the matcher – an introduction to the basics

brief

The matcher provides a kind of easy-to-use regular expression facility for lists and symbols, gives us a couple of different ways to map patterns over sets of data and introduces a new form of method definition.

outline

The main engine for the matcher is the *matches* function. In practice, we rarely use this function directly but it is used below to illustrate the use of wild cards & introduce match variables.

In its simplest form the matcher resembles an equals comparison, returning nil for non-equal structures and non-nil for equal structures.

```
cg-user(13): (matches '(a b c) '(p q r))
nil
cg-user(14): (matches '(a b c) '(a b c))
((*bind (a b c)) (*match (a b c)))
```

The matcher allows a wild card symbol "=" to be used in its first argument (the pattern) which matches with anything in its second argument (the data)...

```
cg-user(15): (matches '(a = c) '(a b c))
((*bind (a = c)) (*match (a b c)))
```

The matcher also allows match variables to be used. These are prefixed with a "?". Notice in the example below, the result of matches contains an association between the match variable "x" and the data value "b"...

```
cg-user(16): (matches '(a ?x c) '(a b c))
((x b) (*bind (a ?x c)) (*match (a b c)))
```

Match variables must be used consistently within a match...

```
cg-user(17): (matches '(cat ?x rat ?x) '(cat dog rat frog))
nil
cg-user(18): (matches '(cat ?x rat ?y) '(cat dog rat frog))
((y frog) (x dog) (*bind (cat ?x rat ?y)) (*match (cat dog rat
frog)))

cg-user(19): (matches '(cat ?x rat ?x) '(cat dog rat dog))
((x dog) (*bind (cat ?x rat ?x)) (*match (cat dog rat dog)))

;; wild cards do not remember their matching
cg-user(20): (matches '(cat = rat =) '(cat dog rat frog))
((*bind (cat = rat =)) (*match (cat dog rat frog)))
```

The matcher also has wild cards & match variable forms to match with many data items. The wild card for this is a double-equals symbol "==”...

```
cg-user(21): (matches '(a ==) '(a b c))
((*bind (a ==)) (*match (a b c)))

cg-user(22): (matches '(a ==) '(a))
((*bind (a ==)) (*match (a)))

cg-user(23): (matches '(a ==) '(a b c d e f banana mango blah))
((*bind (a ==)) (*match (a b c d e f banana mango blah)))

cg-user(24): (matches '(a ==) '(aardvaark b c d e f banana mango blah))
nil

cg-user(25): (matches '(a == blah) '(a b c d e f banana mango blah))
((*bind (a == blah)) (*match (a b c d e f banana mango blah)))

cg-user(26): (matches '(a == mango) '(a b c d e f banana mango blah))
nil

cg-user(27): (matches '(a == mango) '(a mango))
((*bind (a == mango)) (*match (a mango)))

cg-user(28): (matches '(a == mango == == == ==) '(a mango))
((*bind (a == mango == == == ==)) (*match (a mango)))
```

The equivalent for match variables is to use a double question mark "??"...

```
cg-user: (matches '(a ??x blah) '(a b c d e f banana mango blah))
((x (b c d e f banana mango)) (*bind (a ??x blah))
 (*match (a b c d e f banana mango blah)))

cg-user: ;; notice ?? is always 'reluctant'...
cg-user: (matches '(a ??x ??y blah)
                  '(a b c d e f banana mango blah))
((y (b c d e f banana mango)) (x nil) (*bind (a ??x ??y blah))
 (*match (a b c d e f banana mango blah)))

cg-user: (matches '(a ??x ?y blah) '(a b c d e f banana mango blah))
((y mango) (x (b c d e f banana)) (*bind (a ??x ?y blah))
 (*match (a b c d e f banana mango blah)))
```

Often we use match variables because we want to refer to the *outside* of the matcher forms. *mlet* is the matcher equivalent of let. Note the use of **#?**

```
cg-user(32): (mlet ('(?x plus ?y) '(5 plus 2))
                   (+ #?x #?y))
7

;; match>> is convenient for working with a mix of match
;; variables and literal symbols

cg-user(33): (mlet ('(?x plus ?y) '(5 plus 2))
                   (match>> '(i am trying to add ?x and ?y)))
(i am trying to add 5 and 2)
```

We most often use #?var forms inside match methods. Match methods are methods which specialise on data patterns. They can take the normal mix of Lisp arguments but their first argument is always a pattern...

```
(defmatch calc ( (?x plus ?y) )
   (+ #?x #?y))

cg-user(35): (calc '(45 plus 123))
168

;; match methods only run when the argument supplied matches their
;; first/pattern argument

cg-user(36): (calc '(mango banana blah))
nil
```

The nice thing about matcher methods is that we can define multiple versions of them, each one matching on a different pattern...

```
(defmatch calc ( (?x plus ?y) )
   (+ #?x #?y))

(defmatch calc ( (?x minus ?y) )
   (- #?x #?y))

cg-user(38): (calc '(mango banana blah))
nil
cg-user(39): (calc '(45 plus 123))
168
cg-user(40): (calc '(45 minus 123))
-78
```

Note: the order in which you compile matcher forms becomes their order of priority. When you call a match method the one that is used will the highest priority method which matches. If you recompile after editing a pattern or if you want to change the order of priority you may need to clear all previous definitions of a match method. You can do this by evaluating...

```
(makunmatch 'method-name)
```

Matcher methods allow us to rewrite some of our recursive functions in a new style. Think about a length function. A null list has length 0, a non-null list has length 1 more than the length of its tail...

```
(defmatch mlen (nil) 0)

(defmatch mlen ((?head ??tail))
  (1+ (mlen #?tail)))

cg-user(41): (mlen nil)
0
cg-user(42): (mlen '(a b c d e f))
6
cg-user(43): (mlen '(a b c ((((d e)) f)) g))
5
```

```
;; with trace...

cg-user(28): (mlen '(a b c d))
 0[4]: (mlen (a b c d))
   1[4]: (mlen (b c d))
     2[4]: (mlen (c d))
       3[4]: (mlen (d))
         4[4]: (mlen nil)
         4[4]: returned 0
       3[4]: returned 1
     2[4]: returned 2
   1[4]: returned 3
 0[4]: returned 4
4
```

Some more examples...

```
;; factorial

(defmatch mfact (0) 1)

(defmatch mfact (?n) (* #?n (mfact (1- #?n))))

cg-user(44): (fact 5)
120
cg-user(45): (trace fact)
(fact)
cg-user(46): (fact 5)
 0[4]: (fact 5)
   1[4]: (fact 4)
     2[4]: (fact 3)
       3[4]: (fact 2)
         4[4]: (fact 1)
           5[4]: (fact 0)
           5[4]: returned 1
         4[4]: returned 1
       3[4]: returned 2
     2[4]: returned 6
   1[4]: returned 24
 0[4]: returned 120
120
```

```
;; sumlist
;; first item added to the sum of all the others

(defmatch msum-list (nil) 0)

(defmatch msum-list ((?n ??rest))
  (+ #?n (msum-list #?rest)))


cg-user(30): (msum-list '(8 2 4 7 12))
33



;; turn everything to spam...

(defmatch spam-all (nil) nil)

(defmatch spam-all ((= ??rest))
  (cons 'spam (spam-all #?rest)))


cg-user(32): (spam-all '(2 eggs 1 bacon 3 sausage and spam))
(spam spam spam spam spam spam spam spam)



;; turn symbols to spam, leave other stuff unchanged...

(defmatch spam-syms (nil) nil)

(defmatch spam-syms ((?(n numberp) ??rest))
  (cons #?n (spam-syms #?rest)))

(defmatch spam-syms ((= ??rest))
  (cons 'spam (spam-syms #?rest)))


cg-user(33): (spam-syms '(2 eggs 1 bacon 3 sausage and spam))
(2 spam 1 spam 3 spam spam spam)
```

To write some matcher methods we use a simple helper function to organise the pattern argument when it is first called. Check out delete below...

```
;; delete with a helper function

;; this is the helper function. It combines the item to delete
;; and the list to delete it from into a single (nested) list

(defun del (item data)
  (mdel (list item data)))


;; now the  match methods...

(defmatch mdel ((= nil)) nil)

(defmatch mdel ((?item (?item ??rest)))
  (mdel (match>> '(?item ?rest))))

(defmatch mdel ((?item (?first ??rest)))
  (cons #?first (mdel (match>> '(?item ?rest)))))


cg-user(36): (del 'spud '(1 spud 2 spud 3 more spud))
(1 2 3 more)
```