Ripple

make waves

Reference Manual

Team 5

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 – 03 – 25

Contents

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.

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.

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 "`/`".

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, `byte`s, `string`s, and `bool`s; all are specified inline)
4. Operators (`+`, `-`, etc.)
5. Miscellaneous separators (`()`, `;`, etc.)

2.1.1 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.

2.1.2 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	import	final	continue	break
void	bool	byte	int	float
string	dataset	true	false	not
or	and			

2.1.3 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. An `int` is a sequence of digits (0 - 9) without a decimal point (".")
2. `A` float is a sequence of digits containing a decimal point.
3. A `byte` is the string "`0b`" followed by a string of length 8 consisting of `0`s and `1`s. A `byte` can be represented as a decimal value from 0 to 255 as well.
4. A `string` is any sequence of characters enclosed in single or double quotation marks.

2.1.4 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`
	`<`
	`<=`
	`==`
	`!=`
	`=>`
	`>`
	`.`

2.1.5 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

2.2 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.

2.3 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 Types

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

3.1 `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.

3.2 `bool`

`bools` 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 `byte`

`byte`s are included in Ripple so that programmers can manipulate a single byte at a time, allowing finer control over data received from external streams. A `byte` can be assigned an integral value between 0 and 255, inclusive, or a binary value in the form `0bXXXXXXXX`, where `X` can be either 0 or 1. However, unless the type is explicitly specified, an integral constant given as input to a program will always be interpreted as an `int`.

3.4 `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 $[-2^{31},\ 2^{31}-1]$.

3.5 `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 $[\pm 2.23 \times 10^{-308},\ \pm 1.80 \times 10^{308}]$.

3.6 `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 single or double quotation marks: either `'hello'` or `"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.

3.7 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.

3.7.1 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:

$$\begin{aligned} \textit{array\_creation} &\rightarrow \{\ \textit{args}\ \} \\ \textit{args} &\rightarrow \texttt{(}\ \textit{expression arg\_list}\ \texttt{)} \\ &\quad \mid \texttt{(}\ \texttt{)} \end{aligned}$$

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).

3.7.2 `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

$$\begin{aligned} \textit{dataset} &\rightarrow \texttt{dataset}\ ID\ \{\ \textit{declaration\_list}\ \}; \\ \textit{declaration\_list} &\rightarrow \textit{declaration};\ \textit{declaration\_list} \\ &\mid \epsilon \\ \textit{declaration} &\rightarrow TYPE_{OPT}\ ID \\ &\mid TYPE\ \texttt{[}\ \textit{expression}_{OPT}\ \texttt{]}\ ID \\ &\mid ID\ \texttt{[}\ \textit{expression}\ \texttt{]} \end{aligned}$$

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.

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.

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.

4.2 And Expressions

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

`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.

4.3 Equality Expressions

$$
\begin{aligned}
\mathit{eq\_expression} \rightarrow\ & \mathit{eq\_expression}\ \texttt{==}\ \mathit{rel\_expression} \\
\mid\ & \mathit{eq\_expression}\ \texttt{!=}\ \mathit{rel\_expression} \\
\mid\ & \mathit{rel\_expression}
\end{aligned}
$$

`==` 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.

4.4 Relational Expressions

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

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

4.5 Plus and Minus Expressions

$$
\begin{aligned}
\mathit{plus\_expression} \rightarrow\ & \mathit{plus\_expression}\ \texttt{+}\ \mathit{mult\_expression} \\
\mid\ & \mathit{plus\_expression}\ \texttt{-}\ \mathit{mult\_expression} \\
\mid\ & \mathit{mult\_expression}
\end{aligned}
$$

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’’`.

With arrays, the `+` operator can also be used for concatenation. When adding two arrays containing the same element type, it will append the second array to the end of the first. When adding an array and a variable if the variable is the same type as that of the array, `+` adds the value to the beginning of the array if the value was first in the equation, and to the end if it was second.

4.6 Multiplicative Expressions

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

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.

4.7 Unary Expressions

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

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).

4.8 Exponential Expressions

$$\mathit{exp\_expression} \rightarrow \mathit{exp\_expression} \wedge \mathit{exp\_expression}$$

$\mid$ *var* $\mid$ `(` *expression* `)`

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.

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:

statement $\rightarrow$ *conditional_statement*
$\mid$ *loop_statement*
$\mid$ *link_statement*
$\mid$ *declarative_statement*
$\mid$ *jump_statement*

5.1 Conditional Statements

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

conditional_statement $\rightarrow$ `if (` *expression* `)` *statement_block 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.

else_statement $\rightarrow$ `else` *statement_block*
$\mid$ ϵ

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.

loop_statement $\rightarrow$ `while (` *expression* `)` *statement_block*
$\mid$ `for (` *declarative_statement* $expression_{OPT}$; $expression_{OPT}$ `)` *statement_block*
$\mid$ `for ( ;` $expression_{OPT}$; $expression_{OPT}$ `)` *statement_block*

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.

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:

$$\begin{aligned} \textit{link\_statement} &\rightarrow \texttt{link (} \textit{declaration link\_stream} \texttt{) } function\_call_{OPT}; \\ \textit{link\_stream} &\rightarrow \texttt{<-}\ \textit{stream\_reader} \\ &\quad \mid \texttt{<-}\ \textit{expression} \end{aligned}$$

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 function *function_call*, if provided, will be called every time this update occurs. Only a *link function* (i.e., a function declared using the `link` keyword) can be called in this fashion. These functions are unique in that all parameters passed to them are passed by reference.

Furthermore, 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.

5.3.1 `StreamReaders`

An important aspect of Ripple is the ability to treat external variables as local to the program. To this end Ripple provides the `StreamReader` construct which is used in conjunction with `link` statements to “link” a variable to a changing external stream of data. The `link` statement creates a dependency between the `StreamReader` and the variable; the linked variable will always have the latest value provided by the stream. Hence the local variable represents a link to external data.
External data cannot always be easily represented using the primitive types provided by Ripple. Programmers can therefore specify what kind of data they want out of a `StreamReader` through use of a `dataset`. `StreamReader`s are identified by an ID and should only be used within link statements.

5.4 Declarative Statements

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

$$\begin{aligned} \textit{declarative\_statement} &\rightarrow \textit{declaration} \texttt{ = } \textit{expression}; \\ &\quad \mid \textit{expression}; \\ &\quad \mid \textit{declaration}\texttt{;} \\ &\quad \mid \epsilon \\ \textit{declaration} &\rightarrow TYPE_{OPT}\ ID \\ &\quad \mid TYPE\ ID\ ID \\ &\quad \mid TYPE\ [\ \textit{expression}_{OPT}\ ]\ ID \\ &\quad \mid ID\ [\ \textit{expression}\ ] \end{aligned}$$

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:

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

5.5 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:

$$\begin{aligned} jump\_statement \rightarrow\ & \texttt{return}\ expression_{OPT};\\ |\ & \texttt{continue};\\ |\ & \texttt{break}; \end{aligned}$$

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.

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 four top level declarations based on the following grammar

$$\begin{aligned} program\_declarations \rightarrow\ & program\_declarations\ program\_declaration\\ |\ & \epsilon\\ program\_declaration \rightarrow\ & import\_statement\\ |\ & final\_declaration\\ |\ & dataset\\ |\ & function \end{aligned}$$

6.1 `import` Statements

$$import\_statement \rightarrow \texttt{import}\ LITERAL;$$

`import` statements serve to bring in external code into a Ripple program. These statements can copy files from Ripple's standard library, as well as from the local machine. `import` statements are specified by using the `import` keyword followed by a string literal designating the file to be imported. Import statements must end with a semicolon.

6.2 `dataset` declaration

dataset $\rightarrow$ `dataset` *ID* { *declaration_list* }`;`

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

6.3 `final` Declaration

final_declaration $\rightarrow$ `final` *declaration_statement*

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.

6.4 `function` Declarations

function $\rightarrow$ $link_{OPT}$ *declaration declaration_args* { *statement_list* }

Function declarations are used to name and specify the arguments, return type, and code to be executed of a function. Additionally, as specified by the grammar, functions provided to `link` statements must be preceded with the `link` keyword.

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.

8 Functions

Functions are named sets of instructions; as such, they encourage modularity and easy reuse. A function in Ripple takes *expressions* 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.

function $\rightarrow$`link`$_{OPT}$ *declaration declaration_args* { *statement_list* }

Within `link` statements, the developer has the ability to call functions that interact with the linked variables and any other variables that are within the link scope. It is important that any functions called within these linked blocks are linked functions themselves (these functions are said to be "link-safe"). Functions that do not have the `link` keyword in their declaration can cause undefined behavior with the `link` block is run. In order to make a function link-safe, the programmer must use include the keyword `link` within the declaration of the function, before the type declaration.

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

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

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

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_declarations*. To reiterate, terms in *italics* are syntactic constructs, while terms in `monospaced font` are literal strings. Terms in all-caps italics are tokens.

$$
\begin{aligned}
\mathit{program\_declarations} &\rightarrow \mathit{program\_declarations}\ program_{d}eclaration \\
&\mid \epsilon \\
program_{d}eclaration &\rightarrow \mathit{import\_statement} \\
&\mid final\_declaration \\
&\mid \mathit{dataset} \\
&\mid function \\
\mathit{import\_statement} &\rightarrow \texttt{import}\ LITERAL\texttt{;} \\
\mathit{dataset} &\rightarrow \texttt{dataset}\ ID\ \{\ \mathit{declaration\_list}\ \}\texttt{;} \\
final\_declaration &\rightarrow \texttt{final}\ \mathit{declaration\_statement} \\
function &\rightarrow \mathit{link}_{OPT}\ \mathit{declaration}\ \mathit{declaration\_args}\ \{\ \mathit{statement\_list}\ \} \\
\mathit{declaration\_list} &\rightarrow \mathit{declaration}\texttt{;}\ \mathit{declaration\_list} \\
&\mid \epsilon \\
\mathit{declaration} &\rightarrow TYPE_{OPT}\ ID \\
&\mid TYPE\ [\ \mathit{expression}_{OPT}\ ]\ ID \\
&\mid ID\ [\ \mathit{expression}\ ] \\
\mathit{declaration\_args} &\rightarrow \texttt{(}\ \mathit{declaration}\ \mathit{declaration\_arg\_list}\ \texttt{)} \\
&\mid \texttt{(}\ \texttt{)} \\
\mathit{declaration\_arg\_list} &\rightarrow ,\ \mathit{declaration}\ \mathit{declaration\_arg\_list} \\
&\mid \epsilon \\
\mathit{statement\_list} &\rightarrow \mathit{statement\_list}\ \mathit{statement\_block} \\
&\mid \epsilon
\end{aligned}
$$

9.1 Statements

$$
\begin{array}{rl}
\textit{statement} & \rightarrow \textit{conditional\_statement} \\
 & \mid \textit{loop\_statement} \\
 & \mid \textit{link\_statement} \\
 & \mid \textit{declarative\_statement} \\
 & \mid \textit{jump\_statement} \\
\textit{conditional\_statement} & \rightarrow \texttt{if (}\ \textit{expression}\ \texttt{)}\ \textit{statement\_block}\ \textit{else\_statement} \\
\textit{else\_statement} & \rightarrow \texttt{else}\ \textit{statement\_block} \\
 & \mid \epsilon \\
\textit{loop\_statement} & \rightarrow \texttt{while (}\ \textit{expression}\ \texttt{)}\ \textit{statement\_block} \\
 & \mid \texttt{for (}\ \textit{declarative\_statement}\texttt{;}\ \textit{expression}_{OPT}\texttt{;}\ \textit{expression}_{OPT}\ \texttt{)}\ \textit{statement\_block} \\
 & \mid \texttt{for (;}\ \textit{expression}_{OPT}\texttt{;}\ \textit{expression}_{OPT}\ \texttt{)}\ \textit{statement\_block} \\
\textit{link\_statement} & \rightarrow \texttt{link (}\ \textit{declaration}\ \textit{link\_stream}\ \texttt{)}\ \textit{function\_call}_{OPT}\texttt{;} \\
\textit{link\_stream} & \rightarrow \texttt{<-}\ \textit{stream\_reader} \\
 & \mid \texttt{<-}\ \textit{expression} \\
\textit{stream\_reader} & \rightarrow READER\_NAME\ \texttt{(}\ \textit{function\_call\_args}\ \texttt{)} \\
\textit{declarative\_statement} & \rightarrow \textit{declaration}\ \texttt{=}\ \textit{expression}\texttt{;} \\
 & \mid \textit{expression}\texttt{;} \\
 & \mid \textit{declaration}\texttt{;} \\
\textit{statement\_block} & \rightarrow \ \texttt{statement\_list} \\
 & \mid \textit{statement}\texttt{;} \\
\textit{jump\_statement} & \rightarrow \texttt{return}\ \textit{expression}\texttt{;} \\
 & \mid \texttt{continue;} \\
 & \mid \texttt{break;}
\end{array}
$$

9.2 Expressions

$$
\begin{array}{rl}
\textit{expression} & \rightarrow \textit{expression}\ \texttt{or}\ \textit{and\_expression} \\
 & \mid \textit{and\_expression} \\
\textit{and\_expression} & \rightarrow \textit{and\_expression}\ \texttt{and}\ \textit{eq\_expression} \\
 & \mid \textit{eq\_expression} \\
\textit{eq\_expression} & \rightarrow \textit{eq\_expression}\ \texttt{==}\ \textit{rel\_expression} \\
 & \mid \textit{eq\_expression}\ \texttt{!=}\ \textit{rel\_expression} \\
 & \mid \textit{rel\_expression} \\
\textit{rel\_expression} & \rightarrow \textit{rel\_expression}\ \texttt{>=}\ \textit{plus\_expression} \\
 & \mid \textit{rel\_expression}\ \texttt{<=}\ \textit{plus\_expression} \\
 & \mid \textit{rel\_expression}\ \texttt{>}\ \textit{plus\_expression} \\
 & \mid \textit{rel\_expression}\ \texttt{<}\ \textit{plus\_expression} \\
 & \mid \textit{plus\_expression}
\end{array}
$$

$$\begin{aligned}
plus\_expression &\rightarrow plus\_expression \; \texttt{+} \; mult\_expression \\
&\mid plus\_expression \; \texttt{-} \; mult\_expression \\
&\mid mult\_expression \\
mult\_expression &\rightarrow mult\_expression \; \texttt{*} \; unary\_expression \\
&\mid mult\_expression \; \texttt{/} \; unary\_expression \\
&\mid mult\_expression \; \texttt{//} \; unary\_expression \\
&\mid mult\_expression \; \texttt{\%} \; unary\_expression \\
&\mid unary\_expression \\
unary\_expression &\rightarrow \texttt{not} \; unary\_expression \\
&\mid \texttt{-} \; unary\_expression \\
&\mid \texttt{@} \; unary\_expression \\
&\mid \texttt{(} \; TYPE \; \texttt{)} unary_e xpression \\
&\mid exp\_expression \\
exp\_expression &\rightarrow exp\_expression \wedge exp\_expression \\
&\mid \texttt{(} \; expression \; \texttt{)} \\
&\mid var
\end{aligned}$$

9.3 Variables

$$\begin{aligned}
var &\rightarrow \texttt{true} \\
&\mid \texttt{false} \\
&\mid constant \\
&\mid dataset\_access \\
&\mid function\_call \\
&\mid array\_access \\
&\mid array\_creation \\
&\mid ID \\
constant &\rightarrow NUM \\
&\mid LITERAL \\
dataset\_access &\rightarrow ID \texttt{.} dataset\_access\_list \\
dataset\_access\_list &\rightarrow ID \texttt{.} dataset\_access\_list \\
&\mid ID \\
function\_call &\rightarrow ID \; args \\
array\_access &\rightarrow ID \; \texttt{[} \; expression \; \texttt{]} \\
array\_literal &\rightarrow \{ \; args \; \} \\
args &\rightarrow \texttt{(} \; expression \; arg\_list \; \texttt{)} \\
&\mid \texttt{(} \; \texttt{)} \\
arg\_list &\rightarrow , \; expression \; arg\_list \\
&\mid \epsilon
\end{aligned}$$