Racket Pattern Matching

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Review: If

`if` evaluates a predicate, and returns either the consequent or the alternative depending on the result:

```
(if predicate
  consequent
  alternative)
```

Example:

```
(printf "n is ~a~%" (if (even? n)
                       "even"
                       "odd"))
```

Review: Let

The `let` macro will bind local variables:

```
(let ([a 10]
      [b 20])
  (printf "~a ~a~%" a b))
```

`let*` will bind in order, equivalent to nesting a bunch of lets together:

```
(let* ([a 10]
       [b (+ a 10)])
  (printf "~a ~a~%" a b))
```

Equivalent to:

```
(let ([a 10])
  (let ([b (+ a 10)])
    (printf "~a ~a~%" a b)))
```

Introducing cond

cond takes a series of clauses. You can think of it as replacing a big long `if` chain:

```
(cond
  [predicate-1 consequent-1]
  [predicate-2 consequent-2]
  ...
  [predicate-n consequent-n]
  [else alternative])
```

Equivalent to:

```
(if predicate-1
  consequent-1
  (if predicate-2
    consequent-2
    ...
      (if predicate-n
        consequent-n
        alternative)))
```

Example: cond

```
(printf "Your BMI rating is: ~a~%"
        (cond
          [(< bmi 18.5) "underweight"]
          [(< bmi 24.9) "normal"]
          [(< bmi 29.9) "overweight"]
          [else "obese"]))
```

Introducing case

Case evaluates and returns the clause which has a quoted form `equal?` to the value passed.

```
(case value
  [(a-1 a-2 ... a-n) result-1]
  [(b-1 b-2 ... b-n) result-2]
  ...
  [else result])
```

Careful

Parenthesis are needed around the values in each clause, even if there is only one value to match.

Example 1: case

```
(define (pet-classification pet-type)
  (case pet-type
    [(cat dog fish) 'normal]
    [(mouse rat ferret) 'strange]
    [(zebra gorilla elephant) 'not-a-pet]
    [else 'unknown]))

> (pet-classification 'cat)
'normal
> (pet-classification 'ferret)
'strange
> (pet-classification 'gorilla)
'not-a-pet
> (pet-classification 'horse)
'unknown
```

Example 2: case

```
(define (login-as username password)
  (case (list username password)
    [(("bob" "secret") ("jill" "ihatebob")) "Welcome!"]
    [(("badguy" "badguy")) "Not authorized!"]
    [else "Unknown username or password"]))

> (login-as "jill" "ihatebob")
"Welcome!"
> (login-as "jill" "ih8bob")
"Unknown username or password"
> (login-as "badguy" "badguy")
"Not authorized!"
```

Example 3: case

```
;; Note: this algorithm is not efficent
(define (fib n)
  (case n
    [(0 1) n]
    [else (+ (fib (- n 1))
             (fib (- n 2)))]))
```

Learning Group Activity

With your learning group, choose whether to implement each of these functions using either `case` or `cond`. Then, once you have decided, write an implementation:

1. A function, which, when passed an even number, divides it by two. When the function is passed an odd number, it will multiply it by three then add 1. If the function is passed anything else (e.g., a string), it will return `"oh noes!"`.
2. A function `f` of parameter `n` which returns `'magic` when passed 0, 10, 20, or 30, or `(f (floor (/ n 2)))` otherwise.

Pattern Matching: Motivation

What if we could combine the functionality of `let`, `cond`, and `case` (and even more) all into a single syntax?

Racket has this: it's called `match`!

Introducing match

Here is the basic syntax for match:

```
(match value
  [pattern-1 result-1]
  [pattern-2 result-2]
  ...
  [pattern-n result-n])
```

Each of these patterns is a very special syntax form which Racket will use to determine if there is a match, as well as bind variable names inside of the body of the clause.

Basic Patterns

Simple constants will match when they are equivalent:

```
(match user-input
  ["true" #t]
  ["false" #f])
```

Underscore can be used to match anything:

```
(match user-input
  ["true" #t]
  ["false" #f]
  [_ (error "Not a valid input")])
```

Binding Identifiers

If a variable name is used in a pattern, racket will bind what matches to that name:

```
(match (+ a b c)
  [10 "it's 10!"]
  [the-number (format "sadly, only ~a..." the-number)])
```

Testing Predicates

The pattern `(? predicate-function optional-binding)` can be used to test if `predicate-function` returns #t when called on the value.

```
(match (+ a b c)
  [(? odd?) "How odd..."]
  [(? even? x) (format "~a is even!" x)])
```

Matching Lists

The pattern `(list ...)` contains patterns and matches a list whose elements match the patterns in order.

```
(match some-list
  [(list) "that list is empty!"]
  [(list a) (format "the list has a single element: ~a" a)]
  [(list a (? even? b)) (format "~a and ~a, second is even" a b)]
  [_ "something else"])
```

Repeated Variable Names

Repeating a variable name will not only bind the value to the variable, but actually check that the elements are `equal?` too!

```
(match some-list
  [(list (? even? a) a) "two even equivalent elements"]
  [(list a a) "two elements, the same"]
  [(list a b) "two elements, different"]
  [_ "something else"])
```

Variable Length Lists

The `(list-rest a b ... z z-cdr)` pattern can be used to match the elements of a list and dump the rest of the elements in `z-cdr`. For example:

```
(match some-list
  [(list) "empty list"]
  [(list _) "one element"]
  [(list-rest _ tail) tail])
```

list-rest as a Common Recursive Pattern

Consider the following function which sums the elements of a list:

```
(define (sum-list lst)
  (match lst
    [(list) 0]
    [(list-rest head tail) (+ head (sum-list tail))]))
```

Quasiquotataions as a Pattern

Quasiquoted patterns match lists of matching symbols, with commas escaping the quotation (and can be any pattern):

```
(define (extract-value lst)
  (match lst
    [`(sally ,x ,_) x]
    [`(mary ,_ ,y) y]))
```

Like `,@` splices in a list to a quasiquotation, when used in a pattern, it will extract a pattern matching from the list:

```
(define (extract-rest lst)
  (match lst
    [`(sally ,@(list-rest x)) x]))
```

define/match

`match` is used to match the arguments of a function so often, that Racket provides a syntax for putting a syntax right inside of `define`.

Example:

```
;; Return the first n elements of a list
(define/match (take n lst)
  [(0 _) '()]
  [(_ (list-rest head tail))
   (cons head (take (- n 1) tail))])
```

Exercise

With your learning group, write a function using `define`/`match` which drops the first n elements of a list:

```
> (drop 3 '(1 2 3 4))
'(4)
> (drop 1 '(7))
'()
> (drop 0 '(8 4 1 4))
'(8 4 1 4)
> (drop 0 '())
'()

;; OK if error when n is greater than the list length, e.g.,
;;    (drop 1 '())
```