Pattern Matching for Scheme

Andrew K. Wright and Bruce F. Duba

Department of Computer Science
Rice University
Houston, TX 77251-1892

Version 1.12, May 24, 1995

Please direct questions or bug reports regarding this software to wright@research.nj.nec.com. The most recent version of this software can be obtained by anonymous FTP from site ftp.nj.nec.com in file pub/wright/match.tar.gz.

1 Pattern Matching for Scheme

Pattern matching allows complicated control decisions based on data structure to be expressed in a concise manner. Pattern matching is found in several modern languages, notably Standard ML, Haskell and Miranda. This document describes several pattern matching macros for Scheme, and an associated mechanism for defining new forms of structured data.

The basic form of pattern matching expression is:

(**match** *exp* [*pat body*] ...)

where *exp* is an expression, *pat* is a pattern, and *body* is one or more expressions (like the body of a **lambda**-expression).[1] The **match** form matches its first subexpression against a sequence of patterns, and branches to the *body* corresponding to the first pattern successfully matched. For example, the following code defines the usual *map* function:

```
(define map
  (lambda (f l)
    (match l
      [() ()]
      [(x . y) (cons (f x) (map f y))])))
```

The first pattern () matches the empty list. The second pattern (*x* . *y*) matches a pair, binding *x* to the first component of the pair and *y* to the second component of the pair.

1.1 Pattern Matching Expressions

The complete syntax of the pattern matching expressions follows:

[1]The notation "⟨*thing*⟩ ..." indicates that ⟨*thing*⟩ is repeated zero or more times. The notation "⟨$thing_1$ | $thing_2$⟩" means an occurrence of either $thing_1$ or $thing_2$. Brackets "[]" are extended Scheme syntax, equivalent to parentheses "()".

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 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 a list or vector. 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.

1.2 Patterns

Figure 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.

($pat_1 \ldots pat_n$): matches a proper list of n elements that match pat_1 through pat_n.

		Pattern :	*Matches* :
pat	::=	*identifier*	anything, and binds *identifier* as a variable
	$\mid$	_	anything
	$\mid$	()	itself (the empty list)
	$\mid$	#t	itself
	$\mid$	#f	itself
	$\mid$	*string*	an *equal?* string
	$\mid$	*number*	an *equal?* number
	$\mid$	*character*	an *equal?* character
	$\mid$	'*s-expression*	an *equal?* s-expression
	$\mid$	'*symbol*	an *equal?* symbol (special case of s-expression)
	$\mid$	$(pat_1 \ldots pat_n)$	a proper list of n elements
	$\mid$	$(pat_1 \ldots pat_n \;.\; pat_{n+1})$	a list of n or more elements
	$\mid$	$(pat_1 \ldots pat_n \; pat_{n+1} \; \underline{..k})$	a proper list of $n+k$ or more elements[a]
	$\mid$	$\#(pat_1 \ldots pat_n)$	a vector of n elements
	$\mid$	$\#(pat_1 \ldots pat_n \; pat_{n+1} \; \underline{..k})$	a vector of $n+k$ or more elements
	$\mid$	#&*pat*	a box
	$\mid$	($ *struct* $pat_1 \ldots pat_n$)	a structure
	$\mid$	(= *field pat*)	a field of a structure
	$\mid$	(**and** $pat_1 \ldots pat_n$)	if all of pat_1 through pat_n match
	$\mid$	(**or** $pat_1 \ldots pat_n$)	if any of pat_1 through pat_n match
	$\mid$	(**not** $pat_1 \ldots pat_n$)	if none of pat_1 through pat_n match
	$\mid$	(? *predicate* $pat_1 \ldots pat_n$)	if *predicate* true and pat_1 through pat_n all match
	$\mid$	(**set!** *identifier*)	anything, and binds *identifier* as a setter
	$\mid$	(**get!** *identifier*)	anything, and binds *identifier* as a getter
	$\mid$	`*qp*	a quasipattern
		Quasipattern:	*Matches* :
qp	::=	()	itself (the empty list)
	$\mid$	#t	itself
	$\mid$	#f	itself
	$\mid$	*string*	an *equal?* string
	$\mid$	*number*	an *equal?* number
	$\mid$	*character*	an *equal?* character
	$\mid$	*identifier*	an *equal?* symbol
	$\mid$	$(qp_1 \ldots qp_n)$	a proper list of n elements
	$\mid$	$(qp_1 \ldots qp_n \;.\; qp_{n+1})$	a list of n or more elements
	$\mid$	$(qp_1 \ldots qp_n \; qp_{n+1} \; \underline{..k})$	a proper list of $n+k$ or more elements
	$\mid$	$\#(qp_1 \ldots qp_n)$	a vector of n elements
	$\mid$	$\#(qp_1 \ldots qp_n \; qp_{n+1} \; \underline{..k})$	a vector of $n+k$ or more elements
	$\mid$	#&*qp*	a box
	$\mid$	,*pat*	a pattern
	$\mid$	,@*pat*	a pattern, spliced

Figure 1: Pattern Syntax

[a] The notation $\underline{..k}$ denotes a keyword consisting of three consecutive dots (ie., "..."), or two dots and an non-negative integer (eg., "..1", "..2"), or three consecutive underscores (ie., "___"), or two underscores and a non-negative integer. The keywords "..k" and "__k" are equivalent. The keywords "...", "___", "..0", and "__0" are equivalent.

($pat_1 \ldots pat_n$. pat_{n+1})**:** matches a (possibly improper) list of at least n elements that ends in something matching pat_{n+1}.

($pat_1 \ldots pat_n$ pat_{n+1} ...)**:** 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.

($pat_1 \ldots pat_n$ pat_{n+1} ___)**:** This pattern means the same thing as the previous pattern.

($pat_1 \ldots pat_n$ 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.

($pat_1 \ldots pat_n$ pat_{n+1} __k)**:** This pattern means the same thing as the previous pattern.

#($pat_1 \ldots pat_n$)**:** matches a vector of length n, whose elements match pat_1 through pat_n.

#($pat_1 \ldots pat_n$ pat_{n+1} ...)**:** matches a vector of length n or more, where each element beyond n matches pat_{n+1}.

#($pat_1 \ldots pat_n$ pat_{n+1} ..k)**:** matches a vector of length $n + k$ or more, where each element beyond n matches pat_{n+1}.

#&*pat***:** matches a box containing something matching *pat*.

($ *struct* $pat_1 \ldots pat_n$)**:** matches a structure declared with **define-structure** or **define-const-structure**. See Section 2.

(= *field pat*)**:** is intended for selecting a field from a structure. "field" may be any expression; it is applied to the value being matched, and the result of this application is matched against *pat*.

(**and** $pat_1 \ldots pat_n$)**:** matches if all of the subpatterns match. At least one subpattern must be present. This pattern is often used as (**and** x *pat*) to bind x to to the entire value that matches *pat* (cf. "as-patterns" in ML or Haskell).

(**or** $pat_1 \ldots pat_n$)**:** 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** $pat_1 \ldots pat_n$)**:** matches if none of the subpatterns match. At least one subpattern must be present. The subpatterns may not bind any pattern variables.

(? *predicate* $pat_1 \ldots pat_n$): 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 $pat_1 \ldots pat_n$ 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.

1.3 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:

match:error-control:	error action:
'error (default)	call (*match:error unmatched-value*)
'match	call (*match:error unmatched-value* '(match expression ...))
'fail	call *match:error* or die in *car*, *cdr*, ...
'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.

2 Data Definition

The ability to define new forms of data proves quite useful in conjunction with pattern matching. This macro package includes a slightly altered[2] version of Chez Scheme's **define-structure** macro for defining new forms of data [1], and a similar **define-const-structure** macro for defining immutable data.

The following expression defines a new kind of data named *struct*:

(**define-structure** (*struct* $arg_1 \ldots arg_n$))

A *struct* is a composite data structure with *n fields* named arg_1 through arg_n. The **define-structure** macro declares the following procedures for constructing and manipulating data of type *struct*:

Procedure Name:	*Function*:
make-*struct*	constructor requiring n arguments
struct?	predicate
struct-arg_1, ..., *struct*-arg_n	named selectors
set-*struct*-arg_1!, ..., set-*struct*-arg_n!	named mutators
struct-1, ..., *struct-n*	numeric selectors
set-*struct-1*!, ..., set-*struct-n*!	numeric mutators

The field name _ (underscore) is special: no named selectors or mutators are defined for such a field. Such *unnamed* fields can only be accessed through the numeric selectors or mutators, or through pattern matching.

A second form of definition:

(**define-structure** (*struct* $arg_1 \ldots arg_n$) ([$init_1$ exp_1] ... [$init_m$ exp_m]))

declares m additional fields $init_1$ through $init_m$ with initial values exp_1 through exp_m. The expressions exp_1 through exp_m are evaluated in order each time make-*struct* is invoked.

Finally, the macro **define-const-structure**:

(**define-const-structure** (*struct* $arg_1 \ldots arg_n$))
(**define-const-structure** (*struct* $arg_1 \ldots arg_n$) ([$init_1$ exp_1] ... [$init_m$ exp_m]))

is similar to **define-structure**, but allows immutable fields. If a field name arg_i is simply a variable, no (named or numeric) mutator is declared for that field. If a field name has the form (*!* x) where x is a variable, then that field is mutable. Hence (**define-structure** (*Foo* a b)) abbreviates (**define-const-structure** (*Foo* (*!* a) (*!* b))).

By default, structures are implemented as vectors whose first component is the name of the structure as a symbol. Thus a *Foo* structure of one field will match both the patterns ($ *Foo* x) and #('Foo x). Setting the variable *match:structure-control* to 'disjoint causes subsequent **define-structure** definitions to create structures that are disjoint from all other data, including vectors. In this case, *Foo* structures will no longer match the pattern #('Foo x).[3]

[2] This macro generates additional numeric selector and mutator names for use by the pattern matcher, recognizes _ as an unnamed field, and optionally allows structures to be disjoint from vectors. Chez Scheme does not provide **define-const-structure**.

[3] Disjoint structures are implemented as vectors whose first component is a unique symbol (an uninterned symbol for Chez Scheme). The procedure *vector?* is modified to return false for such vectors (hence the 'disjoint option cannot be used with Chez Scheme's optimize-level set higher than 1). For completeness the other vector operations (*vector-ref*, *vector-set!*, etc.) should also be modified to reject structures, but we don't bother.

3 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 cdddr ...
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.

4 Examples

This section illustrates the convenience of pattern matching with some examples. The following function recognizes 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 x "invalid expression")]))

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

(define unparse
  (match-lambda
    [($ Var s) s]
    [($ Const n) n]
    [($ Lam args body) `(lambda ,args ,(unparse body))]
    [($ App f args) `(,(unparse f) ,@(map unparse 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 (+ 1 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 *match:validate-pattern* checks the syntax of match patterns, and converts quasipatterns into ordinary patterns.

```
(define match:validate-pattern
  (lambda (pattern)
    (letrec
      ([simple?
        (lambda (x)
          (or (string? x) (boolean? x) (char? x) (number? x) (null? x)))]
       [ordinary
        (match-lambda
          [(? simple? p) p]
          ['_ '_]
          [(? match:pattern-var? p) p]
          [('quasiquote p) (quasi p)]
          [(and p ('quote _)) p]
          [('? pred ps ...) `(? ,pred ,@(map ordinary ps))]
          [('and ps ..1) `(and ,@(map ordinary ps))]
          [('or ps ..1) `(or ,@(map ordinary ps))]
          [('not ps ..1) `(not ,@(map ordinary ps))]
          [('$ (? match:pattern-var? r) ps ...) `($ ,r ,@(map ordinary ps))]
          [(and p ('set! (? match:pattern-var?))) p]
          [(and p ('get! (? match:pattern-var?))) p]
          [(p (? match:dot-dot-k? ddk)) `(,(ordinary p) ,ddk)]
          [(x . y) (cons (ordinary x) (ordinary y))]
          [(? vector? p) (apply vector (map ordinary (vector->list p)))]
          [(? box? p) (box (ordinary (unbox p)))]
          [p (match:syntax-err pattern "syntax error in pattern")])]
       [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 (? match:dot-dot-k? ddk)) `(,(quasi p) ,ddk)]
          [(x . y) (cons (quasi x) (quasi y))]
          [(? vector? p) (apply vector (map quasi (vector->list p)))]
          [(? box? p) (box (quasi (unbox p)))]
          [p (match:syntax-err pattern "syntax error in pattern")])]
       [ordlist
        (match-lambda
          [() ()]
          [(x . y) (cons (ordinary x) (ordlist y))]
          [p (match:syntax-err pattern
               "invalid use of unquote-splicing in pattern")])])
    (ordinary pattern))))
```

5 Known Bugs

A structure pattern like ($ *foo a b c*) is not checked to ensure that there are enough fields present for a *foo* object. This should be fixed in the future.

Acknowledgments

Several members of the Rice programming languages community exercised the implementation and suggested enhancements. We thank Matthias Felleisen, Cormac Flanagan, Amit Patel, and Amr Sabry for their contributions.

References

[1] Dybvig, R. K. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.