1. Introduction. This is a new version of `pmatch`, called `dmatch`. It is named for Dijkstra's Guarded Commands. How `dmatch` differs from `pmatch` is with more emphasis on error reporting and a simpler specification of patterns. This version of the matcher will show all clauses where the pattern matches and the guards succeed if there are two or more such clauses. We allow for a name to be given to dmatch if an error ensues. Also, a line from the specification has been removed. (see below). Without that line removed, it was impossible for a pattern to be `(quote ,x)`, which might be worth having especially when one writes an interpreter for Scheme, which includes quote as a language form.

The original code was written by Oleg Kiselyov (http://pobox.com/ oleg/ftp/) and this source is based off the code from leanTAP.scm found here:

`http://kanren.cvs.sourceforge.net/kanren/kanren/mini/leanTAP.scm?view=log`

This is a simple, linear pattern matcher. It is efficient (generates code at macro-expansion time) and should work on any R5RS (and R6RS) Scheme system. The overall syntax looks like this:

$$
\begin{aligned}
\mathit{match} &:= (\textbf{dmatch}\ \texttt{exp}\ \mathit{clause}\ldots)\\
&\mid (\textbf{dmatch}\ \texttt{exp}\ \texttt{name}\ \mathit{clause}\ldots)\\
\mathit{clause} &:= (\mathit{pattern}\ \mathit{guard}\ \texttt{exp}\ldots)\\
\mathit{guard} &:= (\textbf{guard}\ \texttt{boolean-expression}\ldots) \mid \epsilon\\
\mathit{pattern} &:= ,\texttt{var}\\
&\mid \texttt{exp}\\
&\mid (\mathit{pattern}_1\ \mathit{pattern}_2\ldots)\\
&\mid (\mathit{pattern}_1\ .\ \mathit{pattern}_2)
\end{aligned}
$$

2. We encapsulate our main code into a module form that hides the implementation details from the rest of the world. However, we use an anonymous module to ensure that loading of the "dmatch.ss" file works without needing to have any special import forms. This makes it as simple to use as any of the other normal Scheme files which have their definitions at the top level, but avoids polluting the namespace.

```
(module (dmatch guard unquote)
  (import (chezscheme))
  (implicit-exports #t)
  ⟨ Package procedures 4 ⟩
  ⟨ Error handling procedures 9 ⟩
  ⟨ Helper procedures 8 ⟩
  ⟨ Helper macros 7 ⟩
  ⟨ Main dmatch rules 3 ⟩)
```

3. Implementation. The main `dmatch` macro simply handles the optional name that we can provide, and passes off control to the auxiliary helpers which do most of the extra work. Our auxiliary macros will give us a package list which is then processed by the `run-a-thunk` procedure.

⟨ Main `dmatch` rules 3 ⟩ ≡
```
(define-syntax dmatch
  (syntax-rules ()
    [(_ v (e ...) ...)
     (let ([pkg* (dmatch-remexp v (e ...) ...)])
       (run-a-thunk 'v v #f pkg*))]
    [(_ v name (e ...) ...)
     (let ([pkg* (dmatch-remexp v (e ...) ...)])
       (run-a-thunk 'v v 'name pkg*))]))
```
This section captures `run-a-thunk`.

This section exports `dmatch`.

See also sections 5and6.

This code is used in section 2.

4. In our case we want to represent a package as a pair of a representation of the pattern and a thunk that will execute the code for that pattern. A package is just the information necessary to run a pattern. In this case, we use the following for our pattern abstraction.

⟨ Package procedures 4 ⟩ ≡
```
(define (pkg cls thk) (cons cls thk))
(define (pkg-clause pkg) (car pkg))
(define (pkg-thunk pkg) (cdr pkg))
```
This section exports `pkg, pkg-thunk, andpkg-clause`.

This code is used in section 2.

5. The first step in processing a `dmatch` syntax is to ensure that we only evaluate the input expression once, which is what the `dmatch-remexp` ensures. We use a `let` to ensure that we pass only an identifier to the `dmatch-aux` macro that does the work of constructing the package list.

⟨ Main `dmatch` rules 5 ⟩ +≡
```
(implicit-exports #t)
(define-syntax dmatch-remexp
  (syntax-rules ()
    [(_ (rator rand ...) cls ...)
     (let ([v (rator rand ...)])
       (dmatch-aux v cls ...))]
    [(_ v cls ...) (dmatch-aux v cls ...)]))
```
This section exports .

6. At each expansion of the `dmatch-aux` macro, we want to create a package list of some type. We have three cases which form two recursive cases and a single base case for our macro rules. The base case is when we have no clauses to match, in which case we can just expand to an empty package list. We then have the normal pattern without a guard; if that pattern matches, we want to add that clause body to the package list as one of the matching clauses. In the case where we have a guard, we want to conditionally add the clause to the package list only if the guard also succeeds.

To do the heavy lifting, we will abstract the actual pattern matching out into another helper macro `ppat` that does the check on the pattern and then expands into one of two forms. This is a sort of patterned `if`. The consequent expression will be the result of the expansion of `ppat` if the pattern was a match, and the alternate expression if it was not. In all cases, the alternate will just be another `dmatch-aux` macro that drops the first pattern and continues the recursive expansion. In the cases when we have a match, we want to either `cons` on the package blindly, or check first for the guarded condition.

To encode the alternative case we use a procedure to avoid needing to do the expansions multiple times, which will make our macro a bit faster.

⟨ Main `dmatch` rules 6 ⟩ +≡

```
(define-syntax dmatch-aux
  (syntax-rules (guard)
    [(_ v) '()]
    [(_ v (pat (guard g ...) e0 e ...) cs ...)
     (let ([fk (lambda () (dmatch-aux v cs ...))])
       (ppat v pat
         (if (not (and g ...))
             (fk)
             (cons (pkg '(pat (guard g ...) e0 e ...) (lambda () e0 e ...))
                   (fk)))
         (fk)))]
    [(_ v (pat e0 e ...) cs ...)
     (let ([fk (lambda () (dmatch-aux v cs ...))])
       (ppat v pat
         (cons (pkg '(pat e0 e ...) (lambda () e0 e ...))
               (fk))
         (fk)))]))
```

This section captures `ppatandpkg`.

This section exports .

7. Now we can consider how to do the actual matching of patterns against values. The `ppat` will do this for us, and we leverage the `syntax-rules` pattern matcher to do most of the work. We have a case for each of the types we may encounter, and then handle them as appropriate. We need to do a bit of tree recursion on our expansion in the pair cases to match the `car` and `cdr` cases.

The use of `equal?` for comparisons of things that are not pairs or variables is important, since we may have vectors or other syntaxes sitting around in there that we want to handle, and these should be compared with `equal?` instead of `eq?`.

⟨ Helper macros 7 ⟩ ≡

```
(define-syntax ppat
  (syntax-rules (unquote)
    [(_ 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)]))
```

This section exports `ppat`.

This code is used in section 2.

8. That completes the work we have to do on the macro side. Now the question becomes, what do we do with all of these package lists? Why do we want them? We want them for two reasons. Firstly, they provide more information about the errors of multiple matches than we might have otherwise. We will deal with how we report this in the next section. Right now, we want to know what to do when we have a package list. In this case, we have only one good cases, which is that we have a package list of a single package, meaning that we have only a single match clause that matched our input. In this case, we should just run the package thunk to evaluate the expression.

We then have two error cases. if the package list is empty, that means we have found no matches, and if the package list has more than one package in it, that means that we have more than one match, which means that our match clauses are ambiguous.

⟨ Helper procedures 8 ⟩ ≡

```
(define (run-a-thunk v-expr v name pkg*)
  (cond
    [(null? pkg*) (no-matching-pattern name v-expr v)]
    [(null? (cdr pkg*)) ((pkg-thunk (car pkg*)))]
    [else (ambiguous-pattern/guard name v-expr v pkg*)]))
```

This section captures `no-matching-pattern, ambiguous-pattern/guard, andpkg-thunk`.

This section exports `run-a-thunk`.

This code is used in section 2.

9. Dealing with errors. We have two error cases to deal with. The first is the simplest, when we have not found a match at all. In this case, the match has failed and we can just report the failure.

⟨ Error handling procedures 9 ⟩ ≡
```
(define (no-matching-pattern name v-expr v)
  (printf "dmatch ~@[~d~] failed~n~d ~d~n" name v-expr v)
  (error 'dmatch "match failed"))
```
This section exports `no-matching-pattern`.

See also section 10.

This code is used in section 2.

10. The other case is when we have an ambiguous pattern set that has lead to some problems. In this case, we report the problem and then we want to print out all of the clauses that matched to aid the user in finding the one to fix.

⟨ Error handling procedures 10 ⟩ +≡
```
(define (ambiguous-pattern/guard name v-expr v pkg*)
  (printf "dmatch ~@[~d~] ambiguous matching clauses~n" name)
  (printf "with ~d evaluating to ~d~n" v-expr v)
  (printf "~n")
  (printf "~{~d~n~}" (map pkg-clause pkg*)))
```
This section captures `pkg-clause`.

This section exports `ambiguous-pattern/guard`.

11. Index.

⟨ Error handling procedures 9, 10 ⟩ Used in section 2.
⟨ Helper macros 7 ⟩ Used in section 2.
⟨ Helper procedures 8 ⟩ Used in section 2.
⟨ Main `dmatch` rules 3, 5, 6 ⟩ Used in section 2.
⟨ Package procedures 4 ⟩ Used in section 2.

Dijkstra Match (DMATCH)

Aug 2012 (Version 2.0)