Lecture 4 – Recursions

Violet Ka I Pun

violet@ifi.uio.no

Definition of functions

- **New from old**:
  splitAt n xs = (take n xs, drop n xs)
  splitAt :: Int → [a] → ([a],[a])

- **Patterns**:
  bothTrue x y = and [x,y]
  bothTrue False _ = False
  bothTrue _ False = False
  bothTrue _ _ = True
  bothTrue False _ = False
  bothTrue True x = x

- **Conditional expressions**:
  (must always have 'else' branch!)
  (avoid "dangling else" problem)

```
bothTrue x y = if x==False
               then False
               else if y==False then False
                                else True
```

Definition of functions

- **Guarded equations**:
  bothTrue x y | x==False = False
  | y==False = False
  | otherwise = True

- **Anonymous functions**:
  $\backslash$ x $\rightarrow$ $\backslash$ y $\rightarrow$ **if** x==False **then** False **else** y

Patterns

- Patterns are composed of as constructions such as `( , )`, `[]`, `:`
- Constructors of user-defined datatype can also be used
- General format:

```
f <pattern_1> = <expr_1>
f <pattern_2> = <expr_2>
...
f <pattern_n> = <expr_n>
```

- Some functional languages prohibits overlapping patterns
- Others use the principle: first satisfied pattern ... Hskl
- Some allow the last pattern to be a variable (overlap!) Hskl
- All require that the variables within one pattern are different Hskl
- The variables in $\langle$`expr_i`$\rangle$ is bound by $\langle$`pattern_i`$\rangle$ Hskl

Lists in Haskell

- Lists in FP vs. array in IP
  Given a type `A` we have a type `[A]`
- Recursion in FP vs. loops in IP
  A list is either empty ([ ]), or . . . . thus basic case, and
  a value followed by a <u>shorter</u> list recursive case
- The idea is also used for other recursive data structures, e.g.:
  `data Tree = Leaf Int | Node Int Tree Tree`

Some list-functions

- length :: [a] -> Int
- head, last :: [a] -> a
- xs !! n – the nth element in the list xs
- tail, init :: [a] -> [a]
- take n xs – take first n elements from the list xs
  take 2 [1,2,3,4,5] = [1,2]
- drop n xs – remove first n elements from the list
  drop 2 [1,2,3,4,5] = [3,4,5]
- splitAt n xs – split the list into two, after the nth *element*
  splitAt 2 [1,2,3,4,5] = ([1,2], [3,4,5])
  splitAt :: Int -> [a] -> ([a], [a])
- filter my-test xs – keep only those satisfy my-test
  filter (>5) [2,4,5,6,7] = [6,7]
  filter odd [2,4,5,6,7] = [5,7]
  filter :: (a -> Bool) -> [a] -> [a]

List comprehension

- Set-comprehension – given a set M: $\{x \in M \mid P(x)\}$,
  $\{f(x) \mid x \in M \wedge P(x)\}$
- List comprehension
  Example 1: .... `[x*x | x <- [1..10] ]`
  `|` is read as 'such as, `x <- [1..10]` is called a generator
- Example 2: .... `[x | x <- [1..10], x `mod` 2 /= 0]`
  `x `mod` 2 /= 0` is called a guard
- Example 3: ....`[(x,y) | x <- [1..3], y <- ['a','b'] ]`
- Example 4: ....`[(x,y) | y <- ['a','b'], x <- [1..3] ]`
- Example 5: ....`[(x,y) | x <- [1..5], y <- [1..x] ]`
- Example 6: ....`[(x,y) | y <- [1..x], x <- [1..5] ]`

More examples

- `factor n = [x | x <- [1..n], n ‘mod‘ x == 0 ]`
- `primes n = [x | x <- [1..n], factor x == [1,x]]`
- `flatten ll = [e | l <- ll, e <- l]`
  flatten :: [[t]] -> [t]
- `firsts ps = [x | (x,_) <- ps]` (_ is a ‘wildcard’)
  firsts :: [(t, t1)] -> [t]
- `lookup k t = [v | (v,k’) <- t, k == k’]` (linear search)
  lookup :: Eq a => a -> [(t, a)] -> [t]
- qs [] = []
  qs(x:xs) = qs [y | y <- xs, y <=x] ++ [x] ++ qs [y | y <- xs, y > x]

Triangular numbers

$$T_n = \sum_{i=1}^{i=n} i$$

- tree 1 = 1
  tree n = n + (tree (n-1))
- tree n = sum [x | x <- [1..n]][1..n]
- tree n = (n+1)*n ‘div‘ 2

The *zip* function

- `zip` – pairs elements in two lists (length does not matter)

  zip [] ys = []   zip xs [] = []

  zip (x:xs) (y:ys) = (x,y) : zip xs ys
- Example: `pairs l = zip l (tail l)`

```
sorted l = and [x <= y | (x,y) <- pairs l]
sorted' l = []==[(x,y) | (x,y) <- pairs l, x > y]
```
- Example: `posilist :: Eq a => a -> [a] -> [Int]`

```
posilist x xs =
          [i | (x',i) <- zip xs [1..length xs], x==x']
```
- password must have at 5 small, 3 capital letters, and 2 digits:

```
lud ps = [length [x | x<-ps, isLower x],
          length [x | x<-ps, isUpper x],
          length [x | x<-ps, isDigit x] ]
check ps l u d =
    and [e<=i | (e,i) <- zip [l,u,d] (lud ps) ]
[l,u,d] > lud ps
```

[6,2,1] > **[5,3,2]**

Strings in Haskell

- Strings in Haskell are lists of characters
- Examples: `‘‘abc’’ == ['a','b','c']`, `‘‘’’ == []`
- List functions and comprehension can be used for strings
- Examples: `length ‘‘INF121’’`, `take 3 ‘‘INF121’’...`
- `numDigit cs = length [c | c <- cs, isDigit c]`
- lower s = [toLower b | b <- s] `import Data.Char`
- Hutton, Ch 5.5!

Recursion: using stacks

Lists

Construction: **[]**, **:** ,
destruction: **head**, **tail**,
basic case test: **null**

Imperative "reverse":

```
list l1 = ...;
list l2 = [];
while (! null(l1)) l2 = head(l1):l2; l1 = tail(l1);
```

Imperative "append":

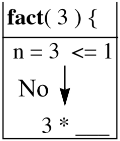
```
list l1 = ...;
list l2 = ...;
list l3 = [];
while (! null(l1)) l3 = head(l1):l3; l1 = tail(l1);
while (! null(l3)) l2 = head(l3):l2; l3 = tail(l3);
```

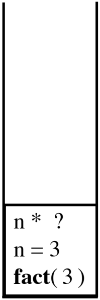
Imperative "append":

a:								
b:	d:		b:	d:			d:	b:
c:	e:		c:	e:	a:	c:	e:	a:
[]	[]	[]	[]	[]	[]	[]	[]	[]
–	–	–	–	–	–	–	–	–
l1	l2	l3	l1	l2	l3	l1	l2	l3

							b:	
		c:		c:			c:	
	d:	b:		d:	b:		d:	
	e:	a:		e:	a:		e:	a:
[]	[]	[]	[]	[]	[]	[]	[]	[]
–	–	–	–	–	–	–	–	–
l1	l2	l3	l1	l2	l3	l1	l2	l3
	a:							
	b:							

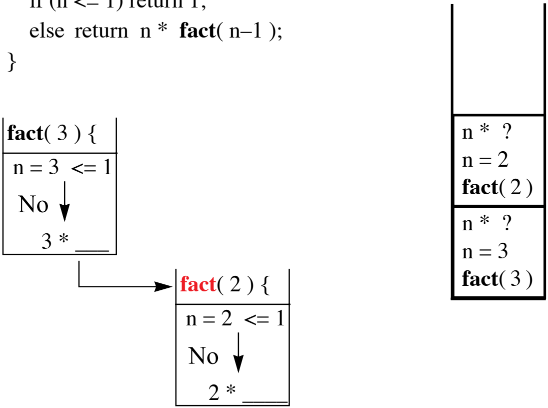
```
public int fact( int n ) {
   if (n <= 1) return 1;
   else return n * fact( n–1 );
}
```





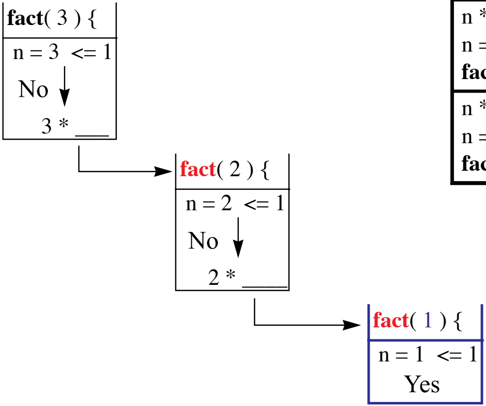
Implementing recursion with a stack

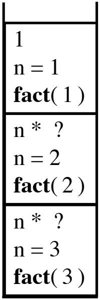
```
public int fact( int n ) {
   if (n <= 1) return 1;
   else return n * fact( n–1 );
}
```



Implementing recursion with a stack

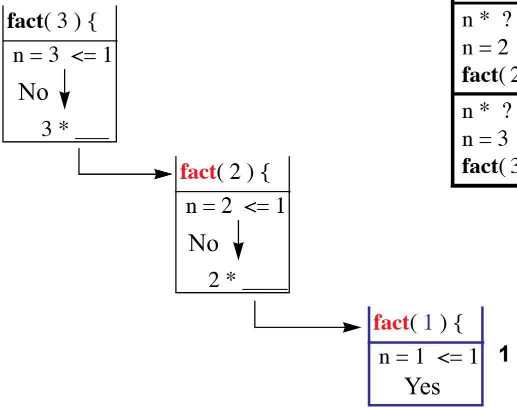
```
public int fact( int n ) {
   if (n <= 1) return 1;
   else return n * fact( n–1 );
}
```

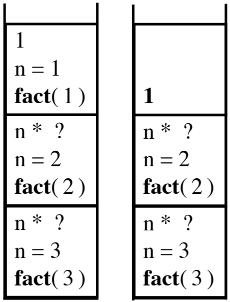




Implementing recursion with a stack

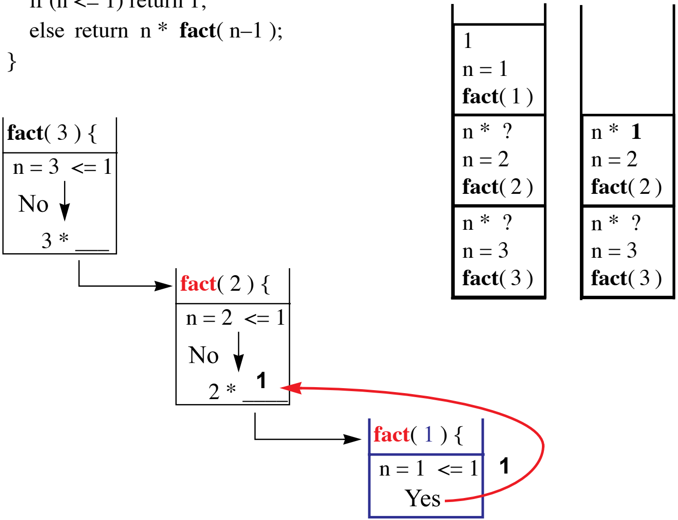
```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```



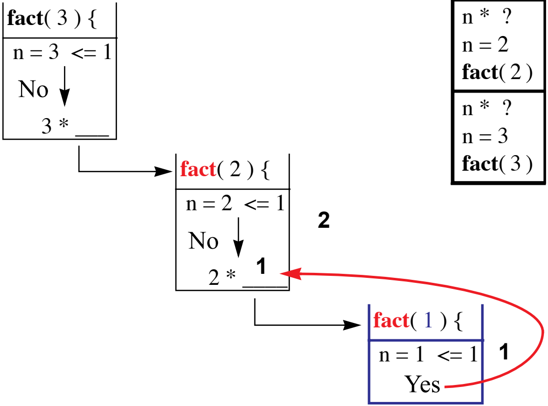


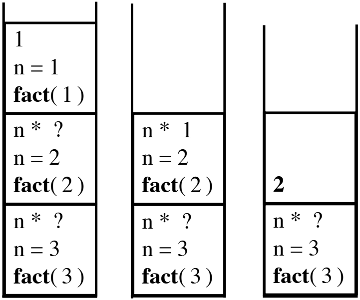
Implementing recursion with a stack

```
public int fact( int n ) {
   if (n <= 1) return 1;
   else  return  n *  fact( n–1 );
}
```

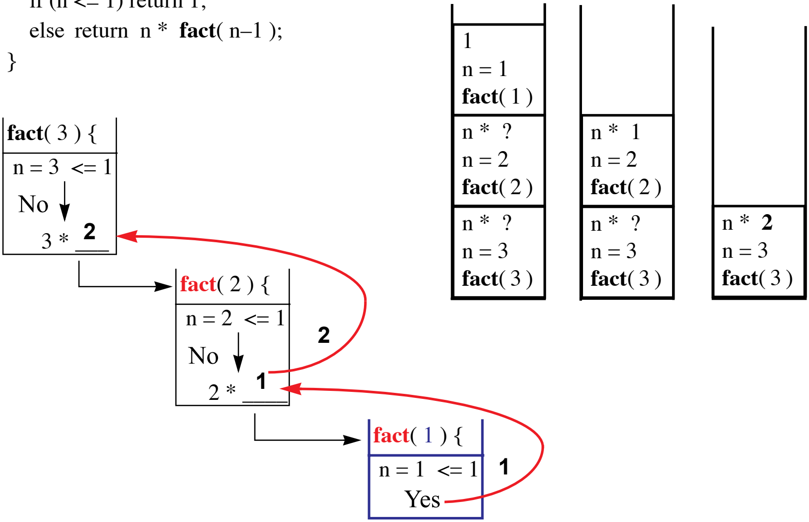


```
public int fact( int n ) {
    if (n <= 1) return 1;
    else return n * fact( n–1 );
}
```



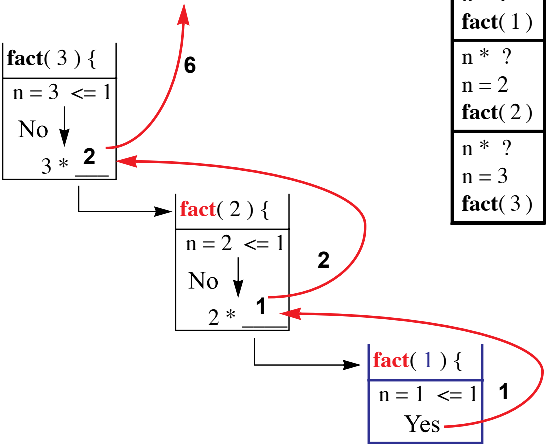


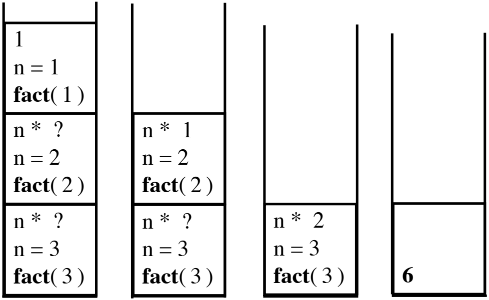
```
public int fact( int n ) {
   if (n <= 1) return 1;
   else return n * fact( n–1 );
}
```



Implementing recursion with a stack

```
public int fact( int n ) {
    if (n <= 1) return 1;
    else  return  n *  fact( n–1 );
}
```



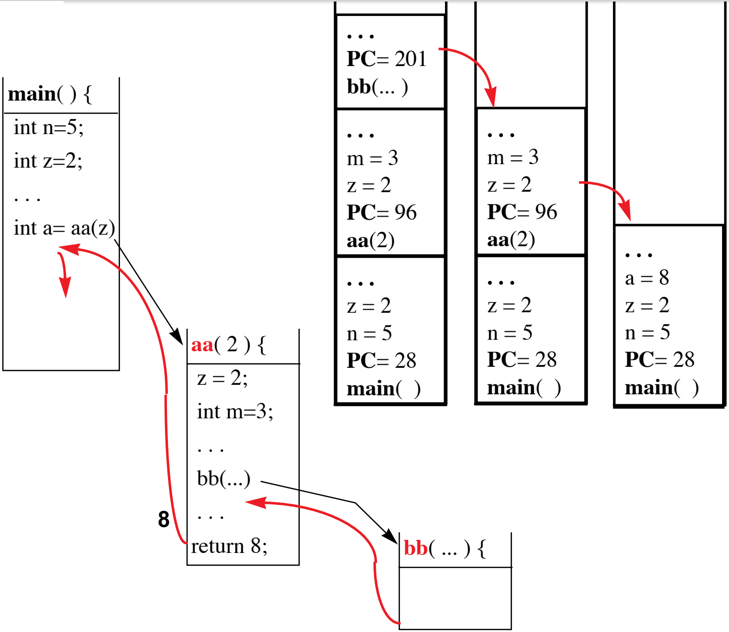


Method stack

```
    main( ) {
        int n=5;
        int z=2;
        . . .
28      int a= aa(z);
        . . .
    }

    int aa(int z) {
        int m=3;
        . . .
96      bb(...);
        . . .
        return 8;
    }

201 void bb(...) {
        . . .
    }
```



Recursion

Imperative

append:

```
while (! null(l1)) l3 = head(l1):l3; l1 = tail(l1);
while (! null(l3)) l2 = head(l3):l2; l3 = tail(l3);
```

reverse:

```
while (! null(l1)) l2 = head(l1):l2; l1 = tail(l1);
```

Recursive

```
app l1 l2 =
 if null(l1) then l2 else head l1:app (tail l1) l2

reverse' l1 =
 if null(l1) then l1 else reverse' (tail l1) ++ head(l1)

rev l1 l2 =
if null(l1) then l2 else rev (tail l1) (head l1:l2)
```

`f x = <body>` where `f` calls itself once

```
app l1 l2 =
 if null(l1) then l2 else head l1:app (tail l1) l2

reverse' l1 =
 if null(l1) then l1 else reverse' (tail l1) ++ head(l1)

rev l1 l2 =
if null(l1) then l2 else rev (tail l1) (head l1:l2)
```

rev is **tail** recursive

Tail recursion

- In general, recursion is implemented with a stack
- Tail recursion does not need such a stack

  rev l1 l2 = **if null** l1 **then** l2 **else** rev (tail l1) (head l1:l2)

  rev [1,2,3] [] = rev [2,3] [1] = rev [3] [2,1] = rev [] [3,2,1]

- Only needs an extra output variable (accumulator l2), and can be written as iteration:

```
while (!null(l1)) l2= head(l1):l2; l1= tail(l1);
```

Tail recursion does not need a stack, since transferring the current state to the next call is trivial

Tail recursion

Fibonacci numbers: fib(n) = fib (n-1) + fib (n-2)

- Naïve solution

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- Tail recursion

```
aux n result pre
    | n == 0 = result
    | otherwise = aux (n-1) (result + pre) result
fib n
    | n == 0 = 1
    | otherwise = aux n 1 0
```

- Implementation of tail-recursion in general does not need a stack
  ⇒ tail recursion uses considerably less memory
- – Tail recursion optimisation is compulsory for FP compilation!
- Materials about recursion: Hutton, Ch 6.6

Iteration	To recursion	With accumulator

sum(n) =
 r := 0;
 while (n > 0)
 r := r+n;
 n := n-1;
 return r;
	sum 0 = 0
	 sum n = n + sum(n-1)
		sum n = sm n 0
		 sm 0 r = r
		 sm n r = sm n-1 r+n
		 tail recursion!

Each iteration can be written as recursion, including tail-recursion.

```
f(n) =
 r := init;
 while (n > last)
  r := body(r,n);
  n := increase(r,n);
 return r;
```

```
f n = fu n init
fu last r = r
fu n r = fu increase(r,n) body(r,n)
```

The approach: "divide and conquer"

1. Which data type is used for recursive solution of the problem P?

INDUCTIVE...

Given a instance n of P:

2. What do I do when n is the base case?
3. How to construct the solution $P(n)$ based on solutions $P(m_i)$ for $m_i < n$?

- P(n) = 2^n bs x l = find the value with the key x in a sorted list l
- Recursion over n (natural numbers) Recursive over the length of the list (natural numbers)
- Base case, $n = 0$, the value: $P\ 0 = 1$ Base case, [] or [x], the value: bs x [] = -1

  bs x [(y,z)] | x==y = z | otherwise = -1
- Induction step, $+1$:

  $P(n) = P(n-1) * 2$ Induction step, halve the list:

  bs x l = **let** m = length l ‘div‘ 2, (n,v) = l !! m **in**

  if n==x **then** v

  else if n<x **then** bs x (drop m l)

The approach: “divide and conquer”

1. Which data type is used for recursive solution of the problem P?

INDUCTIVE...

Given a instance n of P:

2. What do I do when n is the base case?
3. How to construct the solution $P(n)$ based on solutions $P(m_i)$ for $m_i < n$?

- P(n) = sum the numbers in a binary tree
- – with n elements?
- – of height n?
- data BT = Lf Int | Nd BT Int BT
- Recursion over the “complexity” of the tree (inductive definition)
- Base case: `leaf`, value: sum Lf x = x
- Induction step,
  sum Nd lt x rt = x + sum lt + sum rt

Complexity... – size of recursion tree

What do these do (x>=0)

f x =

1) if x<=1 then 1 else x + f(x-1) = $\sum_{i=1}^{x} i = x * (x+1)/2$

2) if x<=1 then 1 else x*x + f(x-1) = $\sum_{i=1}^{x} i^2$

3) if x<=1 then 1 else 1 + f(x-1) = $\sum_{1}^{x} 1 = x$

4) if x<=1 then 1 else 1 + f(x-2) = $\sum_{i=1}^{x/2} 1 = x/2$

5) if x<=1 then 5 else f(x-1) = 5 .... O(x)

6) if x<=0 then 1 else f(x-1)+f(x-1) = 2^x .... O(?) .... O(2^x)

7) if x<=0 then 1 else 2 * f(x-1) = 2^x .... O(x)

Recursion invariant

sum all numbers from 0 to n:

invariant: `sm n` = 0+1+...+n, when $n >= 0$

sm 0 = 0	sm 0 = 0 is correct
sm x = x + sm(x-1)	assume: sm(x-1) = (0+1+...+x-1)
	then: x + sm(x-1) = (0+1+...+x-1)+x is correct

sort a list (quicksort)

invariant: `qs ls` – returns sorted ls

qs [] = []	qs [] = [] is correctly sorted
qs(x:xs) = let R = qs [y | y<-xs, y>x],	assume: R and
L = qs [y | y<-xs, y<=x]	L is correctly sorted
in L++[x]++R	then L++[x]++R is correct

Multiple recursive calls

Not linear recursion (several recursive calls)

```
fib 0 = 1
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

```
qs [] = []
qs (x:xs) = qs [y | y <- xs, y<=x] ++ [x] ++ qs [y | y <- xs, y>x]
```

Several arguments:

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

recursion on which argument (if not both)?

```
exp n x = ...?
```

```
exp 0 x = 0
exp n x = ...      exp n 0 =
1
exp n x = n * (exp n x-1)
```

Mutual recursion

```
even 0 = True
even (n+1) = odd n

odd 0 = False
odd (n+1) = even n

odd 2 = even 1 = odd 0 = False
```

divide a list into two: one with odd and the other with even indices

```
di [1,2,3,4,5,6] = ([1,3,5],[2,4,6])

di [] = ([],[])
di (x:xs) = let (a,b) = di2 xs in (x:a,b)

di2 [] = ([],[])
di2 (x:xs) = let (a,b) = di xs in (a,x:b)
```

Variations: mutual recursion

```
flat [] = []
flat (x:xs) = aux x xs
aux [] ls = flat ls
aux (x:xs) ls = x : aux xs ls
```

```
fl [[1,2],[3,4]] = aux [1,2] [[3,4]] = 1:aux [2] [[3,4]] =
1:2:aux [] [[3,4]] = 1:2:fl [[3,4]] = 1:2:aux [3,4] [] =
1:2:3:4:aux [] [] = 1:2:3:4:fl [] = 1:2:3:4:[]
```

```
flat :: [[a]] -> [a]            aux :: [a] -> [[a]] -> [a]
```

```
flatt lili = [ x | li <- lili, x <- li ]
```

Recursion – summary

- "Divide and conquer":
  - Decide what to do for the base case(s)
  - Construct (conquer) a solution from the recursive solutions (divide) for the other instances
- Each inductive data type gives rise to recursive algorithms
- In general, recursion is implemented iteratively with stack
  – tail recursion can be implemented without stack (essential for FP)
- Terminating
  – each recursive call must bring us closer to the base case
- Correctness – the invariant
  - Verify that each base case satisfies the invariant
  - Assuming that each recursive call satisfies the invariant, show that their combination will maintain invariant
- Hutton, Ch 6 (especially, 6.6!).