Programming Languages

Recollecting Haskell, Part III: Recursion

CIS 352/Spring 2016

January 21, 2016

Based on LYH, Chapter 5.

Recursion: Defining something in terms of itself

Question: Who counts as being jewish?
One answer: Either:

a. You are Abraham, or
b. you are a convert, or
c. your mother was jewish. (the recursive case)

The Haudenosaunee (Iroquois) have similar rules for clan membership.

The standard first example of recursion: Factorial

$0! = 1$
$1! = 1$
$2! = 1 * 2$
$3! = 1 * 2 * 3$
$4! = 1 * 2 * 3 * 4$
$5! = 1 * 2 * 3 * 4 * 5$
$\ldots$
$k! = k * (k-1)! \quad \text{where } k > 0$

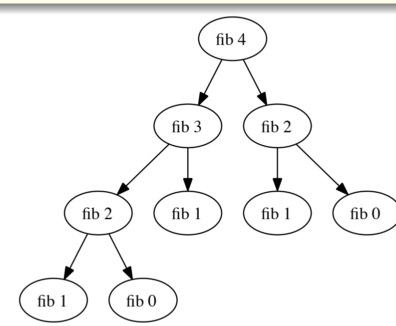
```
-- fact n = n factorial
-- NOTE: n < 0 causes an error
fact :: Integer -> Integer
fact n
  | n==0 = 1
  | n>0  = n * fact (n-1)
```

```
  fact 4
=     { n = 4 }
  4 * fact (4-1)
=
  4 * fact 3
=     { n = 3 }
  4 * 3 * fact (3-1)
=
  4 * 3 * fact 2
=     { n = 2 }
  4 * 3 * 2 * fact (2-1)
=
  4 * 3 * 2 * fact 1
=     { n = 1 }
  4 * 3 * 2 * 1 * fact (1-0)
=
  4 * 3 * 2 * 1 * fact 0
=     { n = 0 }
  4 * 3 * 2 * 1 * 1
```


The standard 2nd example of recursion: Fibonacci

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$, for $n > 1$.

```
-- Fibonacci numbers
fib :: Integer -> Integer
fib n
    | n==0      = 0
    | n==1      = 1
    | n>1       = fib(n-1) + fib(n-2)
    | otherwise = error "fib given negative argument"
```



Recursion on lists, 1

Typical recursions on lists have *at least* two cases:

1. The list you are recurring on looks like: `[]`
   in which case, the recursion bottoms out (i.e., stops).
2. The list you are recurring on looks like: `(x:xs)`
   in which case you probably have subcase where the recursion continues on `xs`.

```
sum' :: (Num a) => [a] -> a
sum' []     = 0
sum' (x:xs) = x + sum' xs
```


Recursion on lists, 2

A messier example

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list!"        (1)
maximum' [x] = x                                    (2)
maximum' (x:xs) = max x (maximum' xs)               (3)
```

```
    maximum' [2,5,1]
=       { (3) succeeds with x = 2, xs = [5,1] }
    max 2 (maximum' [5,1])
=       { (3) succeeds with x = 5, xs = [1] }
    max 2 (max 5 (maximum' [1]))
=       { (2) succeeds with x = 1 }
    max 2 (max 5 1)
=
    max 2 5
=
    5
```


Recursions on lists, 3

Class exercises

- `replicate' :: Int -> a -> [a]`
  `replicate' n x` = a list of n copies of x
- `take' :: Int -> [a] -> [a]`
  `take' n xs` = the first n elements of xs
- `reverse' :: [a] -> [a]`
  `reverse' xs` = the reverse of xs
- `zip' :: [a] -> [b] -> [(a,b)]`
  `zip' xs ys` = the zip of xs and ys
- `elem' :: (Eq a) => a -> [a] -> Bool`
  `elem' x xs` tests if x is an element of xs


```
reverse :: [a] -> [a]
reverse' [ ] = [ ]
reverse' [x:xs] = reverse' xs + [x]

reverse'' xs = helper xs [ ]
    where
      helper [ ] ans = ans
      helper (x:xs) ans = helper xs (x:ans)
```

doesn't need stack

```
zip' :: [a] -> [b] -> [(a,b)]

zip' [ ] _ = [ ]
zip _ [ ] = [ ]
zip (x:xs) (y:ys) = (x,y) : zip' xs ys

elem': (Eq a) => a -> [a] -> Bool

elem' x [ ] = False
elem' x (y:ys)
    | x==y = True
    | otherwise = elem' x ys

elem'' x xs = helper xs
    where helper [ ] = False
          helper [y:ys] = if (x==y) then True else helper ys
```