A Racket Glossary

Racketfest 2023

Stefan Schwarzer, SSchwarzer.com
info@sschwarzer.com

Berlin, Germany, 2023-03-18

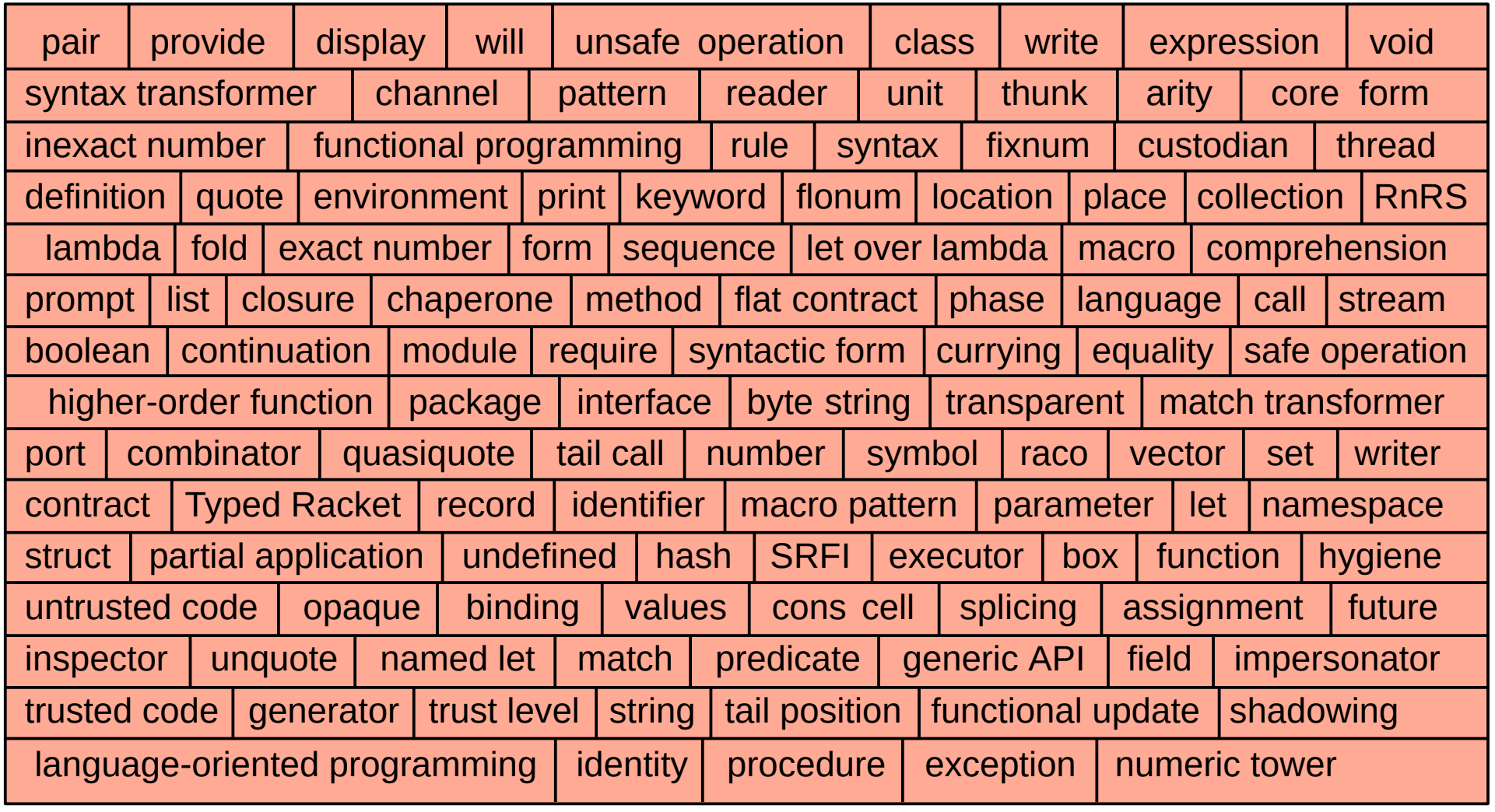
Links

- Rendered at
  https://docs.racket-lang.org/racket-glossary
- Project home
  https://sr.ht/~sschwarzer/racket-glossary
- Github PRs
  https://github.com/sschwarzer/racket-glossary

Motivation

Lots of concepts

- Based on Scheme, and “Scheme is a simple language”
- But Scheme has concepts unfamiliar to many learners
- Racket has *additional* (often complex) concepts
- The terms often don’t make it clear whether something is a fundamental or an advanced concept


pair
provide
display
will
unsafe operation
class
write
expression
void
syntax transformer
channel
pattern
reader
unit
thunk
arity
core form
inexact number
functional programming
rule
syntax
fixnum
custodian
thread
definition
quote
environment
print
keyword
flonum
location
place
collection
RnRS
lambda
fold
exact number
form
sequence
let over lambda
macro
comprehension
prompt
list
closure
chaperone
method
flat contract
phase
language
call
stream
boolean
continuation
module
require
syntactic form
currying
equality
safe operation
higher-order function
package
interface
byte string
transparent
match transformer
port
combinator
quasiquote
tail call
number
symbol
raco
vector
set
writer
contract
Typed Racket
record
identifier
macro pattern
parameter
let
namespace
struct
partial application
undefined
hash
SRFI
executor
box
function
hygiene
untrusted code
opaque
binding
values
cons cell
splicing
assignment
future
inspector
unquote
named let
match
predicate
generic API
field
impersonator
trusted code
generator
trust level
string
tail position
functional update
shadowing
language-oriented programming
identity
procedure
exception
numeric tower

“Read the Racket Guide first”

= “Read the Racket Guide before the Racket Reference”

- Too little: You need the Racket Reference to check details, for example on lists, strings and structs.
- Too much: The Racket Guide itself contains some esoteric/advanced sections, like prompts, aborts, units and inspectors. “Is this the introduction or not?”

“Check out the advanced stuff later”

- If a term/concept is too foreign, you have to read the documentation anyway
- *Then* you find out that the concept is advanced stuff and that you don’t need it 😆

Glossary features

Scribble-based

- The usual ...
  - Nice layout
  - Racket examples
- Project visibility
  - Rendered on package server
  - Integrated with documentation – including search! ☺

Levels

Levels describe the relative importance of concepts when learning Racket

- basic – focus on these if you start with Racket
- intermediate – may be needed for some tasks
- advanced – less important, read this later

Hash

Title

Level: basic

Level

Hashes, also called hash tables, hash maps or dictionaries, map keys to values. For example, the following hash table maps numbers to symbols:

```
> (define my-hash
    (hash 1 'a
          2 'b
          3 'c))
; Look up the value for the key 2.
> (hash-ref my-hash 2)
'b
```

Intro with code examples

Keys and values can be arbitrary values:

```
> (define my-hash
    (hash #(1 2 3) '(a b c)
          '(1 2)   #t
          #f       map))
> (hash-ref my-hash #(1 2 3))
'(a b c)
> (hash-ref my-hash '(1 2))
#t
> (hash-ref my-hash #f)
#<procedure:map>
```

However, usually all keys are of the same type and all values are of the same type.

The API for hashes in Racket is more complicated than for the other compound data structures like lists and vectors. Hashes can differ in the following criteria; all of the combinations are possible:

- Comparison for keys: `equal?`, `eq?`, `eqv?`
- Mutability: immutable, mutable
  Mutability applies to the hash as a whole, not to the mutability of the keys or the values.
- Strength: strong, weak, ephemerons
  This influences when hash entries can be garbage-collected.

These are $3 \times 2 \times 3 = 18$ combinations, but in practice you can almost always get by with this list of just four combinations:

- Comparison for keys: `equal?`, `eq?`
- Mutability: immutable, mutable
- Strength: strong

Simplification for common use cases

Here's an overview of the most important APIs for these four equality/mutability combinations:

Combination	**Construction** (1, 2)	**Set or update value** (3)	**Get value**
`equal?`/immutable	`(hash key1 value1 key2 value2 ...)` or `(make-immutable-hash pair1 pair2 ...)`	`(hash-set hash key value)`	`(hash-ref hash key)`
`eq?`/immutable	`(hasheq key1 value1 key2 value2 ...)` or `(make-immutable-hasheq pair1 pair2 ...)`	`(hash-set hash key value)`	`(hash-ref hash key)`
`equal?`/mutable	`(make-hash pair1 pair2 ...)`	`(hash-set! hash key value)`	`(hash-ref hash key)`
`eq?`/mutable	`(make-hasheq pair1 pair2 ...)`	`(hash-set! hash key value)`	`(hash-ref hash key)`

API information

(1) You can create empty hashes by calling the constructor without arguments. For example, `(hash)` creates an empty immutable hash with `equal?` key comparison.

(2) A *pair* here is a regular Scheme/Racket pair, for example `(cons 1 'a)`. Pairs that contain only literals can also be written as `'(1 . a)`.

(3) Setting or updating a value in an immutable hash may sound contradictory. The solution is that `hash-set` causes a so-called functional update. That is, it returns a new hash with the modification applied and leaves the *hash argument* unchanged. This is the same principle that `cons` or `struct-copy` use.

Warnings: Caveats

- If a hash entry has a mutable key (for example a vector) don't change the key in-place
- Don't change a hash while iterating over it.

See also: References

- Collection, Equality, Functional update, List, Pair, Struct, Vector in this glossary
- Hash Tables in the Racket Guide
- Hash Tables in the Racket Reference

Development

Tools

- Git
- Scribble
- LibreOffice Draw
- Make
- Custom statistics script

Statistics script

```
$ make stats
racket glossary-stats.rkt
Completion stats, ignoring cross references:

basic        |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■···············| 39 of  55 (71 %) done
intermediate |■■■■■■■···········································|  5 of  34 (15 %) done
advanced     |··················································|  0 of  25 ( 0 %) done

total        |■■■■■■■■■■■■■■■■■■■·······························| 44 of 114 (39 %) done

...
```

Great for motivation! 😊
Do something like this in your projects.

Statistics script

```
$ make stats
...
Missing entries for level basic:
  Collection
  Debugging
  Display
  DrRacket
  Form
  Formatting
  Interface (API)
  Module
  Pattern (regular expressions)
  Port
  Raco
  Require
  Scheme
  ...
```

Preparation for glossary entries

- Read Racket Guide entry
- Read Racket Reference entry

Depending on glossary entry:

- Read other documentation
- Experiments
- Ask questions on Racket Discourse

Notes file

`glossary-notes.md`

Notes for each entry, for example:

- Things I want to include/emphasize
- Links to Racket Guide and Reference
- Wikipedia articles
- Blog entries

Challenges

Expanding scope

- Started as “just” Racket glossary
- But Scheme is so similar, and I like Scheme
- Functional Programming concepts are helpful

That’s enough, really! ☺

- More FP concepts (lifting, monoids, monads, ...)
- Recommend libraries

I don’t know everything

- Some things I’m experienced with
- Some things I can find out with experimentation
- From here on, I need help ...
- Obvious idea: I need pull requests – didn’t work
- → Ask for help/feedback on Racket Discourse 💡

Contributions

- I feel more comfortable with writing the main document alone
- On the other hand, everyone can contribute to the notes file, `glossary-notes.md`
- Project is hosted on Sourcehut – but you can send me your suggestions any way you like