Match and Recursive Types

CS 151: Introduction to Computer Science I

Pattern Matching

Syntax `match`:

```
(match exp0
  [pat1 exp1]
  [pat2 exp2]
  ...
  [patk expk])
```

- Similar to `cond` evaluation
- `exp0` is evaluated to a value we'll call v
- v is compared against each pattern in turn, in the order given: `pat1`, then `pat2`, and so on, up to `patk`

Pattern Matching

A new expression, `match`:

```
(match (+ 1 1)
  [1 "one plus one is one"]
  [2 "one plus one is two"]
  [3 "one plus one is three"])
```

Used like this, `match` looks and behaves very similar to `cond`:

```
(:  is-it-5 :  Integer -> String)
(define (is-it-5 n)
  (match n
    [5 "It's 5!!"]
    [_ "Not 5..."]))
```

...but swap `else` for the wildcard pattern `_`

Pattern Matching

```
(match (+ 2 3)
  [0 0]
  [n (* n 2)])
```

First line is called a *constant pattern*
Second line is called a *variable pattern*, and it always succeeds

```
(:  x :  Integer)
(define x 100)
(match (* x 2)
  [150 "very good"]
  [_ (error "2x must be 150")])
```

Difference between variable and wildcard: using a variable creates a new variable

Aside: Variable Scope

The *scope* of a variable is the region of code in which its value is accessible.
For normal variables, their scope is the entire program:

```
(:  my-variable :  String)
(define my-variable "CS is awesome")
```

For function arguments, their scope is the function body:

```
(:  square :  Real -> Real)
(define (square x)
  (* x x))
```

For pattern matches, their scope is a single case:

```
(match (+ 2 3)
  [n (* n 2)])
```

Aside: Variable Scope

To *bind* a variable means to assign it a value

When you use a variable, the value comes from the innermost binding of that variable:

```
(:  x :  String)
(define x "I live outside")

(:  func :  String -> String)
(define (func x)
  (append x " and thank you"))
```

```
(func "please") ⟹ "please and thank you"
```

Matching Structures

```
(define-struct Point
  ([x :  Real]
   [y :  Real]))
(:  my-point :  Point)
(define my-point (Point -3 2))
```

We can use `match` to “pull out" the parts of the structure:

```
(match my-point
  [(Point a b) (Point (+ a 1) (+ b 1))])
```

Equivalent to using the selector notation:

```
(Point (+ (Point-x my-point) 1)
       (+ (Point-y my-point) 1))
```

Match on Union Types

```
(define-struct Circle
  ([radius : Real]
   [center : Point]
   [color : Color]))
(define-struct Square
  ([side-length : Real]
   [center : Point]
   [color : Color]))
(define-type Shape (U Circle Square))
```

```
(: area : Shape -> Real)
(define (area shape)
  (match shape
    [(Circle r _ _) (* pi r r)]
    [(Square s _ _) (* s s)]))
```

Recursive Types: Russian Dolls

A type for Russian dolls:

```
(define-struct Doll
  ([inches-tall :  Exact-Rational]
   [color :  (U 'blue 'red 'yellow)]
   [inside :  (U 'nothing Doll)]))
```

Recursive Types: Russian Dolls

Some things we might want to do with Dolls:

```
(:  count-dolls :  Doll -> Integer)
(:  doll=?  :  Doll Doll -> Boolean)
```

```
(:  count-dolls :  Doll -> Integer)
(define (count-dolls doll)
  (match (Doll-inside doll)
    ['nothing 1]
    [(Doll _) (+
               1
               (count-dolls (Doll-inside doll)))]))
```

Will it Run?

Add some new definitions for Russian dolls:

```
(define-type Color (U 'blue 'red 'yellow))
(define-type Inside (U 'nothing Doll))

(define-struct Doll
  ([inches-tall :  Integer]
   [color :  Color]
   [inside :  Inside]))
```

Will it Run?

```
(:  func :  Integer -> Integer)
(define (func x)
  (cond
    [(= 0 x) 1]
    [else (tion (- x 1))]))

(:  tion :  Integer -> Integer)
(define (tion y)
  (cond
    [(= 0 y) 0]
    [else (+
           2
           (func y))]))
```

Recursive Types: Person

```
(define-struct Person
  ([name : String]
   [year-of-birth : Integer]
   [children : (Listof Person)]))
```

```
(: ivanka : Person)
(: eric : Person)
(: donald : Person)
(define ivanka (Person "Ivanka Trump" 1981 empty))
(define eric (Person "Eric Trump" 1984 empty))
(define donald (Person "Donald Trump" 1946
  (list ivanka eric)))
```

Find Descendant

The characters in *Game of Thrones* can get pretty complicated. . . you just want to know, who is an ancestor of Jon Snow?

```
(:  ancestor?  :  Person -> Boolean)
(:  any-ancestor?  :  (Listof Person) -> Boolean)
```

Find Descendant

```
(: any-ancestor? : (Listof Person) -> Boolean)
(define (any-ancestor? character-list)
  (cond
    [(empty? character-list) #f]
    [else (or
           (ancestor? (first character-list))
           (any-ancestor? (rest
character-list)))]))

(: ancestor? : Person -> Boolean)
(define (ancestor? character)
  (match character
    [(Person "Jon Snow" _ _) #t]
    [(Person _ _ children) (any-ancestor?
children)]))
```

Natural Numbers

The natural numbers are $0, 1, 2, 3, 4, \ldots$

There is a Racket type for these, `Natural`... but rarely used

```
(define-struct Succ
  ([nat :  Nat]))
(define-type Nat (U 'zero Succ))
```

How to represent 0:

```
'zero
```

How to represent 1:

```
(Succ 'zero)
```

How to represent 2:

```
(Succ (Succ 'zero))
```

Natural Numbers

This is **our very own** definition the type `Natural`

```
(:  zero?  :  Nat -> Boolean)
(:  add :  Nat Nat -> Nat)
(:  pred :  Nat -> Nat)
(:  sub :  Nat Nat -> Nat)
```

```
(:  zero?  :  Nat -> Boolean)
(define (zero?  nat)
  (match nat
    ['zero #t]
    [_ #f]))
```

Natural Numbers

```
(: pred : Nat -> Nat)
(define (pred nat)
  (match nat
    ['zero (error "pred: input is zero")]
    [(Succ m) m]))
```

```
(: sub : Nat Nat -> Nat)
(define (sub nat-left nat-right)
  (match nat-right
    ['zero nat-left]
    [(Succ m) (sub (pred nat-left) m)]))
```

Integers

Can use our `Nat` to make our own definition of `Integer`:

```
(define-struct Int
  ([value : Nat]
   [sign : (U 'positive 'negative)]))
```

Implementing `(: add-int : Int Int -> Int)` is painful...

Rationals and Reals

Can also use to define `Exact-Rational`:

```
(define-struct Rat
  ([numerator : Nat]
   [denominator : Nat]
   [sign : (U 'positive 'negative)]))
```

And even `Real`:

```
(define-type Digit (U '0 '1 '2 '3 '4 '5 '6 '7 '8
'9))
(define-type Infinite-Decimal (Nat -> Digit))
(define-struct Real
  ([whole-number : Int]
   [decimal : Infinite-Decimal]))
```

What to know

- Pattern matching
- Variable scope
- Recursive structs, mutual recursion
- Recursive definition of Natural