1. pmatch

Here is a simple example of **pmatch**, a pattern-matching macro written by Oleg Kiselyov.

```
(define h
  (λ (x y)
    (pmatch `(,x . ,y)
      ((,a . ,b) (guard (number? a) (number? b)) (+ a b))
      (,c (guard (number? (cdr c))) (* (cdr c) (cdr c)))
      (else (* x x)))))

(list (h 1 2) (h 'w 5) (h 6 'w)) ⇒ (3 25 36)
```

In this example, a dotted pair is matched against three different kinds of patterns.

In the first pattern, the value of x is lexically bound to a and the value of y is lexically bound to b. Before the pattern match succeeds, however, an optional guard is run within the scope of a and b. The guard succeeds only if x and y are numbers; if so, the value of the call to h is the sum of x and y.

The second pattern matches against any data value, provided that the the optional guard succeeds. If so, the value of `` `(,x . ,y) `` is lexically bound to c, and the square of y is returned.

If the second pattern fails to match against `` `(,x . ,y) ``, because y is not a number, then the third and final clause is tried. An **else** pattern matches *any* input, and never includes a guard. In this case, the clause returns the square of x, which must be a number in order to avoid a type error at runtime.

Here is the definition of **pmatch**, which is implemented using continuation-passing style macros (Hilsdale and Friedman 2000).

```
(define-syntax pmatch
  (syntax-rules (else guard)
    ((_ (rator rand ...) cs ...)
     (let ((v (rator rand ...)))
       (pmatch v cs ...)))
    ((_ v) (error 'pmatch "failed: ~s" v))
    ((_ v (else e0 e ...)) (begin e0 e ...))
    ((_ v (pat (guard g ...) e0 e ...) cs ...)
     (let ((fk (λ () (pmatch v cs ...))))
       (ppat v pat
         (if (and g ...) (begin e0 e ...) (fk))
         (fk))))
    ((_ v (pat e0 e ...) cs ...)
     (let ((fk (λ () (pmatch v cs ...))))
       (ppat v pat (begin e0 e ...) (fk))))))
```

The first clause ensures that the expression whose value is to be **pmatch**ed against is evaluated only once. The second clause signals an error if none of the patterns match against that value.

The remaining clauses represent the three types of patterns supported by **pmatch**. The first is the trivial **else** clause, which matches against any datum, and which behaves identically to an **else** clause in a **cond** expression. The other two clauses are identical, except that the first one includes a guard containing one or more expressions—if the datum matches against the pattern, the guard expressions are evaluated in left-to-right order. If a guard expression evaluates to `#f`, then it is as if the datum had failed to match against the pattern: the remaining guard expressions are ignored, and the next clause is tried.

(*fk*) is an expression that should be evaluated if the pattern fails to match, or if the pattern matches but the guard returns false.

ppat does the actual pattern matching over constants and pairs.

```
(define-syntax ppat
  (syntax-rules (_ quote unquote)
    ((_ v _ kt kf) kt)
    ((_ v () kt kf) (if (null? v) kt kf))
    ((_ v (quote lit) kt kf) (if (equal? v (quote lit)) kt kf))
    ((_ v (unquote var) kt kf) (let ((var v)) kt))
    ((_ v (x . y) kt kf)
     (if (pair? v)
         (let ((vx (car v)) (vy (cdr v)))
           (ppat vx x (ppat vy y kt kf) kf))
         kf))
    ((_ v lit kt kf) (if (equal? v (quote lit)) kt kf))))
```

The wild-card pattern _ matches against any datum. The second pattern matches against the empty list. The third pattern matches against a quoted value. The fourth pattern matches against *any* value, and binds that value to a lexical variable with the specified identifier. The pattern matches against a pair, tearing it apart, and recursively matching the *car* of the value against the *car* of the pattern. If that succeeds, the *cdr* of the value is recursively matched against the *cdr* of the pattern. (We use **let** to avoid building long *car/cdr* chains.) The last pattern matches against constants, including symbols.

Here is the definition of h after expansion.

```
(define h
  (λ (x y)
    (let ((v `(,x . ,y)))
      (let ((fk (λ ()
                  (let ((fk (λ () (* x x))))
                    (let ((c v))
                      (if (number? (cdr c))
                          (* (cdr c) (cdr c))
                          (fk)))))))
        (if (pair? v)
            (let ((vx (car v)) (vy (cdr v)))
              (let ((a vx))
                (let ((b vy))
                  (if (and (number? a) (number? b))
                      (+ a b)
                      (fk)))))
            (fk))))))
```

References

Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000*, September 5, 2000.