Rhine

Language Reference Manual

COMS W4115

Stephen A. Edwards

Ramkumar Ramachandra (rr2893) and Gudbergur Erlendsson (ge2187)

July 23, 2014

Contents

1 Introduction

This manual intends to describe the Rhine language. The syntax and semantics will be described in painful detail. However, it is a broad proposed standard, without an implementation.

2 Lexical Conventions

A program is entirely contained within a single file. Rhine files have the extension ".rh". There is no pre-processor system, and the tokens are parsed in one go using a program generated via ocamllex.

2.1 Tokens

The tokens we have are identifiers, keywords, operators, constants, and separators. White space (which consists of newlines, spaces and tabs) is ignored by the tokenizer.

2.2 Comments

The character sequence `;;` begins a comment, and it continues until the end of the line. There are no multi-line comments. Comments do not nest, and they do not occur within a string literals.

2.3 Identifiers

An identifier (or symbol) is a sequence of letters and digits, not beginning with a digit. The characters - and ? count as letters. Identifiers are case-sensitive. They are bound to values via `let` and `def`, and have lexical scope.

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise: `let`, `def`, `defn`, and `fn`.

2.5 Operators

The following are builtin operators and are further described in table under section 10: `' + - / * > < = >= <= and or not`.

2.6 Constants

Integers, floats, string literals, `true`, `false`, `nil` are all constants in Rhine.

2.6.1 Integers

Integer constants are a sequence of digits that can optionally begin with - or + signs which signifies negative and positive integers.

2.6.2 Floats

Float constants consist of an integer part, a decimal point, a fraction part. Furthermore float constants can optionally include an integer exponent using e notation.

2.6.3 String literals

A string literal is a sequence of characters surrounded by double quotes. Double quotes must be preceded by a backslash to be included in the string.

2.6.4 Others

`true`, `false` and `nil` are regular constants.

3 Syntactic structure

The syntax consists of parenthesis and square brackets. The square brackets enclose a "list" that is not to be evaluated, while the parenthesis encloses a S-expression to be used for computation.

An S-expression looks like:

```
(first-argument second-argument third-argument ...)
```

To illustrate a list, the following forms are equivalent:

```
'(first-argument second-argument third-argument ...)
[first-argument second-argument third-argument ...]
```

4 Type System

Rhine has 6 fundamental types (that are described further above): Integer, Float, Boolean, Symbol, nil, String, and List. The first five are fundamental data types, while the last one is a composite.

4.1 List type

Lists are essentially S-expressions that are not evaluated. They are internally represented by linked lists of cons pairs. They can contain elements of different data types, although care must be taken while mixing types.

4.2 Type checking

Functions have specific type signatures: the arguments (or their conversions, which is described next) must conform to the types, or there's a runtime error. Compile-time typechecking is much weaker, and can only catch certain types of errors.

4.3 Type conversions

Rhine does most common-sense implicit conversions. Integers get promoted to floats as necessitated by functions operating on them. Integers and floats are converted to Boolean as necessary. Nil is converted to the Boolean false. All types are promoted to strings, while printing, for example.

5 Expressions

Expressions are all S-expressions. A Rhine program consists of several S-expressions. Each S-expression can contain several other S-expressions, making a recursive structure. In this structure, the first element is a function, an operator, a keyword, or a builtin, and the other elements are the arguments. Each S-expression can be thought of as a linked-list made up of cons cells.

5.1 Underlying representation

To illustrate how each S-expression is a cons cell:

```
(first-argument second-argument third-argument ...)
(cons first-argument '(second-argument third-argument ...))
(cons first-argument (cons second-argument '(third-argument ...))
```

5.2 Expression nesting

To illustrate S-expression nesting, and the first argument being treated as a function:

```
(first-argument0 second-argument0 ...
  (first-argument1 second-argument1 ...
    (...
    ...)))
```

6 Scope rules

All bindings in Rhine are lexically scoped. `def` and `defn` definitions extend to the end of the file from where they appear. `let` definitions extend to where the `let` parentheses ends.

7 Control Structures

Statement	Description
(do sexpr*)	Evaluates sexpr's in order and returns the value of the last one.
(if predicate sexpr sexpr)	Evaluates predicate, if true first sexpr is executed else second sexpr.
(when predicate sexpr)	Evaluates predicate and executes sexpr if predicate is true.
(dotimes [x n] sexpr)	Executes sexpr n-times with n bound to x.

8 Function definition and lexical bindings

Statement	Description
(fn [arg*] sexp)	Defines an anonymous function with the given arguments.
(def identifier sexp)	Creates a variable with the name of the identifier and initializes to the sexp.
(defn identifier [arg*] sexp)	Defines a function with the given arguments, and binds it to an identifier.
(let [binding*] sexp*)	Evaluates the expressions sexp with the bindings list bound in its lexical context.

9 Operators

Statement	Description
'sexpr	Returns the sexpr unevaluated.
(+ sexpr ...)	Adds together all the arguments.
(- sexpr ...)	Subtracts from the first argument, the successive arguments.
(* sexpr ...)	Multiplies together all the arguments.
(/ sexpr ...)	Successively divides the first argument with the other arguments.
(mod sexpr1 sexpr2)	Returns the reminder of (/ sexpr1 sexpr2).
(> sexpr1 sexpr2)	Evaluates to the boolean sexpr1 > sexpr2.
(< sexpr1 sexpr2)	Evaluates to the boolean sexpr1 < sexpr2.
(= sexpr1 sexpr2)	Tests equality between sexpr1 and sexpr2.
(>= sexpr1 sexpr2)	Evaluates to the boolean sexpr1 >= sexpr2.
(<= sexpr1 sexpr2)	Evaluates to the boolean sexpr1 <= sexpr2.
(and sexpr1 sexpr2)	Evaluates to the logical and between sexpr1 and sexpr2.
(or sexpr1 sexpr2)	Evaluates to the logical or between sexpr1 and sexpr2.
(not sexpr)	Evaluates to the logical not of sexpr.

10 List functions

Statement	Description
(first list)	Returns the first element of the list.
(rest list)	Returns everything but the first element of the list.
(cons item list)	Returns the concatenation of [item] and list.
(length list)	Returns the length of the list.

11 String functions

Statement	Description
(str-split string)	Splits a string into a list of its characters.
(str-join list)	Takes a list of strings and returns them joined together in one string.

12 I/O functions

Statement	Description
(print sexpr)	Evaluates the sexpr and prints to stdout.
(println sexpr)	Evaluates the sexpr, prints to stdout and appends a newline character.

13 Examples

Listing 1: Sample snippets

```
;; FUNCTION DEFINITION
(defn abs
  "Absolute value of argument"
  [n]
  (if (< n 0)
    (* n -1)
    n))

;; RECURSIVE FUNCTIONS
(defn factorial
  "Returns the factorial of number n"
  [n]
  (if (< n 2)
    n
    (* n (factorial (dec n)))))

;; Recursive function acting on lists
(defn concat
  "Concatenates two lists"
  [c1 c2]
  (if (not (= [] c1))
    (cons (first c1)
          (concat (rest c1) c2))
    c2))

;; A functional one
(defn take
  "Return first n items of coll"
  [n coll]
  (when (and
         (> n 0)
         (not (= [] coll)))
    (cons (first coll)
          (take (dec n) (rest coll)))))

;; Similar to take
(defn drop
  "Drop the first n items of coll"
  [n coll]
  (if (and
       (> n 0)
       (not (= [] coll)))
    (drop (dec n) (rest coll))
    coll))

;; Building up to nth
(defn nth
  "Return the nth element of coll"
  [n coll]
  (first (drop (dec n) coll)))
```

```
;; since we don’t have variadic functions, here is map1 accepting
;; function of one argument + one list
(defn map1
  "Returns the result of applying f to each element of coll"
  [f coll]
  (when coll
    (cons (f (first coll))
          (map1 f (rest coll)))))

;; and map2 accepting a function of two arguments + two lists
(defn map2
  "Returns the result of applying f to each of (c1, c2)"
  [f c1 c2]
  (when (and c1 c2)
    (cons (f (first c1) (first c2))
             (map2 f (rest c1) (rest c2))))))
```