# Outline of Lecture 9

- Using higher-order functions: example
- Folding revisited
- Additional Haskell libraries and other resources

# Summary: various ways to define a function (reminder)

A function that negates all the elements in the integer list and then multiplies them together.

```
prodNegated ::  [Integer] -> Integer
prodNegated xs = foldr (*) 1 (map negate xs)

prodNegated_2 xs = (foldr (*) 1 . map negate) xs

prodNegated_3 xs = foldr (*) 1 $ map negate $ xs

prodNegated_4 = \xs -> (foldr (*) 1 . map negate) xs

prodNegated_5 = foldr (*) 1 . map negate

prodNegated_6 = foldr (\x y -> negate x * y) 1
```

# Datatype constructors

Datatype constructors are functions too ⇒ they can be partially applied, passed as arguments or returned as results

Example:

```
data People = Person String Int deriving (Show)

somePeople = zipWith Person ["Bernie Stauskas",
"Bob Dyllan"] [25,71]

> print somePeople
[Person "Bernie Stauskas" 25,
Person "Bob Dyllan" 71]
```

# Some properties of higher-order functions

- `f . (g . h) = (f . g) . h`
- `map (f . g) = map f . map g`
- `map f (xs ++ ys) = map f xs ++ map f ys`
- `filter p (xs ++ ys) = filter p xs ++ filter p ys`
- `filter p . map f = map f . filter (p . f)`
- `foldr f st (xs ++ ys) =`
  `f (foldr f st xs) (foldr f st ys)`
- `foldr f st . map g = foldr (\x y -> f (g x) y) st`

# Functions as data

Example: recognising regular expressions – patterns on strings of characters:

- $\epsilon$ – empty string
- $x$ – any single character
- $r_1|r_2$ – either pattern $r_1$ or $r_2$
- $r_1 r_2$ – $r_1$ followed by $r_2$
- (r)* – repeating $r$ zero or more times

Matching arbitrary strings against such patterns

# Functions as data (cont.)

A Haskell implementation of regular expressions:

```
type RegExp = String -> Bool

epsilon ::  RegExp
epsilon = (=="")

char ::  Char -> RegExp
char ch = (==[ch])

(|||) ::  RegExp -> RegExp -> RegExp
e1 ||| e2 = \x -> e1 x || e2 x
```

# Functions as data (cont.)

A Haskell implementation of regular expressions (cont.):

```
(<*>) ::  RegExp -> RegExp -> RegExp
e1 <*> e2 = \x ->
    or [e1 y && e2 z | (y,z) <- splits x]

star ::  RegExp -> RegExp
star p = epsilon ||| (p <*> star p)
```

`splits :: String -> [(String,String)]` returns all the ways a string can be split into two

# Folding revisited

- Folding: deconstructing data, reducing their structure
- Folding to the right (using `foldr` function):

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
```

- Recursive definition of folding to the right:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- Evaluation unfolding according to `foldr`:

```
foldr f z [1,2,3] =
1 `f` (foldr f z [2,3]) =
1 `f` ( 2 `f` (foldr f z [3])) =
1 `f` ( 2 `f` (3 `f` (foldr f z []))) =
1 `f` ( 2 `f` (3 `f` z))
```

# Folding to the right (cont.)

- Example: evaluation unfolding according to `foldr` (with `(+)`):

  `foldr (+) 0 [1,2,3]` $=$ `1 + ( 2 + (3 + 0)) = 6`

  or

  `foldr (+) 0 [1,2,3]` $=$ `(+) 1 ( (+) 2 ( (+) 3 0)) = 6`

- If the folding function is *non-strict* in the second argument, i.e., it does not require evaluation of the second argument to return a result, `foldr` can be applied to infinite data structures :

  `foldr const 0 [1..]  = 1`

  or (since the function `any` can be expressed by `foldr`)

  `any even [1..]  = True`

  Here `[1..]` – infinite list of integer numbers starting with 1

# Folding to the left (cont.)

- Folding to the left:

```
foldl ::  (b -> a -> b) -> b -> [a] -> b
```

- Recursive definition of folding to the right:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- Evaluation unfolding according to `foldl`:

```
foldl f z [1,2,3]
foldl f (z `f` 1) [2,3]
foldl f ((z `f` 1) `f` 2) [3]
foldl f (((z `f` 1) `f` 2) `f` 3) []
((z `f` 1) `f` 2) `f` 3
```

# Folding to the left (cont.)

- Example: evaluation unfolding according to `foldl` (with `(+)`):

  $$\texttt{foldl (+) 0 [1,2,3]} = \texttt{((0 + 1) + 2) + 3}$$

  or

  $$\texttt{foldl (+) 0 [1,2,3]} = \texttt{(+) ((+) ((+) 0 1) 2) 3}$$

- The relationship between `foldr` and `foldl` (only for finite lists!):

```
foldr f z xs = foldl (flip f) z (reverse xs)
```

- `foldl` traverses to the list end before evaluation starts, hence cannot be applied to infinite lists

# Folding revisited (cont.)

- For associative functions like `(+)`, both versions of folding produce the same results:

```
foldr (+) 0 [1,2,3] == 6
foldl (+) 0 [1,2,3] == 6
```

- For non-associative function, the results can be quite different:

```
foldr (^) 2 [1..3] == 1
foldl (^) 2 [1..3] == 64
```

  or

```
foldr (:) [] [1..3] == [1,2,3]
foldl (flip (:)) [] [1..3] == [3,2,1]
```

  where `flip` creates a binary function with the reversed order of parameters

# Folding revisited (summary)

- `foldr` associates to the right when evaluating
- Can be thought as alternation between applications `foldr` and the folding function *f*
- The next invocation of `foldr` is thus *conditional* (if necessary), allowing to work with infinite lists:

```
foldr const 0 [1..]  == 1
```

- `foldl` associates to the left when evaluating
- `foldl` self-calls (tail-calls) through the list, only beginning to produce values after reaching the end of the list
- Because of that, `foldl` cannot be used with infinite lists

# Folding revisited (summary)

- `foldl` can be also inefficient with very large lists
- The reason: evaluation and simplification is postponed until all list structure is unfolded
- `foldl'` – a more efficient version of `foldl` (located in the module Data.List)
- Forcefully evaluates and simplifies the inner expression  z ‘f‘ x before a recursive call foldl f (z ‘f‘ x) xs
- More about evaluation order as well as strict and non-strict (lazy) computations in Haskell – in later lectures

# Scans

- A combination of mapping and folding that produces all the intermediate results of folding as a list
- Scanning to the right:

```
scanr ::  (a -> b -> b) -> b -> [a] -> [b]
scanr (+) 0 [1..5] == [15,14,12,9,5,0]
```

- Scanning to the left:

```
scanl ::  (b -> a -> b) -> b -> [a] -> [b]
scanl (+) 0 [1..5] == [0,1,3,6,10,15]
```

- Properties of `scanr` and `scanl`:

```
head(scanr f z xs) == foldr f z xs
last(scanl f z xs) == foldl f z xs
```

# Additional Haskell libraries and other resources

- Standard Haskell installation – the Haskell Platform
- In addition to the definitions in `Prelude`, many other functions/ modules/ libraries (hierarchical modules) / packages are available (either in the Haskell Platform or externally)
- Module names are often hierarchical (examples: `QuickCheck.Test`, `Data.Char`, `Data.List`, `Foreign.Marshal.Alloc.Data.Bool`)
- Moreover, additional (package) downloading and installing via using the tool Cabal (a part of the Haskell platform)

# `import` command revisited

- **import Mod** – all the `Mod` definitions are imported (simple identifiers x, y, … or qualified ones `Mod`.x, `Mod`.y, …)
- **import Mod (x,y)** – only x and y are imported from `Mod`
- **import qualified Mod (x,y)** – only qualified identifiers, e.g., `Mod`.x, `Mod`.y, can be used
- **import Mod hiding (x,y)** – all except x and y are imported
- **import Mod as Foo** – the imported module is renamed
- We can use **qualified, as, hiding** keywords in one command
- `Prelude` can be hidden, qualified, and renamed as well: **import qualified Prelude as P hiding (zip)**

# Some libraries from the Haskell Platform

- **Data** – contain additional datatypes (like `Data.Array`) or additional operations on the existing types (like `Data.List` or `Data.Char`)
- **Control** – provides application control (e.g., sequencing of computations), basic IO mechanism, concurrent executions, exception handling
- **Numeric** – contains functions to read and print numbers in a variety of formats
- **Foreign** – supports interworking with other programming languages
- **System** – support various forms of IO handling (e.g., interaction with command line)

# Additional Haskell resources : Hackage and Cabal

- **Hackage** – an online repository for Haskell packages and libraries (currently over 5000 packages)
- `http://hackage.haskell.org`
- A package: a collection of Haskell modules. Can contain also C code, documentation, test cases, and so on
- **Cabal** – a command line tool for installing packages (and the packages they depend on)
- **Cabal** is a part of the Haskell Platform distribution (quick documentation – `https://wiki.haskell.org/Cabal-Install`)

# Additional Haskell resources : Documentation

- `http://hackage.haskell.org/package` – documentation for many external packages listed by category but also searchable
- `http://www.haskell.org/hoogle` – search for many standard libraries (by name and type)