1.md

class: center, middle

Functional Programming

1. Lists and Functions

Frank C Langbein
frank@langbein.org

Version 1.0

Welcome!

```
primes = filterPrime [2..] -- all prime numbers
  where filterPrime (p:xs) = p : filterPrime [ x | x <- xs, x `mod` p /= 0 ]
```

- `haskell.org`
    - Glasgow Haskell Compiler (ghc[i], version 7 or 8)
    - On Linux
    - Editor, IDE, etc. for you to choose
        - vi and a shell is all that is needed
- Reading
    - M Lipovaca. Learn You a Haskell for Great Good, No Strach Press, 2011. http://learnyouahaskell.com/
    - C Allen, J Moronuki. Haskell Programming from First Principles [Early Access], Gumroad, 2015.
- Slides in markdown!
    - Using remark.js, readable via browser
    - Or available as PDF

Introduction

- All about **functions**: `f: R -> R, x -> x^2; g: R -> R, x -> sqrt(x)`
    - And **combining** them: `g(f(x)) = |x|`

    ```
    g x = sqrt x -- Space as operator: apply function to argument
    f x = x * x  -- Unless there is an infix operator (for readability)
    g (f (-4))   -- Parenthesis needed (space is left associated),
                -- and - is also an operator
    ```

- Increasingly important in industry
    - High-level concepts
    - Concurrency!
- Operate on data structure as a whole
    - Imperative: operate on items in sequence in a loop
    - Functional: operate on the sequence
- *Garbage collection*, *higher-order functions*, *generics*, *list comprehensions*, *type classes*

Functions

- A **function** is a special relation, `f: X -> Y, y = f(x), (x,y) in XxY`
    - A set of `(input,output)` value pairs
    - Input and output are taken from sets (types): domain, codomain
        - Those sets usually have some structure, can be quite complex
        - There can be multiple arguments; also see *currying* later
    - Each input value is present *exactly once* in the relation
        - Only one output value for each input value
        - Output values can be present multiple times, once, or not at all
- Functions have **no side effects**!
    - There are also **actions**, but these are **values**!
    - To realise sideffects, such as I/O
- **Type signature**: `NAME :: DOMAIN -> CODOMAIN`
- **Function declaration**: `NAME ARGS = FUNCTION_BODY`

```
f: Int -> Int
f x = x * x
```

Lists

- **List**: sequence of elements from a single type
    - Most important data structure

```
someNumbers :: [Int]
someNumbners = [1,2,3]

moreNumbers :: [Int]
moreNumbers = [1..10]

someChars :: String -- Equivalent to [Char] type
someChars = [ 'T', 'e', 's', 't', '1' ]

someLists :: [[Int]]
someLists= [[1],[1,2],[1,2,3],[],[5,6,7]]

someFunctions :: [Int -> Int]
someFunctions = [f,g,\x -> x + 1]

someStuff = [1, "Test", [2,3]] -- Cannot mix types
```

List Comprehensions: Generators and Guards

- **Generators**, `x <- EXPR` , to draw values from expression

```
[ x*x | x <- [1,2,3] ]
[ toLower c | c <- "Hello, World!" ]
[ (x, even x) | x <- [1..10] ]
```

- **Guards** as predicates (map values to `Bool`) to filter elements

```
[ x | x <- [1..10], odd x ]
[ x*x | x <- [1..10], even x ]
[ x | x <- [42, -11, 23, 42, 0, -1], x > 0 ]
```

```
[ toLower c | c <- "Hello, World!", isAlpha c ]
```

- Sums and products

```
sum [1,2,3]
sum []                               -- What value should this be?
sum [ x*x | x <- [1..10], odd x]     -- Generator and guard
product [1,2,3]
product []                           -- Why is this 1?
product [ x*x | x <- [1..10], odd x]
factorial n = product [2..n]         -- Defines a new function
```

Functions and Lists

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

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

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

main :: IO () -- main function for compilation
              -- IO indicates this is an action to handle sideffects
main = do -- execute things in sequence
          putStrLn("Sum of odd Squares from 1 to ?:")
          inp <- getLine       -- read any stdin line
          x = read inp :: Int  -- and convert to integer
          print (sum (squares (odds [1..x])))
          print (sumSqOdd [1..x])
```

Cons and Append

- **Cons** (:) combines an element and a list: `(:) :: a -> [a] -> [a]`
- **Append** (++) merges two lists: `(++) :: [a] -> [a] -> [a]`

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

  - So a list can be written as `1 : (2 : (3 : []))`
    - A list is either '[]' (empty) or
    - `x:xs` where `x` is an element and `xs` is a list
- **Recursive definition** of a list
  - Head: `(head [1,2,3]) = 1`
  - Tail: `(tail [1,2,3]) = [2,3]`
  - Null: `null [] = True` , `null [1,2,3] = False`
- Not meaningless statements!
  - "Brexit means Brexit" [T May] -- infinite loop: `while (true) do nothing`
  - But you can represent **infinite data**, e.g. natural numbers:
    - There is one number
    - Every number has a successor

Square every element in a list

```
squares :: [Int] -> [Int] -- Comprehension
```

```
squares xs = [ x*x | x <- xs ]
```

--

```
squaresRec :: [Int] -> [Int] -- Recursive (with pattern matching)
squaresRec []     = []
squaresRec (x:xs) = x*x : squaresRec xs
```

--

```
squaresCond :: [Int] -> [Int] -- Conditionals (with binding)
squaresCond ls =
  if null ls then
    []
  else
    let
      x = head ls
      xs = tail ls
    in
      x * x : squaresCond xs
```

Filtering: odds

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

--

```
oddsRec :: [Int] -> [Int] -- Recursion (with Guards)
oddsRec []                 = []
oddsRec (x:xs) | odd x     = x : oddsRec xs
               | otherwise = oddsRec xs
               -- Checked in order to decide which to use!
```

--

```
oddsCond :: [Int] -> [Int] -- Conditionals (with binding)
oddsCond ls =
  if null ls then
    []
  else
    let
      x = head ls
      xs = tail ls
    in
      if odd x then
        x : oddsCond xs
      else
        oddsCond xs
```

Append and Complexity

- Definition of append `++` operator

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

- This executes

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

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

- Length of the first argument determines number of operations

Associative Operators

- **associative**
  - `(xs ++ yz) ++ zs == xs ++ (ys ++ zs)`
  - This is useful for efficiency and concurrency
    - `"abcdef...y" ++ "z"` vs. `"a" ++ "bc...z"` --
- **Left** vs **right** associated append
  - **Left**: `((x1 ++ x2) ++ x3) ++ x4)`
    - Number of operations: `n1 + (n1 + n2) + (n1 + n2 + n3)`
  - **Right**: `x1 ++ (x2 ++ (x3 ++ x4))`
    - Number of operations: `n1 + n2 + n3`
  - If we have `m` lists of length `n`
    - Left: `m^2 * n`
    - Right: `m * n` --
- **Parallelise** `sum`
  - `x1 + (x2 + (x3 + (x4 + (x5 + x6)))) ==  ((x1+x2) + (x3+x4)) + ((x5+x6) + (x7+x8))`
  - `m-1` vs `log m` for `m` numbers

More Operator Properties

- **identity**
  - Does the operator have an identity?
  - `xs ++ [] == xs && [] ++ xs == xs` --
- **commutativity**
  - `xs ++ ys \= ys ++ xs` - append is not commutative!
    - Cannot reorder sequence for speedup.
  - `a+b == b+a` - addition is commutative
    - Can reorder! --
- **distributivity**
  - `x * y + x * z == x * (x + z)`
  - Helps to reduce number of operations --
- **zero**
  - `0 * (...) == 0`
  - Avoid executing dead code! --
- **idempotence**
  - `f (f x) = f x` - fixed point of `f`
  - E.g. set union and intersection
  - Avoid doing unnecessary things.

Counting

```
enumFromTo :: Int -> Int -> [Int] -- construct a list of integers from m to n
enumFromTo m n | m >  n = []
               | m <= n = m : enumFromTo (m+1) n
```

```
factorial :: Int -> Int -- as enum, but multiply instead of cons
factorial n = fact 1 n
  where                 -- to introduce helper function, etc
    fact :: Int -> Int -> Int
    fact m n | m >  n = 1
             | m <= n = m * fact (m+1) n
  -- Of course there is product [1..n] and
  -- you do not really need a helper function!

enumFrom :: Int -> [Int] -- Count forever!
enumFrom m = m : enumForm (m+1)
-- Thanks to lazy evaluation!
-- Defines an infinite data structure!
-- Works unless you really need or ask to create all numbers
```

Zipping

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

zipConservative :: [a] -> [b] -> [(a,b)] -- (not so liberal)
zipConservative [] []         = []
zipConservative (x:xs) (y:ys) = (x,y) : zipConservative xs ys
```

- For example

```
zip [1..] "counting"

let pairs xs = zip xs (tail xs)
pairs "counting"
```

Searching

```
search "needle" 'e'

search :: Eq a => [a] -> a -> [Int]
          -- "Eq a": equality must work with type a
search 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)

-- Or:

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

Select, Take and Drop

- `xs !! n` selects the `n` th character from the list
  - `"words" !! 3`
- `take n xs` returns the first `n` items from the list
  - `take 3 "words"`

- `drop n xs` returns all except the first n items in the list
    - `drop 3 "words"`
- How would you implement these?

Map

- Map operator defined as

```
map :: (a->b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

- So we can define squares as

```
squares = map (\x -> x * x) -- lambda expression!
                            -- note the missing argument!
```

Filter

- We used guards or comprehension to select elements from a list

```
positives :: [Int] -> Int
positives []                = []
positives (x:xs) | x > 0     = [ x | x <- xs, x > 0]
                 | otherwise = positives xs

-- Or:

positives :: [Int] -> Int
positives xs = [ x | x <- xs, x > 0]
```

- Instead, define `filter` operator as

```
filter :: (a-> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

positives = filter (\x -> x > 0) -- with predicate (lambda expression)
```

Fold

- `sum` , `product` , `concatenate` combines elments in lists with `+` , `*` , `++`
- Better, use `foldr` (fold right) defined as

```
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)

-- Or with infix notation:

foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f v []     = v
foldr f v (x:xs) = x `f` (foldr f v xs)
```

- Then

```
sum = foldr (+) 0

product = foldr (*) 1
```

```
concatenate = foldr (++) []
```

Sum of Positive Squares

```
f :: [Int] -> Int
f xs = sum [ x*x | x <- xs , x > 0]

-- Or with foldr

f :: [Int] -> Int
f xs = foldr (+) 0 (map sqr (filter pos xs))
  where
    sqr x = x * x
    pos x = x > 0

-- And now add lambda expressions

f xs = foldr (+) 0 (map (\x -> x * x) (filter (\x -> x>0) xs))
```

Currying

- Finally, time to explain the notation
  - `f :: a -> b -> c`
    - mapsto (->) associated to the right
  - f x y
    - function-application ( ) associates to the left
- So...

  ```
  add :: Int -> (Int -> Int)
  (add x) y = x + y
  ```

  is executed as...

  ```
  (add 1) 2 = (1+) 2 = 3
  ```

  - Hence,
    function of two numbers
    ==
    function of the first number that returns a function of the second number

Haskell Curry

```
add :: Int -> Int -> Int
add x y = x + y
-- is the same than
add :: Int -> (Int -> Int)
add x = h
  where
    h y = x + y

-- So
add 3 4
-- is
(add 3) 4
```

- Named after **Haskell Curry** (1900-1982); also mentioned by **Moses Ilyich Schönfinkel** (aka Моисей Исаевич Шейнфинкель; 1889-1942), **Gottlob Frege** (1848-1925)
- **Currying** allows to **apply a function partially**

  ```
  sum :: [Int] -> Int
  ```

```
sum xs = foldr (+) 0 xs
-- Or (as I already sneaked in)
sum = foldr (+) 0
```