Informatics 1

Introduction to Computation

Functional Programming

Lecture 3

Lists and Recursion

Philip Wadler

Part I

Lists and Recursion

Cons and append

Cons takes an element and a list.

Append takes two lists.

```
(:)  :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]

1 : [2,3]       =  [1,2,3]
[1] ++ [2,3]    =  [1,2,3]
[1,2] ++ [3]    =  [1,2,3]
'l' : "ist"     =  "list"
"l" ++ "ist"    =  "list"
"li" ++ "st"    =  "list"

[1] : [2,3]               -- type error!
1 ++ [2,3]                -- type error!
[1,2] ++ 3                -- type error!
"l" : "ist"               -- type error!
'l' ++ "ist"              -- type error!
```

`(:)` is pronounced *cons*, for *construct*

`(++)` is pronounced *append*

Lists

Every list can be written using only `(:)` and `[]`.

```
[1,2,3]   =  1 : (2 : (3 : []))

"list"    =  ['l','i','s','t']
          =  'l' : ('i' : ('s' : ('t' : [])))
```

A *recursive* definition: A *list* is either

- *empty*, written `[]`, or
- *constructed*, written `x:xs`, with *head* `x` (an element), and *tail* `xs` (a list).

So every list matches exactly one of the following two *patterns*

```
[ ]              -- only matches the empty list
( x : xs )       -- matches any non-empty list
```

We can use any two distinct variables in the *cons* pattern

```
( head : tail ) -- matches any non-empty list
```

Patterns

List patterns can be used in declarations – `<pattern> = <value>`

```
myList = [ 0, 1, 2, 3, 4 ]
(x:xs)              = myList
[ a, b, c, d, e ] = myList -- matches lists of length 5
[ p, q, r ]         = myList -- matches lists of length 3

*Main> x    --   ( x : xs )          = [ 0, 1, 2, 3, 4 ]
0
*Main> xs   --   ( x : xs )          = [ 0, 1, 2, 3, 4 ]
[ 1, 2, 3, 4 ]
*Main> c    --   [ a, b, c, d, e ] = [ 0, 1, 2, 3, 4 ]
2
*Main> p    --   [ p, q, r ]         = [ 0, 1, 2, 3, 4 ]
*** Exception: ... -- pattern and value must match!
```

Recursion versus meaningless self-reference

A *list* is either

- *empty*, written `[]`, or
- *constructed*, written `x:xs`, with *head* `x` (an element), and *tail* `xs` (a list).

Recursion versus meaningless self-reference

A *list* is either

- *empty*, written `[]`, or
- *constructed*, written `x:xs`, with *head* `x` (an element), and *tail* `xs` (a list).

“Brexit means Brexit.”

Theresa May

A list of numbers

```
Prelude> null [1,2,3]
False
Prelude> head [1,2,3]
1
Prelude> tail [1,2,3]
[2,3]
Prelude> null [2,3]
False
Prelude> head [2,3]
2
Prelude> tail [2,3]
[3]
Prelude> null [3]
False
Prelude> head [3]
3
Prelude> tail [3]
[]
Prelude> null []
True
```

Part II

Mapping: Square every element of a list

Two styles of definition—squares

Comprehension

```
squares :: [Int] -> [Int]
squares xs  =  [ x*x | x <- xs ]
```

Recursion

```
squaresRec :: [Int] -> [Int]
squaresRec []      =  []
squaresRec (x:xs)  =  x*x : squaresRec xs
```

Pattern matching and conditionals

Pattern matching

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs
```

Conditionals with binding

```
squaresCond :: [Int] -> [Int]
squaresCond ws  =
  if null ws then
    []
  else
    let
      x  = head ws
      xs = tail ws
    in
      x*x : squaresCond xs
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs
```

```
   squaresRec [1,2,3]
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
=        { x = 1, xs = (2 : (3 : [])) }
   1*1 : squaresRec (2 : (3 : []))
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
=
   1*1 : squaresRec (2 : (3 : []))
=        { x = 2, xs = (3 : []) }
   1*1 : (2*2 : squaresRec (3 : []))
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

    squaresRec [1,2,3]
=
    squaresRec (1 : (2 : (3 : [])))
=
    1*1 : squaresRec (2 : (3 : []))
=
    1*1 : (2*2 : squaresRec (3 : []))
=         { x = 3, xs = [] }
    1*1 : (2*2 : (3*3 : squaresRec []))
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
=
   1*1 : squaresRec (2 : (3 : []))
=
   1*1 : (2*2 : squaresRec (3 : []))
=
   1*1 : (2*2 : (3*3 : squaresRec []))
=
   1*1 : (2*2 : (3*3 : []))
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
=
   1*1 : squaresRec (2 : (3 : []))
=
   1*1 : (2*2 : squaresRec (3 : []))
=
   1*1 : (2*2 : (3*3 : squaresRec []))
=
   1*1 : (2*2 : (3*3 : []))
=
   1 : (4 : (9 : []))
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
=
   1*1 : squaresRec (2 : (3 : []))
=
   1*1 : (2*2 : squaresRec (3 : []))
=
   1*1 : (2*2 : (3*3 : squaresRec []))
=
   1*1 : (2*2 : (3*3 : []))
=
   1 : (4 : (9 : []))
=
   [1,4,9]
```

How recursion works—squaresRec

```
squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

   squaresRec [1,2,3]
=
   squaresRec (1 : (2 : (3 : [])))
=
   1*1 : squaresRec (2 : (3 : []))
=
   1*1 : (2*2 : squaresRec (3 : []))
=
   1*1 : (2*2 : (3*3 : squaresRec []))
=
   1*1 : (2*2 : (3*3 : []))
=
   1 : (4 : (9 : []))
=
   [1,4,9]
```

QuickCheck

```
-- squares.hs
import Test.QuickCheck

squares :: [Int] -> [Int]
squares xs  =  [ x*x | x <- xs ]

squaresRec :: [Int] -> [Int]
squaresRec []       =  []
squaresRec (x:xs)   =  x*x : squaresRec xs

prop_squares :: [Int] -> Bool
prop_squares xs  =  squares xs == squaresRec xs
```

```
[jitterbug]dts: ghci squares.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
*Main> quickCheck prop_squares
+++ OK, passed 100 tests.
*Main>
```

Part III

Filtering: Select odd elements from a list

Two styles of definition—odds

Comprehension

```
odds :: [Int] -> [Int]
odds xs  =  [ x | x <- xs, odd x ]
```

Recursion

```
oddsRec :: [Int] -> [Int]
oddsRec []                     =  []
oddsRec (x:xs) | odd x         =  x : oddsRec xs
               | otherwise     =  oddsRec xs
```

Pattern matching and conditionals

Pattern matching with guards

```
oddsRec :: [Int] -> [Int]
oddsRec []                         =   []
oddsRec (x:xs) | odd x             =   x : oddsRec xs
               | otherwise         =   oddsRec xs
```

Conditionals with binding

```
oddsCond :: [Int] -> [Int]
oddsCond ws  =
  if null ws then
    []
  else
    let
      x  = head ws
      xs = tail ws
    in
      if odd x then
        x : oddsCond xs
      else
        oddsCond xs
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                       =  []
oddsRec (x:xs) | odd x           =  x : oddsRec xs
               | otherwise       =  oddsRec xs
```

```
oddsRec [1,2,3]
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                      =  []
oddsRec (x:xs) | odd x          =  x : oddsRec xs
               | otherwise      =  oddsRec xs
```

```
   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                      =  []
oddsRec (x:xs) | odd x          =  x : oddsRec xs
               | otherwise      =  oddsRec xs

   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
=        { x = 1, xs = (2 : (3 : [])), odd 1 = True }
   1 : oddsRec (2 : (3 : []))
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                       =  []
oddsRec (x:xs) | odd x           =  x : oddsRec xs
               | otherwise       =  oddsRec xs

   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
=
   1 : oddsRec (2 : (3 : []))
=        { x = 2, xs = (3 : []), odd 2 = False }
   1 : oddsRec (3 : [])
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                         =  []
oddsRec (x:xs) | odd x             =  x : oddsRec xs
               | otherwise         =  oddsRec xs

   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
=
   1 : oddsRec (2 : (3 : []))
=
   1 : oddsRec (3 : [])
=       { x = 3, xs = [], odd 3 = True }
   1 : (3 : oddsRec [])
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                   =  []
oddsRec (x:xs) | odd x       =  x : oddsRec xs
               | otherwise   =  oddsRec xs

   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
=
   1 : oddsRec (2 : (3 : []))
=
   1 : oddsRec (3 : [])
=
   1 : (3 : oddsRec [])
=
   1 : (3 : [])
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                    =  []
oddsRec (x:xs) | odd x        =  x : oddsRec xs
               | otherwise    =  oddsRec xs
```

```
   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
=
   1 : oddsRec (2 : (3 : []))
=
   1 : oddsRec (3 : [])
=
   1 : (3 : oddsRec [])
=
   1 : (3 : [])
=
   [1,3]
```

How recursion works—oddsRec

```
oddsRec :: [Int] -> [Int]
oddsRec []                      =  []
oddsRec (x:xs) | odd x          =  x : oddsRec xs
               | otherwise      =  oddsRec xs

   oddsRec [1,2,3]
=
   oddsRec (1 : (2 : (3 : [])))
=
   1 : oddsRec (2 : (3 : []))
=
   1 : oddsRec (3 : [])
=
   1 : (3 : oddsRec [])
=
   1 : (3 : [])
=
   [1,3]
```

QuickCheck

```
-- odds.hs
import Test.QuickCheck

odds :: [Int] -> [Int]
odds xs  =  [ x | x <- xs, odd x ]

oddsRec :: [Int] -> [Int]
oddsRec []                     =  []
oddsRec (x:xs) | odd x         =  x : oddsRec xs
               | otherwise     =  oddsRec xs

prop_odds :: [Int] -> Bool
prop_odds xs  =  odds xs == oddsRec xs
```

```
[jitterbug]dts: ghci odds.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
*Main> quickCheck prop_odds
+++ OK, passed 100 tests.
*Main>
```

Part IV

Accumulation: Sum a list

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs
```

```
    sum [1,2,3]
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
```

Sum

```
sum :: [Int] -> Int
sum []        =  0
sum (x:xs)    =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=       {x = 1, xs = (2 : (3 : []))}
    1 + sum (2 : (3 : []))
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=
    1 + sum (2 : (3 : []))
=      {x = 2, xs = (3 : [])}
    1 + (2 + sum (3 : []))
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=
    1 + sum (2 : (3 : []))
=
    1 + (2 + sum (3 : []))
=      {x = 3, xs = []}
    1 + (2 + (3 + sum []))
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=
    1 + sum (2 : (3 : []))
=
    1 + (2 + sum (3 : []))
=
    1 + (2 + (3 + sum []))
=
    1 + (2 + (3 + 0))
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=
    1 + sum (2 : (3 : []))
=
    1 + (2 + sum (3 : []))
=
    1 + (2 + (3 + sum []))
=
    1 + (2 + (3 + 0))
=
    6
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=
    1 + sum (2 : (3 : []))
=
    1 + (2 + sum (3 : []))
=
    1 + (2 + (3 + sum []))
=
    1 + (2 + (3 + 0))
=
    6
```

Sum

```
sum :: [Int] -> Int
sum []       =  0
sum (x:xs)   =  x + sum xs

    sum [1,2,3]
=
    sum (1 : (2 : (3 : [])))
=
    1 + sum (2 : (3 : []))
=
    1 + (2 + sum (3 : []))
=
    1 + (2 + (3 + sum []))
=
    1 + (2 + (3 + 0))
=
    6
```

Product

```
product :: [Int] -> Int
product []       =  1
product (x:xs)   =  x * product xs

    product [1,2,3]
=
    product (1 : (2 : (3 : [])))
=
    1 * product (2 : (3 : []))
=
    1 * (2 * product (3 : []))
=
    1 * (2 * (3 * product []))
=
    1 * (2 * (3 * 1))
=
    6
```

Part V

Putting it all together:

Sum of the squares of the odd numbers in a list

Two styles of definition

Comprehension

```
sumSqOdd :: [Int] -> Int
sumSqOdd xs  =  sum [ x*x | x <- xs, odd x ]
```

Recursion

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                     =  0
sumSqOddRec (x:xs) | odd x         =  x*x + sumSqOddRec xs
                   | otherwise     =  sumSqOddRec xs
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                      =  0
sumSqOddRec (x:xs) | odd x          =  x*x + sumSqOddRec xs
                   | otherwise      =  sumSqOddRec xs
```

```
sumSqOddRec [1,2,3]
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                      =  0
sumSqOddRec (x:xs) | odd x          =  x*x + sumSqOddRec xs
                   | otherwise      =  sumSqOddRec xs
```

```
   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                    =  0
sumSqOddRec (x:xs) | odd x        =  x*x + sumSqOddRec xs
                   | otherwise    =  sumSqOddRec xs

   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
=        { x = 1, xs = (2 : (3 : [])), odd 1 = True }
   1*1 + sumSqOddRec (2 : (3 : []))
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                      =  0
sumSqOddRec (x:xs) | odd x          =  x*x + sumSqOddRec xs
                   | otherwise      =  sumSqOddRec xs

    sumSqOddRec [1,2,3]
=
    sumSqOddRec (1 : (2 : (3 : [])))
=
    1*1 + sumSqOddRec (2 : (3 : []))
=         { x = 2, xs = (3 : []), odd 2 = False }
    1*1 + sumSqOddRec (3 : [])
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                      =  0
sumSqOddRec (x:xs) | odd x          =  x*x + sumSqOddRec xs
                   | otherwise      =  sumSqOddRec xs
```

```
   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
=
   1*1 + sumSqOddRec (2 : (3 : []))
=
   1*1 + sumSqOddRec (3 : [])
=        { x = 3, xs = [], odd 3 = True }
   1*1 + (3*3 : sumSqOddRec [])
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                   =  0
sumSqOddRec (x:xs) | odd x       =  x*x + sumSqOddRec xs
                   | otherwise   =  sumSqOddRec xs
```

```
   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
=
   1*1 + sumSqOddRec (2 : (3 : []))
=
   1*1 + sumSqOddRec (3 : [])
=
   1*1 + (3*3 + sumSqOddRec [])
=
   1*1 + (3*3 + 0)
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                         =  0
sumSqOddRec (x:xs) | odd x             =  x*x + sumSqOddRec xs
                   | otherwise         =  sumSqOddRec xs
```

```
   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
=
   1*1 + sumSqOddRec (2 : (3 : []))
=
   1*1 + sumSqOddRec (3 : [])
=
   1*1 + (3*3 + sumSqOddRec [])
=
   1*1 + (3*3 + 0)
=
   1 + (9 + 0)
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                         =   0
sumSqOddRec (x:xs) | odd x             =   x*x + sumSqOddRec xs
                   | otherwise         =   sumSqOddRec xs
```

```
   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
=
   1*1 + sumSqOddRec (2 : (3 : []))
=
   1*1 + sumSqOddRec (3 : [])
=
   1*1 + (3*3 + sumSqOddRec [])
=
   1*1 + (3*3 + 0)
=
   1 + (9 + 0)
=
   10
```

How recursion works—sumSqOddRec

```
sumSqOddRec :: [Int] -> Int
sumSqOddRec []                        =   0
sumSqOddRec (x:xs) | odd x            =   x*x + sumSqOddRec xs
                   | otherwise        =   sumSqOddRec xs
```

```
   sumSqOddRec [1,2,3]
=
   sumSqOddRec (1 : (2 : (3 : [])))
=
   1*1 + sumSqOddRec (2 : (3 : []))
=
   1*1 + sumSqOddRec (3 : [])
=
   1*1 + (3*3 + sumSqOddRec [])
=
   1*1 + (3*3 + 0)
=
   1 + (9 + 0)
=
   10
```