# Rippl

Recursively Inferred Pure functional

Programming Language

**Reference Manual**

*Riddler:* Da Hua Chen

*Gallbladder:* Hollis Lehv

*Language Yoda:* Amanda Liu

*Prime Minister:* Hans Montero

{dc2802, hml2138, al3623, hjm2133}@columbia.edu

# Contents

# 1 Overview

Rippl (Recursively Inferred Pure functional Pythonic Language) is a functional language that leverages the safety and elegance of pure languages like Haskell with the intuitive syntax of Python. With list comprehensions, lazily evaluated infinite lists, a strong static type system implementing Hindley-Milner style inference, higher-order functions, and simple syntax, Rippl sets out to provide a clean and intuitive functional programming experience like no other language can.

# 2 Type System

Rippl has primitive types, higher-order types, and arrow types.

## 2.1 Primitive Types

The primitive types in Rippl are `int`, `float`, `bool`, and `char`.

### 2.1.1 Integers

`int` is the signed integer type. An integer literal may be preceded by a unary minus (-) to denote a negative integer. Unary plus (+) is not supported. Integers are written in base 10 (decimal) and cannot be written in other bases (e.g. binary, octal, hexadecimal).

### 2.1.2 Floating-point Numbers

`float` is the floating-point numerical type. A float literal may be preceded by a unary minus (-). Unary plus (+) is not supported. Float literals must be written with a decimal point and at least one digit to the left and right of the decimal point (e.g. `-4.2`, `10.0`). This format was chosen to avoid ambiguity with integer literals when used with the list range operator (. . . ). Numbers not written in this format, e.g. `6.`, `.53`, and `-.0`, are not valid floats. Floating-point numbers written in scientific notation, e.g. `1.37e15`, are also not supported.

### 2.1.3 Booleans

`bool` is the Boolean type. There are two Boolean literals, `true` and `false`.

### 2.1.4 Characters

`char` is the character type. A character literal is written as a single character in single quotes e.g. `'A'`, `'!'`, `'+'`. Special characters, such as the newline character '`\n`', are escaped with a backslash (\).

### 2.1.5 Summary of Primitive Types

| *Type* | *Size* | *Examples* |
|---|---|---|
| `int` | 4 bytes | `0, 37, -2, 2019` |
| `float` | 8 bytes | `0.0, -6.9, 3.1415926` |
| `bool` | 1 byte | `true, false` |
| `char` | 1 byte | `'#', 'x', 'D', '\n'` |

## 2.2 Higher-Order Types

Higher-order types are represented as type constructors that are parametrically polymorphic in terms of type variables. The constructors for these types can be viewed as functions that take in a proper type as an argument and return a new type abstracted over the argument. The higher-order types in Rippl are lists, tuples, and the `maybe` sum type, which comprises `none` and the `just` constructor.

### 2.2.1 Lists

Lists are a first-order type and all elements of a list must have the same type. This means that a function that acts on lists need not concern itself with the type of the elements of the list. Such a function is said to be parameterized by the type of the elements in the list. All list operators in Rippl (see section 3.5.5) act on lists of arbitrary type.

Strings are handled and represented internally as lists of `char`, but string literals may be written in the usual sugared representation with double quotes (e.g. `"Hello, World!"`).

Lists as well as list types are delimited by brackets, and list elements are separated by commas (e.g. `[0,3,1]`, which has type `[int]`, `['R','i','p','p','l']` ⇔ `"Rippl"`, which has type `[char]`).

### 2.2.2 Tuples

Tuples are also a first-order type, parametrically polymorphic in the types of its elements. A tuple must contain exactly two elements, but each of them can be of any type. Tuples as well as their types are delimited by parentheses, and elements are separated by commas (e.g. `("PLT",4118)`, which has type `([char],int)`).

### 2.2.3 `maybe` Type

The `maybe` type is a tool to handle errors without side effects by representing an optional value. In the case that a computation cannot return a valid value, it may return the `none` constructor of the `maybe` type. Otherwise, it can return a proper value wrapped in the `just` constructor. Because `just` can wrap any type, `maybe` is polymorphic.

## 2.3 Arrow Types

An arrow type is a sequence of types separated by an arrow (->) and represents the type of a function or operator.

```
sum_tup tup = (first tup) + (sec tup)
# sum_tup :: (int, int) -> int

apply_sec tup f = f (sec tup)
# apply_sec :: (a, b) -> b -> c -> c
```

The definition of the `sum_tup` function performs a established computation on its argument, which has a value. On the other hand, the `apply_sec` function takes in an arrow type argument, which means this argument can be applied to its other argument.

## 2.4 Type Annotations and Inference

Rippl supports type inference through an implementation of the Hindley-Milner type inference system over the core syntactic language constructs of lambda abstractions, let-bindings, application, and variable names, as well as the additional language construct of if-then-else expressions.

By performing type inference in an environment where literals and language native operators (see section 3.5) are bootstrapped with a particular type, types of expressions can be inferred and checked in a complete and decidable manner. These type signatures may be concrete types like the ones below.

```
sum_ints :: int -> int -> int

count_chars :: [char] -> int

is_mutually_prime :: int -> int -> bool
```

However, these inferred types may also be parametrically polymorphic. These polymorphic types are represented as type variables like the ones below.

```
identity :: a -> a

len :: [a] -> int

empty_list :: [a]

list_map :: a -> b -> [a] -> [b]
```

Rippl also allows programmers to provide their own type annotations for function definitions. This is done using the Haskell-like syntax shown in the previous examples, which involves specifying the variable name followed by its full curried type signature, separated by a double colon "`::`". Type annotations are optional, but when they are provided, type-checking is performed to make sure the user-annotated types are at most as general as the inferred type. In other words, the type annotation must be a subtype of the inferred type.

The following function definition is an example of how an inferred type and annotated type can type-check directly if they are equal.

```
succ :: int -> int
succ n = n + 1        # inferred type is int -> int
```

The following type annotation yields a typing error, since the inferred type and annotated type contradict each other. This is due to the built-in type of the integer addition and division operators and the fact that Rippl doesn't perform any implicit type promotions.

```
avg :: int -> int -> float
avg x y = (x + y) / 2   # inferred type is int -> int -> int
```

The following function definition and annotation type-check correctly because the inferred type, which is parametrically polymorphic with type variable `a`, can be properly concretized by substituting `a` with the concrete type `[int]` to match the user-annotated type.

```
nest_int_list :: [int] -> [[int]]
nest_int_list l = l cons []     # inferred type is a -> [a]
```

The following function definition is an example of a type annotation that doesn't type-check with the inferred type. The inferred type enforces that the return type of `identity` be the same as its argument, so it's polymorphic in a single type variable `a`. This is a stronger type restriction than the annotated type which states that the function is polymorphic and may return a type different than that of its argument. Since the annotated type must be a subtype of the inferred type, this yields a type error.

```
identity :: a -> b
identity x = x          # inferred type is a -> a
```

# 3 Syntax and Grammar

## 3.1 Syntax

The following grammar represents a high-level overview of Rippl syntax.

| | |
|---|---|
| $\mu$ ::= | *Entrypoint* |
| \| main $e = e$ | main method |
| | |
| $e$ ::= | *Expressions* |
| \| $c$ | literals |
| \| $x$ | variables |
| \| $e\ e$ | application |
| \| fun $x$ -> $e$ | lambda abstraction |
| \| let $x = e$ in $e$ | let binding |
| \| if $e$ then $e$ else $e$ | if-then-else |
| \| $\gamma$ | list comprehension |
| | |
| $\gamma$ ::= | *List Comprehensions* |
| \| [$e$...] | infinite list |
| \| [$e$...$e$] | ranged list |
| \| [$e$ \| $x$ over $\gamma$,] | parametric list |
| \| [$e$ \| $x$ over $\gamma$,$e$] | qualified list |
| | |
| $\sigma$ ::= | *Primitive Types* |
| \| int | integer |
| \| float | floating point number |
| \| bool | boolean |
| \| char | character |
| | |
| $\delta$ ::= | *Higher-Order Type Constructors* |
| \| [] | list |
| \| () | tuple |
| \| none \| just | maybe |
| | |
| $\tau$ ::= | *Types* |
| \| $\sigma$ | proper type |
| \| $\delta\ \tau$ | higher-order type |
| \| $\tau$->$\tau$ | arrow type |
| | |
| $\theta$ ::= | *Type Annotations* |
| \| $x$ :: $\tau$ | type annotations |

## 3.2 Comments

Comments in Rippl draw from Python and Haskell. Single line comments are marked by a pound sign, while multiline comments are delimited by curly braces and dashes. Both forms of commenting support nested comments.

```
{- The following code will first declare a variable with value 2
   After that, it will be raised to the power of 5 -}

let x = 2 in
x ^ 5 # 32... what a #cool operation!

{- All programmers eventually fall victim to writing
silly  {- superfluous -} comments, so it is important to
learn how to write {- concise -} documentation! -}
```

## 3.3 Keywords

There are several keywords that are reserved in Rippl and thus not available for use as identifiers.

```
{- Data Types -}
# int, float, bool, char, maybe, just, none

{- Boolean Logic -}
# and, or, not, true, false

{- Program Structure -}
# over, let, in, fun, if, then, else, main

{- Higher-Order Type Operators -}
# head, tail, cons, cat, len, first, sec
```

## 3.4 Identifiers

Identifiers can be any sequence of characters that start with either a letter or an underscore followed by any combination of letters, numbers, and underscores.

```
my_identifier123             # Valid!
_my_other_indentifier456     # Also valid!
1_crazy_name                 # What the heck? NO!
```

## 3.5 Operators

The following subsections detail the various operators associated with each type in Rippl. It is worth noting that arithmetic expressions are evaluated using a mathematical order of precedence with left associativity. That is, expressions are evaluated using the following rules in order of decreasing precedence: parentheses, exponents, unary operators, multiplication/division/modulus, addition/subtraction.

### 3.5.1 Integer Operators

| *Operator* | *Type* | *Function* |
|---|---|---|
| + | $int \rightarrow int \rightarrow int$ | addition |
| - | $int \rightarrow int \rightarrow int$ | subtraction |
| * | $int \rightarrow int \rightarrow int$ | multiplication |
| / | $int \rightarrow int \rightarrow int$ | division |
| % | $int \rightarrow int \rightarrow int$ | modulus |
| ˆ | $int \rightarrow int \rightarrow int$ | power |
| > | $int \rightarrow int \rightarrow int$ | greater than |
| >= | $int \rightarrow int \rightarrow int$ | greater than or equal |
| < | $int \rightarrow int \rightarrow int$ | less than |
| <= | $int \rightarrow int \rightarrow int$ | less than or equal |
| == | $int \rightarrow int \rightarrow int$ | equal |
| != | $int \rightarrow int \rightarrow int$ | not equal |

### 3.5.2 Floating-point Operators

Note that float operators differ from integer operators in that they require an additional '.' to aid in type inference.

| *Operator* | *Type* | *Function* |
|---|---|---|
| +. | $float \rightarrow float \rightarrow float$ | addition |
| -. | $float \rightarrow float \rightarrow float$ | subtraction |
| *. | $float \rightarrow float \rightarrow float$ | multiplication |
| /. | $float \rightarrow float \rightarrow float$ | division |
| ˆ. | $float \rightarrow float \rightarrow float$ | power |
| >. | $float \rightarrow float \rightarrow float$ | greater than |
| >=. | $float \rightarrow float \rightarrow float$ | greater than or equal |
| <. | $float \rightarrow float \rightarrow float$ | less than |
| <=. | $float \rightarrow float \rightarrow float$ | less than or equal |
| ==. | $float \rightarrow float \rightarrow float$ | equal |
| !=. | $float \rightarrow float \rightarrow float$ | not equal |

### 3.5.3 Boolean Operators

| *Operator* | *Type* | *Function* |
|---|---|---|
| `and` | $bool \rightarrow bool \rightarrow bool$ | and |
| `or` | $bool \rightarrow bool \rightarrow bool$ | or |
| `not` | $bool \rightarrow bool$ | negation |

### 3.5.4 Character Operators

| *Operator* | *Type* | *Function* |
|---|---|---|
| `==` | $char \rightarrow char \rightarrow bool$ | equal |

### 3.5.5 List Operators

| *Operator* | *Type* | *Function* |
|---|---|---|
| `cons` | $a \rightarrow [a] \rightarrow [a]$ | construct |
| `head` | $[a] \rightarrow a$ | head |
| `tail` | $[a] \rightarrow [a]$ | tail |
| `len` | $[a] \rightarrow int$ | length |
| `cat` | $[a] \rightarrow [a] \rightarrow [a]$ | concatenate |

### 3.5.6 Tuple Operators

| *Operator* | *Type* | *Function* |
|---|---|---|
| `first` | $(a, b) \rightarrow a$ | first element of tuple |
| `sec` | $(a, b) \rightarrow b$ | second element of tuple |

### 3.5.7 Maybe Operators

| *Operator* | *Type* | *Function* |
|---|---|---|
| `is_nothing` | $maybe\ a \rightarrow bool$ | return `true` if none |
| `from_just` | $maybe\ a \rightarrow a$ | extract element |

## 3.6 Let Bindings

Given the functional nature of Rippl, there are no imperative assignment statements. Instead, Rippl consists mainly of nested expressions strung together using the let-in construct. This takes the form of `let identifier = expr1 in expr2`, where `identifier` serves as a name binded to the value of `expr1` to be used in `expr2`. The value of a let binding chain is the value of the last expression in the chain.

```
let x = 1 in
let y = 2 in
let y = 3 in
x + y + z # 6
```

## 3.7 If-then-else Expressions

Like most programming languages, Rippl supports basic control flow through if-then-else expressions in the following form: `if bool then expr else expr`.

```
let num = 5 in
let my_string = if num == 5 then "five"
        else "not five" # "five"
```

## 3.8 Lambda Abstractions

One cornerstone of Rippl is to permit computations that require higher-order functions, which is made possible by lambda abstractions. These anonymous functions are written using the `fun` keyword and are arrow type expressions.

```
fun x -> x + 1 # int -> int
(fun x -> x + 1) 9 # 10 - application with an anonymous function
```

We can also bind these lambda abstractions to identifiers using a `let` expression.

```
let add_one = fun x -> x + 1 in
add_one 9 # 10
```

Hope you have a sweet tooth! Rippl provides some syntactic sugar when it comes to name-binding functions. The following expressions (after de-sugaring the syntax) are equivalent.

```
let add_one = fun x -> x + 1 # add_one :: int -> int
let cooler_add_one x = x + 1 # cooler_add_one :: int -> int
```

To top it all off, Rippl embraces the Haskell paradigm that all functions are curried. In other words, all functions in Rippl actually take just one parameter, either in arrow type form or proper/higher-order type form. This allows a Rippl user to employ partial application on functions!

```
let sum_three x y z = x + y + z in
# sum_three :: int -> int -> int -> int
let sum_two_add_one = sum_three 1 in
#sum_two_add_one :: int -> int -> int
```

```
let add_four = sum_two_add_one 3 in
# add_four :: int -> int
add_four 5  # 9
```

## 3.9 List Comprehensions

List comprehensions are an elegant and concise way to define and construct a list. They use a well-known mathematical notation to substitute complex list operations like map and filter.

### 3.9.1 Ranged List

With ranged lists, we can create a list without specifying every element. Ranges will always have an interval step of +1 and are only allowed for integer lists.

```
[18...24] # [18, 19, 20, 21, 22, 23, 24]

['a'...'z'] # not allowed

[1.2...2.7] # no sir, uncountable number of elements
```

### 3.9.2 Parametric list

Parametric lists allow users to specify a parameter.

```
[x * 2 | x over [3, 1, 4]] # [6, 2, 8]

[x | x over [9...12]] # [9, 10, 11, 12]
```

### 3.9.3 Qualified List

With qualified lists, lists can be filtered by one or more conditions that are represented as computations on the bound list parameters and return a boolean value.

```
[x ^ 2 | x over [1...10], x % 2 == 0] # [4, 16, 36, 64, 100]
```

### 3.9.4 Parallel List

Parallel list comprehensions are lists comprehensions with multiple parameters.

```
[x - y - z | x over [10, 1], y over [2, 3], z over [1, 5]]
# [7, 3, 6, 2, -2, -6, -3, -7]

[x + y | x over [10, 30, 50], y over [10...12], x != y]
# [20, 21, 22, 40, 41, 42, 60, 61, 62]
```

### 3.9.5 Infinite List

Thanks to lazy evaluation, Rippl supports infinite lists.

```
[3...] # [3, 4, 5, ...]
```

### 3.9.6 Lazy Evaluation

Rippl uses lazy evaluation for list comprehensions. This means that only the head of the list is initially fully evaluated and all other elements are evaluated when their value is requested. We can see the consequences of lazy evaluation in the example below:

```
let funky_list = [1/(x-1) | x over [0...]] # no error
    # even though when x == 1, 1/(x-1) is undefined
head funky_list # -1
    # only evaluates the first element of the list
tail funky_list # causes an error
    # will evaluate the second element, which divides by 0
```

We can see that when only the first element of the list is evaluated, there is no error. Although the second element of `funky_list` will have an error from dividing by 0, this element is not evaluated until we get the tail of the list. When we do get the tail and thus evaluate the second element of the list, there will be an error.

## 4 Immutability

As a pure functional language, Rippl enforces immutable semantics. This means that once an expression has been assigned to a variable name, the value can't be changed later on in the program (inducing a change in state). Rather than performing actions on objects and altering them, Rippl performs computations that return new values. This protects programs from side effects that may occur from having having multiple operations computing and mutating the same data.

Take for example the code snippet below. The use of the `cons` operator returns a new list with a first element of `0` rather than changing the value of `natural_nums`.

```
num_set :: bool
num_set = let natural_nums = [1...] in
let whole_nums = 0 cons natural_nums in
(head natural_nums) == (head whole_nums)    # false
```

By nature of being a functional language, Rippl has no constructs supporting reassignment of variables. However, in the case of nested let bindings, the program semantics may resemble mutability in variable reassignment. The program shown below for a definition of `change_a` is an example of such a case. Nevertheless, it's important to note that this is not a case of mutability, but a case of variable shadowing. The a in the final expression binds more tightly to the a assigned in inner let binding so the final value returned is that of the second a.

```
change_a :: int
change_a =  let a = 1 in
            let a = 2 in
            a       # 2
```

For a more concrete example of how this semantically differs from mutability, consider the modified definition of change_a below. By the second let-binding, the local definition of a binds more tightly than the higher-level binding for a and instead becomes a recursive definition. This results in a program error in both Rippl and Haskell.

```
change_a2 :: int
change_a2 = let a = 1 in
            let a = a + 1 in
            a       # error!
```

## 5 Entrypoints and IO

Like all pure functional languages, Rippl ensures that the evaluation of all expressions is free of side effects, or stateful interactions with the outside world. This prohibits the insertion of print statements inside functions that execute as they are evaluated.

Also as a pure functional language, Rippl guarantees that any computation given an argument will always return the same output. This means there can be no assumption of order of evaluation in a program as there is in the imperative paradigm. This prohibits the use of language constructs like input or scanf that read and return a value read from the user with no arguments (save the format-friendly strings present; this is more like a separate print/IO action than a true argument). With no varying arguments, the function call in a pure language should return the same value each time which should not be the behavior of an IO operation.

Pure functional languages like Haskell get around this by giving all the IO function calls a hidden argument and strings each call together as arguments and dependencies to create a proper ordering by using an IO monad. This operation is sugared up into an imperative-looking do construct in their main methods.

Like in Haskell, the main method in Rippl is the top-level entrypoint into a program. In order to avoid the use of higher-kinded types like in Haskell, the main method is used such that there is only one input operation allowed and one output operation allowed, strictly evaluated in that order.

User input is represented as the argument to the main function. It can be bound as a named argument that is then used for evaluation of the body of the main function if it requires user input.

```
# this main method takes user input and stores it in ‘arg‘
main arg = ...
```

Otherwise, to signify that nothing is to be read from user input, an underscore takes the place of the first argument (much like the wildcard in Haskell).

```
# this main method takes no user input
main _ = ...
```

The evaluated value of the body of the main function is printed. The ”Hello, world!” program is written as follows in Rippl.

```
# this prints "Hello, world!" to the terminal
main _ = "Hello, world!"
```

If the right-hand side of the main method were a more complex computation expressed in terms of other function definitions provided in the file, then the full expression would be evaluated and printed. This means that the main function is also fully polymorphic in terms of its input and output.

```
main :: a -> b
```

It might be noted that this construct of IO in a language doesn’t allow for a function not to have output (Rippl doesn’t have a `void` value representing a bottom type). However, this design should be reasonable since Rippl currently supports no other output operations so programs would be moot otherwise.

## 6 Sample Programs

### 6.1 Infinite Sum

This program returns the sum of the first $n$ positive integers using an infinite list.

```
inf_sum :: int -> int
inf_sum n = let infinity = [1...] in
    let rec_inf_sum x list acc = if x == 0 or (len list) == 0
        then acc
        else rec_inf_sum (x-1) (tail list) ((head list) + acc)
    in rec_inf_sum n infinity 0
```

### 6.2 Collatz Conjecture

The Collatz conjecture is a conjecture in mathematics surrounding the iterative function shown below:

$$\begin{cases} \frac{1}{2}x & x \texttt{ is even} \\ 3x+1 & x \texttt{ is odd} \end{cases}$$

Lothar Collatz proposed that this sequence will always converge to 1 starting from an arbitrary positive integer. Given a starting integer, the following program returns a list of iterations that ends in 1. The program hence uses 1 as its base case even though it hasn't been mathematically proven that all numbers will eventually reach this case, but we believe in Lothar.

```
collatz :: int -> [int]
collatz n =
    let rec_collatz n list =
        if n == 1
            then 1 cons list
        else if n % 2 == 0
            then rec_collatz (n / 2) (n cons list)
            else rec_collatz (3*n +1) (n cons list)
    in rec_collatz n []
```

### 6.3 Weak Prime Number Theorem

Bertrand's Postulate stated in the Weak Prime Number Theorem that there is always a prime number to be found between some $n$ and its double $2n$. This postulate was later proven by Pafnuty Chebyshev and refined by Paul Erdös. The following program includes a function that determines the primality of a number and a function that takes a $n$ and returns the first prime between $n$ and $2n$.

```
is_prime :: int -> bool
is_prime n =
    let max = n / 2 in
    let range = [2...max] in
    let divisors = [x | x over range, n % x == 0] in
    len divisors == 0

prime_number_theorem :: int -> int
prime_number_theorem n =
    let range = [(n+1)...2*n] in
    let odd_range = [x | x over range, x % 2 != 0 ] in
    foldl (fun prev -> fun curr -> if is_prime then prev else curr)
        (head odd_range) odd_range
```

## 6.4 Entrypoint and IO

This program demonstrates how a main method would be written to take in a user-inputted number and print out the full list of iterations generated by the formula in the Collatz Conjecture.

```
main :: int -> [int]
main n = collatz n      # 10 -> [10,5,16,8,4,2,1]
```