radicle Documentation

Monadic

Mar 26, 2020

Contents

1 The radicle guide **1**
- 1.1 Basics . . . 1
  - 1.1.1 Comments . . . 1
  - 1.1.2 Built-in datatypes . . . 1
  - 1.1.3 Atoms . . . 2
  - 1.1.4 Simple expressions . . . 4
  - 1.1.5 Quote and eval . . . 6
- 1.2 Modules . . . 6
- 1.3 Pattern matching . . . 8
  - 1.3.1 Basics . . . 8
  - 1.3.2 Pre-defined patterns . . . 8
  - 1.3.3 Binding variables . . . 9
  - 1.3.4 Non-linearity . . . 9
  - 1.3.5 Custom patterns . . . 9
- 1.4 Data Validation . . . 11

2 Testing **13**
- 2.1 Test Definition . . . 13
- 2.2 Test Setup . . . 14
- 2.3 Running Tests . . . 14

3 Issue machine tutorial **15**
- 3.1 Adding an removing issues . . . 15
- 3.2 Comments on issues . . . 17
- 3.3 Validating data . . . 18
- 3.4 Verifying authors . . . 18

4 Radicle Reference **21**
- 4.1 Primitive functions . . . 21
  - 4.1.1 `*` . . . 21
  - 4.1.2 `+` . . . 21
  - 4.1.3 `-` . . . 21
  - 4.1.4 `/` . . . 21
  - 4.1.5 `<` . . . 21
  - 4.1.6 `>` . . . 22
  - 4.1.7 `eq?` . . . 22
  - 4.1.8 `apply` . . . 22

4.1.9 show . . . 22
4.1.10 throw . . . 22
4.1.11 exit! . . . 22
4.1.12 read-annotated . . . 22
4.1.13 read-many-annotated . . . 22
4.1.14 base-eval . . . 22
4.1.15 ref . . . 22
4.1.16 read-ref . . . 23
4.1.17 write-ref . . . 23
4.1.18 match-pat . . . 23
4.1.19 cons . . . 23
4.1.20 first . . . 23
4.1.21 rest . . . 23
4.1.22 add-right . . . 23
4.1.23 <> . . . 23
4.1.24 list . . . 23
4.1.25 list-to-vec . . . 23
4.1.26 vec-to-list . . . 23
4.1.27 zip . . . 24
4.1.28 map . . . 24
4.1.29 length . . . 24
4.1.30 foldl . . . 24
4.1.31 foldr . . . 24
4.1.32 drop . . . 24
4.1.33 sort-by . . . 24
4.1.34 take . . . 24
4.1.35 nth . . . 24
4.1.36 seq . . . 24
4.1.37 dict . . . 25
4.1.38 lookup . . . 25
4.1.39 insert . . . 25
4.1.40 delete . . . 25
4.1.41 member? . . . 25
4.1.42 map-keys . . . 25
4.1.43 map-values . . . 25
4.1.44 string-append . . . 25
4.1.45 string-length . . . 25
4.1.46 string-replace . . . 25
4.1.47 foldl-string . . . 26
4.1.48 type . . . 26
4.1.49 atom? . . . 26
4.1.50 keyword? . . . 26
4.1.51 boolean? . . . 26
4.1.52 string? . . . 26
4.1.53 number? . . . 26
4.1.54 integral? . . . 26
4.1.55 vector? . . . 26
4.1.56 list? . . . 26
4.1.57 dict? . . . 26
4.1.58 file-module! . . . 27
4.1.59 find-module-file! . . . 27
4.1.60 import . . . 27
4.1.61 pure-state . . . 27
4.1.62 get-current-state . . . 27

4.1.63 set-current-state . . . 27
4.1.64 get-binding . . . 27
4.1.65 set-binding . . . 27
4.1.66 set-env . . . 27
4.1.67 state->env . . . 27
4.1.68 timestamp? . . . 28
4.1.69 unix-epoch . . . 28
4.1.70 from-unix-epoch . . . 28
4.1.71 now! . . . 28
4.1.72 to-json . . . 28
4.1.73 from-json . . . 28
4.1.74 uuid! . . . 28
4.1.75 uuid? . . . 28
4.1.76 default-ecc-curve . . . 28
4.1.77 verify-signature . . . 28
4.1.78 public-key? . . . 29
4.1.79 gen-key-pair! . . . 29
4.1.80 gen-signature! . . . 29
4.1.81 get-args! . . . 29
4.1.82 put-str! . . . 29
4.1.83 get-line! . . . 29
4.1.84 load! . . . 29
4.1.85 cd! . . . 29
4.1.86 stdin! . . . 29
4.1.87 stdout! . . . 29
4.1.88 stderr! . . . 29
4.1.89 read-file! . . . 30
4.1.90 read-line-handle! . . . 30
4.1.91 open-file! . . . 30
4.1.92 close-handle! . . . 30
4.1.93 system! . . . 30
4.1.94 wait-for-process! . . . 30
4.1.95 write-handle! . . . 30
4.1.96 subscribe-to! . . . 30
4.1.97 doc . . . 30
4.1.98 doc! . . . 30
4.1.99 apropos! . . . 31
4.2 Prelude modules . . . 31
4.3 prelude/basic . . . 31
4.3.1 (or x y) . . . 31
4.3.2 (some xs) . . . 31
4.3.3 (empty-seq? xs) . . . 31
4.3.4 length . . . 31
4.3.5 (maybe->>= v f) . . . 31
4.3.6 (maybe-foldlM f i xs) . . . 31
4.3.7 (elem? x xs) . . . 31
4.3.8 head . . . 31
4.3.9 tail . . . 32
4.3.10 (read s) . . . 32
4.3.11 (read-many s) . . . 32
4.3.12 (<= x y) . . . 32
4.4 prelude/patterns . . . 32
4.4.1 (match-pat pat v) . . . 32
4.4.2 (_ v) . . . 32

4.4.3 (/? p) . . . 32
4.4.4 (/as var pat) . . . 32
4.4.5 (/cons x-pat xs-pat) . . . 32
4.4.6 (/nil v) . . . 33
4.4.7 (/just pat) . . . 33
4.4.8 (/member vs) . . . 33
4.5 prelude/bool . . . 33
4.5.1 (not x) . . . 33
4.5.2 (and x y) . . . 33
4.5.3 (all xs) . . . 33
4.5.4 (and-predicate f g) . . . 33
4.6 prelude/seq . . . 33
4.6.1 (empty? seq) . . . 33
4.6.2 (seq? x) . . . 33
4.6.3 (reverse xs) . . . 34
4.6.4 (filter pred ls) . . . 34
4.6.5 (take-while pred ls) . . . 34
4.6.6 (starts-with? s prefix) . . . 34
4.6.7 (/prefix prefix rest-pat) . . . 34
4.6.8 (concat ss) . . . 34
4.7 prelude/list . . . 34
4.7.1 nil . . . 34
4.7.2 (range from to) . . . 34
4.8 prelude/strings . . . 34
4.8.1 (intercalate sep strs) . . . 34
4.8.2 (unlines x) . . . 35
4.8.3 (unwords x) . . . 35
4.8.4 (split-by splitter? xs) . . . 35
4.8.5 (words xs) . . . 35
4.8.6 (lines xs) . . . 35
4.8.7 (map-string f xs) . . . 35
4.8.8 (reverse-string str) . . . 35
4.8.9 (ends-with? str substr) . . . 35
4.8.10 (pad-right-to l word) . . . 35
4.9 prelude/error-messages . . . 35
4.9.1 (missing-arg arg cmd) . . . 36
4.9.2 (too-many-args cmd) . . . 36
4.9.3 (missing-arg-for-opt opt valid-args) . . . 36
4.9.4 (invalid-arg-for-opt arg opt valid-args) . . . 36
4.9.5 (invalid-opt-for-cmd opt cmd) . . . 36
4.9.6 (dir-already-exists dir-name) . . . 36
4.9.7 (git-clone-failure origin name) . . . 36
4.9.8 (upstream-commit-failure) . . . 36
4.9.9 (upstream-push-failure) . . . 36
4.9.10 (item-not-found item item-number) . . . 36
4.9.11 (whole-item-number item) . . . 36
4.9.12 (missing-item-number item action) . . . 37
4.9.13 (state-change-failure item state) . . . 37
4.9.14 (no-number-returned item) . . . 37
4.9.15 (unknown-command cmd) . . . 37
4.9.16 (unknown-commit commit) . . . 37
4.9.17 (parent-commit-not-master commit) . . . 37
4.9.18 (checkout-new-branch-failure branch) . . . 37
4.9.19 (checkout-master-failure) . . . 37

4.9.20 (applying-patch-failure) . . . 37
4.9.21 (applying-accepted-patch-failure) . . . 37
4.9.22 (push-patch-failure) . . . 37
4.9.23 (missing-key-file) . . . 38
4.9.24 (rad-ipfs-name-publish-failure stderr) . . . 38
4.9.25 (rad-ipfs-key-gen-failure stderr) . . . 38
4.9.26 (process-exit-error command args exit-code stderr) . . . 38
4.10 prelude/dict . . . 38
4.10.1 (dict-from-seq xs) . . . 38
4.10.2 (keys d) . . . 38
4.10.3 (values d) . . . 38
4.10.4 (rekey old-key new-key d) . . . 38
4.10.5 (modify-map k f d) . . . 38
4.10.6 (delete-many ks d) . . . 38
4.10.7 (lookup-default key default dict) . . . 39
4.10.8 (lookup-maybe key dict) . . . 39
4.10.9 (safe-modify-map k f d) . . . 39
4.10.10 (group-by f xs) . . . 39
4.11 prelude/io . . . 39
4.11.1 (print! x) . . . 39
4.11.2 (shell! command to-write) . . . 39
4.11.3 (process! command args to-write) . . . 39
4.11.4 (read-line!) . . . 39
4.11.5 (read-file-value! file) . . . 39
4.11.6 (read-file-values! file) . . . 40
4.11.7 (shell-with-stdout! command to-write) . . . 40
4.11.8 (shell-no-stdin! command to-write) . . . 40
4.11.9 (write-file! filename contents) . . . 40
4.11.10 (process-with-stdout! command args to-write) . . . 40
4.11.11 (process-with-stdout-stderr-exitcode! command args to-write) . . . 40
4.11.12 (process-with-stdout-strict! command args to-write) . . . 40
4.11.13 (init-file-dict! file) . . . 40
4.11.14 (read-file-key! file k) . . . 40
4.11.15 (write-file-key! file k v) . . . 40
4.11.16 (delete-file-key! file k) . . . 41
4.11.17 (ls!) . . . 41
4.11.18 (modify-file! file f) . . . 41
4.11.19 (install-fake-filesystem! files) . . . 41
4.11.20 (prompt! prompt) . . . 41
4.12 prelude/exception . . . 41
4.13 prelude/set . . . 41
4.13.1 empty . . . 41
4.13.2 (insert x s) . . . 41
4.13.3 (delete x s) . . . 41
4.13.4 member? . . . 42
4.13.5 (to-vec s) . . . 42
4.13.6 (from-seq xs) . . . 42
4.13.7 (key-set d) . . . 42
4.13.8 (subset? xs ys) . . . 42
4.14 prelude/ref . . . 42
4.14.1 (modify-ref r f) . . . 42
4.15 prelude/lens . . . 42
4.15.1 (make-lens g s) . . . 42
4.15.2 (view lens target) . . . 42

4.15.3 (set lens new-view target) . . . 42
4.15.4 id-lens . . . 43
4.15.5 (.. lens1 lens2) . . . 43
4.15.6 (... lenses) . . . 43
4.15.7 (over lens f target) . . . 43
4.15.8 (@ k) . . . 43
4.15.9 (@def k default) . . . 43
4.15.10 (@nth n) . . . 43
4.15.11 (view-ref r lens) . . . 43
4.15.12 (set-ref r lens v) . . . 43
4.15.13 (over-ref r lens f) . . . 43
4.16 prelude/io-utils . . . 43
4.16.1 (fzf-select! xs) . . . 44
4.16.2 (edit-in-editor! orig) . . . 44
4.16.3 (get-git-config! key) . . . 44
4.16.4 (set-git-config! key value) . . . 44
4.16.5 (get-git-commit-data! format commit) . . . 44
4.16.6 (get-git-username!) . . . 44
4.16.7 (process-git-with-exit! args msg) . . . 44
4.16.8 (base-path!) . . . 44
4.17 prelude/key-management . . . 44
4.17.1 (read-keys!) . . . 44
4.17.2 (get-keys!) . . . 45
4.17.3 (create-keys!) . . . 45
4.17.4 (set-fake-keys! keys) . . . 45
4.17.5 (use-fake-keys!) . . . 45
4.18 prelude/machine . . . 45
4.18.1 (updatable-eval sub-eval) . . . 45
4.18.2 (eval-fn-app state f arg cb) . . . 45
4.18.3 (send-prelude! machine-id) . . . 45
4.18.4 (new-machine!) . . . 45
4.18.5 (send-code! machine-id filename) . . . 45
4.18.6 (send! machine-id inputs) . . . 45
4.18.7 (query! machine-id expr) . . . 46
4.18.8 (install-remote-machine-fake) . . . 46
4.18.9 (send-signed-command! machine machine-id cmd payload) . . . 46
4.18.10 (catch-daemon! f) . . . 46
4.19 prelude/state-machine . . . 46
4.20 prelude/validation . . . 46
4.20.1 (= x) . . . 46
4.20.2 (member xs) . . . 46
4.20.3 (and vs) . . . 46
4.20.4 (or vs) . . . 46
4.20.5 (type t) . . . 47
4.20.6 (pred name p) . . . 47
4.20.7 (integral n) . . . 47
4.20.8 (optional-key k v) . . . 47
4.20.9 (contains k) . . . 47
4.20.10 (contains-all ks) . . . 47
4.20.11 (contains-only ks) . . . 47
4.20.12 (key k v) . . . 47
4.20.13 (optional-keys ks) . . . 47
4.20.14 (keys d) . . . 47
4.20.15 (every v) . . . 47

4.20.16 (uuid x) 48
4.20.17 (signed x) 48
4.20.18 (timestamp x) 48
4.20.19 (string-of-max-length max-len) 48
4.20.20 (always-valid x) 48
4.21 prelude/util 48
4.21.1 (make-counter) 48

5 Indices and tables **49**

CHAPTER 1

The radicle guide

radicle is a language for building and interacting with P2P Radicle Machines. What these machines do–how they behave–is entirely up to you.

This guide provides an introduction to radicle–as a language, as a tool for building chains, and as a means of interacting with existing chains.

Please see the [installation instructions](http://radicle.xyz/docs/index.html#installation-setup) to get started.

1.1 Basics

Though `radicle` is designed with a particular use-case in mind–namely, defining and interacting with decentralised state machines–it is ultimately a general purpose language, quite similar to Scheme or Clojure. To effectively use it, it is helpful to learn the basics of it independent of its application.

This chapter introduces `radicle` as a language. In each section we present a core datatype of the language, and show some basic operations that can be performed on them.

1.1.1 Comments

The string `;;` starts a comment. This comments all the text until the next line-break. In this guide we will often add a comment as follows:

```
(+ 1 1) ;; ==> 2
```

to indicate that an expression (in this case `(+ 1 1)`) evaluates to a specific value (in this case `2`).

1.1.2 Built-in datatypes

`radicle` supports the following datatypes:

Booleans

The literal syntax for booleans is `#t` for true, and `#f` for false. Booleans can be combined with `and` and `or`:

```
(and #t #f) ;; ==> #f
(or  #t #f) ;; ==> #t
```

Strings

Strings are enclosed in double quotes:

```
"this is a string"
```

Keywords

Keywords are identifiers that evaluate to themselves. To create a keyword you prefix `:` onto an identifier (`:foo` is a keyword, for example). Keywords are often used as keys in maps which act like records in other languages. The colon at the front is a prefix form of the colon that follows the name of a field in other languages. For example, `{name: "foo", age:  22}` in JavaScript becomes `{:name "foo" :age 22}` in Radicle.

```
:foo ;; ==> :foo
```

1.1.3 Atoms

Atoms are identifiers which, when evaluated, lookup the value bound to that identifier in the current environment:

```
(def foo 42)

foo ;; ==> 42
```

Vectors

`radicle`'s *vectors* are an array-like container for values. Literal notation for vectors uses square brackets:

```
["this" "is" "a" "vector"]
```

You can access any of the elements in the vector using `nth`:

```
(nth 2 ["start at zero!" "one" "two" "three"]) ;; ==> "two"
```

But be careful: if the size of the argument isn't less than (<) the size of the vector then this will result in an error.

Concatenate vectors using <>:

```
(<> [1 2 3] [4 5 6]) ;; ==> [1 2 3 4 5 6]
```

Add an element to the left using `cons`:

```
(cons 0 [1 2 3]) ;; ==> [0 1 2 3]
```

Add an element to the right using `add-right`:

```
(add-right 3 [0 1 2]) ;; ==> [0 1 2 3]
```

Lists

Lists are one of the fundamental data structures of functional programming. Just like vectors are enclosed in square brackets, lists are enclosed in parentheses.

However, as you may have noticed, parentheses are also used for function application. This is because lists are the representation for *code* that evaluates to the result of applying the first element of the list to the other elements.

Don't worry if this is still a little unclear; we will discuss it at greater length when we talk about evaluation and quotation. For now, what this in practice means is that if you want to create a list, you can use the function `list` on the elements you want in the list. You can also use `list-from-vec` to make a vector into a list.

You can prepend an element to a list with `cons`:

```
(cons 0 (list 1)) ;; ==> (0 1)
```

Retrieve the head of a list with `head`:

```
(head (list 0 1 2 3)) ;; ==> 0
```

The rest of it with `tail`:

```
(tail (list 0 1 2 3)) ;; ==> (1 2 3)
```

Or a specific index with `nth`:

```
(nth 2 (list 0 1 2 3)) ;; ==> 2
```

Lists or vectors?

Vectors differ from lists in a couple of important ways. First, lists are implemented as linked lists, and accessing the `nth` element of a list has linear time complexity on `n` (the function `nth` works for both vectors and lists). Additionally, appending to the end of a list is also linear. Vectors, on the other hand, allow for faster (logarithmic) indexed access and appending (constant time).

Dictionaries

Dictionaries are mappings between keys and values. (You may know them as maps, associative arrays, or symbol tables.) Literal syntax for dictionaries is formed by including the keys and values inside curly braces:

```
{ :key1 1
  :key2 "foo"
}
```

It is an error to have an odd number of elements inside the braces.

Keys may be any value, including other dictionaries:

```
{ :key1 1
  { :nested "foo" } "bar"
}
```

You can access the value associated with a key with the function `lookup`:

```
(lookup :key1 { :key1 1 }) ;; ==> 1
```

And add new values with `insert`:

```
(insert :key2 2 { :key1 1 }) ;; ==> { :key1 1 :key2 2 }
```

1.1.4 Simple expressions

Branching

To dispatch on a boolean value use `if`:

```
(def num-bugs 50)

(if (> num-bugs 0)
  "Oh no, so many bugs!"
  "All done!")
```

To test multiple booleans use `cond`:

```
(cond
  (> num-bugs 100) "Oh no, so many bugs!"
  (eq? num-bugs 0)   "All done!"
  #t               "A few bugs.")
```

In fact *all* values which are not `#f` are treated as true, so one can also write:

```
(cond
  (> num-bugs 100) "Oh no, so many bugs!"
  (eq? num-bugs 0)   "All done!"
  :else              "A few bugs.")
```

If none of the conditions in the `cond` are true then the result is an error!

Function application

A function call takes prefixed, parenthesized form. That is, the call appears within parenthesis, with the function at the head, followed by its arguments:

```
(+ 3 2) ;; => 5
```

Definitions

Definitions bind values to an identifier.

```
(def five 5)
five ;; ==> 5
```

Functions

A function is created using `fn`. The first argument is a vector of symbols representing the parameters of the function, and the rest is a sequence of expressions which are run when the function is called.

So a function which adds one to a number can be defined like so:

```
(fn [x] (+ x 1))
```

You can call it like any other function:

```
((fn [x] (+ x 1)) 1) ;; ==> 2
```

But most likely you want to `def`ine a new variable:

```
(def f (fn [x] (+ x 1)))

(f 41) ;; ==> 42
```

Functions are not by default recursive. If you want to define a recursive function, use the `def-rec` form:

```
(def-rec range
 (fn [from to]
  (if (<= to from)
    (list)
    (cons from (range (+ 1 from) to)))))
```

Note that `def-rec` may only be used to define functions.

For creating one-off anonymous functions quickly, there is a short-form. This is formed by a `\` character (meant to look like the main stroke of a λ) and then a valid radicle expression, which will be the return value of the function. To make a single parameter function use `?` for the argument. To make a multi-parameter function, use `?1`, `?2`, etc. Here are some examples of short-form lambdas and the long-form equivalent:

```
;; The identity function.
\?
(fn [x] x)

;; Partial application:
\(foo 0 ?)
(fn [x] (foo 0 x))

;; A function to make singleton vectors:
\[?]
(fn [x] [x])

;; Multiple arguments:
\(string-append ?1 " is the parent of " ?2)
(fn [x y] (string-append x " is the parent of " y))

;; Thunks:
\(put-str! "hello there!")
(fn [] (put-str! "hello there"))
```

Note that:

- Only `?1`, `?2`, ..., `?9` are allowed for positional arguments. If you want more that 9 parameters please seek professional help.

- Short-form lambdas cannot be nested.
- You cannot use a mixture of `?` and `?1`, `?2`, etc. inside a short-form lambda.
- To avoid hard to detect bugs, atoms starting with a `?` followed by a sequence of digits that isn't one of the allowed positional atoms will cause an error.

1.1.5 Quote and eval

1.2 Modules

To create a module, use the `module` special form. All modules must start with a *module declaration* which provides the module with a name, documentation string and an exports vector. Module declarations are evaluated, so you must quote any exports.

Here is a simple example of a module:

```
(module
  {:module  'my-module
   :doc     "Just for showing off modules."
   :exports '[foo bar]}

  (def foo-helper
    (fn [x] (+ x 1)))

  (def foo
    (fn [x] (foo-helper (foo-helper x))))

  (def bar
    (fn [x] (foo (+ x 2)))))
```

To import a module, use `import`:

```
(import my-module)
```

The content of a module is evaluated in a new scope that is identical to the scope of the module definition site. This means you you can use whatever has already been defined before the module was, and that definitions in the module don't leak outside the module. The form returns the module value, which can be passed around just like any other value. This means that we can write functions which create modules:

```
(def make-taxes
  (fn [VAT]
    (module
     {:module 'taxes
      :doc    "Calculates taxes."
      :exports ['add-VAT]}
     (def add-VAT (fn [x] (* (+ 1 VAT) x))))))
```

And then we could use this module with different values of `VAT`:

```
(import (make-taxes 2/10))
;; We also need to import `prelude/io` to be able to use `print!`
(import (file-module! "prelude/io.rad") :unqualified)

(print! (string-append "100 EUR + VAT is: " (show (taxes/add-VAT 100))))
```

By default `import` will add all the definitions of the module in fully-qualified form, which means `my-module/foo` and `my-module/bar` (but not `my-module/foo-helper`) are in scope:

```
(print! (my-module/bar 0))
```

You can narrow down exactly which definitions you would like to import by including a list of atoms:

```
(import my-module ['foo])
```

Which would only import `my-module/foo`. And you can also use a custom qualifier:

```
(import my-module :as 'baz)
```

After which we can:

```
(print! (baz/bar 42))
```

So we could import our taxes module twive using different qualifiers:

```
(def taxes-fr (make-taxes 20/100))
(def taxes-de (make-taxes 19/100))

(import taxes-de :as 'de)
(import taxes-fr :as 'fr)

(print!
 (string-append
  "100 EUR + VAT is "
  (show (de/add-VAT 100))
  " EUR in Germany and "
  (show (fr/add-VAT 100))
  " EUR in France."))
```

If you really want to import all the definitions without a qualification, then you can use the keyword `:unqualified` like so:

```
(import my-module :unqualified)
```

And then we can:

```
(print! (foo 128))
```

Qualification and import lists can be combined like so:

```
(import my-module ['bar] :as 'useful)
```

Would only add `useful/bar` to the current scope:

```
(print! (useful/bar 0))
```

When working at the REPL, the function `file-module!` is also available, which can create a module from a file. It assumes the file starts with a module declaration, and is then equivalent to wrapping the contents of the file in `(module ...)`.

Modules can be passed around and manipulated because they are just dicts with some metadata, and an `:env` key containing a radicle environment.

1.3 Pattern matching

1.3.1 Basics

Radicle has first class patterns, which means you can use `match` expressions to bind values according to the shape of some value, and create new patterns to be used in `match` expressions.

Pattern matching is invoked as follows:

```
;; Just getting an import out of the way
(import (file-module! "prelude/io.rad") :unqualified)

(match 42
  'x (print! (string-append "x was: " (show x))))
```

That's not very interesting, but it's all we can do before we import some more patterns:

```
(import (file-module! "prelude/test.rad") :unqualified)
(file-module! "prelude/basic.rad")
(import (file-module! "prelude/patterns.rad") :unqualified)
```

Now we can pattern match on vectors, numbers, dicts, etc. For example, evaluating:

```
(print!
  (match [42 [:a 3] {:key "val" :key2 "don't care"}]
    [43 [:a _] {:key 'v}]  :not-this
    [42 "hello" {:key 'v}] :not-this
    [42 [:a 'x] {:key 'v}] [:yes x v]
    _                      :not-this))
```

will print `[:yes 3 "val"]`.

A valid match expression has the shape `(match v ...patterns...)` where `...patterns...` is an even number of expressions. The *patterns* are the expressions at the even indices; the expressions at the odd indices are the *branches*. When evaluated, the value `v` is matched against the patterns, one at a time. For the first one that matches, the corresponding branch is then evaluated, and the result of that is the result of the expression. If none of the patterns matches `v`, then an exception is thrown.

1.3.2 Pre-defined patterns

Patterns are expressions which describe shapes that values can have, and specify atoms to bind certain parts of the value to variables.

These are the patterns that come included in the prelude:

- _ is the wildcard pattern. It will match any value. It's mostly useful for ignoring sub-parts of structures, or as a catch all for the last pattern.
- An atom will match against any value too, but it will then be bound to the matched value in the corresponding branch.
- Numbers, keywords and strings match against themselves by equality. E.g. `42` as a pattern will only match the value `42`.
- A vector of patterns `[p_0 ... p_n]` will match against vectors `[v_0 ... v_n]` of the same size, as long as `p_i` matches `v_i`.

- A dict `{k_0 p_0 ... k_n p_n}` of patterns will match against dicts `{k_0 v_0 ... k_n v_n ...}` that have *at least* the keys `k_0`, ..., `k_n`, and such that `p_i` matches `v_i`.
- The pattern `(/just p)` will match values `[:just v]` as long as the pattern `p` matches `v`.
- The pattern `(/cons h t)` will match against non-empty lists whose head matches `h`, and whose tail matches `t`.
- `/nil` will only match the empty list.
- The pattern `(/? p)`, where `p` is a predicate function, will match all values `v` such that `(p v)` is truthy.
- The pattern `(/as x p)`, where `x` is an atom and `p` is a pattern, matches any values that `p` matches while also binding this value to `x` in the branch.

1.3.3 Binding variables

Variables used in patterns must be quoted because patterns are themselves evaluated. This means that if `x` is already bound to a value then it can be used in a pattern. For example:

```
(def x 1)

(print!
  (match [2 2]
    [x       'y] (+ y y)
    [(+ x 1) 'y] (+ x y)))
```

will print the number `3`, that is, evaluate the second branch. This is because `(+ x 1)` evaluates to `2`, and so this matches the first `2` of the value `[2 2]`.

Quoting variables is also what allows custom patterns to be defined.

1.3.4 Non-linearity

All of the patterns included in the prelude support non-linear matching, which means re-using a variable will result in testing for equality. For example the pattern `['x 'x]` will match vectors of length `2` with the same value repeated twice. And

```
(print!
  (match [1 2 2]
    ['x 'x 'x'] [:three x]
    ['x 'y 'y]  [:one x :and-two y]
    ['x 'y 'z]  [:all-different x y z]))
```

will print `[:one 1 :and-two 2]` because `[1 2 2]` matches the second pattern but not the first, as `1` is not equal to `2`.

1.3.5 Custom patterns

You can easily create you own patterns: they are just functions with return `[:just b]` where `b` is a dict of bindings in case of a match, and `:nothing` when the value does not match.

For example let's assume that we have to deal with a lot of values of the following shape:

```
(def alice
  {:user-id "123"
   :first-name "Alice"
   :last-name "Smith"
   :char-attributes {:class "dwarf"
                     :max-hp 132
                     :current-hp 122
                     :spell-radius 134
                     :and-lots-more "stuff"}})

(def bob
  {:user-id "345"
   :first-name "Bob"
   :last-name "Kane"
   :char-attributes {:class "elf"
                     :max-hp 96
                     :current-hp 55
                     :spell-radius 76
                     :and-lots-more "stuff"}})
```

And for some reason we keep having to extract the full name and spell radius if `:spell-radius` is over `100`, but otherwise we just want the character `:class` and the `:current-hp`. We can do this with two custom patterns:

```
(def danger
  (fn [name spell-rad]
    (fn [v]
      (match v
        {:first-name 'fn
         :last-name  'ln
         :char-attributes {:spell-radius 'sr}}
          (if (> sr 100)
            [:just {name (string-append fn " " ln)
                    spell-rad sr}]
            :nothing)
           _ :nothing))))

(def prey
  (fn [class current-hp]
    (fn [v]
      (match v
             {:char-attributes {:class 'c
                                :current-hp 'hp}}
             [:just {class c
                     current-hp hp}]
             _ :nothing))))
```

And now we can make a function for alerting players to other nearby players:

```
(def nearby
  (fn [p]
    (match p
      (danger 'n 'sr)
        (print! (string-append "DANGER! powerful player " n " with spell radius "␣
↪(show sr) " is approaching!"))
      (prey 'c 'hp)
        (print! (string-append "A weak " c " with only " (show hp) " health is nearby!
↪ Attack!")))))
```

And calling

```
(nearby alice)
```

Will print `"DANGER! powerful player Alice Smith with spell radius 134 is approaching!"`, but calling:

```
(nearby bob)
```

will print `"A weak elf with only 55 health is nearby! Attack!"`.

1.4 Data Validation

Most of the entities that get manipulated and stored on chains are represented by structured radicle data. To maintain the integrety of the chain it can be important to refuse malformed/invalid data. To detect this, use *validator functions*. These are functions which take a piece of data as input and either throw an exception (thus cancelling the current transaction), or return the data unchanged. The radicle prelude contains functions for creating and combining validators.

For example let's assume that we want to validate the `:priority` field on an issue entity, with the requirement that it be an integer p such that $0 \leq p \leq 10$. To do this we can define a validator:

```
(def validator/priority
  (validator/and
    [(validator/type :number)
     (validator/pred "is integer"
                     integral?)
     (validator/pred "0 <= p <= 10"
                     (fn [p] (and (< -1 p) (< p 11))))]))
```

Here we have used `validator/and` to combine several validators together: the priority is valid only if it passes all three validators:

- The first one checks that the value is a number.
- The second uses `validator/pred`, which takes a description and a predicate,

and checks that the predicate holds. In this case we ensure that the priority is a whole number.

- The third validator ensures the priority is in the required range.

Very often entities are represented using dicts with specific required keys. Validators for these entities can be created using the `validator/keys` function. For example to validate an issue we could use:

```
(def validator/issue
  (validator/keys
    {:id     (validator/and
               [(validator/type :string)
                (validator/pred "valid UUID"
                                uuid?)])
     :author (validator/pred "valid public key"
                             public-key?)
     :title  (validator/and
               [(validator/type :string)
                (validator/pred "< 60 chars"
                                (fn [s] (< (string-length s) 60)))])
     :body   (validator/and
```

(continues on next page)

(continued from previous page)

```
               [(validator/type :string)
                (validator/pred "valid markdown"
                                markdown?)
                (validator/pred "< 4000 chars"
                                (fn [s] (< (string-length s) 4000)))])
    :tags    (validator/every
               (validator/and
                 [(validator/type :string)
                  (validator/pred "< 40 chars"
                                  (fn [s] (< (string-length s) 40)))]))
    :priority validator/priority}))
```

This checks that an issue is a dict, with required keys:

- `:id` that's a valid UUID string,
- `:author` that's a valid public key,
- `:title` that's a string shorter than 60 characters,
- `:body` that's a markdown string less that 4000 characters,
- `:tag` that's a vector of strings, each shorted than 40 characters.
- `:priority` that's a integers p such that $0 \leq p \leq 10$.

Because validators return their input if it is valid, we can just wrap any value which should be validated with the appropriate validator:

```
(fn [issue]
  (modify-ref
    issues
    (fn [is]
      (add-right (validator/issue issue) is))))
```

CHAPTER 2

Testing

Radicle provides a `:test` macro which allows you to write tests next to your code.

```
(def not
  (fn [x] (if x #f #t)))

(:test "not"
   [ (not #t) ==> #f ]
   [ (not #f) ==> #t ]
   )
```

The `test/prelude.rad` script runs all tests defined in the prelude.

2.1 Test Definition

Each test definition consists of a test name and a list of steps

```
(:test "my test" step1 step2 ...)
```

Each step is a vector triple with the symbol ==> in the middle. For example

```
[ (not #t) ==> #f ]
```

When a test step is run the value left of ==> is evaluated in the environment captured at the definition site of the test. The resulting value is then compared with the right-hand side. The test passes if both are equal. The right-hand side is *not* evaluated.

If the evaluation of the left-hand side throws an error the test fails with the message produced by the error.

Changes to reference in a test step evaluation have no effect and all tests steps are run independently.

2.2 Test Setup

There is a special setup test step that allows you to change the environment that tests steps run in.

```
(:test "with setup"
  [ :setup (do
     (def foo 5)
     )]
  [ (+ foo 2) ==> 7 ]
  [ (- foo 2) ==> 3 ]
  )
```

Similar to test steps the body of the `:setup` step is evaluated in the environment of the definition site. Changes to the environment introduced by evaluating the setup code are then available in all test steps.

2.3 Running Tests

All tests defined with the `:test` macro are collected in the `tests` reference. Tests can be executed using the `run-all` function from the `prelude/test` module.

CHAPTER 3

Issue machine tutorial

3.1 Adding an removing issues

In this example we will create a machine which stores some issues about a code repository. At the start it will only support adding and removing issues, but then we will add more features.

This is a literate radicle file, which means you can run it like this: `stack exec doc -- docs/source/guide/Issues.lrad`.

First we load the prelude:

```
(load! (find-module-file! "prelude.rad"))
```

We'll use a ref to store the issues, as a dictionary from issue IDs to issues.

```
(def issues (ref {}))
```

To update the state of our issues we'll use `modify-ref`, a function which takes a ref and a function. It updates the value held in the ref using the function and then returns the *new* value. We'll be modifying `issues` a lot, so let's create a function for that:

```
(def mod-issues
  (fn [f] (modify-ref issues f)))
```

Next we'll define a function for creating a new issue. An issue is a dict which looks like:

```
{ :id     "ab12-xy23"
  :author "user-id-here"
  :title  "I want pattern matching"
  :body   "I can't continue using radicle without it."
}
```

The `:id` should be a valid UUID that is supplied by whoever submits the issue; we just check that the ID isn't already used for another issue:

```
(def id-valid-and-free?
  (fn [id]
    (and (uuid? id)
         (not (member? id (read-ref issues))))))
```

To add an issue we just add it to the `issues` dict, using the `:id` as the key. Note that in the case of an error we use `print!` in this tutorial, but in a real machine one would `throw` to refuse the transaction.

```
(def new-issue
  (fn [i]
    (def id (lookup :id i))
    (if (id-valid-and-free? id)
      (mod-issues
        (fn [is] (insert id i is)))
      (print! "Issue ID was not free or was invalid."))))
```

Closing an issue is also easy–we just delete that issue from the dict:

```
(def close-issue
  (fn [id]
    (mod-issues (fn [is] (delete id is)))))
```

The behaviour we want from our machine is to accept some commands to perform issue-related tasks. We'll store these commands in a dict, keyed by the command symbol. We put the whole thing in a ref so that we can add more commands later, or even update existing commands.

```
(def commands
  (ref
    {'create new-issue
     'close  close-issue}))
```

Now we create a function for handling these commands. We assume that the input is a list with at least 2 elements, the first of which is a symbol describing the command. We look up the command handler in the `commands` ref and then run it on the arguments using `apply`, which calls a function on a list of arguments. Finally we print the issues so that we can see how the state evolves.

```
(def process-command
  (fn [expr]
    (def command (head expr))
    (def args    (tail expr))
    (def do-this (lookup command (read-ref commands)))
    (apply do-this args)
    (print! (read-ref issues))
    :ok))
```

Now we update the eval. We will check for a special command `update` for updating the semantics of our machine (by executing arbitrary code), but otherwise we delegate to `process-command`. In general of course you wouldn't want to leave such an `update` command with no restrictions; we'll talk about that later.

```
(load! "rad/machine.rad")

(def eval
  (updatable-eval
    (fn [expr state]
      (eval-fn-app state 'process-command expr print!))))
```

Now we can start creating some issues. To generate some UUIDs, use the `uuid!` function at the REPL.

```
(create {:id     "f621caec-b0a3-4c5e-9fdd-147066a35af1"
         :author "james"
         :title  "Pattern matching"
         :body   "Pleeease"})

(create {:id     "a37e56bd-b66a-4f3f-af06-9eaeb4afdae9"
         :author "julian"
         :title  "Better numbers"
         :body   "The ids are floats!"})
```

3.2 Comments on issues

Now it would be nice to be able to comment on issues. For this we'll just use a `:comment` field in the issue, which will be a vector of comments. The first thing we need to do therefore is add an empty vector of comments to all our current issues. To do this we'll use the update command which allows us to run some arbitrary updating code:

```
(update
  (modify-ref issues
              (fn [is]
                (map-values (fn [i] (insert :comments [] i))
                            is))))
```

When creating issues we now need to be careful to add the comments:

```
(create {:id       "4a4d4479-468e-46e5-b026-1f84288aa682"
         :author   "Alice"
         :title    "Can issues have comments please?"
         :body     "So that we can talk about them."
         :comments []})
```

A more sophisticated handler might add an empty comments field if it doesn't exist.

To add the commenting feature, first we will add a new function to add comments to issues.

```
(update
  (def add-comment
    (fn [issue-id comment]
      (mod-issues
        (fn [is]
          (modify-map issue-id
                      (fn [i]
                        (modify-map :comments
                                    (fn [cs] (add-right comment cs))
                                    i))
                      is))))))
```

TODO: modify-map should be renamed to modify-dict

(You'll note that there is a lot of painful nested updating going on; later we will see how this can be made a lot easier with lenses.)

Adding this command to our commands dictionary will now make comments work:

```
(update
  (modify-ref
```

(continues on next page)

(continued from previous page)

```
    commands
    (fn [cs] (insert 'comment add-comment cs))))
```

Now we can comment on issues:

```
(comment "a37e56bd-b66a-4f3f-af06-9eaeb4afdae9"
         {:author  "james"
          :comment "Yes that is a good idea."})
```

3.3 Validating data

It would be nice to validate issues and comments before they are added, e.g. checking if all the right keys exist.

TODO

3.4 Verifying authors

This machine has issues and comments coming from users but there is nothing that enforces that the issues and comments are actually submitted by these users. I (Alice) could easily submit the comment:

(update { :author "bobs-id" :comment "LGTM"})

to the machine and no one would know it wasn't Bob. To remedy this we will use signatures. First of all we will assume that this machine has a unique ID:

```
(update
  (def machine-id "some-unique-id"))
```

Signatures are created using the `gen-signature!` function, which takes a secret key `sk` and a string `msg`. The signature `sig` that it returns can be used with the function `verify-signature` as follows: `(verify-signature pk sig msg)`. If this returns `#t` then this tells us that the person who signed the message is the person who knows the secret key.

We create a function to verify that issues are valid:

```
(update
  (def issue-valid?
    (fn [i]
      (verify-signature (lookup :author i)
                        (lookup :signature i)
                        (string-append machine-id
                                       (lookup :id i)
                                       (lookup :title i)
                                       (lookup :body i))))))
```

Finally we modify the `add-issue` function. Note that we re-use our old `add-issue` function, just adding the extra layer of security:

```
(update
  (def add-verified-issue
    (fn [issue]
      (if (issue-valid? issue)
          (new-issue issue)
```

(continues on next page)

(continued from previous page)

```
          ;; In a real machine we would throw, not print a message.
          (print! "The issue was not valid.")
          ))))
```

And we update our commands:

```
(update
  (modify-ref
    commands
    (fn [cs] (insert 'create add-verified-issue cs))))
```

Issues now have to be signed to be valid, here is some code one could run in another file to create such commands:

```
;; (def machine-id "some-unique-id")

;; (def kp (gen-key-pair! (default-ecc-curve)))

;; (def sk (lookup :private-key kp))
;; (def pk (lookup :public-key  kp))

;; (def make-issue
;;   (fn [title body]
;;     (def id (uuid!))
;;     (def msg
;;          (string-append machine-id
;;                         id
;;                         title
;;                         body))
;;     {:id        id
;;      :author    pk
;;      :title     title
;;      :body      body
;;      :signature (gen-signature! sk msg)}))
```

To generate a signed issue we can call, after loading that code into a REPL:

```
;; (make-issue "This issue has a verified author"
;;             "This is the body of the first issue with a verified author.")
```

And submit the result to the machine:

```
(create
  {:author [:public-key
   {:public_curve [:curve-fp
    [:curve-prime
    1.15792089237316195423570985008687907853269984665640564039457584007908834671663e77
    [:curve-common
    {:ecc_a 0.0
     :ecc_b 7.0
     :ecc_g [:Point
     5.506626302227734366957871889516853432625060345377759417550018736038911672924e76
     3.
↪2670510020758816978083085130507043184471273380659243275938904335757337482424e76]
     :ecc_h 1.0
     :ecc_n 1.
↪15792089237316195423570985008687907852837564279074904382605163141518161494337e77}]]]
    :public_q [:Point
```

(continues on next page)

(continued from previous page)

```
    1.110546924872650168002926498121575048209371485859895263046216140940612572329 89e77
    3.
↪60592724795916246124205817195260729342618668337794467252193400584386517345 23e76]}]
   :body "This is the body of the first issue with a verified author."
   :id "76dd218b-fbc1-4384-9962-8bfbec5da2a2"
   :signature [:Signature
   {:sign_r 1.
↪07685492960818947345554835683887719269111710108141784367526228085824476440 077e77
    :sign_s 3.
↪74076707669392540151933947566955789136040644083942885188837225336805849089 6e76}]
   :title "This issue has a verified author"})
```

Signing comments would be done in a similar way.

CHAPTER 4

Radicle Reference

This is the `radicle` reference document, with documentation for all functions which come as part of the standard distribution.

4.1 Primitive functions

Primitive functions are those that are built into the compiler. They are available on all machines but may be shadowed by later definitions. Those that end in a `!` are only available locally, not on 'pure' machines.

4.1.1 *

Multiplies two numbers together.

4.1.2 +

Adds two numbers together.

4.1.3 -

Substracts one number from another.

4.1.4 /

Divides one number by another. Throws an exception if the second argument is 0.

4.1.5 <

Checks if a number is strictly less than another.

4.1.6 >

Checks if a number is strictly greater than another.

4.1.7 eq?

Checks if two values are equal.

4.1.8 apply

Calls the first argument (a function) using as arguments the elements of the the second argument (a list).

4.1.9 show

Returns a string representing the argument value.

4.1.10 throw

Throws an exception. The first argument should be an atom used as a label for the exception, the second can be any value.

4.1.11 exit!

Exit the interpreter immediately with the given exit code.

4.1.12 read-annotated

`(read-anotated label s)` parses the string `s` into a radicle value. The resulting value is not evaluated. The `label` argument is a string which is used to annotate the value with line numbers.

4.1.13 read-many-annotated

(read-many-annotated label s) parses a string into a vector of radicle values. The resulting values are not evaluated. The `label` argument is a string which is used to annotate the values with line numbers.

4.1.14 base-eval

The default evaluation function. Expects an expression and a radicle state. Return a list of length 2 consisting of the result of the evaluation and the new state.

4.1.15 ref

Creates a ref with the argument as the initial value.

4.1.16 `read-ref`

Returns the current value of a ref.

4.1.17 `write-ref`

Given a reference `r` and a value `v`, updates the value stored in `r` to be `v` and returns `v`.

4.1.18 `match-pat`

The most basic built-in pattern-matching dispatch function.

4.1.19 `cons`

Adds an element to the front of a sequence.

4.1.20 `first`

Retrieves the first element of a sequence if it exists. Otherwise throws an exception.

4.1.21 `rest`

Given a non-empty sequence, returns the sequence of all the elements but the first. If the sequence is empty, throws an exception.

4.1.22 `add-right`

Adds an element to the right side of a vector.

4.1.23 `<>`

Merges two structures together. On vectors and lists this performs concatenation. On dicts this performs the right-biased merge.

4.1.24 `list`

Turns the arguments into a list.

4.1.25 `list-to-vec`

Transforms lists into vectors.

4.1.26 `vec-to-list`

Transforms vectors to lists.

4.1.27 zip

Takes two sequences and returns a sequence of corresponding pairs. In one sequence is shorter than the other, the excess elements of the longer sequence are discarded.

4.1.28 map

Given a function `f` and a sequence (list or vector) `xs`, returns a sequence of the same size and type as `xs` but with `f` applied to all the elements.

4.1.29 length

Returns the length of a vector, list, or string.

4.1.30 foldl

Given a function `f`, an initial value `i` and a sequence (list or vector) `xs`, reduces `xs` to a single value by starting with `i` and repetitively combining values with `f`, using elements of `xs` from left to right.

4.1.31 foldr

Given a function `f`, an initial value `i` and a sequence (list or vector) `xs`, reduces `xs` to a single value by starting with `i` and repetitively combining values with `f`, using elements of `xs` from right to left.

4.1.32 drop

Returns all but the first `n` items of a sequence, unless the sequence is empty, in which case an exception is thrown.

4.1.33 sort-by

Given a sequence `xs` and a function `f`, returns a sequence with the same elements `x` of `xs` but sorted according to `(f x)`.

4.1.34 take

Returns the first `n` items of a sequence, unless the sequence is too short, in which case an exception is thrown.

4.1.35 nth

Given an integral number `n` and `xs`, returns the `n`th element (zero indexed) of `xs` when `xs` is a list or a vector. If `xs` does not have an `n`-th element, or if it is not a list or vector, then an exception is thrown.

4.1.36 seq

Given a structure `s`, returns a sequence. Lists and vectors are returned without modification while for dicts a vector of key-value-pairs is returned: these are vectors of length 2 whose first item is a key and whose second item is the associated value.

4.1.37 dict

Given an even number of arguments, creates a dict where the `2i`-th argument is the key for the `2i+1`th argument. If one of the even indexed arguments is not hashable then an exception is thrown.

4.1.38 lookup

Given a value `k` (the 'key') and a dict `d`, returns the value associated with `k` in `d`. If the key does not exist in `d` then `()` is returned instead. If `d` is not a dict then an exception is thrown.

4.1.39 insert

Given `k`, `v` and a dict `d`, returns a dict with the same associations as `d` but with `k` associated to `d`. If `d` isn't a dict or if `k` isn't hashable then an exception is thrown.

4.1.40 delete

Given `k` and a dict `d`, returns a dict with the same associations as `d` but without the key `k`. If `d` isn't a dict then an exception is thrown.

4.1.41 member?

Given `v` and structure `s`, checks if `x` exists in `s`. The structure `s` may be a list, vector or dict. If it is a list or a vector, it checks if `v` is one of the items. If `s` is a dict, it checks if `v` is one of the keys.

4.1.42 map-keys

Given a function `f` and a dict `d`, returns a dict with the same values as `d` but `f` applied to all the keys. If `f` maps two keys to the same thing, the greatest key and value are kept.

4.1.43 map-values

Given a function `f` and a dict `d`, returns a dict with the same keys as `d` but `f` applied to all the associated values.

4.1.44 string-append

Concatenates a variable number of string arguments. If one of the arguments isn't a string then an exception is thrown.

4.1.45 string-length

DEPRECATED Use `length` instead. Returns the length of a string.

4.1.46 string-replace

Replace all occurrences of the first argument with the second in the third.

4.1.47 `foldl-string`

A left fold on a string. That is, given a function `f`, an initial accumulator value `init`, and a string `s`, reduce `s` by applying `f` to the accumulator and the next character in the string repeatedly.

4.1.48 `type`

Returns a keyword representing the type of the argument; one of: `:atom`, `:keyword`, `:string`, `:number`, `:boolean`, `:list`, `:vector`, `:function`, `:dict`, `:ref`.

4.1.49 `atom?`

Checks if the argument is a atom.

4.1.50 `keyword?`

Checks if the argument is a keyword.

4.1.51 `boolean?`

Checks if the argument is a boolean.

4.1.52 `string?`

Checks if the argument is a string.

4.1.53 `number?`

Checks if the argument is a number.

4.1.54 `integral?`

Checks if a number is an integer.

4.1.55 `vector?`

Checks if the argument is a vector.

4.1.56 `list?`

Checks if the argument is a list.

4.1.57 `dict?`

Checks if the argument is a dict.

4.1.58 file-module!

Given a file whose code starts with module metadata, creates the module. That is, the file is evaluated as if the code was wrapped in `(module ...)`.

4.1.59 find-module-file!

Find a file according to radicle search path rules. These are: 1) If RADPATH is set, first search there; 2) If RADPATH is not set, search in the distribution directory 3) If the file is still not found, search in the current directory.

4.1.60 import

Import a module, making all the definitions of that module available in the current scope. The first argument must be a module to import. Two optional arguments affect how and which symbols are imported. `(import m :as 'foo)` will import all the symbols of `m` with the prefix `foo/`. `(import m '[f g])` will only import `f` and `g` from `m`. `(import m '[f g] :as 'foo')` will import `f` and `g` from `m` as `foo/f` and `foo/g`. To import definitions with no qualification at all, use `(import m :unqualified)`.

4.1.61 pure-state

Returns a pure initial radicle state. This is the state of a radicle chain before it has processed any inputs.

4.1.62 get-current-state

Returns the current radicle state.

4.1.63 set-current-state

Replaces the radicle state with the one provided.

4.1.64 get-binding

Lookup a binding in a radicle env.

4.1.65 set-binding

Add a binding to a radicle env.

4.1.66 set-env

Sets the environment of a radicle state to a new value. Returns the updated state.

4.1.67 state->env

Extract the environment from a radicle state.

4.1.68 timestamp?

Returns true if the input is an ISO 8601 formatted CoordinatedUniversal Time (UTC) timestamp string. If the input isn't a string, an exception is thrown.

4.1.69 unix-epoch

Given an ISO 8601 formatted Coordinated Universal Time (UTC) timestamp, returns the corresponding Unix epoch time, i.e., the number of seconds since Jan 01 1970 (UTC).

4.1.70 from-unix-epoch

Given an integer the represents seconds from the unix epock return an ISO 8601 formatted Coordinated Universal Time (UTC) timestamp representing that time.

4.1.71 now!

Returns a timestamp for the current Coordinated Universal Time (UTC), right now, formatted according to ISO 8601.

4.1.72 to-json

Returns a JSON formatted string representing the input value. Numbers are only converted if they have a finite decimal expansion. Strings and booleans are converted to their JSON counterparts. Atoms and keywords are converted to JSON strings (dropping the initial ':' for keywords). Lists and vectors are converted to JSON arrays. Dicts are converted to JSON objects as long as all the keys are strings, atoms, keywords, booleans or numbers.

4.1.73 from-json

Converts a JSON string into Radicle data. If the string is not valid JSON then `:nothing` is returned, otherwise `[:just v]` is returned where `v` is a Radicle representation of the JSON data.

4.1.74 uuid!

Generates a random UUID.

4.1.75 uuid?

Checks if a string has the format of a UUID.

4.1.76 default-ecc-curve

Returns the default elliptic-curve used for generating cryptographic keys.

4.1.77 verify-signature

Given a public key `pk`, a signature `s` and a message (string) `m`, checks that `s` is a signature of `m` for the public key `pk`.

4.1.78 `public-key?`

Checks if a value represents a valid public key.

4.1.79 `gen-key-pair!`

Given an elliptic curve, generates a cryptographic key-pair. Use `default-ecc-curve` for a default value for the elliptic curve.

4.1.80 `gen-signature!`

Given a private key and a message (a string), generates a cryptographic signature for the message.

4.1.81 `get-args!`

Returns the list of the command-line arguments the script was called with

4.1.82 `put-str!`

Prints a string.

4.1.83 `get-line!`

Reads a single line of input and returns it as a string.

4.1.84 `load!`

Evaluates the contents of a file. Each seperate radicle expression is `eval`uated according to the current definition of `eval`.

4.1.85 `cd!`

Change the current working directory.

4.1.86 `stdin!`

A handle for standard in.

4.1.87 `stdout!`

A handle for standard out.

4.1.88 `stderr!`

A handle for standard error.

4.1.89 read-file!

Reads the contents of a file and returns it as a string.

4.1.90 read-line-handle!

Read a single line from a handle. Returns the string read, or the keyword `:eof` if an EOF is encountered.

4.1.91 open-file!

Open file in the specified mode (`:read`, `:write`, `:append`, `:read-write`).

4.1.92 close-handle!

Close a handle

4.1.93 system!

(system! proc) execute a system process. Returns the dict with the form `{ :stdin maybe-handle :stdout maybe-handle :stderr maybe-handle :proc prochandle }` Where `maybe-handle` is either `[:just handle]` or `:nothing`. Note that this is quite a low-level function; higher-level ones are more convenient.

4.1.94 wait-for-process!

Block until process terminates.

4.1.95 write-handle!

Write a string to the provided handle.

4.1.96 subscribe-to!

Expects a dict `s` (representing a subscription) and a function `f`. The dict `s` should have a function `getter` at the key `:getter`. This function is called repeatedly (with no arguments), its result is then evaluated and passed to `f`.

4.1.97 doc

Returns the documentation string for a variable. To print it instead, use `doc!`.

4.1.98 doc!

Prints the documentation attached to a value and returns `()`. To retrieve the docstring as a value use `doc` instead.

4.1.99 `apropos!`

Prints documentation for all documented variables in scope.

4.2 Prelude modules

These are the modules included in the radicle prelude and the functions these modules expose.

4.3 `prelude/basic`

Basic function used for checking equality, determining the type of a value, etc.

4.3.1 `(or x y)`

Returns `x` if `x` is not `#f`, otherwise returns `y`

4.3.2 `(some xs)`

Checks that there is a least one truthy value in a list.

4.3.3 `(empty-seq? xs)`

Returns true if `xs` is an empty sequence (either list or vector).

4.3.4 `length`

Returns the length of a vector, list, or string.

4.3.5 `(maybe->>= v f)`

Monadic bind for the maybe monad.

4.3.6 `(maybe-foldlM f i xs)`

Monadic fold over the elements of a sequence `xs`, associating to the left (i.e. from left to right) in the maybe monad.

4.3.7 `(elem? x xs)`

Returns true if `x` is an element of the sequence `xs`

4.3.8 `head`

Backwards compatible alias for `first`.

4.3.9 `tail`

Backwards compatible alias for `rest`.

4.3.10 `(read s)`

Reads a radicle value from a string.

4.3.11 `(read-many s)`

Reads many radicle values from a string.

4.3.12 `(<= x y)`

Test if `x` is less than or equal to `y`.

4.4 `prelude/patterns`

Pattern matching is first-class in radicle so new patterns can easily be defined. These are the most essential.

4.4.1 `(match-pat pat v)`

The pattern matching dispatch function. This function defines how patterns are treated in `match` expressions. Atoms are treated as bindings. Numbers, keywords and strings are constant patterns. Dicts of patterns match dicts whose values at those keys match those patterns. Vectors of patterns match vectors of the same length, pairing the patterns and elements by index.

4.4.2 `(_ v)`

The wildcard pattern.

4.4.3 `(/? p)`

Predicate pattern. Takes a predicate function as argument. Values match against this pattern if the predicate returns a truthy value.

4.4.4 `(/as var pat)`

As pattern. Takes a variable and a sub-pattern. If the subpattern matches then the whole pattern matches and furthermore the variable is bound to the matched value.

4.4.5 `(/cons x-pat xs-pat)`

A pattern for sequences with a head and a tail.

4.4.6 `(/nil v)`

Empty-sequence pattern. Matches `[]` and `(list)`

4.4.7 `(/just pat)`

Pattern which matches `[:just x]`.

4.4.8 `(/member vs)`

Matches values that are members of a structure.

4.5 `prelude/bool`

Functions for dealing with truthiness and #f.

4.5.1 `(not x)`

True if `x` is `#f`, false otherwise.

4.5.2 `(and x y)`

Returns `y` if `x` is not `#f`, otherwise returns `x`

4.5.3 `(all xs)`

Checks that all the items of a list are truthy.

4.5.4 `(and-predicate f g)`

Pointwise conjunction of predicates.

4.6 `prelude/seq`

Functions for manipulating sequences, that is lists and vectors.

4.6.1 `(empty? seq)`

True if `seq` is empty, false otherwise.

4.6.2 `(seq? x)`

Returns `#t` if `x` is a list or a vector.

4.6.3 (reverse xs)

Returns the reversed sequence `xs`.

4.6.4 (filter pred ls)

Returns `ls` with only the elements that satisfy `pred`.

4.6.5 (take-while pred ls)

Returns all elements of a sequence `ls` until one does not satisfy `pred`

4.6.6 (starts-with? s prefix)

Returns `#t` if `prefix` is a prefix of the sequence `s`. Also works for strings

4.6.7 (/prefix prefix rest-pat)

Matches sequences that start with `prefix` and bind the rest of that sequence to `rest-pat`. Also works for strings.

4.6.8 (concat ss)

Concatenate a sequence of sequences.

4.7 prelude/list

Functions for creating lists. See also `prelude/seq`.

4.7.1 nil

The empty list.

4.7.2 (range from to)

Returns a list with all integers from `from` to `to`, inclusive.

4.8 prelude/strings

String manipulation functions.

4.8.1 (intercalate sep strs)

Intercalates a string in a list of strings

4.8.2 (unlines x)

Concatenate a list of strings, with newlines in between.

4.8.3 (unwords x)

Concatenate a list of strings, with spaces in between.

4.8.4 (split-by splitter? xs)

Splits a string `xs` into a list of strings whenever the function `splitter?` returns true for a character.

4.8.5 (words xs)

Splits a string `xs` into a list of strings by whitespace characters.

4.8.6 (lines xs)

Splits a string `xs` into a list of strings by linebreaks.

4.8.7 (map-string f xs)

Returns a string consisting of the results of applying `f` to each character of `xs`. Throws a type error if `f` returns something other than a string

4.8.8 (reverse-string str)

Reverses `str`. E.g.: `(reverse-string "abc") == "cba"`.

4.8.9 (ends-with? str substr)

True if `str` ends with `substr`

4.8.10 (pad-right-to l word)

Appends the `word` with whitespace to get to length `l`. If `word` is longer than `l`, the whole word is returned without padding.

4.9 prelude/error-messages

Functions for user facing error messages. Functions should either have a descriptive name or additional comment so that the text can be edited without knowledge of where they are used. To verify changes, tests can be run with `stack exec -- radicle test/all.rad`

4.9.1 (missing-arg arg cmd)

Used for command line parsing when an argument to a command is missing.

4.9.2 (too-many-args cmd)

Used for command line parsing when there are too many arguments passed to a command.

4.9.3 (missing-arg-for-opt opt valid-args)

Used for command line parsing when an option requires an argument.

4.9.4 (invalid-arg-for-opt arg opt valid-args)

Used for command line parsing when the argument for an option is invalid.

4.9.5 (invalid-opt-for-cmd opt cmd)

Used for command line parsing when the option for a given command is unkown

4.9.6 (dir-already-exists dir-name)

`rad project checkout` is aborted, if there is already a directory with the name of the project `dir-name` in the current directory.

4.9.7 (git-clone-failure origin name)

`rad project checkout` is aborted, if cloning the repo `name` form `origin` failed.

4.9.8 (upstream-commit-failure)

`rad project init` is aborted when creating an empty commit failed in preparation to setting the upstream master branch.

4.9.9 (upstream-push-failure)

`rad project init` is aborted when pushing the empty commit failed while setting the upstream master branch.

4.9.10 (item-not-found item item-number)

Any command on a specific patch/issue aborts if it does not exist.

4.9.11 (whole-item-number item)

Any command on a specific patch/issue aborts if the provided `item-number` is not a whole number.

4.9.12 (missing-item-number item action)

Any command on a specific patch/issue aborts if the `item-number` is not provided.

4.9.13 (state-change-failure item state)

On changing the state of a patch/issue if the daemon returned an error.

4.9.14 (no-number-returned item)

On creating a patch/issue, when the creation was successful, but no patch/issue number was returned.

4.9.15 (unknown-command cmd)

An unknown command for an app. E.g. `rad issue foobar`

4.9.16 (unknown-commit commit)

`rad patch propose` aborts if the provided commit is unknown.

4.9.17 (parent-commit-not-master commit)

`rad patch propose` aborts if the provided commit is unknown.

4.9.18 (checkout-new-branch-failure branch)

`rad patch checkout` aborts if creating and switching to the patch branch fails.

4.9.19 (checkout-master-failure)

`rad patch accept` aborts if checking out the master branch fails.

4.9.20 (applying-patch-failure)

`rad patch checkout` aborts if applying the patch to the patch branch fails. Conflicts have to be resolved manually.

4.9.21 (applying-accepted-patch-failure)

`rad patch accept` aborts if applying the patch to master fails. Conflicts have to be resolved manually as well as pushing the commit.

4.9.22 (push-patch-failure)

`rad patch accept` aborts if pushing the patch failed.

4.9.23 (missing-key-file)

Any request to the machine is aborted, when the key file can't be found.

4.9.24 (rad-ipfs-name-publish-failure stderr)

Printed when the `rad ipfs name publish` command in `init-git-ipfs-repo` in `rad-project` fails. Takes stderr of the command as an argument.

4.9.25 (rad-ipfs-key-gen-failure stderr)

Printed when the `rad ipfs key gen` command in `init-git-ipfs-repo` in `rad-project` fails. Takes stderr of the command as an argument.

4.9.26 (process-exit-error command args exit-code stderr)

Printed when the a sub process exits with a non-zero exit code. Includes the stderr output in the message.

4.10 prelude/dict

Functions for manipualting dicts.

4.10.1 (dict-from-seq xs)

Creates a dictionary from a list of key-value pairs.

4.10.2 (keys d)

Given a dict `d`, returns a vector of its keys.

4.10.3 (values d)

Given a dict `d`, returns a vector of its values.

4.10.4 (rekey old-key new-key d)

Change the key from `old-key` to `new-key` in a dict `d`. If `new-key` already exists, it is overwritten.

4.10.5 (modify-map k f d)

Given a key `k`, a function `f` and a dict `d`, applies the function to the value associated to that key.

4.10.6 (delete-many ks d)

Delete several keys `ks` from a dict `d`.

4.10.7 (lookup-default key default dict)

Like `lookup` but returns `default` if the key is not in the map.

4.10.8 (lookup-maybe key dict)

Like `lookup` but returns `[:just x]` if the key is not in the map and `:nothing` otherwise.

4.10.9 (safe-modify-map k f d)

Modifies the association of a value to a key `k` in a dict `d`. The function `f` will receive `[:just v]` if `(eq? (lookup k d) v)`, otherwise it will receive `:nothing`. It should return `[:just new-v]` to change the value, and `:nothing` to remove it.

4.10.10 (group-by f xs)

Partitions the values of a sequence `xs` according to the images under `f`. The partitions are returned in a dict keyed by the return value of `f`.

4.11 prelude/io

Some basic I/O functions.

4.11.1 (print! x)

Print a value to the console or stdout.

4.11.2 (shell! command to-write)

Executes `command` using the shell with `to-write` as input. Stdout and stderr are inherited. WARNING: using `shell!` with unsanitized user input is a security hazard! Example: `(shell! "ls -Glah" "")`.

4.11.3 (process! command args to-write)

Executes `command` using `execvp` with `to-write` as input. Stdout and stderr are inherited. See `man exec` for more information on `execvp`. Returns `:ok` if the process exited normally and `[:error n]` otherwise. Example: `(process! "ls" ["-Glah"] "")`.

4.11.4 (read-line!)

Read a single line of input and interpret it as radicle data.

4.11.5 (read-file-value! file)

Read a single radicle value from a file.

4.11.6 (read-file-values! file)

Read many radicle values from a file.

4.11.7 (shell-with-stdout! command to-write)

Like `shell!`, but captures the stdout and returns it.

4.11.8 (shell-no-stdin! command to-write)

Like `shell!`, but inherits stdin. WARNING: using `shell!` with unsanitized user input is a security hazard! Example: `(shell-no-stdin! "ls -Glah")`.

4.11.9 (write-file! filename contents)

Write `contents` to file `filename`.

4.11.10 (process-with-stdout! command args to-write)

Like `process!`, but captures stdout.

4.11.11 (process-with-stdout-stderr-exitcode! command args to-write)

Like `process-with-stdout!`, but returns a vec `[stdout stderr exitcode]`. `exitcode` is either `:ok` or `[:error n]` where `n` is a number.

4.11.12 (process-with-stdout-strict! command args to-write)

Like `process-with-stdout!`, but prints an error message and exits if the command fails.

4.11.13 (init-file-dict! file)

Initiate a file with an empty dict, but only if the file doesn't already exist.

4.11.14 (read-file-key! file k)

Read a file key. Assumes that the file contents is a serialised dict.

4.11.15 (write-file-key! file k v)

Write a key to a file. Assumes that the file contents is a serialised dict.

4.11.16 (delete-file-key! file k)

Delete a key from a file. Assumes that the file contents is a serialised dict.

4.11.17 (ls!)

List the contents of the current working directory

4.11.18 (modify-file! file f)

Modified the value stored in a file according to the function `f`.

4.11.19 (install-fake-filesystem! files)

Installs a fake for `read-file!` that simulates the presence of files in the `files` dictionary.

If `(read-file! path)` is called and `path` is a key in `files` then the value from `files` is returned. Otherwise the original `read-file!` is used.

This requires the `prelude/test/primitive-stub` script to be loaded.

4.11.20 (prompt! prompt)

Ask for user input with a prompt.

4.12 prelude/exception

Tests for exceptions.

4.13 prelude/set

Sets, built using dicts.

4.13.1 empty

An empty set.

4.13.2 (insert x s)

Insert a value into a set.

4.13.3 (delete x s)

Delete a value from a set.

4.13.4 member?

Query if a value is an element of a set.

4.13.5 (to-vec s)

Convert a set to a vector.

4.13.6 (from-seq xs)

Create a set from a sequence.

4.13.7 (key-set d)

The set of keys of a dict.

4.13.8 (subset? xs ys)

Checks if `xs` is a subset of `ys`.

4.14 prelude/ref

Functions for dealing with reference cells.

4.14.1 (modify-ref r f)

Modify `r` by applying the function `f`. Returns the new value.

4.15 prelude/lens

Functional references.

4.15.1 (make-lens g s)

Makes a lens out of a getter and a setter.

4.15.2 (view lens target)

View a value through a lens.

4.15.3 (set lens new-view target)

Set a value though a lens.

4.15.4 id-lens

The identity lens.

4.15.5 (.. lens1 lens2)

Compose two lenses.

4.15.6 (... lenses)

Compose multiple lenses.

4.15.7 (over lens f target)

Modify a value through a lens.

4.15.8 (@ k)

Returns a lens targetting keys of dicts.

4.15.9 (@def k default)

Returns a lens targetting keys of dicts with a default value for getting if the key does not exist in the target.

4.15.10 (@nth n)

Lenses into the nth element of a vector

4.15.11 (view-ref r lens)

Like `view`, but for refs.

4.15.12 (set-ref r lens v)

Like `set`, but for refs.

4.15.13 (over-ref r lens f)

Like `over`, but for refs.

4.16 prelude/io-utils

IO-related utilities

4.16.1 (fzf-select! xs)

Select one of many strings with `fzf`. Requires that `fzf` be on the path. Returns `[:just x]` where `x` is the selected string, or `:nothing` if nothing was selected.

4.16.2 (edit-in-editor! orig)

Open `$EDITOR` on a file prepopulated with `orig`. Returns the contents of the edited file when the editor exits.

4.16.3 (get-git-config! key)

Get the value associated with a key in git config.

4.16.4 (set-git-config! key value)

Set the value associated with a key in git config.

4.16.5 (get-git-commit-data! format commit)

Get data from a `commit` via `show` specified by `format`

4.16.6 (get-git-username!)

Get the user name stored in git config.

4.16.7 (process-git-with-exit! args msg)

Processes a git command `args`. If it fails, the message `msg` is shown and the process exits, otherwise `:ok` is passed.

4.16.8 (base-path!)

Returns the base path for storage of radicle related config files. By default this is `$HOME/.config/radicle`. This can be adjusted by setting `$XDG_CONFIG_HOME`.

4.17 prelude/key-management

Providing functions for creating and reading key pairs for signing send commands. Per default, key pairs are stored in `$HOME/.config/radicle/my-keys.rad` this can be adjusted by setting `$XDG_CONFIG_HOME`.

4.17.1 (read-keys!)

Reads the keys stored in `my-keys.rad` or returns `:nothing` if the file doesn't exist.

4.17.2 (get-keys!)

Like `read-keys` but prints an error message and exits the process if no key file was found.

4.17.3 (create-keys!)

Creates a new key pair and stores it in `my-keys.rad`. Returns the full absolute path of the created file.

4.17.4 (set-fake-keys! keys)

Bypass reading the keys from `my-keys.rad`, using instead the provided keys. This is intended for testing.

4.17.5 (use-fake-keys!)

Bypass reading the keys from `my-keys.rad`, using newly-generated ones. This is intended for testing.

4.18 prelude/machine

Functions for simulating remote machines.

4.18.1 (updatable-eval sub-eval)

Given an evaluation function `f`, returns a new one which augments `f` with a new command `(update expr)` which evaluates arbitrary expression using `base-eval`.

4.18.2 (eval-fn-app state f arg cb)

Given a state, a function, an argument and a callback, returns the result of evaluating the function call on the arg in the given state, while also calling the callback on the result.

4.18.3 (send-prelude! machine-id)

Send the pure prelude to a machine.

4.18.4 (new-machine!)

Creates a new machine. Returns the machine name.

4.18.5 (send-code! machine-id filename)

Send code from a file to a remote machine.

4.18.6 (send! machine-id inputs)

Update a machine with the vector of `inputs` to evaluate. Returns a vector with the evaluation results.

4.18.7 `(query! machine-id expr)`

Send an expression to be evaluated on a machine. Does not alter the machine.

4.18.8 `(install-remote-machine-fake)`

Install test doubles for the `send!`, `query!`, and `new-machine! primitives that use a mutable dictionary to store RSMs. Requires`rad/test/stub-primitives‘ to be loaded

4.18.9 `(send-signed-command! machine machine-id cmd payload)`

Send a command signed by the keys in `my-keys.rad`.

4.18.10 `(catch-daemon! f)`

Catches all `radicle-daemon` related errors and just prints them out to the user.

4.19 `prelude/state-machine`

An eval for running a state-machine with an updatable transition function.

4.20 `prelude/validation`

Functions for creating or combining *validators*, which are functions which return the input unchanged or throw with an error message. These can be used for checking data before accepting it onto a chain.

4.20.1 `(= x)`

Given `x`, returns a validator that checks for equality with `x`.

4.20.2 `(member xs)`

Given a structure, returns a validator which checks for membership in the structure.

4.20.3 `(and vs)`

Given a sequence of validators `vs`, returns a new validator which, given a value, checks if it conforms to all the validators in `vs`.

4.20.4 `(or vs)`

Given a vector of validators `vs`, returns a new validator which, given a value, checks if it conforms to at least one of the `vs`.

4.20.5 (type t)

Checks that a value has a type. Expects a keyword describing the type, as returned by the `type` function.

4.20.6 (pred name p)

Given a description and a predicate, returns a validator that checks if the predicate is true.

4.20.7 (integral n)

Validator for whole numbers.

4.20.8 (optional-key k v)

Given a key `k` and a validator `v`, returns a validator which checks that the value associated to `k` in a dict conforms to `v`. If the key is absent, the validator passes.

4.20.9 (contains k)

Given a value, returns a validator which checks for membership of that value.

4.20.10 (contains-all ks)

Given a vector of keys, returns a validator which checks that a structure contains all of them.

4.20.11 (contains-only ks)

Validator which checks that a dict only contains a subset of a vector of keys.

4.20.12 (key k v)

Combines existence and validity of a key in a dict.

4.20.13 (optional-keys ks)

Given a dict associating keys to validators, returns a validator which checks that the values associated to those keys in a dict conform to the corresponding validators.

4.20.14 (keys d)

Given a dict `d`, returns a validator which checks that a dict contains all the keys that `d` does, and that the associated values a valid according to the associated validators.

4.20.15 (every v)

Given a validator, creates a new validator which checks that all the items in a sequence conform to it.

4.20.16 (uuid x)

Validates UUIDs.

4.20.17 (signed x)

Checks that a value is a dict with `:signature` and `:author` keys, and that the signature is valid for the rest of the dict for that author. The rest of the dict is turned into a string according to `show`.

4.20.18 (timestamp x)

A validator which checks if a string is an ISO 8601 formatted Coordinated Universal Time (UTC) timestamp.

4.20.19 (string-of-max-length max-len)

A validator which checks that it's argument is a string and less than the specified length.

4.20.20 (always-valid x)

A validator that is always valid.

4.21 prelude/util

Utility functions. For the moment just a counter.

4.21.1 (make-counter)

Creates a stateful counter. Returns a dict with two keys: the function at `:next-will-be` will return the next number (without incrementing it), while the function at `:next` increments the number and returns it.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search