the matcher – an introduction to the basics

brief

The matcher provides a kind of easy-to-use regular expression facility for structured symbolic data, 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.

```
user=> (matches '(a b c) '(p q r))
nil
user=> (matches '(a b c) '(a b c))
{:pat (a b c), :it (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)...

```
user=> (matches '(a ?_ c) '(a b c))
{:pat (a ?_ c), :it (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"...

```
user=> (matches '(a ?x c) '(a b c))
{x b, :pat (a ?x c), :it (a b c)}
```

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

```
user=> (matches '(cat ?x rat ?x) '(cat dog rat frog))
nil
user=> (matches '(cat ?x rat ?y) '(cat dog rat frog))
{y frog, x dog, :pat (cat ?x rat ?y), :it (cat dog rat frog)}

user=> (matches '(cat ?x rat ?x) '(cat dog rat dog))
{x dog, :pat (cat ?x rat ?x), :it (cat dog rat dog)}

;; wild cards do not remember their matching
user=> (matches '(cat ?_ rat ?_) '(cat dog rat frog))
{:pat (cat ?_ rat ?_), :it (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 the symbol "??_"...

```
user=> (matches '(a ??_) '(a b c))
{tmp539 (b c), :pat (a ??_), :it (a b c)}

user=> (matches '(a ??_) '(a))
{tmp542 (), :pat (a ??_), :it (a)}

user=> (matches '(a ??_) '(a b c d e f banana mango blah))
{tmp545 (b c d e f banana mango blah),
   :pat (a ??_), :it (a b c d e f banana mango blah)}

user=> (matches '(a ??_) '(aardvaark b c d e f banana mango blah))
nil

user=> (matches '(a ??_ blah) '(a b c d e f banana mango blah))
{tmp550 (b c d e f banana mango),
   :pat (a ??_ blah), :it (a b c d e f banana mango blah)}

user=> (matches '(a ??_ mango) '(a b c d e f banana mango blah))
nil

user=> (matches '(a ??_ mango) '(a mango))
{tmp556 (), :pat (a ??_ mango), :it (a mango)}

user=> (matches '(a ??_ mango ??_ ??_ ??_ ??_) '(a mango))
{tmp563 (), tmp562 (), tmp561 (), tmp560 (), tmp559 (),
   :pat (a ??_ mango ??_ ??_ ??_ ??_), :it (a mango)}
```

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

```
user=> (matches '(a ??x blah) '(a b c d e f banana mango blah))
{x (b c d e f banana mango),
   :pat (a ??x blah), :it (a b c d e f banana mango blah)}

;; notice ?? is always 'reluctant'...
user=> (matches '(a ??x ??y blah) '(a b c d e f banana mango blah))
{y (b c d e f banana mango), x (),
   :pat (a ??x ??y blah), :it (a b c d e f banana mango blah)}

user=> (matches '(a ??x ?y blah) '(a b c d e f banana mango blah))
{y mango, x (b c d e f banana),
   :pat (a ??x ?y blah), :it (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 **(? *var*)**

```
user=> (mlet ['(?x plus ?y) '(5 plus 2)]
          (+ (? x) (? y)))
7

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

user=> (mlet ['(?x plus ?y) '(5 plus 2)]
               (mout '(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 Clojure arguments then specify pattern forms...

```
(defmatch calc []
  ([?x plus ?y] (+ (? x) (? y))))

user=> (calc '(45 plus 123))
168

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

user=> (calc '(mango banana blah))
nil
```

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

```
(defmatch calc []
  ([?x plus ?y]  (+ (? x) (? y)))
  ([?x minus ?y] (- (? x) (? y)))
  )

user=> (calc '(mango banana blah))
nil
user=> (calc '(45 plus 123))
168
user=> (calc '(45 minus 123))
-78
```

we can give match methods a more explicit rule-like flavour by using :=>

```
(defmatch calc []
  ([?x plus ?y]  :=> (+ (? x) (? y)))
  ([?x minus ?y] :=> (- (? x) (? y)))
  )

user=> (calc '(23 minus 32))
-9
```

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 length []
  ([]       :=> 0)
  ([?_ ??r] :=> (inc (length (? r))))
  )

user=> (length ())
0
user=> (length '(a b c d e f))
6
user=> (length '(a b c ((((d e)) f)) g))
5

;; but notice...
user=> (length nil)
nil
```

Some more examples...

```
; check out the argument list & default match

(defmatch math2 [x]
  ((add ?y)  :=> (+ x (? y)))
  ((subt ?y) :=> (- x (? y)))
  ( ?_       :=> x)
  )


user=> (math2 '(add 6) 10)
16
user=> (math2 '(mango melon pawpaw) 6)
6

;; factorial
(defmatch mfact []
  (0  :=> 1)
  (?n :=> (* (? n) (mfact (dec (? n)))))
  )

user=> (mfact 5)
120
```

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

(defmatch msum []
  ([]       :=> 0)
  ([?n ??r] :=> (+ (? n) (msum (? r))))
  )

user=> (msum '(5 3 1 4 2))
15



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

(defmatch spamn []
  ([]  :=> ())

  ([(-> ?n number?) ??r]
    (cons 'spam (spamn (? r))))

  ([?x ??r] :=> (cons (? x) (spamn (? r))))
  )

user=> (spamn '(1 egg 2 beans 3 chips))
(spam egg spam beans spam chips)
```

NB: We can also use match methods as dispatchers, see the matcher user-guide for details