205     float test_4 = - test_3;
206     print("\tVariable Float Negation:\t", test_4);
207     print("\tVariable Negation Tests Passed!\n");
208 }
209
210 void test_link() {
211     print("[LINK TEST]");
212
213 }
214
215 void test_arrays() {
216     print("[ARRAY TEST]");
217     int[5] test_array;
218     test_array[0] = 1;
219     test_array[1] = 2;
220     test_array[2] = 3;
221     test_array[3] = 4;
222     test_array[4] = 5;
223     print("Int Array Initialization Test Passed!");
224
225 }
226
227 void test_dataset() {
228     print("[DATASET TEST]");
229
230 }
231
232 void test_functions() {
233     print("[FUNCTION TEST]");
234 }
235
236 int main(){
237
238     #~
239     Test
240     ~#
241
242     print("[MAIN TEST]\tStart\n");
243
244     test_integer_decl();
245     test_float_decl();
246     test_string_decl();
247     test_bool_decl();
248     test_int_arithmetic();
249     test_float_arithmetic();
250     test_string_concatenation();
251     test_negation();
252     test_link();
253     test_array();
```

```
254     test_dataset();
255     test_functions();
256
257     print("[MAIN TESTS]\tStop");
258
259 }
```

../source–code/tests/test.rpl