Rhine

COMS W4115

Stephen A. Edwards

Ramkumar Ramachandra (rr2893) and Gudbergur Erlendsson (ge2187)

July 14, 2014

Introduction

Rhine is a language in the LISP family to configure a programmable text editor. Like Emacs; the name is a recursive acronym for "Rhine is not Emacs". While building an entire text editor is out of the scope of the project, we want to implement a language that is ideal to configure a text editor.

Key language features

The syntax of Rhine is inspired by Clojure, which is richer than language like Elisp, used for Emacs. Like Scheme and Clojure, Rhine is a Lisp-1 which means that variables and functions share a single namespace. The base features are lexical closures and named functions. Including the basic ones to do conditionals and looping, Rhine includes some functional programming primitives from Clojure.

For the purpose of configuring an editor, producing ahead-of-time (AOT) executables is useless; instead, the compiler will use just-in-time (JIT) compilation to dynamically load programs. Generating optimized assembly is a hard problem, so Rhine will use an LLVM backend.

No Lisp is complete without a macro system, but its implementation can be quite complex; therefore, Rhine does not include a macro system. A long-running editor will also require garbage collection, but that is out of the scope of the project.

Primitives

Operators: ' + - / * > < = >= <= and or not
Integer operations: inc, dec
Conditionals: if, when
Loops: dotimes, while
Binding: defn, def, let
IO functions: print, println
Data types: integer, float, bool, string, list, nil
List operations: first, rest, cons, length

Examples

Listing 1: Sample snippets

```
;; SIMPLE EXPRESSIONS
(+ 2 3)
;; Returns: 5

(/ 2.0 3)
;; Returns: 0.666666666

;; CONDITIONALS
(if (> 2 3) "2 is bigger than 3" "3 is bigger than 2")
;; Returns: 3 is bigger than 2

(if (= 3 3) "3 and 3 are equal" "3 and 3 are not equal")
;; Returns: 3 and 3 are equal
```

```
;; LET BINDINGS ARE RECURSIVE
(let [x 1
      y x]
  y)

;; PRINTING
(println "Hello world!")
;; Prints Hello world!\n

(print "Hello world!")
;; Prints Hello world!

;; LOOPING
(while (print "foo"))

(dotimes [n 5] (print n))
;; Returns: 1 2 3 4 5

;; LISTS
(def onetwothree [1 2 3])
;; Creates list "onetwothree"

(first [1 2 3 4])
;; Returns: 1

(rest onetwothree)
;; Returns: [2 3]

(cons 1 '(2 3 4))
;; Returns: [1 2 3 4]

(length [1 3 3])
;; Returns: 3

;; STRINGS
(str-split "a string")
;; returns ("a", " ", "s", "t", "r", "i", "n", "g")

(str-join ["a" " " "s" "t" "r" "i" "n" "g"])
;; returns "a string"

;; FUNCTION DEFINITION
(defn square [x]
  (* x x))

(defn average
  "Docstring for function"
  [x y]
  (/ (+ x y) 2))

(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))))))

;; functions that operate on strings
(defn str-length
  "Length of string"
  [str]
  (length (str-split str)))

;; A more involved string function
(defn str-sub
  "Substring starting at index n1 and ending at index n2"
  [str n1 n2]
  (str-join
   (take (- n2 n1)
         (drop (dec n1)
               (str-split str)))))
```

End notes

From the examples, we see that nil, false, and () are distinct values; nil is reserved for indicating end-of-sequence in operations. Although strings could have been implemented as a list of characters, we have chosen to make them a first-class type with a family of associated functions. While the main focus of the examples is recursion, pure recursion will only be performant with tail call optimization (TCO), which most modern compilers implement.

A future editor built out of Rhine would rely heavily on OCaml's foreign function interface (FFI) to Objective C, for the Cocoa UI. For the editor to be truly programmable, every keystroke captured by the UI has to be sent over the FFI, processed by the JIT, and sent back over the FFI as a UI command. In Emacs for example, every keystroke that is unbound invokes self-insert-command, which prints the key on the screen; keys are bound to commands via global-set-key.