Match

`match` is a special form like **`cond`**

But instead of having clauses that evaluate to either true or false, it takes one item and tests it against the set of clauses

Match

match is a special form like **cond**

But instead of having clauses that evaluate to either true or false, it takes one item and tests it against the set of clauses

```
(match num-presents
  [0 "Christmas is ruined!"]
  [1 "I guess that's alright"]
  [2 "Good"]
  [_ "Great"])
```

Match

```
(match num-presents
  [0 "Christmas is ruined!"]
  [1 "I guess that's alright"]
  [2 "Good"]
  [_ "Great"])
```

Compare that to using **cond**

```
(cond
  [(= num-presents 0) "Christmas is ruined!"]
  [(= num-presents 1) "I guess that's alright"]
  [(= num-presents 2) "Good"]
  [#t "Great"])
```

Match

If that's all that **match** did, it would still be useful, but not very often. But **match** can do much more, it can match against structures and lists

```
(match (list 1 2 3)
  [(list 1 2 3) 'something]
  ['() 'nothing]
  [_ 'something-else])
=> 'something
```

Bindings

The full power of **`match`** is in it's ability to give names to things.

```
(define (rotate lst)
   (match lst
     [(list)       (list)]
     [(list a)     (list a)]
     [(list a b)   (list b a)]
     [(list a b c) (list c a b)]))

> (rotate '(1 2 3))
'(3 1 2)
```

Bindings

I find this really nice when writing recursive code, as I don't need to use `first` or `rest` anymore, and I never accidentally apply them to the empty list

```
(define (map f lst)
  (match lst
    ['() '()]
    [(cons x xs)
     (cons (f x)
           (map f xs))]))
```

Just a note, you see this pattern of a list being matched into the first and rest and them being called `x` and `xs`, (pronouced like "excess") or `a` and `as`, or `b` and `bs`. It help keep the relationship between the variables clear.

Bindings

Another example

```
(define (filter pred lst)
  (match lst
    [(cons x xs) #:when (pred x)
                 (cons x (filter pred xs))]
    [(cons _ xs) (filter pred xs)]
    ['() '()]))
```

Warning

Note one tricky point, `empty` is not a literal. On line 87 of **`racket/collects/list.rkt`** there is the line of code:

```
(define empty '())
```

This means, if you try and match against it, the same thing happens like you try and match against a different variable, like `x`

```
(match (list 1 2 3)
  [empty 'true]
  [_ 'false])
```

Evaluates to `'true`!!!

Haskell

Now, everything I've shown off here is the basic functionally of pattern matching, and if only use this, I think it'll make your code more clear. Also, pattern matching like this is avalible in Haskell

```
map f [] = []
map f (x : xs) = f x : map f xs
```

This will no longer be true from here on, as Racket's `match` is actually quite sophisticated. Checkout the documentation for the full list of things it can do.

Equality

If we repeat a binding, what happens? Racket checks to see if the two instances are equal for us.

```
(match (list 1 2)
 [(list a a) 'same]
 [(list a b) 'different])
=> 'different

(match (list 1 1)
 [(list a a) 'same]
 [(list a b) 'different])
=> 'same
```

...

Racket also lets you use three dots in a row "..." to collect many elements into a list. For example:

```
(match (list 1 2 3 4)
  [(list a as ...) (list as a)])
=> '((2 3 4) 1)
```

as `as` is `'(2 3 4)`.

Note, it can match 0 items.

...

This even works inside nested lists, which is really cool. Say we were storing students' information in structs:

```
(struct student (name mark))

(match (list (student "Alice" 89)
             (student "Bob" 87)
             (student "Eve" 88))
  [(list (student name mark) ...)
   (average mark)])
=> 88
```

will return the average mark of the class

and and or

You can also take conjunctions and disjunctions of patterns.

`(and pat1 pat2)` matches when both `pat1` and `pat2` match. Likewise `(or pat1 pat2)` matches when either pattern matches.

```
(match '(1 (2 3) 4)
  [(list _ (and a (list _ ...)) _) a])
=> '(2 3)

(match '(1 2 3)
  [(or (list 1 _ _) (list 2 _ _)) 'yup])
=> 'yup
```

and

and is quite useful since it can also bind names.

This is a common pattern in my code.

```
(match (map read-parse (rest line))
  [(and children
        (list
          (list 'terminal "LPAREN" "(")
          (list 'typed-rule lvalue-type _ ...)
          (list 'terminal "RPAREN" ")")))
   (cons 'typed-rule
         (cons lvalue-type
               (cons line
                     children)))])
```

?

Sometimes you want to transform the data before matching it, that's where `(? pred pat ...)` comes in handy. It takes a predicate which must return a true value before the patterns can match. You can supply no extra patterns if you want.

```
(match '(1 2 3)
  [(list (? odd? a) 2 _) a])
=> 1
```

app

In a more general way, you can use `(app f pat ...)` to match against the result of any function, not just predicates.

```
(match '(1 2)
  [(app length 2) 'yes])
=> 'yes
```

Match extenders

We can write any of these previous helpers ourselves, or any other ones we can thing ok.

```
(define-match-expander aba?
  (lambda (stx)
    (syntax-case stx ()
      [(_ a b)
       #'(list a b a)])))
(match (list 1 2 1)
  [(aba? 1 2) 'worked])
```