Informatics 1

Functional Programming Lectures 5 and 6

Monday 8 and Tuesday 9 October 2012

More fun with recursion

Don Sannella

University of Edinburgh

Tutorials

Attendance is compulsory.

Tuesday/Wednesday	Computation and Logic
Thursday/Friday	*Functional Programming*

You *must* do each week's tutorial exercise! Do it *before* the tutorial!

Bring a *printout* of your work to the tutorial!

You may *collaborate*, but you are responsible for knowing the material.

Mark of 0% on tutorial exercises means you have no incentive to *plagiarize*.

But *you will fail the exam if you don't do the tutorial exercises!*

Start work on the tutorial as *early* as possible.

Required text and reading

Haskell: The Craft of Functional Programming (Third Edition),
Simon Thompson, Addison-Wesley, 2011.

Reading assignment

Monday 24 September 2012	Chapters 1–3 (pp. 1–66)
Monday 1 October 2012	Chapters 4–7 (pp. 67–176)
Monday 8 October 2012	Chapters 8–9 (pp. 177–212)

The assigned reading covers the material very well with plenty of examples.

There will be no lecture notes, just the book. *Get it and read it!*

Part I

Booleans and characters

Boolean operators

```
not :: Bool -> Bool
(&&), (||) :: Bool -> Bool -> Bool

not False  =  True
not True   =  False

False && False  =  False
False && True   =  False
True  && False  =  False
True  && True   =  True

False || False  =  False
False || True   =  True
True  || False  =  True
True  || True   =  True
```

Defining operations on characters

```
isLower :: Char -> Bool
isLower x  =  'a' <= x && x <= 'z'

isUpper :: Char -> Bool
isUpper x  =  'A' <= x && x <= 'Z'

isDigit :: Char -> Bool
isDigit x  =  '0' <= x && x <= '9'

isAlpha :: Char -> Bool
isAlpha x  =  isLower x || isUpper x
```

Defining operations on characters

```
digitToInt :: Char -> Int
digitToInt c | isDigit c  =  ord c - ord '0'

intToDigit :: Int -> Char
intToDigit d | 0 <= d && d <= 9  =  chr (ord '0' + d)

toLower :: Char -> Char
toLower c | isUpper c  =  chr (ord c - ord 'A' + ord 'a')
          | otherwise  =  c

toUpper :: Char -> Char
toUpper c | isLower c  =  chr (ord c - ord 'a' + ord 'A')
          | otherwise  =  c
```

These rely on the conversion functions:

```
ord :: Char -> Int      -- same as: fromEnum :: Char -> Int
chr :: Int -> Char      -- same as: toEnum :: Int -> Char
```

Part II

Conditionals and Associativity

Conditional equations

```
max :: Int -> Int -> Int
max x y | x >= y    =  x
        | y >= x    =  y

max3 :: Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z   =  x
           | y >= x && y >= z   =  y
           | z >= x && z >= y   =  z
```

Conditional equations with otherwise

```
max :: Int -> Int -> Int
max x y | x >= y       =  x
        | otherwise    =  y

max3 :: Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z   =  x
           | y >= x && y >= z   =  y
           | otherwise          =  z
```

Conditional equations with otherwise

```
max :: Int -> Int -> Int
max x y | x >= y      =  x
        | otherwise   =  y

max3 :: Int -> Int -> Int -> Int
max3 x y z | x >= y && x >= z  =  x
           | y >= x && y >= z  =  y
           | otherwise         =  z

otherwise :: Bool
otherwise =  True
```

Conditional expressions

```
max :: Int -> Int -> Int
max x y  =  if x >= y then x else y

max3 :: Int -> Int -> Int -> Int
max3 x y z  =  if x >= y && x >= z then x
               else if y >= x && y >= z then y
               else z
```

Another way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  if x >= y then
                  if x >= z then x else z
               else
                  if y >= z then y else z
```

Key points about conditionals

- As always: write your program in a form that is easy to read. Don't worry (yet) about efficiency: premature optimization is the root of much evil.
- Conditionals are your friend: without them, programs could do very little that is interesting.
- Conditionals are your enemy: each conditional doubles the number of test cases you must consider. A program with five two-way conditionals requires $2^5 = 32$ test cases to try every path through the program. A program with ten two-way conditionals requires $2^{10} = 1024$ test cases.

A better way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  max (max x y) z
```

An even better way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  x `max` y `max` z

max :: Int -> Int -> Int
max x y | x >= y      =  x
        | otherwise   =  y
```

An even better way to define max3

```
max3 :: Int -> Int -> Int -> Int
max3 x y z  =  x `max` y `max` z

max :: Int -> Int -> Int
x `max` y | x >= y      =  x
          | otherwise   =  y
```

`x + y` *stands for* `(+) x y`

`x >= y` *stands for* `(>=) x y`

``x `max` y`` *stands for* `max x y`

Associativity

```
prop_max_assoc :: Int -> Int -> Int -> Bool
prop_max_assoc x y z  =
   (x `max` y) `max` z  ==  x `max` (y `max` z)
```

It doesn't matter where the parentheses go with an associative operator, so we often omit them.

Associativity

```
prop_max_assoc :: Int -> Int -> Int -> Bool
prop_max_assoc x y z  =
  (x `max` y) `max` z  ==  x `max` (y `max` z)
```

It doesn't matter where the parentheses go with an associative operator, so we often omit them.

Why we use infix notation

```
prop_max_assoc :: Int -> Int -> Int -> Bool
prop_max_assoc x y z  =
  max (max x y) z  ==  max x (max y z)
```

This is much harder to read than infix notation!

Key points about associativity

- There are a few key properties about operators: *associativity*, *identity*, *commutativity*, *distributivity*, *zero*, *idempotence*. You should know and understand these properties.

- When you meet a new operator, the first question you should ask is "Is it associative?" The second is "Does it have an identity?"

- Associativity is our friend, because it means we don't need to worry about parentheses. The program is easier to read.

- Associativity is our friend, because it is key to writing programs that run twice as fast on dual-core machines, and a thousand times as fast on machines with a thousand cores. We will study this later in the course.

Part III

Append

Append

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)

   "abc" ++ "de"
=
   ('a' : ('b' : ('c' : []))) ++ ('d' : ('e' : []))
=
   'a' : (('b' : ('c' : [])) ++ ('d' : ('e' : [])))
=
   'a' : ('b' : (('c' : []) ++ ('d' : ('e' : []))))
=
   'a' : ('b' : ('c' : ([] ++ ('d' : ('e' : [])))))
=
   'a' : ('b' : ('c' : ('d' : ('e' : []))))
=
   "abcde"
```

Append

```
(++) :: [a] -> [a] -> [a]
[] ++ ys        =  ys
(x:xs) ++ ys    =  x : (xs ++ ys)

   "abc" ++ "de"
=
   'a' : ("bc" ++ "de")
=
   'a' : ('b' : ("c" ++ "de"))
=
   'a' : ('b' : ('c' : ("" ++ "de")))
=
   'a' : ('b' : ('c' : "de"))
=
   "abcde"
```

Properties of append

```
prop_append_assoc :: [Int] -> [Int] -> [Int] -> Bool
prop_append_assoc xs ys zs  =
  (xs ++ ys) ++ zs  ==  xs ++ (ys ++ zs)

prop_append_ident :: [Int] -> Bool
prop_append_ident xs  =
  xs ++ [] == xs  &&  xs == [] ++ xs

prop_append_cons :: Int -> [Int] -> Bool
prop_append_cons x xs  =
  [x] ++ xs  ==  x : xs
```

Efficiency

```
(++) :: [a] -> [a] -> [a]
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

```
  "abc" ++ "de"
=
  'a' : ("bc" ++ "de")
=
  'a' : ('b' : ("c" ++ "de"))
=
  'a' : ('b' : ('c' : ("" ++ "de")))
=
  'a' : ('b' : ('c' : "de"))
=
  "abcde"
```

Computing `xs ++ ys` takes about n steps, where n is the length of `xs`.

A useful fact

```
-- prop_sum.hs
import Test.QuickCheck

prop_sum :: Integer -> Property
prop_sum n  =  n >= 0 ==>  sum [1..n]  ==  n * (n+1) `div` 2
```

```
[melchior]dts: ghci prop_sum.hs
GHCi, version 6.8.3: http://www.haskell.org/ghc/ :? for help
*Main> quickCheck prop_sum
+++ OK, passed 100 tests.
*Main>
```

Associativity and Efficiency: Left vs. Right

Compare computing (associated to the left)

$$((\texttt{xs}_1 \mathbin{++} \texttt{xs}_2) \mathbin{++} \texttt{xs}_3) \mathbin{++} \texttt{xs}_4$$

with computing (associated to the right)

$$\texttt{xs}_1 \mathbin{++} (\texttt{xs}_2 \mathbin{++} (\texttt{xs}_3 \mathbin{++} \texttt{xs}_4))$$

where n_1, n_2, n_3, n_4 are the lengths of $\texttt{xs}_1, \texttt{xs}_2, \texttt{xs}_3, \texttt{xs}_4$.
Associating to the left takes

$$n_1 + (n_1 + n_2) + (n_1 + n_2 + n_3)$$

steps. If we have m lists of length n, it takes about $m^2 n$ steps.
Associating to the right takes

$$n_1 + n_2 + n_3$$

steps. If we have m lists of length n, it takes about mn steps.

When $m = 1000$, the first is a thousand times slower than the second!

Associativity and Efficiency: Sequential vs. Parallel

Compare computing (sequential)

$$x_1 + (x_2 + (x_3 + (x_4 + (x_5 + (x_6 + (x_7 + x_8))))))$$

with computing (parallel)

$$((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$$

In sequence, summing 8 numbers takes 7 steps.
If we have m numbers it takes $m - 1$ steps.

In parallel, summing 8 numbers takes 3 steps.

$$x_1 + xs_2 \text{ and } x_3 + x_4 \text{ and } x_5 + xs_6 \text{ and } x_7 + x_8$$
$$(x_1 + xs_2) + (x_3 + x_4) \text{ and } (x_5 + xs_6) + (x_7 + x_8),$$
$$((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8))$$

If we have m numbers it takes $\log_2 m$ steps.

When $m = 1000$, the first is a hundred times slower than the second!

Part IV

Counting

Counting

```
Prelude [1..3]
[1,2,3]
Prelude enumFromTo 1 3
[1,2,3]
```

`[m..n]` *stands for* `enumFromTo m n`

Recursion

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo m n | m > n     =  []
               | m <= n    =  m : enumFromTo (m+1) n
```

How enumFromTo works (recursion)

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo m n | m > n    =  []
               | m <= n   =  m : enumFromTo (m+1) n

  enumFromTo 1 3
=
  1 : enumFromTo 2 3
=
  1 : (2 : enumFromTo 3 3)
=
  1 : (2 : (3 : enumFromTo 4 3))
=
  1 : (2 : (3 : []))
=
  [1,2,3]
```

Factorial

```
Main*> factorial 3
```

Library functions

```
factorial :: Int -> Int
factorial n  =  product [1..n]
```

Recursion

```
factorialRec :: Int -> Int
factorialRec n  =  fact 1 n
  where
  fact :: Int -> Int -> Int
  fact m n | m > n     =  1
           | m <= n    =  m * fact (m+1) n
```

How factorial works (recursion)

```
factorialRec :: Int -> Int
factorialRec n  =  fact 1 n
  where
  fact :: Int -> Int -> Int
  fact m n | m > n     =  1
           | m <= n    =  m * fact (m+1) n

   factorialRec 3
=
   fact 1 3
=
   1 * fact 2 3
=
   1 * (2 * fact 3 3)
=
   1 * (2 * (3 * fact 4 3))
=
   1 * (2 * (3 * 1))
=
   6
```

Counting forever!

```
Prelude [0..]
[0,1,2,3,4,5,...
Prelude enumFrom 0
[0,1,2,3,4,5,...
```

`[m..]` *stands for* `enumFrom m`

Recursion

```
enumFrom :: Int -> [Int]
enumFrom m = m : enumFrom (m+1)
```

How enumFrom works (recursion)

```
enumFrom :: Int -> [Int]
enumFrom m  =  m : enumFrom (m+1)

  enumFrom 0
=
  0 : enumFrom 1
=
  0 : (1 : enumFrom 2)
=
  0 : (1 : (2 : enumFrom 3))
=
  ...
=
  [0,1,2,...   -- computation goes on forever!
```

Part V

Zip and search

Zip

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys             =  []
zip xs []             =  []
zip (x:xs) (y:ys)     =  (x,y) : zip xs ys

    zip [0,1,2] "abc"
=
    (0,'a') : zip [1,2]"bc"
=
    (0,'a') : ((1,'b') : zip [2] "c")
=
    (0,'a') : ((1,'b') : ((2,'c') : zip [] ""))
=
    (0,'a') : ((1,'b') : ((2,'c') : []))
=
    [(0,'a'),(1,'b'),(2,'c')]
```

Two alternative definitions of zip

Liberal

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys             =  []
zip xs []             =  []
zip (x:xs) (y:ys)     =  (x,y) : zip xs ys
```

Conservative

```
zipHarsh :: [a] -> [b] -> [(a,b)]
zipHarsh [] []             =  []
zipHarsh (x:xs) (y:ys)     =  (x,y) : zipHarsh xs ys
```

Lists of different lengths

```
Prelude> zip [0,1,2] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zipHarsh [0,1,2] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zip [0,1,2] "abcde"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zipHarsh [0,1,2] "abcde"
[(0,'a'),(1,'b'),(2,'c')*** Exception:
Non-exhaustive patterns in function zipHarsh

Prelude> zip [0,1,2,3,4] "abc"
[(0,'a'),(1,'b'),(2,'c')]

Prelude> zipHarsh [0,1,2,3,4] "abc"
[(0,'a'),(1,'b'),(2,'c')*** Exception:
Non-exhaustive patterns in function zipHarsh
```

More fun with zip

```
Prelude> zip [0..] "words"
[(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')]

Prelude> let pairs xs = zip xs (tail xs)
Prelude> pairs "words"
[('w','o'),('o','r'),('r','d'),('d','s')]
```

Zip with an infinite list

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys            =  []
zip xs []            =  []
zip (x:xs) (y:ys)    =  (x,y) : zip xs ys
```

```
  zip [0..] "abc"
=
  (0,'a') : zip [1..] "bc"
=
  (0,'a') : ((1,'b') : zip [2..] "c")
=
  (0,'a') : ((1,'b') : ((2,'c') : zip [3..] ""))
=
  (0,'a') : ((1,'b') : ((2,'c') : zip (3 : [4..]) ""))
=
  (0,'a') : ((1,'b') : ((2,'c') : []))
=
  [(0,'a'),(1,'b'),(2,'c')]
```

Computer can determine `(3 : [4..])` $\neq$ `[]` without computing `[4..]`.

Dot product of two lists

Comprehensions and library functions

```
dot :: Num a => [a] -> [a] -> a
dot xs ys  =  sum [ x*y | (x,y) <- zipHarsh xs ys ]
```

Recursion

```
dotRec :: Num a => [a] -> [a] -> a
dotRec [] []             =  0
dotRec (x:xs) (y:ys)     =  x*y + dotRec xs ys
```

How dot product works (comprehension)

```
dot :: Num a => [a] -> [a] -> a
dot xs ys  =  sum [ x*y | (x,y) <- zip xs ys ]

    dot [2,3,4] [5,6,7]
=
    sum [ x*y | (x,y) <- zip [2,3,4] [5,6,7] ]
=
    sum [ x*y | (x,y) <- [(2,5), (3,6), (4,7)] ]
=
    sum [ 2*5, 3*6, 4*7 ]
=
    sum [ 10, 18, 28 ]
=
    56
```

How dot product works (recursion)

```
dotRec :: Num a => [a] -> [a] -> a
dotRec [] []            =  0
dotRec (x:xs) (y:ys)   =  x*y + dotRec xs ys

  dotRec [2,3,4] [5,6,7]
=
  dotRec (2:(3:(4:[]))) (5:(6:(7:[])))
=
  2*5 + dotRec (3:(4:[])) (6:(7:[]))
=
  2*5 + (3*6 + dotRec (4:[]) (7:[]))
=
  2*5 + (3*6 + (4*7 + dotRec [] []))
=
  2*5 + (3*6 + (4*7 + 0))
=
  10 + (18 + (28 + 0))
=
  56
```

Search

```
Main*> search "bookshop" 'o'
[1,2,6]
```

Comprehensions and library functions

```
search :: Eq a => [a] -> a -> [Int]
search xs y = [ i | (i,x) <- zip [0..] xs, x==y ]
```

Recursion

```
searchRec :: Eq a => [a] -> a -> [Int]
searchRec xs y  =  srch xs y 0
  where
  srch :: Eq a => [a] -> a -> Int -> [Int]
  srch [] y i       =  []
  srch (x:xs) y i
    | x == y            =  i : srch xs y (i+1)
    | otherwise         =  srch xs y (i+1)
```

How search works (comprehension)

```
search :: Eq a => [a] -> a -> [Int]
search xs y = [ i | (i,x) <- zip [0..] xs, x==y ]

  search "book" 'o'
=
  [ i | (i,x) <- zip [0..] "book", x=='o' ]
=
  [ i | (i,x) <- [(0,'b'),(1,'o'),(2,'o'),(3,'k')], x=='o' ]
=
  [0|'b'=='o']++[1|'o'=='o']++[2|'o'=='o']++[3|'k'=='o']
=
  []++[1]++[2]++[]
=
  [1,2]
```

How search works (recursion)

```
searchRec xs y  =  srch xs y 0
  where
  srch [] y i                          =  []
  srch (x:xs) y i   | x == y           =  i : srch xs y (i+1)
                    | otherwise        =  srch xs y (i+1)

  searchRec "book" 'o'
=
  srch "book" 'o' 0
=
  srch "ook" 'o' 1
=
  1 : srch "ok" 'o' 2
=
  1 : (2 : srch "k" 'o' 3)
=
  1 : (2 : srch "" 'o' 4)
=
  1 : (2 : [])
=
  [1,2]
```

Part VI

Select, take, and drop

Select, take, and drop

```
Prelude> "words" !! 3
'd'

Prelude> take 3 "words"
"wor"

Prelude> drop 3 "words"
"ds"
```

Select, take, and drop (comprehensions)

```
selectComp :: [a] -> Int -> a  -- (!!)
selectComp xs i  =  the [ x | (j,x) <- zip [0..] xs, j == i ]
  where
  the [x]  =  x

takeComp :: Int -> [a] -> [a]
takeComp i xs  =  [ x | (j,x) <- zip [0..] xs, j < i ]

dropComp :: Int -> [a] -> [a]
dropComp i xs  =  [ x | (j,x) <- zip [0..] xs, j >= i ]
```

How take works (comprehension)

```
takeComp :: Int -> [a] -> [a]
takeComp i xs  =  [ x | (j,x) <- zip [0..] xs, j < i ]

  take 3 "words"
=
  [ x | (j,x) <- zip [0..] "words", j < 3 ]
=
  [ x | (j,x) <- [(0,'w'),(1,'o'),(2,'r'),(3,'d'),(4,'s')],
        j < 3 ]
=
  ['w'|0<3]++['o'|1<3]++['r'|2<3]++['d'|3<3]++['s'|4<3]
=
  ['w']++['o']++['r']++[]++[]
=
  "wor"
```

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

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

Natural numbers

Every natural number can be written using only `(+1)` and `0`.

```
3  =  ((0 + 1) + 1) + 1
```

A *recursive* definition: A *natural number* is either

- *zero*, written `0`, or
- *successor*, written `n+1`
  with *predecessor* `n` (a natural number).

Select, take, and drop (recursion)

```
(!!) :: [a] -> Int -> a
(x:xs) !! 0       =  x
(x:xs) !! (i+1)   =  xs !! i

take :: Int -> [a] -> [a]
take 0 xs            =  []
take i []            =  []
take (i+1) (x:xs)    =  x : take i xs

drop :: Int -> [a] -> [a]
drop 0 xs            =  xs
drop i []            =  []
drop (i+1) (x:xs)    =  drop i xs
```

Pattern matching and conditionals (squares)

Pattern matching

```
squares :: [Integer] -> [Integer]
squares []       =  []
squares (x:xs)   =  x*x : squares xs
```

Conditionals with binding

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

Pattern matching and conditionals (take)

Pattern matching

```
take :: Int -> [a] -> [a]
take 0 xs            =  []
take i []            =  []
take (i+1) (x:xs)    =  x : take i xs
```

Conditionals with binding

```
take :: Int -> [a] -> [a]
take j ws
  if j == 0 || null ws then
    []
  else
    let
      x  = head ws
      xs = tail ws
      i  = j-1
    in
      x : take i xs
```

Pattern matching and guards (take)

Pattern matching

```
take :: Int -> [a] -> [a]
take 0 xs             =   []
take i []             =   []
take (i+1) (x:xs)     =  x : take i xs
```

Guards

```
take :: Int -> [a] -> [a]
take 0 xs                     =   []
take j []                     =   []
take j (x:xs) | j > 0         =  x : take (j-1) xs
```

How take works (recursion)

```
take :: Int -> [a] -> [a]
take 0 xs           =  []
take i []           =  []
take (i+1) (x:xs)   =  x : take i xs

  take 3 "words"
=
  'w' : take 2 "ords"
=
  'w' : ('o' : take 1 "rds")
=
  'w' : ('o' : ('r' : take 0 "ds"))
=
  'w' : ('o' : ('r' : []))
=
  "wor"
```

How take works (recursion reprise)

```
take :: Int -> [a] -> [a]
take 0 xs          =  []
take i []          =  []
take (i+1) (x:xs)  =  x : take i xs

   take 3 "words"
=
   take (((0+1)+1)+1) ('w':('o':('r':('d':('s':[])))))
=
   'w' : take ((0+1)+1) ('o':('r':('d':('s':[]))))
=
   'w' : ('o' : take (0+1) ('r':('d':('s':[]))))
=
   'w' : ('o' : ('r' : take 0 ('d':('s':[]))))
=
   'w' : ('o' : ('r' : []))
=
   "wor"
```

The infinite case

```
take :: Int -> [a] -> [a]
take 0 xs              =  []
take i []              =  []
take (i+1) (x:xs)      =  x : take i xs

takeComp :: Int -> [a] -> [a]
takeComp i xs  =  [ x | (j,x) <- zip [0..] xs, j < i ]

Prelude> take 3 [10..]
[10,11,12]

Prelude> takeComp 3 [10..]
[10,11,12    -- computation goes on forever!
```

The infinite case explained

Function `takeComp` is equivalent to `takeCompRec`.

```
takeCompRec :: Int -> [a] -> [a]
takeCompRec i xs  =  helper 0 i xs
  where
  helper j i []                        = []
  helper j i (x:xs) | j > i            =  x : helper (j+1) i xs
                    | otherwise        =  helper (j+1) i xs

  takeCompRec 3 [10..]
=
  helper 0 3 [10..]
=
  10 : helper 1 3 [11..]
=
  10 : (11 : helper 2 3 [12..])
=
  10 : (11 : (12 : helper 3 3 [13..]))
=
  10 : (11 : (12 : helper 4 3 [14..]))
= ...
```

Part VII

Arithmetic

Arithmetic (recursion)

```
 (+) :: Int -> Int -> Int
m + 0       =  m
m + (n+1)   =  (m + n) + 1

 (*) :: Int -> Int -> Int
m * 0       =  0
m * (n+1)   =  (m * n) + m

 (^) :: Int -> Int -> Int
m ^ 0       =  1
m ^ (n+1)   =  (m ^ n) * m
```

How arithmetic works (recursion)

```
(+) :: Int -> Int -> Int
m + 0         =  m
m + (n+1)     =  (m + n) + 1

   2 + 3
=
   (2 + 2) + 1
=
   ((2 + 1) + 1) + 1
=
   (((2 + 0) + 1) + 1) + 1
=
   ((2 + 1) + 1) + 1
=
   5
```

How arithmetic works (recursion reprise)

```
(+) :: Int -> Int -> Int
m + 0       =  m
m + (n+1)   =  (m + n) + 1

    2 + 3
=
    ((0 + 1) + 1) + (((0 + 1) + 1) + 1)
=
    (((0 + 1) + 1) + ((0 + 1) + 1)) + 1
=
    ((((0 + 1) + 1) + (0 + 1)) + 1) + 1
=
    (((((0 + 1) + 1) + 0) + 1) + 1) + 1
=
    ((((0 + 1) + 1) + 1) + 1) + 1
=
    5
```

Giuseppe Peano (1858–1932)

The definition of the natural numbers is named the *Peano axioms* in his honour.
Made key contributions to the modern treatment of mathematical induction.