Racket

9 Pattern Matching

Pattern Matching in **The Racket Guide** introduces pattern matching.

The `match` form and related forms support general pattern matching on Racket values. See also Regular Expressions for information on regular-expression matching on strings, bytes, and streams.

```
(require racket/match)                                    package: base
```

The bindings documented in this section are provided by the `racket/match` and `racket` libraries, but not `racket/base`.

```
(match val-expr clause ...)                                        syntax

  clause = [pat body ...+]
         | [pat (=> id) body ...+]
         | [pat #:when cond-expr body ...+]
```

Finds the first *pat* that matches the result of *val-expr*, and evaluates the corresponding *body*s with bindings introduced by *pat* (if any). The last *body* in the matching clause is evaluated in tail position with respect to the `match` expression.

To find a match, the *clause*s are tried in order. If no *clause* matches, then the `exn:misc:match?` exception is raised.

An optional `#:when` *cond-expr* specifies that the pattern should only match if *cond-expr* produces a true value. *cond-expr* is in the scope of all of the variables bound in *pat*. *cond-expr* must not mutate the object being matched before calling the failure procedure, otherwise the behavior of matching is unpredictable. See also `failure-cont`, which is a lower-level mechanism achieving the same ends.

An optional `(=> id)` between a *pat* and the *body*s is bound to a *failure procedure* of zero arguments. If this procedure is invoked, it escapes back to the pattern matching expression, and resumes the matching process as if the pattern had failed to match. The *body*s must not mutate the object being matched before calling the failure procedure, otherwise the behavior of matching is unpredictable.

The grammar of *pat* is as follows, where non-italicized identifiers are recognized symbolically (i.e., not by binding).

pat	::=	*id*	match anything, bind identifier
	|	(var *id*)	match anything, bind identifier
	|	_	match anything
	|	*literal*	match literal
	|	(quote *datum*)	match equal? value
	|	(list *lvp* ...)	match sequence of *lvp*s
	|	(list-rest *lvp* ... *pat*)	match *lvp*s consed onto a *pat*
	|	(list-no-order *pat* ...)	match *pat*s in any order
	|	(list-no-order *pat* ... *lvp*)	match *pat*s in any order
	|	(vector *lvp* ...)	match vector of *pat*s
	|	(hash-table (*pat* *pat*) ...)	match hash table
	|	(hash-table (*pat* *pat*) ...+ *ooo*)	match hash table
	|	(cons *pat* *pat*)	match pair of *pat*s
	|	(mcons *pat* *pat*)	match mutable pair of *pat*s
	|	(box *pat*)	match boxed *pat*
	|	(*struct-id* *pat* ...)	match *struct-id* instance
	|	(struct *struct-id* (*pat* ...))	match *struct-id* instance
	|	(regexp *rx-expr*)	match string
	|	(regexp *rx-expr* *pat*)	match string, result with *pat*
	|	(pregexp *px-expr*)	match string
	|	(pregexp *px-expr* *pat*)	match string, result with *pat*
	|	(and *pat* ...)	match when all *pat*s match
	|	(or *pat* ...)	match when any *pat* match
	|	(not *pat* ...)	match when no *pat* matches
	|	(app *expr* pats ...)	match (*expr* value) output values to *pat*s
	|	(? *expr* *pat* ...)	match if (*expr* value) and *pat*s
	|	(quasiquote *qp*)	match a quasipattern
	|	*derived-pattern*	match using extension
literal	::=	#t	match true
	|	#f	match false
	|	*string*	match equal? string
	|	*bytes*	match equal? byte string
	|	*number*	match equal? number
	|	char	match equal? character
	|	keyword	match equal? keyword
	|	regexp	match equal? regexp literal
	|	pregexp	match equal? pregexp literal
lvp	::=	*pat* *ooo*	greedily match *pat* instances
	|	*pat*	match *pat*
qp	::=	*literal*	match literal
	|	*id*	match symbol
	|	(*qp* ...)	match sequences of *qp*s

```
          |  (qp ... . qp)               match qps ending qp
          |  (qp ooo . qp)               match qps beginning with repeated qp
          |  #(qp ...)                   match vector of qps
          |  #&qp                        match boxed qp
          |  ,pat                        match pat
          |  ,@(list lvp ...)            match lvps, spliced
          |  ,@(list-rest lvp ... pat)   match lvps plus pat, spliced
          |  ,@'qp                       match list-matching qp, spliced
ooo      ::= ...                         zero or more; ... is literal
          |  ___                         zero or more
          |  ..k                         k or more
          |  __k                         k or more
```

In more detail, patterns match as follows:

- *id* (excluding the reserved names _, ..., .._, ..*k*, and ..*k* for non-negative integers *k*) or (var *id*) — matches anything, and binds *id* to the matching values. If an *id* is used multiple times within a pattern, the corresponding matches must be the same according to (match-equality-test), except that instances of an *id* in different or and not sub-patterns are independent.

  Examples:

  ```
  > (match '(1 2 3)
      [(list a b a) (list a b)]
      [(list a b c) (list c b a)])
  '(3 2 1)
  > (match '(1 (x y z) 1)
      [(list a b a) (list a b)]
      [(list a b c) (list c b a)])
  '(1 (x y z))
  ```

- _ — matches anything, without binding any identifiers.

  Example:

  ```
  > (match '(1 2 3)
      [(list _ _ a) a])
  3
  ```

- #t, #f, *string*, *bytes*, *number*, *char*, or (quote *datum*) — matches an equal? constant.

  Example:

  ```
  > (match "yes"
      ["no" #f]
      ["yes" #t])
  #t
  ```

- (list *lvp* ...) — matches a list of elements. In the case of (list *pat* ...), the pattern matches a list with as many element as *pat*s, and each element must match the corresponding *pat*. In the more general case, each *lvp* corresponds to a “spliced” list of greedy matches.

For spliced lists, `...` and `___` are aliases for zero or more matches. The `..`*k* and `__`*k* forms are also aliases, specifying *k* or more matches. Pattern variables that precede these splicing operators are bound to lists of matching forms.

Examples:

```
> (match '(1 2 3)
    [(list a b c) (list c b a)])
'(3 2 1)
> (match '(1 2 3)
    [(list 1 a ...) a])
'(2 3)
> (match '(1 2 3)
    [(list 1 a ..3) a]
    [_ 'else])
'else
> (match '(1 2 3 4)
    [(list 1 a ..3) a]
    [_ 'else])
'(2 3 4)
> (match '(1 2 3 4 5)
    [(list 1 a ..3 5) a]
    [_ 'else])
'(2 3 4)
> (match '(1 (2) (2) (2) 5)
    [(list 1 (list a) ..3 5) a]
    [_ 'else])
'(2 2 2)
```

- `(list-rest` *`lvp`* `...` *`pat`*`)` — similar to a `list` pattern, but the final *pat* matches the “rest” of the list after the last *lvp*. In fact, the matched value can be a non-list chain of pairs (i.e., an “improper list”) if *pat* matches non-list values.

  Examples:

  ```
  > (match '(1 2 3 . 4)
      [(list-rest a b c d) d])
  4
  > (match '(1 2 3 . 4)
      [(list-rest a ... d) (list a d)])
  '((1 2 3) 4)
  ```

- `(list-no-order` *`pat`* `...)` — similar to a `list` pattern, but the elements to match each *pat* can appear in the list in any order.

  Example:

  ```
  > (match '(1 2 3)
      [(list-no-order 3 2 x) x])
  1
  ```

- `(list-no-order pat ... lvp)` — generalizes `list-no-order` to allow a pattern that matches multiple list elements that are interspersed in any order with matches for the other patterns.

  Example:

  ```
  > (match '(1 2 3 4 5 6)
      [(list-no-order 6 2 y ...) y])
  '(1 3 4 5)
  ```

- `(vector lvp ...)` — like a `list` pattern, but matching a vector.

  Example:

  ```
  > (match #(1 (2) (2) (2) 5)
      [(vector 1 (list a) ..3 5) a])
  '(2 2 2)
  ```

- `(hash-table (pat pat) ...)` — similar to `list-no-order`, but matching against hash table's key–value pairs.

  Example:

  ```
  > (match #hash(("a" . 1) ("b" . 2))
      [(hash-table ("b" b) ("a" a)) (list b a)])
  '(2 1)
  ```

- `(hash-table (pat pat) ...+ ooo)` — Generalizes `hash-table` to support a final repeating pattern.

  Example:

  ```
  > (match #hash(("a" . 1) ("b" . 2))
      [(hash-table (key val) ...) key])
  '("a" "b")
  ```

- `(cons pat1 pat2)` — matches a pair value.

  Example:

  ```
  > (match (cons 1 2)
      [(cons a b) (+ a b)])
  3
  ```

- `(mcons pat1 pat2)` — matches a mutable pair value.

  Example:

  ```
  > (match (mcons 1 2)
      [(cons a b) 'immutable]
      [(mcons a b) 'mutable])
  'mutable
  ```

- `(box pat)` — matches a boxed value.

  Example:

```
> (match #&1
    [(box a) a])
1
```

- `(struct-id pat ...)` or `(struct struct-id (pat ...))` — matches an instance of a structure type named `struct-id`, where each field in the instance matches the corresponding `pat`. See also `struct*`.

  Usually, `struct-id` is defined with `struct`. More generally, `struct-id` must be bound to expansion-time information for a structure type (see Structure Type Transformer Binding), where the information includes at least a predicate binding and field accessor bindings corresponding to the number of field `pat`s. In particular, a module import or a `unit` import with a signature containing a `struct` declaration can provide the structure type information.

  Examples:

  ```
  (define-struct tree (val left right))

  > (match (make-tree 0 (make-tree 1 #f #f) #f)
      [(tree a (tree b  _ _) _) (list a b)])
  '(0 1)
  ```

- `(struct struct-id _)` — matches any instance of `struct-id`, without regard to contents of the fields of the instance.

- `(regexp rx-expr)` — matches a string that matches the regexp pattern produced by `rx-expr`; see Regular Expressions for more information about regexps.

  Examples:

  ```
  > (match "apple"
      [(regexp #rx"p+") 'yes]
      [_ 'no])
  'yes
  > (match "banana"
      [(regexp #rx"p+") 'yes]
      [_ 'no])
  'no
  ```

- `(regexp rx-expr pat)` — extends the `regexp` form to further constrain the match where the result of `regexp-match` is matched against `pat`.

  Examples:

  ```
  > (match "apple"
      [(regexp #rx"p+(.)" (list _ "l")) 'yes]
      [_ 'no])
  'yes
  > (match "append"
      [(regexp #rx"p+(.)" (list _ "l")) 'yes]
      [_ 'no])
  'no
  ```

- `(pregexp rx-expr)` or `(regexp rx-expr pat)` — like the `regexp` patterns, but if *rx-expr* produces a string, it is converted to a pattern using `pregexp` instead of `regexp`.

- `(and pat ...)` — matches if all of the *pat*s match. This pattern is often used as `(and id pat)` to bind *id* to the entire value that matches *pat*.

  Example:

  ```
  > (match '(1 (2 3) 4)
     [(list _ (and a (list _ ...)) _) a])
  '(2 3)
  ```

- `(or pat ...)` — matches if any of the *pat*s match. **Beware**: the result expression can be duplicated once for each *pat*! Identifiers in *pat* are bound only in the corresponding copy of the result expression; in a module context, if the result expression refers to a binding, then all *pat*s must include the binding.

  Example:

  ```
  > (match '(1 2)
     [(or (list a 1) (list a 2)) a])
  1
  ```

- `(not pat ...)` — matches when none of the *pat*s match, and binds no identifiers.

  Examples:

  ```
  > (match '(1 2 3)
     [(list (not 4) ...) 'yes]
     [_ 'no])
  'yes
  > (match '(1 4 3)
     [(list (not 4) ...) 'yes]
     [_ 'no])
  'no
  ```

- `(app expr pats ...)` — applies *expr* to the value to be matched; the result of the application is matched against *pats*.

  Examples:

  ```
  > (match '(1 2)
     [(app length 2) 'yes])
  'yes
  > (match '(1 2)
     [(app (lambda (v) (split-at v 1)) '(1) '(2)) 'yes])
  'yes
  ```

- `(? expr pat ...)` — applies *expr* to the value to be matched, and checks whether the result is a true value; the additional *pat*s must also match; i.e., `?` combines a predicate application and an `and` pattern. However, `?`, unlike `and`, guarantees that *expr* is matched before any of the *pat*s.

Example:

```
> (match '(1 3 5)
    [(list (? odd?) ...) 'yes])
'yes
```

- (quasiquote *qp*) — introduces a quasipattern, in which identifiers match symbols. Like the quasiquote expression form, unquote and unquote-splicing escape back to normal patterns.

  Example:

  ```
  > (match '(1 2 3)
      [`(1 ,a ,(? odd? b)) (list a b)])
  '(2 3)
  ```

- *derived-pattern* — matches a pattern defined by a macro extension via define-match-expander.

Note that the matching process may destructure the input multiple times, and may evaluate expressions embedded in patterns such as (app expr *pat*) in arbitrary order, or multiple times. Therefore, such expressions must be safe to call multiple times, or in an order other than they appear in the original program.

9.1 Additional Matching Forms

```
(match* (val-expr ...+) clause* ...)                    syntax

  clause* = [(pat ...+) body ...+]
          | [(pat ...+) (=> id) body ...+]
          | [(pat ...+) #:when cond-expr body ...+]
```

Matches a sequence of values against each clause in order, matching only when all patterns in a clause match. Each clause must have the same number of patterns as the number of *val-expr*s.

Examples:

```
> (match* (1 2 3)
   [(_ (? number?) x) (add1 x)])
4
> (match* (15 17)
   [((? number? a) (? number? b))
    #:when (= (+ a 2) b)
    'diff-by-two])
'diff-by-two
```

```
(match/values expr clause clause ...)                   syntax
```

If `expr` evaluates to `n` values, then match all `n` values against the patterns in `clause ...`. Each clause must contain exactly `n` patterns. At least one clause is required to determine how many values to expect from `expr`.

```
(define/match (head args)                                  syntax
  match*-clause ...)

          head  =  id
                |  (head args)

          args  =  arg ...
                |  arg ... . rest-id

           arg  =  arg-id
                |  [arg-id default-expr]
                |  keyword arg-id
                |  keyword [arg-id default-expr]

  match*-clause =  [(pat ...+) body ...+]
                |  [(pat ...+) (=> id) body ...+]
                |  [(pat ...+) #:when cond-expr body ...+]
```

Binds `id` to a procedure that is defined by pattern matching clauses using `match*`. Each clause takes a sequence of patterns that correspond to the arguments in the function header. The arguments are ordered as they appear in the function header for matching purposes.

Examples:

```
> (define/match (fact n)
    [(0) 1]
    [(n) (* n (fact (sub1 n)))])
> (fact 5)
120
```

The function header may also contain optional or keyword arguments, may have curried arguments, and may also contain a rest argument.

Examples:

```
> (define/match ((f x) #:y [y '(1 2 3)])
    [((regexp #rx"p+") `(,a 2 3)) a]
    [(_ _) #f])
> ((f "ape") #:y '(5 2 3))
5
> ((f "dog"))
#f
> (define/match (g x y . rst)
    [(0 0 '()) #t]
    [(5 5 '(5 5)) #t]
    [(_ _ _) #f])
> (g 0 0)
#t
> (g 5 5 5 5)
#t
> (g 1 2)
#f
```

```
(match-lambda clause ...)                                    syntax
```

Equivalent to `(lambda (id) (match id clause ...))`.

```
(match-lambda* clause ...)                                   syntax
```

Equivalent to `(lambda lst (match lst clause ...))`.

```
(match-lambda** clause* ...)                                 syntax
```

Equivalent to `(lambda (args ...) (match* (args ...) clause* ...))`, where the number of `args ...` is computed from the number of patterns appearing in each of the *clause**.

```
(match-let ([pat expr] ...) body ...+)                       syntax
```

Generalizes `let` to support pattern bindings. Each *expr* is matched against its corresponding *pat* (the match must succeed), and the bindings that *pat* introduces are visible in the *body*s.

Example:

```
> (match-let ([(list a b) '(1 2)]
              [(vector x ...) #(1 2 3 4)])
    (list b a x))
'(2 1 (1 2 3 4))
```

```
(match-let* ([pat expr] ...) body ...+)                      syntax
```

Like `match-let`, but generalizes `let*`, so that the bindings of each *pat* are available in each subsequent *expr*.

Example:

```
> (match-let* ([(list a b) '(#(1 2 3 4) 2)]
                [(vector x ...) a])
    x)
'(1 2 3 4)
```

```
(match-let-values ([(pat ...) expr] ...) body ...+)          syntax
```

Like `match-let`, but generalizes `let-values`.

```
(match-let*-values ([(pat ...) expr] ...) body ...+)         syntax
```

Like `match-let*`, but generalizes `let*-values`.

```
(match-letrec ([pat expr] ...) body ...+)                    syntax
```

Like `match-let`, but generalizes `letrec`.

```
(match-define pat expr)                                      syntax
```

Defines the names bound by *`pat`* to the values produced by matching against the result of *`expr`*.

Examples:

```
> (match-define (list a b) '(1 2))
> b
2
```

```
(match-define-values (pat pats ...) expr)                    syntax
```

Like `match-define` but for when expr produces multiple values. Like match/values, it requires at least one pattern to determine the number of values to expect.

Examples:

```
> (match-define-values (a b) (values 1 2))
> b
2
```

```
(exn:misc:match? v) → boolean?                            procedure
  v : any/c
```

A predicate for the exception raised in the case of a match failure.

```
(failure-cont)                                                    syntax
```

Continues matching as if the current pattern failed. Note that unlike use of the => form, this does *not* escape the current context, and thus should only be used in tail position with respect to the match form.

9.2 Extending match

```
(define-match-expander id proc-expr)                              syntax
(define-match-expander id proc-expr proc-expr)
```

Binds *id* to a *match expander*.

The first *proc-expr* sub-expression must evaluate to a transformer that produces a *pat* for match. Whenever *id* appears as the beginning of a pattern, this transformer is given, at expansion time, a syntax object corresponding to the entire pattern (including *id*). The pattern is replaced with the result of the transformer.

A transformer produced by a second *proc-expr* sub-expression is used when *id* is used in an expression context. Using the second *proc-expr*, *id* can be given meaning both inside and outside patterns.

Match expanders are not invoked unless *id* appears in the first position in a sequence. Instead, identifiers bound by define-match-expander are used as binding identifiers (like any other identifier) when they appear anywhere except the first position in a sequence.

For example, to extend the pattern matcher and destructure syntax lists,

```
(define (syntax-list? x)
  (and (syntax? x)
       (list? (syntax->list x))))
(define-match-expander syntax-list
  (lambda (stx)
    (syntax-case stx ()
      [(_ elts ...)
       #'(? syntax-list?
            (app syntax->list (list elts ...)))])))
(define (make-keyword-predicate keyword)
  (lambda (stx)
    (and (identifier? stx)
         (free-identifier=? stx keyword))))
(define or-keyword? (make-keyword-predicate #'or))
(define and-keyword? (make-keyword-predicate #'and))

> (match #'(or 3 4)
    [(syntax-list (? or-keyword?) b c)
```

```
     (list "OOORRR!" b c)]
    [(syntax-list (? and-keyword?) b c)
     (list "AAANND!" b c)])
'("OOORRR!" #<syntax:59:0 3> #<syntax:59:0 4>)
> (match #'(and 5 6)
    [(syntax-list (? or-keyword?) b c)
     (list "OOORRR!" b c)]
    [(syntax-list (? and-keyword?) b c)
     (list "AAANND!" b c)])
'("AAANND!" #<syntax:60:0 5> #<syntax:60:0 6>)
```

And here is an example showing how `define-match-expander`-bound identifiers are not treated specially unless they appear in the first position of pattern sequence.

```
(define-match-expander nil
  (λ (stx) #''())
  (λ (stx) #''()))
(define (len l)
  (match l
    [nil 0]
    [(cons hd tl) (+ 1 (len tl))]))

> (len nil)
0
> (len (cons 1 nil))
0
> (len (cons 1 (cons 2 nil)))
0
```

`prop:match-expander` : `struct-type-property?` value

A structure type property to identify structure types that act as match expanders like the ones created by `define-match-expander`.

The property value must be an exact non-negative integer or a procedure of one or two arguments. In the former case, the integer designates a field within the structure that should contain a procedure; the integer must be between `0` (inclusive) and the number of non-automatic fields in the structure type (exclusive, not counting supertype fields), and the designated field must also be specified as immutable.

If the property value is a procedure of one argument, then the procedure serves as the transformer for match expansion. If the property value is a procedure of two arguments, then the first argument is the structure whose type has `prop:match-expander` property, and the second argument is a syntax object as for a match expander..

If the property value is a assignment transformer, then the wrapped procedure is extracted with `set!-transformer-procedure` before it is called.

This binding is provided `for-syntax`.

```
prop:legacy-match-expander : struct-type-property?                value
```

Like `prop:match-expander`, but for the legacy match syntax.

This binding is provided `for-syntax`.

```
(match-expander? v) → boolean?                                 procedure
  v : any/c
(legacy-match-expander? v) → boolean?                          procedure
  v : any/c
```

Predicates for values which implement the appropriate match expander properties.

```
(match-equality-test) → (any/c any/c . -> . any)               parameter
(match-equality-test comp-proc) → void?
  comp-proc : (any/c any/c . -> . any)
```

A parameter that determines the comparison procedure used to check whether multiple uses of an identifier match the “same” value. The default is `equal?`.

```
(match/derived val-expr original-datum clause ...)                syntax
(match*/derived (val-expr ...) original-datum clause* ...)        syntax
```

Like `match` and `match*` respectively, but includes a sub-expression to be used as the source for all syntax errors within the form. For example, `match-lambda` expands to `match/derived` so that errors in the body of the form are reported in terms of `match-lambda` instead of `match`.

9.3 Library Extensions

```
(== val comparator)                                               syntax
(== val)
```

A match expander which checks if the matched value is the same as *val* when compared by *comparator*. If *comparator* is not provided, it defaults to `equal?`.

Examples:

```
> (match (list 1 2 3)
    [(== (list 1 2 3)) 'yes]
    [_ 'no])
'yes
> (match (list 1 2 3)
    [(== (list 1 2 3) eq?) 'yes]
    [_ 'no])
'no
> (match (list 1 2 3)
    [(list 1 2 (== 3 =)) 'yes]
    [_ 'no])
'yes
```

```
(struct* struct-id ([field pat] ...))                    syntax
```

A match pattern form that matches an instance of a structure type named *struct-id*, where the field *field* in the instance matches the corresponding *pat*.

Any field of *struct-id* may be omitted, and such fields can occur in any order.

Examples:

```
(define-struct tree (val left right))

> (match (make-tree 0 (make-tree 1 #f #f) #f)
    [(struct* tree ([val a]
                    [left (struct* tree ([right #f] [val b]))]))
     (list a b)])
'(0 1)
```