

Pattern Matcher

This document describes the use of Erik Hilsdale's pattern matcher `match`, as modified by Kent Dybvig. Additional information is given in the `match` source code.

A `match` expression looks a lot like a `case` expression, except that the keys are replaced with a pattern to be matched against the input. A `match` expression has the general form below.

```
(match input-expr clause)
```

Each clause consists of an input pattern, an optional guard, and a set of expressions.

```
[input-pattern expr1 expr2 ...]
```

or

```
[input-pattern (guard guard-expr ...) expr1 expr2 ...]
```

As with `case`, the input expression is evaluated to produce the input value, and the first clause the input value matches, if any, is selected. The output expressions of the selected clause are evaluated in sequence, and the value of the last expression is returned.

An input value matches a clause if it fits the clause's pattern and passes the clause's guards, if any. Patterns may contain symbolic constants, which must match exactly, and pattern variables, which match any input. Pattern variables are prefixed by commas; symbolic constants are not.

```
(match '(a 17 37)
  [(a ,x) 1]
  [(b ,x ,y) 2]
  [(a ,x ,y) 3]) ⇒ 3
```

The first clause fails to match because there are three items in the input list, and the pattern has only two. The second clause fails because `b` does not match `a`.

In the output expression, the values of the pattern variables are bound to the corresponding pieces of input.

```
(match '(a 17 37)
  [(a ,x) (- x)]
  [(b ,x ,y) (+ x y)]
  [(a ,x ,y) (* x y)]) ⇒ 629
```

When followed by an ellipsis (`...`), a pattern variable represents a sequence of input values.

```
(match '(a 17 37) [(a ,x* ...) x*]) ⇒ (17 37)
```

By convention, we place a `*` suffix on each pattern variable that matches a sequence of input expressions. This is just a convention, however, and not part of the syntax of `match`.

Ellipses can follow a structured pattern containing one or more pattern variables.

```
(match '(say (a time) (stitch saves) (in nine))
  [(say (,x* ,y*) ...) (append x* y*)]) ⇒ (a stitch in time saves nine)
```

Ellipses can be nested, producing sequences of sequences of values.

```
(match '((a b c d) (e f g) (h i) (j))
  [((,x* ,y** ...) ...)
  (list x* y**)]) ⇒ ((a e h j) ((b c d) (f g) (i) ()))
```

Recursion is frequently required while processing an input expression with `match`. Here is a simple definition

of length using match.

```
(define length
  (lambda (ls)
    (match ls
      [() 0]
      [(,x . ,x*) (add1 (length x*))])))
```

Using ellipses may make this more clear.

```
(define length
  (lambda (ls)
    (match ls
      [() 0]
      [(,x ,x* ...) (add1 (length x*))])))
```

Here is a more realistic example of recursion. It also illustrates the use of guards and the use of `error` to signal match errors.

```
(define simple-eval
  (lambda (x)
    (match x
      [,i (guard (integer? i)) i]
      [(+ ,x* ...) (apply + (map simple-eval x*))]
      [(* ,x* ...) (apply * (map simple-eval x*))]
      [(- ,x ,y) (- (simple-eval x) (simple-eval y))]
      [(/ ,x ,y) (/ (simple-eval x) (simple-eval y))]
      [,otherwise (error 'simple-eval "invalid expression ~s" x)])))
```

Try out `simple-eval` using Chez Scheme's `new-cafe` with `simple-eval` as the evaluation procedure:

```
> (new-cafe simple-eval)
>> (+ 1 2 3)
6
>> (+ (- 0 1) (+ 2 3))
4
>> (- 1 2 3)
```

```
Error in simple-eval: invalid expression (- 1 2 3).
Type (debug) to enter the debugger.
>>
```

An even simpler version of the above uses `match`'s "catamorphism" feature to perform the recursion automatically.

```
(define simple-eval
  (lambda (x)
    (match x
      [,i (guard (integer? i)) i]
      [(+ ,[x*] ...) (apply + x*)]
      [(* ,[x*] ...) (apply * x*)]
      [(- ,[x] ,[y]) (- x y)]
      [(/ ,[x] ,[y]) (/ x y)]
      [,otherwise (error 'simple-eval "invalid expression ~s" x)])))
```

In the second version of `simple-eval`, the explicit recursive calls are gone. Instead, the pattern variables have

been written as `,[var]`; this tells `match` to recur on the matching subpart of the input before evaluating the output expressions of the clause. Parentheses may be used in place of brackets, i.e., `(var)`; we use brackets for readability.

Here is another definition of `length`, this time using the catamorphism feature.

```
(define length
  (lambda (x)
    (match x
      [(()) 0]
      [(,x . ,[y]) (+ y 1)])))
```

Since we usually use `match` to write translators from one language to another, we usually need to build an output expression, rather than return an output value. For example, the following converts `let` expressions into equivalent `lambda` applications.

```
(define translate
  (lambda (x)
    (match x
      [(let ((,var* ,expr*) ...) ,body ,body* ...)
       '(lambda ,var* ,body ,@body*) ,@expr*])
      [else (error 'translate "invalid expression: ~s" x)])))

(translate '(let ((x 3) (y 4)) (+ x y))) ⇒ ((lambda (x y) (+ x y)) 3 4)
```

This procedure uses Scheme’s `quasiquote` (backquote), `unquote` (comma), and `unquote-splicing` (comma-at) to piece together the output form. These are not described here, but are described in *The Scheme Programming Language* and “The Revised⁵ Report on Scheme.”

One useful feature of `quasiquote` not described in these documents is a `match` extension that allows ellipses to be used in place of `unquote-splicing`, which often leads to more readable code.

```
(define translate
  (lambda (x)
    (match x
      [(let ((,var* ,expr*) ...) ,body ,body* ...)
       '(lambda ,var* ,body ,body* ...) ,expr* ...])
      [else (error 'translate "invalid expression: ~s" x)])))
```

Sometimes it is useful to explicitly name the operator in a “cata” subpattern. Whereas `,[id* ...]` recurs to the top of the current `match`, `,[cata -> id* ...]` recurs to `cata`. `cata` must evaluate to a procedure that accepts one argument, the input expression, and returns as many values as there are following the `->`, just as `match` does.

This allows processing to be split into several mutually recursive procedures, as in the following parser for the simple language defined by the grammar below.

```
<Prog> → (program <Stmnt>* <Expr>)
<Stmnt> → (if <Expr> <Stmnt> <Stmnt>)
| (set! <var> <Expr>)
<Expr> → <var>
| <integer>
| (if <Expr> <Expr> <Expr>)
| (<Expr> <Expr>*)
```

```
(define parse
  (lambda (x)
```

```

(define Prog
  (lambda (x)
    (match x
      [(program ,[Stmt -> s*] ... ,[Expr -> e])
       '(begin ,s* ... ,e)]
      [,other (error 'parse "invalid program ~s" other)])))

(define Stmt
  (lambda (x)
    (match x
      [(if ,[Expr -> e] ,[Stmt -> s1] ,[Stmt -> s2])
       '(if ,e ,s1 ,s2)]
      [(set! ,v ,[Expr -> e])
       (guard (symbol? v))
       '(set! ,v ,e)]
      [,other (error 'parse "invalid statement ~s" other)])))

(define Expr
  (lambda (x)
    (match x
      [,v (guard (symbol? v)) v]
      [,n (guard (integer? n)) n]
      [(if ,[e1] ,[e2] ,[e3])
       '(if ,e1 ,e2 ,e3)]
      [(,[rator] ,[rand*] ...) '(,rator ,rand* ...)]
      [,other (error 'parse "invalid expression ~s" other)])))

(Prog x))

(parse '(program (set! x 3) (+ x 4))) ⇒ (begin (set! x 3) (+ x 4))

```