

Pattern Matching for Scheme

Andrew K. Wright

March 1996

Department of Computer Science – MS 132
Rice University
6100 Main Street
Houston, Texas 77005-1892

Contents

1	Match	1
1.1	Definition	1
1.1.1	Patterns	3
1.1.2	Match Failure	4
1.2	Code Generation	4
1.3	Examples	5

1. Match

1.1 Definition

The complete syntax of the pattern matching expressions follows:

```
exp ::= (match exp clause ...)
      (match-lambda clause ...)
      (match-lambda* clause ...)
      (match-let ([pat exp] ...) body)
      (match-let* ([pat exp] ...) body)
      (match-letrec ([pat exp] ...) body)
      (match-let var ([pat exp] ...) body)
      (match-define pat exp)

clause ::= [pat body] | [pat (=> identifier) body]
```

Figure 1.1 gives the full syntax for patterns. The next subsection describes the various patterns.

The **match-lambda** and **match-lambda*** forms are convenient combinations of **match** and **lambda**, and can be explained as follows:

```
(match-lambda [pat body] ...) = (lambda (x) (match x [pat body] ...))
(match-lambda* [pat body] ...) = (lambda x (match x [pat body] ...))
```

where x is a unique variable. The **match-lambda** form is convenient when defining a single argument function that immediately destructures its argument. The **match-lambda*** form constructs a function that accepts any number of arguments; the patterns of **match-lambda*** should be lists.

The **match-let**, **match-let***, **match-letrec**, and **match-define** forms generalize Scheme's **let**, **let***, **letrec**, and **define** expressions to allow patterns in the binding position rather than just variables. For example, the following expression:

```
(match-let ((x y z) (list 1 2 3))) $body$)
```

binds x to 1, y to 2, and z to 3 in *body*. These forms are convenient for destructuring the result of a function that returns multiple values. As usual for **letrec** and **define**, pattern variables bound by **match-letrec** and **match-define** should not be used in computing the bound value.

The **match**, **match-lambda**, and **match-lambda*** forms allow the optional syntax `(=> identifier)` between the pattern and the body of a clause. When the pattern match for such a clause succeeds, the *identifier* is bound to a *failure procedure* of zero arguments within the *body*. If this procedure is invoked, it jumps back to the pattern matching expression, and resumes the matching process as if the pattern had failed to match. The *body* must not mutate the object being matched, otherwise unpredictable behavior may result.

		<i>Pattern :</i>	<i>Matches :</i>
<i>pat</i>	::=	<i>identifier</i>	anything, and binds <i>identifier</i> as a variable
		-	anything
		()	itself (the empty list)
		#t	itself
		#f	itself
		<i>string</i>	an <i>equal?</i> string
		<i>number</i>	an <i>equal?</i> number
		<i>character</i>	an <i>equal?</i> character
		' <i>s-expression</i>	an <i>equal?</i> s-expression
		' <i>symbol</i>	an <i>equal?</i> symbol (special case of s-expression)
		(<i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	a proper list of <i>n</i> elements
		(<i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>} . <i>pat</i> _{<i>n</i>+1})	a list of <i>n</i> or more elements
		(<i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>} <i>pat</i> _{<i>n</i>+1} ...)	a proper list of <i>n</i> or more elements ¹
		(<i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>} <i>pat</i> _{<i>n</i>+1} .. <i>k</i>)	a proper list of <i>n</i> + <i>k</i> or more elements
		#(<i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	a vector of <i>n</i> elements
		#& <i>pat</i>	a box
		(\$ <i>struct pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	a structure
		(and <i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	if all of <i>pat</i> ₁ through <i>pat</i> _{<i>n</i>} match
		(or <i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	if any of <i>pat</i> ₁ through <i>pat</i> _{<i>n</i>} match
		(not <i>pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	if none of <i>pat</i> ₁ through <i>pat</i> _{<i>n</i>} match
		(? <i>predicate pat</i> ₁ ... <i>pat</i> _{<i>n</i>})	if <i>predicate</i> true and <i>pat</i> ₁ through <i>pat</i> _{<i>n</i>} all match
		(set! <i>identifier</i>)	anything, and binds <i>identifier</i> as a setter
		(get! <i>identifier</i>)	anything, and binds <i>identifier</i> as a getter
		' <i>qp</i>	a quasipattern
		<i>Quasipattern:</i>	<i>Matches :</i>
<i>qp</i>	::=	()	itself (the empty list)
		#t	itself
		#f	itself
		<i>string</i>	an <i>equal?</i> string
		<i>number</i>	an <i>equal?</i> number
		<i>character</i>	an <i>equal?</i> character
		<i>identifier</i>	an <i>equal?</i> symbol
		(<i>qp</i> ₁ ... <i>qp</i> _{<i>n</i>})	a proper list of <i>n</i> elements
		(<i>qp</i> ₁ ... <i>qp</i> _{<i>n</i>} . <i>qp</i> _{<i>n</i>+1})	a list of <i>n</i> or more elements
		(<i>qp</i> ₁ ... <i>qp</i> _{<i>n</i>} <i>qp</i> _{<i>n</i>+1} ...)	a proper list of <i>n</i> or more elements
		(<i>qp</i> ₁ ... <i>qp</i> _{<i>n</i>} <i>qp</i> _{<i>n</i>+1} .. <i>k</i>)	a proper list of <i>n</i> + <i>k</i> or more elements
		#(<i>qp</i> ₁ ... <i>qp</i> _{<i>n</i>})	a vector of <i>n</i> elements
		#& <i>qp</i>	a box
		, <i>pat</i>	a pattern
		,@ <i>pat</i>	a pattern, spliced

Figure 1.1: Pattern Syntax

1.1.1 Patterns

Figure 1.1 gives the full syntax for patterns. Explanations of these patterns follow.

- *identifier* (excluding the reserved names `?`, `$`, `_`, **and**, **or**, **not**, **set!**, **get!**, ..., and `..k` for non-negative integers k): matches anything, and binds a variable of this name to the matching value in the *body*.
- `_`: matches anything, without binding any variables.
- `()`, `#t`, `#f`, *string*, *number*, *character*, *'s-expression*: These constant patterns match themselves, *ie.*, the corresponding value must be *equal?* to the pattern.
- $(pat_1 \dots pat_n)$: matches a proper list of n elements that match pat_1 through pat_n .
- $(pat_1 \dots pat_n . pat_{n+1})$: matches a (possibly improper) list of at least n elements that ends in something matching pat_{n+1} .
- $(pat_1 \dots pat_n pat_{n+1} \dots)$: matches a proper list of n or more elements, where each element of the tail matches pat_{n+1} . Each pattern variable in pat_{n+1} is bound to a list of the matching values. For example, the expression:

```
(match '(let ([x 1] [y 2]) z)
  [('let ((binding values) ...) exp) body])
```

binds `binding` to the list `'(x y)`, `values` to the list `'(1 2)`, and `exp` to `'z` in the body of the **match**-expression. For the special case where pat_{n+1} is a pattern variable, the list bound to that variable may share with the matched value.

- $(pat_1 \dots pat_n pat_{n+1} ..k)$: This pattern is similar to the previous pattern, but the tail must be at least k elements long. The pattern keywords `..0` and `...` are equivalent.
- $\#(pat_1 \dots pat_n)$: matches a vector of length n , whose elements match pat_1 through pat_n .
- $\\&pat$: matches a box containing something matching `pat`.
- $(\$ struct pat_1 \dots pat_n)$: matches a structure declared with **define-structure** or **define-const-structure**.
- **(and** $pat_1 \dots pat_n$): matches if all of the subpatterns match. This pattern is often used as **(and** $x pat$) to bind x to the entire value that matches `pat` (cf. “as-patterns” in ML or Haskell).
- **(or** $pat_1 \dots pat_n$): matches if any of the subpatterns match. At least one subpattern must be present. All subpatterns must bind the same set of pattern variables.
- **(not** $pat_1 \dots pat_n$): matches if none of the subpatterns match. The subpatterns may not bind any pattern variables.
- $(? predicate pat_1 \dots pat_n)$: In this pattern, `predicate` must be an expression evaluating to a single argument function. This pattern matches if `predicate` applied to the corresponding value is true, and the subpatterns $pat_1 \dots pat_n$ all match. The `predicate` should not have side effects, as the code generated by the pattern matcher may invoke predicates repeatedly in any order. The `predicate` expression is bound in the same scope as the match expression, *ie.*, free variables in `predicate` are not bound by pattern variables.
- **(set!** *identifier*): matches anything, and binds *identifier* to a procedure of one argument that mutates the corresponding field of the matching value. This pattern must be nested within a pair, vector, box, or structure pattern. For example, the expression:

```
(define x (list 1 (list 2 3)))
(match x [_ (_ (set! setit))] (setit 4))
```

mutates the *cadadr* of *x* to 4, so that *x* is '(1 (2 4)).

- **(get! *identifier*):** matches anything, and binds *identifier* to a procedure of zero arguments that accesses the corresponding field of the matching value. This pattern is the complement to **set!**. As with **set!**, this pattern must be nested within a pair, vector, box, or structure pattern.
- **Quasipatterns:** Quasiquote introduces a quasipattern, in which identifiers are considered to be symbolic constants. Like Scheme's quasiquote for data, **unquote** (,) and **unquote-splicing** (,@) escape back to normal patterns.

1.1.2 Match Failure

If no clause matches the value, the default action is to invoke the procedure *match:error* with the value that did not match. The default definition of *match:error* calls *error* with an appropriate message:

```
> (match 1 [2 2])
```

Error: no clause matched 1.

For most situations, this behavior is adequate, but it can be changed either by redefining *match:error*, or by altering the value of the variable *match:error-control*. Valid values for *match:error-control* are:

<i>match:error-control</i> :	error action:
'error (default)	call (match:error unmatched-value)
'match	call (match:error unmatched-value '(match expression ...))
'fail	call <i>match:error</i> or die in <i>car</i> , <i>cdr</i> , ...
'unspecified	return unspecified value

Setting *match:error-control* to **'match** causes the entire match expression to be quoted and passed as a second argument to *match:error*. The default definition of *match:error* then prints the match expression before calling *error*; this can help identify which expression failed to match. This option causes the macros to generate somewhat larger code, since each match expression includes a quoted representation of itself.

Setting *match:error-control* to **'fail** permits the macros to generate faster and more compact code than **'error** or **'match**. The generated code omits *pair?* tests when the consequence is to fail in *car* or *cdr* rather than call *match:error*.

Finally, if *match:error-control* is set to **'unspecified**, non-matching expressions will either fail in *car* or *cdr*, or return an unspecified value. This results in still more compact code, but is unsafe.

1.2 Code Generation

Pattern matching macros are compiled into **if**-expressions that decompose the value being matched with standard Scheme procedures, and test the components with standard predicates. Rebinding or lexically shadowing the names of any of these procedures will change the semantics of the **match** macros. The names that should not be rebound or shadowed are:


```

null? pair? number? string? symbol? boolean? char? procedure? vector? box? list?
equal?
car cdr cadr caddr ...
vector-length vector-ref
unbox
reverse length call/cc

```

Additionally, the code generated to match a structure pattern like ($\$ Foo pat_1 \dots pat_n$) refers to the names `Foo?`, `Foo-1` through `Foo-n`, and `set-Foo-1!` through `set-Foo-n!`. These names also should not be shadowed.

1.3 Examples

This section illustrates the convenience of pattern matching with some examples. The following function recognizes some s-expressions that represent the standard Y operator:

```

(define Y?
  (match-lambda
    [(lambda (f1)
      (lambda (y1)
        (((lambda (x1) (f2 (lambda (z1) ((x2 x3) z2))))
          (lambda (a1) (f3 (lambda (b1) ((a2 a3) b2))))
          y2)))
      (and (symbol? f1) (symbol? y1) (symbol? x1) (symbol? z1) (symbol? a1) (symbol? b1)
        (eq? f1 f2) (eq? f1 f3) (eq? y1 y2)
        (eq? x1 x2) (eq? x1 x3) (eq? z1 z2)
        (eq? a1 a2) (eq? a1 a3) (eq? b1 b2))]
    [_ #f]))

```

Writing an equivalent piece of code in raw Scheme is tedious.

The following code defines abstract syntax for a subset of Scheme, a parser into this abstract syntax, and an unparser.

```

(define-structure (Lam args body))
(define-structure (Var s))
(define-structure (Const n))
(define-structure (App fun args))

(define parse
  (match-lambda
    [(and s (? symbol?) (not 'lambda))
     (make-Var s)]
    [(? number? n)
     (make-Const n)]
    [(lambda (and args ((? symbol?) ...) (not (? repeats?))) body)
     (make-Lam args (parse body))]
    [(f args ...)
     (make-App
      (parse f)
      (map parse args))]
    [x (error 'syntax "invalid expression")]))

(define repeats?
  (lambda (l)

```

```

    (and (not (null? l))
         (or (memq (car l) (cdr l)) (repeats? (cdr l))))))

(define unparses
  (match-lambda
    [($$$$ Var s) s]
    [($$$$ Const n) n]
    [($$$$ Lam args body) '(lambda ,args ,(unparses body))]
    [($$$$ App f args) '(',(unparses f) ,@(map unparses args))]))

```

With pattern matching, it is easy to ensure that the parser rejects *all* incorrectly formed inputs with an error message.

With **match-define**, it is easy to define several procedures that share a hidden variable. The following code defines three procedures, *inc*, *value*, and *reset*, that manipulate a hidden counter variable:

```

(match-define (inc value reset)
  (let ([val 0])
    (list
      (lambda () (set! val (add1 val)))
      (lambda () val)
      (lambda () (set! val 0)))))

```

Although this example is not recursive, the bodies could recursively refer to each other.

The following code is taken from the macro package itself. The procedure *validate-match-pattern* checks the syntax of match patterns, and converts quasipatterns into ordinary patterns.

```

(define validate-match-pattern
  (lambda (p)
    (letrec
      ([name?
        (lambda (x)
          (and (symbol? x)
               (not (dot-dot-k? x))
               (not (memq x '(quasiquote quote unquote unquote-splicing
                             ? _ $\$$ and or not set! get! ...))))))]
      [simple?
        (lambda (x)
          (or (string? x) (boolean? x) (char? x) (number? x) (null? x)))]
      [ordinary
        (match-lambda
          [(? simple? p) p]
          [(? name? p) p]
          ['_ '_]
          [('quasiquote p) (quasi p)]
          [(and p ('quote _)) p]
          [(? pred ps ...) '(and (? ,pred) ,(map ordinary ps))]
          [(? and ps ...) '(and ,(map ordinary ps))]
          [(? or ps ...) '(or ,(map ordinary ps))]
          [(? not ps ...) '(not (or ,(map ordinary ps)))]
          [(? $\$$ (? name? r) ps ...) '($\$$ ,r ,(map ordinary ps))]
          [(and p ('set! (? name?))) p]
          [(and p ('get! (? name?))) p]
          [(p '...) '(,(ordinary p) ..0)]
          [(p (? dot-dot-k? ddk)) '(,(ordinary p) ,ddk)]
          [(x . y) (cons (ordinary x) (ordinary y))]
          [(? vector? p) (apply vector (map ordinary (vector->list p)))]
          [#&p (box (ordinary p))]
          [p (err "invalid pattern at ~a" p)]]]
      [quasi
        (match-lambda
          [(? simple? p) p]
          [(? symbol? p) ('quote ,p)]
          [(? unquote p) (ordinary p)]
          [(? unquote-splicing p) . () (ordinary p)]
          [(? unquote-splicing p) . y (append (ordlist p) (quasi y))]
          [(p '...) '(,(quasi p) ..0)]
          [(p (? dot-dot-k? ddk)) '(,(quasi p) ,ddk)]
          [(x . y) (cons (quasi x) (quasi y))]
          [(? vector? p) (apply vector (map quasi (vector->list p)))]
          [#&p (box (quasi p))]
          [p (err "invalid quasipattern at ~a" p)]]]
      [ordlist
        (match-lambda
          [() ()]
          [(x . y) (cons (ordinary x) (ordlist y))]
          [p (err "invalid unquote-splicing at ~a" p)]]])
    (ordinary p))))

```