Ripple

make waves

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 – 02 – 25

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With the proliferation of devices that are constantly connected today, information becomes outdated faster than ever.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.

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.

Hello, world!

This simple program begins by printing "Hello world!" Then it assigns `w` a filestream, which determines the file via command line arguments. It reads a file line by line and every time the program reads a new line, the value of `w` changes to the last line read, and the print statement within the link statement executes. Essentially, the program says hello to the world, and to every line in the file.

```
1  #include <FileStreamReader>;
2
3  func main(list<String> args) {
4
5      String filename = args[1];
6
7      String h = "Hello ";
8      String w = "world!";
9
10     print(h,w);
11
12     link(w -> FileStreamReader(filename)) {
13         print(h,w);
14     }
15 }
```

hello.rpl

What is Ripple?

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.

Ripple is connected

From our previous example, the values of `y` and `z` do not need to live within tthe 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 `StreamReader`s for common input types, and an interface that allows developers to easily build their own `StreamReader`s. Developers will thus have the flexibility to handle a variety of input formats, turning data of any kind into a Ripple-compatible stream.

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.

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 `StreamReader` can be defined. This generalization significantly broadens the class of problems that can be solved by a reactive program.

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.

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 all the standard data primitives from C, with the addition of `byte` and `string` types.

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  ...
2
3  int x = 5;
4  int y, z, q, v;
5
6  link (x) {
7      link (y -> x + 2) {
8          link (z -> y + 10) {}
9          link (v -> y + 42) {}
10     }
11     link (q -> x - 4) {}
12 }
13 ...
```

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`. On line 6, we start the linking structure. In order to create a link, the compiler must determine which variable will be the initial or **root** node for the link. The compiler will interpret line 6 to mean that the variable `x` is being linked to one or more variables, and to initialize `x` as

the root of the link tree. Lines 7 through 11 introduce the nested structures of **link blocks**. As we traverse deeper into the nesting of the link block, we can create a **link tree** that shows how the linking structure is suppose to work. The **link tree** for this code is shown in Figure 1.

As we traverse down the link tree, we see `y` links to `x`. Thus, we create a child node of `x` on the link tree and name it `y`. Moving deeper into the nesting, we see that there are two other links (`z` and `v`) that link to `y`. These two variables will become children nodes of `y`. Finally, we leave the inner link block and discover the link to `x` by `q`; `q` now becomes another child node for `x`.

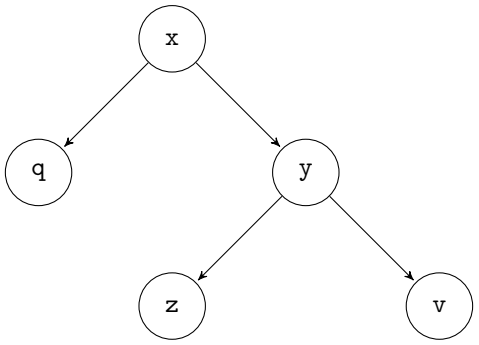

Figure 1: The link tree for *sample.rpl*

All links are hierarchical and uni-directional in nature. For example, if the value of `y` from our previous example were to change, only the children of `y` would be affected (`z` and `v`). Siblings, parents, and ancestors of the updated node will not be notified of the update.

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

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. Links can be made between two lists of equal length through a one-to-one mapping of indices within the lists.

`StreamReader` is Ripple's construct for allowing users to define data streams, and serves to specify how data will enter the program. A `StreamReader` consists of two methods: `OpenStream()`, which facilitates any initial setup of the stream before it can be read (e.g. establishing an HTTP connection), and `ReadStream()`, which returns the next piece of data from available from the stream. Common `StreamReader`s, such as for files, keyboard input, and XML documents, are provided in Ripple's standard library, and others can be defined by the programmer, maximizing flexibility.

Functions are defined using the keyword `func`. Every function can take a variable number of arguments, and uses the keyword `returns` with a data type to represent what the function returns. If there is no returns statement present, then the function returns nothing.