Kaappi ⇒ R7RS-small Scheme · Cheatsheet

v0.22.1 · kaappi-lang.org

⇒ LITERALS & LEXICAL

`#t #f` booleans (also `#true #false`); only `#f` is false
`#\a #\space #\x41` characters — `#\x41` is `#\A`
`"a\n b"` string; escapes `\n \t \" \\ \x41;`
`#""C:\path""` raw string — no escapes (SRFI 267)
`42 -7 1/3 6.02e23` exact ints and rationals, floats
`#x1F #b1010 #o17` hex, binary, octal
`#e1.5 #i1/3` force exact / inexact
`'foo |two words|` symbols; `|...|` allows any chars
`#(1 2 3)` vector literal — immutable
`#u8(1 2 255)` bytevector literal
`; line #| block |# #;(datum)` comment forms

Names: `pred?` asks, `mut!` mutates, `a->b` converts. On this sheet `;=>` means "evaluates to".

⇒ DEFINE & LAMBDA

`(define x expr)` bind a value
`(define (f a b) body ...)` define a procedure
`(define (f a . rest) body ...)` rest args as a list
`(lambda (a b) body ...)` anonymous procedure; `(lambda args ...)` takes all args as a list
`(define-values (a b) expr)` bind multiple values
`(case-lambda (formals body ...) ...)` dispatch on arity [case-lambda]
`(set! var expr)` assign an existing binding

```
(define (add a b) (+ a b))
(add 2 3)                  ;=> 5
```

⇒ BINDING FORMS

`(let ((x 1) (y 2)) body ...)` parallel bindings
`(let* ((x 1) (y (+ x 1))) ...)` sequential — each sees the previous
`(letrec ((f ...) (g ...)) ...)` mutually recursive (also `letrec*`)
`(let-values (((a b) expr) ...) ...)` bind multiple values (also `let*-values`)

```
(let loop ((n 5) (acc 1))  ; named let
  (if (= n 0) acc
      (loop (- n 1) (* n acc))))  ;=> 120
```

Tail calls run in constant stack — recurse freely; a named `let` is the idiomatic loop.

⇒ CONDITIONALS & FLOW

`(if test then else)` two-way branch; `else` optional
`(cond (test expr ...) ... (else ...))` first true clause wins; `(test => proc)` passes the value on
`(case key ((d1 d2) e ...) (else ...))` dispatch on `eqv?` against datum lists
`(and e ...)` `(or e ...)` short-circuit; return the deciding value
`(when test body ...)` one-armed `if` (and `unless`)
`(begin e ...)` sequence; value of the last
`(do ((v init step) ...) (test res) body ...)` general loop

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  (else 'composite))  ;=> composite

(do ((i 0 (+ i 1))
     (sum 0 (+ sum i)))
    ((= i 5) sum))    ;=> 10
```

⇒ QUOTE & QUASIQUOTE

`'(a b c)` quoted literal data
`` `(1 ,x ,@xs) `` template: `,` inserts a value, `,@` splices a list
`(let ((x 2) (xs '(3 4))) `(1 ,x ,@xs))` ;=> (1 2 3 4)`

⇒ MACROS

`(define-syntax name (syntax-rules (lits) (pattern template) ...))` hygienic rewrite rules; `...` repeats the preceding pattern
`(let-syntax ...)` `(letrec-syntax ...)` local macros
`(syntax-error msg args ...)` fail at expansion time

```
(define-syntax my-when
  (syntax-rules ()
    ((my-when test body ...)
     (if test (begin body ...)))))
```

⇒ RECORDS

```
(define-record-type <point>
  (make-point x y)   ; constructor
  point?             ; predicate
  (x point-x)        ; getter
  (y point-y set-point-y!))
(point-x (make-point 3 4)) ;=> 3
```

⇒ EXCEPTIONS

`(error msg irritant ...)` raise an error object
`(guard (e (test expr ...) ... (else ...)) body ...)` catch: first clause matching e handles it
`(raise obj)` `(raise-continuable obj)` raise any value; continuable handlers may return a replacement
`(with-exception-handler handler thunk)` install a handler around thunk
`(error-object-message e)` `(error-object-irritants e)` inspect (also `error-object?`)

```
(guard (e (#t (error-object-message e)))
  (error "boom"))          ;=> "boom"
```

⇒ MULTIPLE VALUES

`(values a b ...)` return several values
`(call-with-values producer consumer)` consume them
`(let-values (((a b) (values 1 2))) (+ a b))` `;=> 3`

⇒ CONTINUATIONS

`(call/cc proc)` capture the continuation — escape or resume
`(call/ec proc)` one-shot escape, much cheaper (Kaappi)
`(dynamic-wind before thunk after)` guaranteed entry/exit around thunk

⇒ PARAMETERS & PROMISES

`(make-parameter init)` dynamic variable, called to read
`(parameterize ((p v) ...) body ...)` rebind within body
`(delay expr)` promise; `(force p)` runs once, memoizes [lazy]

⇒ LIBRARIES

```
(define-library (my lib)
  (import (scheme base))
  (export hello)
  (begin
    (define (hello) 'hi)))
```

`(import (only (srfi 1) fold))` selective import
`(import (prefix (srfi 1) s:))` prefixed; also `(rename ... (old new))` and `(except ...)`
`(cond-expand (kaappi ...) (else ...))` feature-conditional code
`(include "file.scm")` splice a source file
`(my lib)` loads from `my/lib.sld` on the library path.

⇒ KAAPPI CLI

`kaappi` REPL · `kaappi f.scm` runs · reads stdin
`kaappi compile f.scm -o app` native binary via LLVM
`kaappi check f.scm` static analysis; KP-code lints
`kaappi test tests/` SRFI-64 runner, one subprocess per file
`kaappi fmt --check` canonical formatter (2-space R7RS)
`kaappi explain KP4001` what a diagnostic means, with fix
`kaappi doctor · features --json` install self-check · build info
`--lib-path DIR --sandbox --timeout MS` key flags; sandbox blocks FFI, files, eval
Install: `curl -fsSL https://kaappi-lang.org/install.sh | bash`

⇒ REPL

`kaappi>` prompt; unbalanced parens continue on the next line
`_` the last result
`,help ,quit ,version` meta-commands start with a comma
`,time e ,type e ,describe s ,apropos str` inspect
`,expand e ,dis e ,profile e` macro-expand · bytecode · profile
`,import (srfi 1) ,load f.scm` bring in libraries and files
`,break f ,step e` debugger: `step next continue locals backtrace`
`~/.kaappi/history · ~/.kaappi/config` persistence; Ctrl-D exits

⇒ THOTTAM · PACKAGES

`thottam install kaappi-web` with deps, into `~/.kaappi/lib`
`thottam install PKG@v1.0.0` pin a tag; ranges: `@">=0.2.0"`
`thottam install PKG::URL` install from any git URL
`thottam remove · list · update [PKG]` manage installed packages
`thottam verify · --locked install` check / enforce `~/.kaappi/thottam.lock`
`kaappi.pkg` manifest: `name version depends build`

Libraries are plain `.sld` files — import after install: `(import (kaappi web))`.

⇒ NUMBERS

`(+ a ...)` `(- a b ...)` `(* a ...)` `(/ a b ...)` arithmetic; exact / yields rationals
`(/ 1 3)` `;=> 1/3`
`(expt 2 64)` `;=> 18446744073709551616`
`(= a b ...)` `(< ...)` `(> ...)` `(<= ...)` `(>= ...)` numeric compare, chainable
`zero? positive? negative? odd? even?` sign and parity
`(abs n)` `(min a ...)` `(max a ...)`
`(quotient a b)` `(remainder a b)` `(modulo a b)` integer division; `modulo` follows the divisor's sign
`(floor/ a b)` `(truncate/ a b)` two-value division (also `floor-quotient` etc.)
`(gcd a ...)` `(lcm a ...)`
`(floor x)` `(ceiling x)` `(truncate x)` `(round x)` round halves to even
`(expt a b)` `(square a)` `(exact-integer-sqrt n)` powers; integer sqrt with remainder
`(sqrt x)` `(exp x)` `(log x b?)` `(sin x)` `(cos x)` `(tan x)` `(atan y x?)` [inexact]
`finite? infinite? nan?` float predicates [inexact]
`(exact x)` `(inexact x)` convert exactness
`(exact 0.5)` `;=> 1/2`
`(numerator q)` `(denominator q)` `(rationalize x tol)` rational parts
`(number->string n radix?)` `(string->number s radix?)`
`(number->string 255 16)` `;=> "ff"`
`(string->number "ff" 16)` `;=> 255`
`number? integer? rational? real? complex?` tower predicates
`exact? inexact? exact-integer?` exactness predicates

⇒ EQUIVALENCE

`(eq? a b)` identity — symbols, small ints, same object
`(eqv? a b)` `eq?` plus numbers and chars by value
`(equal? a b)` deep structural equality
`(equal? '(1 2) (list 1 2))` `;=> #t`
`(eqv? 1.0 1)` `;=> #f`
`(not x)` `#t` only for `#f`
`(boolean=? a b ...)` `(symbol=? a b ...)`

⇒ TYPE PREDICATES

`number? string? char? symbol? boolean?` atoms
`pair? null? list? vector? bytevector?` compound
`procedure? port? eof-object? promise?` other

⇒ PAIRS & LISTS

`(cons a d)` `(car p)` `(cdr p)` build · first · rest
`(cons 1 '(2 3))` `;=> (1 2 3)`
`(set-car! p v)` `(set-cdr! p v)` mutate a pair
`(list a ...)` `(make-list n fill?)`
`(length ls)` `(reverse ls)` `(append ls ...)`
`(list-ref ls k)` `(list-tail ls k)` `(list-set! ls k v)` zero-based access
`(map f ls ...)` `(for-each f ls ...)` apply elementwise / for effect
`(map (lambda (x) (* x x)) '(1 2 3))` `;=> (1 4 9)`
`(member x ls)` `(memv x ls)` `(memq x ls)` find tail by `equal?`/`eqv?`/`eq?`
`(assoc k al)` `(assv k al)` `(assq k al)` association-list lookup
`(assq 'b '((a 1) (b 2)))` `;=> (b 2)`
`(list-copy ls)` shallow copy
`(caar p)` `(cadr p)` ... `(cddddr p)` compound accessors [cxr]

⇒ STRINGS

`(string ch ...)` `(make-string n ch?)`
`(string-length s)` `(string-ref s k)` by codepoint — UTF-8 native
`(string-set! s k ch)` strings are mutable
`(substring s a b)` half-open `[a,b)`
`(substring "hello" 1 4)` `;=> "ell"`
`(string-append s ...)`
`(string->list s)` `(list->string ls)` also `string->vector`, `vector->string`
`(string-copy s a? b?)` `(string-copy! to at from)` `(string-fill! s ch)`
`(string-map f s ...)` `(string-for-each f s ...)`
`string=? string<? string>? string<=? string>=?` lexicographic compare
`string-ci=? ...` `string-upcase` `string-downcase` `string-foldcase` case ops [char]
`(string->symbol s)` `(symbol->string sym)`

⇒ CHARACTERS

`(char->integer c)` `(integer->char n)` codepoint conversion
`(char->integer #\A)` `;=> 65`
`char=? char<? char>? char<=? char>=?` compare (`char-ci...` [char])
`char-alphabetic? char-numeric? char-whitespace?` classify [char]
`char-upper-case? char-lower-case?` [char]
`(char-upcase c)` `(char-downcase c)` `(char-foldcase c)` [char]
`(digit-value c)` digit as number, else `#f` [char]

⇒ VECTORS

`(vector a ...)` `(make-vector n fill?)`
`(vector-length v)` `(vector-ref v k)` `(vector-set! v k x)`
`(vector->list v a? b?)` `(list->vector ls)`
`(vector->list #(1 2 3) 1)` `;=> (2 3)`
`(vector-copy v a? b?)` `(vector-copy! to at from)` `(vector-fill! v x)`
`(vector-append v ...)`
`(vector-map f v ...)` `(vector-for-each f v ...)`
`(vector-map + #(1 2) #(10 20))` `;=> #(11 22)`

⇒ BYTEVECTORS

`(bytevector b ...)` `(make-bytevector n fill?)` bytes 0–255
`(bytevector-length bv)` `(bytevector-u8-ref bv k)` `(bytevector-u8-set! bv k b)`
`(bytevector-copy bv a? b?)` `(bytevector-copy! to at from)` `(bytevector-append bv ...)`
`(utf8->string bv a? b?)` `(string->utf8 s a? b?)`
`(utf8->string #u8(104 105))` `;=> "hi"`

⇒ CONTROL

`(apply f a ... args)` call with a list as the final args
`(apply + 1 '(2 3))` `;=> 6`
`(read-error? e)` `(file-error? e)` classify raised errors

⇒ PORTS & I/O

`(display x port?)` `(write x port?)` human-readable vs re-readable [write]
`(newline port?)` `(write-char c)` `(write-string s)`
`(read port?)` read one datum [read]
`(read-line port?)` `(read-char port?)` `(read-string k port?)`
`(read-line (open-input-string "one"))` `;=> "one"`
`(peek-char port?)` look ahead without consuming
`(read-u8)` `(write-u8 b)` `(read-bytevector k)` `(write-bytevector bv)` binary I/O
`(eof-object)` `(eof-object? x)` end-of-input value
`(open-input-string s)` `(open-output-string)` string ports; `(get-output-string p)` extracts
`(read (open-input-string "(1 2)"))` `;=> (1 2)`
`(call-with-input-file f proc)` `(call-with-output-file f proc)` auto-close [file]
`(with-input-from-file f thunk)` `(with-output-to-file f thunk)` redirect current ports [file]
`(open-input-file f)` `(open-output-file f)` [file]
`(close-port p)` `(call-with-port p proc)` `(flush-output-port p?)`
`(current-input-port)` `(current-output-port)` `(current-error-port)` default ports, parameterizable
`(file-exists? f)` `(delete-file f)` [file]
`port? input-port? output-port? textual-port? binary-port?` predicates

⇒ SYSTEM & EVAL

`(exit code?)` terminate [process-context]
`(command-line)` program args as a list [process-context]
`(get-environment-variable name)` one env var; `-variables` for all [process-context]
`(current-second)` `(current-jiffy)` `(jiffies-per-second)` wall clock · monotonic ticks [time]
`(features)` feature symbols — pairs with `cond-expand`
`(load "f.scm")` [load]
`(eval expr env)` `(environment import-set ...)` [eval]
`(real-part z)` `(imag-part z)` `(magnitude z)` `(angle z)` complex accessors [complex]

⇒ HASH TABLES · SRFI 69

`(import (srfi 69))` — for insertion-ordered iteration use SRFI 250 instead.
`(make-hash-table equal? hash?)` create; equality and hash optional
`(hash-table-set! ht k v)` associate key with value
`(hash-table-ref ht k fail?)` look up; calls `fail` thunk on a miss
`(hash-table-ref/default ht k default)` look up with fallback
`(hash-table-update!/default ht k f default)` update via `f`
`(hash-table-delete! ht k)` `(hash-table-exists? ht k)`
`(hash-table-size ht)` `(hash-table-keys ht)` `(hash-table-values ht)`
`(hash-table->alist ht)` `(alist->hash-table al)` convert; also `hash-table-walk`, `hash-table-copy`, `hash-table?`

⇒ BEYOND R7RS-SMALL

`(import (srfi 1))` list library: `fold filter reduce iota any every delete-duplicates ...`
`(import (srfi 18))` OS threads; green fibers are built in

600+ built-in procedures, 170+ SRFIs, JSON · HTTP · SQLite · Postgres · Redis via thottam packages ⇒ kaappi-lang.org/procedures/