CS 403: Introduction to functional programming

Gregory Mierzwinski

Fall 2022

FUNCTIONAL PROGRAMMING IS PROGRAMMING WITHOUT...

- Selective assignments (`a[i] = 6` is not allowed)
  - The goal of an imperative program is to change the state [of the machine]
  - The goal of a functional programs is to evaluate (reduce, simplify) expressions
- More generally updating assignments (`y = x + 1` is fine but `x = x + 1` is not fine)
  - A variable in an imperative program: a name for a container
  - There is no proper concept of “variable” in functional programs. What is called “variable” is a name for an expression
- Explicit pointers, storage and storage management
- Input/output
- Control structures (loops, conditional statements)
- Jumps (break, goto, exceptions)

WHAT'S LEFT?

- Expressions (without side effects)
  - Referential transparency (i.e., substitutivity, congruence)
- Definitions (of constants, functions)
  - Functions defined almost as in mathematics

Math	Haskell
$\mathrm{square}(x) = x \times x$	

- Types (including higher-order, polymorphic, and recursively-defined)
  - tuples, lists, trees, shared sub-structures, implicit cycles
- Automatic storage management (garbage collection)
  - Actually we do not care about storage at all; we abstract over the physical machine instead!

WHAT'S LEFT?

- Expressions (without side effects)
  - Referential transparency (i.e., substitutivity, congruence)
- Definitions (of constants, functions)
  - Functions defined almost as in mathematics

Math	Haskell
$\text{square} : \mathbb{N} \to \mathbb{N}$ $\text{square}(x) = x \times x$	

- Types (including higher-order, polymorphic, and recursively-defined)
  - tuples, lists, trees, shared sub-structures, implicit cycles
- Automatic storage management (garbage collection)
  - Actually we do not care about storage at all; we abstract over the physical machine instead!

WHAT'S LEFT?

- Expressions (without side effects)
  - Referential transparency (i.e., substitutivity, congruence)
- Definitions (of constants, functions)
  - Functions defined almost as in mathematics

Math	Haskell
$\text{square} : \mathbb{N} \to \mathbb{N}$	`square ::  Integer -> Integer`
$\text{square}(x) = x \times x$	`square x = x * x`

- Types (including higher-order, polymorphic, and recursively-defined)
  - tuples, lists, trees, shared sub-structures, implicit cycles
- Automatic storage management (garbage collection)
  - Actually we do not care about storage at all; we abstract over the physical machine instead!

- Expressions (without side effects)
  - Referential transparency (i.e., substitutivity, congruence)
- Definitions (of constants, functions)
  - Functions defined almost as in mathematics

Math	Haskell
$\mathrm{square} : \mathbb{N} \to \mathbb{N}$	`square :: Integer -> Integer`
$\mathrm{square}(x) = x \times x$	`square x = x * x`

  - A function is defined by a type definition and a set of rewriting rules
  - The type definition may appear optional, but is not
- Types (including higher-order, polymorphic, and recursively-defined)
  - tuples, lists, trees, shared sub-structures, implicit cycles
- Automatic storage management (garbage collection)
  - Actually we do not care about storage at all; we abstract over the physical machine instead!

Sessions, Scripts, Evaluation

```
< godel:306/slides > ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 66
66
Prelude> 6 * 7
42
Prelude> square 35567
<interactive>:4:1: Not in scope:
‘square’
Prelude> :load example
[1 of 1] Compiling Main
( example.hs, interpreted )
Ok, modules loaded: Main.
*Main> square 35567
1265011489
*Main> square (smaller (5, 78))
25
*Main> square (smaller (5*10, 5+10))
225
*Main>
```

(file example.hs)

```
-- a value (of type Integer):

infty :: Integer
infty = infty + 1

-- a function
-- (from Integer to Integer):

square :: Integer -> Integer
square x = x * x

-- another function:

smaller :: (Integer,Integer)->Integer
smaller (x,y) = if x<=y then x else y
```

- Functions are first order objects

```
twice :: (Integer -> Integer) -> (Integer -> Integer)
twice f = g
    where g x = f (f x)
```

- A program (or script) is a collection of definitions
- Predefined data types in a nutshell:
  - Numerical: *Integer*, *Int*, *Float*, *Double*
  - Logical: *Bool* (values: *True*, *False*)
  - Characters: *Char* (`'a'`, `'b'`, etc.)
  - Composite:
    - Functional: *Integer* $\rightarrow$ *Integer*;
    - Tuples: (*Int*, *Int*, *Float*);
    - Combinations: (*Int*, *Float*) $\rightarrow$ (*Float*, *Bool*), *Int* $\rightarrow$ (*Int* $\rightarrow$ *Int*)
- Sole solution for iterative/repeated computations:

What's left? (cont'd)

- Functions are first order objects

```
twice :: (Integer -> Integer) -> (Integer -> Integer)
twice f = g
    where g x = f (f x)
```

- A program (or script) is a collection of definitions
- Predefined data types in a nutshell:
  - Numerical: *Integer*, *Int*, *Float*, *Double*
  - Logical: *Bool* (values: *True*, *False*)
  - Characters: *Char* (`'a'`, `'b'`, etc.)
  - Composite:
    - Functional: *Integer* $\rightarrow$ *Integer*;
    - Tuples: (*Int*, *Int*, *Float*);
    - Combinations: (*Int*, *Float*) $\rightarrow$ (*Float*, *Bool*), *Int* $\rightarrow$ (*Int* $\rightarrow$ *Int*)
- Sole solution for iterative/repeated computations: recursion

- A script is a collection of definitions of values (including functions)
- Syntactical sugar: definitions by guarded equations:

```
smaller :: (Integer, Integer) -> Integer
smaller (x,y)
   | x <= y   = x
   | x > y    = y
```

- Recursive definitions:

```
fact   :: Integer -> Integer
fact x = if x==0 then 1 else x * fact (x-1)
```

- Syntactical sugar: definitions by pattern matching (aka by cases):

```
fact   :: Integer -> Integer
fact 0 = 1
fact x = x * fact (x-1)
```

LOCAL DEFINITIONS

- Two forms:

```
let v1 = e1                         definition
    v2 = e2                           where v1 = e1
    .                                       v2 = e2
    .                                       .
    .                                       .
    vk = ek                                 .
in expression                               vk = ek
```

- Definitions are qualified by **where** clauses, while expressions are qualified by **let** clauses

- Haskell uses static scoping.

```
cylinderArea :: Float -> Float -> Float
cylinderArea h r = h * 2 * pi * r + 2 * pi * r * r

cylinderArea1 :: Float -> Float -> Float
cylinderArea1 h r = x + 2 * y
    where x = h * circLength r
          y = circArea r
          circArea x = pi * x * x
          circLength x = 2 * pi * x

cylinderArea2 :: Float -> Float -> Float
cylinderArea2 h r = let x = h * circLength r
                        y = circArea r
                        in x + 2 * y
    where circArea x = pi * x * x
          circLength x = 2 * pi * x
```

Types

- Each type has associated operations that are not necessarily meaningful to other types
  - Arithmetic operations ($+$, $-$, $*$, $/$) can be applied to numerical types, but it does not make any sense to apply them on, say, values of type *Bool*
  - It does, however make sense to compare (using $=$ (==), $\neq$ (/=), $\leq$ (<=), $<$, etc.) both numbers and boolean values
- Every well formed expression can be assigned a type (strong typing)
  - The type of an expression can be inferred from the types of the constituents of that expression
  - Those expression whose type cannot be inferred are rejected by the compiler

```
badType x
  | x == 0 = 0
  | x > 0  = 'p'
  | x < 0  = 'n'
```

```
fact :: Integer -> Integer
fact x
  | x < 0  = error "Negative argument."
  | x == 0 = 1
  | x > 0  = x * fact (x-1)
```

What is the type of `error`?

TWO DATA TYPES

- **Booleans**. Values: *True*, *False*
  - operations on *Bool*: logic operators: $\vee$ (||), $\wedge$ (`&&`), $\neg$ (`not`); comparisons: $=$ (==), $\neq$ (/=); relational $<$, $\leq$ (<=), $>$, $\geq$ (>=)
- **Characters**. Values: 256 of them, e.g., 'a', 'b', '\n'
  - Oerations on characters: comparison, relational, plus:

```
ord :: Char -> Int
chr :: Int -> Char
```

```
Prelude> import Data.Char
Prelude Data.Char> ord 'a'
97
Prelude Data.Char> chr 100
'd'
```

```
toLower :: Char -> Char
toLower c | isUpper c = chr (ord c - (ord 'A' - ord 'a'))
          | True      = c
    where isUpper c = 'A' <= c && c <= 'Z'
```

- A list is an ordered set of values.

$[1, 2, 3] :: [Int]$	$[[1, 2], [3]] :: [[Int]]$	$['h', 'i'] :: [Char]$
$[div, rem] :: \;??$	$[1, 'h'] :: \;??$	$[\,] :: \;??$

- Syntactical sugar:

```
Prelude> ['h','i']
"hi"
Prelude> "hi" == ['h','i']
True
Prelude> [['h','i'],"there"]
["hi","there"]
```

Constructing Lists

- Constructors: [] (the empty list) and : (constructs a longer list)

```
Prelude> 1:[2,3,4]
[1,2,3,4]
Prelude> 'h':'i':[]
"hi"
```

  - The operator : (pronounced "cons") is right associative
  - The operator : **does not** concatenate lists together! (++ does this instead)

```
Prelude> [1,2,3] : [4,5]
    No instance for (Num [t0])
      arising from the literal ‘4’
    Possible fix: add an instance declaration for (Num [t0])
    In the expression: 4
    In the second argument of ‘(:)’, namely ‘[4, 5]’
    In the expression: [1, 2, 3] : [4, 5]
Prelude> [1,2,3] : [[4,5]]
[[1,2,3],[4,5]]
Prelude> [1,2,3] ++ [4,5]
[1,2,3,4,5]
Prelude>
```

Operations and Pattern Matching on Lists

- Comparisons ($<$, $\geq$, $==$, etc.), if possible, are made in lexicographical order
- Subscript operator: !! (e.g., $[1, 2, 3]$!! 1 evaluates to 2) – expensive
- Arguably the most common list processing: Given a list, do something with each and every element of that list
  - In fact, such a processing is so common that there exists the predefined *map* that does precisely this:

    ```
    map f [] = []
    map f (x:xs) = f x : map f xs
    ```
  - This is also an example of pattern matching on lists
    - Variant to pattern matching: *head* and *tail* (predefined)

      ```
      head (x:xs) = x
      tail (x:xs) = xs
      ```

      ```
      map f l = if l == []
                then []
                else f (head l) : map f (tail l)
      ```

TUPLES

- While lists are homogenous, tuples group values of (posibly) diferent types

```
divRem :: Integer -> Integer -> (Integer, Integer)
divRem x y = (div x y, rem x y)

divRem1 :: (Integer, Integer) -> (Integer, Integer)
divRem1 (x, 0) = (0, 0)
divRem1 (x, y) = (div x y, rem x y)
```

- The latter variant is also an example of pattern matching on tuples

Operators and functions

- An operator contains symbols from the set !#$%&*+./<=>?@\^|: (− and ~ may also appear, but only as the first character)
- Some operators are predefined (+, −, etc.), but you can define your own as well
- An (infix) operator becomes (prefix) function if surrounded by brackets. A (prefix) function becomes operator if surrounded by backquotes:

```
divRem :: Integer -> Integer -> (Integer, Integer)
x `divRem` y = (div x y, rem x y)
-- precisely equivalent to
-- divRem x y = (div x y, rem x y)

(%%) :: Integer -> Integer -> (Integer, Integer)
(%%) x y = (div x y, rem x y)
-- precisely equivalent to
-- x %% y = (div x y, rem x y)
```

```
Main> 3 %% 2
(1,1)
Main> (%%) 3 2
(1,1)
Main> divRem 3 2
(1,1)
Main> 3 `divRem` 2
(1,1)
Main>
```

  - These are just lexical conventions

IDENTIFIERS

- Identifiers consist in letters, numbers, simple quotes ('), and underscores (_), but they must start with a letter
- For the time being, they must actually start with a lower case letter
  - A Haskell idenitifer starting with a capital letter is considered a type (e.g., *Bool*) or a type constructor (e.g., *True*)—we shall talk at length about those later
  - By convention, types (i.e., class names) in Java start with capital letters, and functions (i.e., method names) start with a lower case letter. What is a convention in Java is the rule in Haskell!
- Some identifiers are language keywords and cannot be redefined (`if`, `then`, `else`, `let`, `where`, etc.).
  - Some identifiers (e.g., `either`) are defined in the standard prelude and possibly cannot be redefined (depending on implementation, messages like "`Definition of variable "either" clashes with import`")

- An inductive proof for a fact $P(n)$, for all $n \geq \alpha$ consists in two steps:
  - Proof of the base case $P(\alpha)$, and
  - The inductive step: assume that $P(n-1)$ is true and show that $P(n)$ is also true

Example

Proof that all the crows have the same colour: For all sets C of crows, $|C| \geq 1$, it holds that all the crows in set C are identical in colour

- Base case, $|C| = 1$: immediate.
- For a set of crows C, $|C| = n$, remove a crow for the set; the remaining (a set of size $n-1$) have the same colour by inductive assumption. Repeat by removing other crow. The desired property follows

Note: According to the Webster's Revised Unabridged Dictionary, a crow is "A bird, usually black, of the genus Corvus [...]."

- The same process is used for building recursive functions: One should provide the base case(s) and the recursive definition(s):
  - To write a function $f :: Integer \rightarrow Integer$, write the base case (definition for $f\ 0$) and the inductive case (use $f\ (n-1)$ to write a definition for $f\ n$)

    Example: computing the factorial
    - Base case: `fact 0 = 1`
    - Induction step: `fact n = n * fact (n-1)`
  - To write a function $f :: [a] \rightarrow \beta$, use induction over the length of the argument; the base case is $f\ []$ and the inductive case is $f\ (x : xs)$ defined using $f\ xs$

    Example: function that concatenates two lists together; we perform induction on the length of the first argument:
    - Base case: `concat [] ys = ys`
    - Induction step: `concat (x:xs) ys = x : concat xs ys`
- Induction is also an extremely useful tool to prove functions that are already written

EXAMPLE: LISTS AS SETS

- Membership ($x \in A$):

```
member x [] = False
member x (y:ys) | x == y = True
                | True   = member x ys
```

- Union ($A \cup B$), intersection ($A \cap B$), difference ($A \setminus B$):

```
union [] t = t
union (x:xs) t | member x t = union xs t
               | True       = x : union xs t
intersection [] t = []
intersection (x:xs) t | member x t = x : intersection xs t
                      | True       = intersection xs t
difference [] t = []
difference (x:xs) t | member x t = difference xs t
                    | True       = x : difference xs t
```

- Constructor: no recursion. `makeSet x = [x]`

Higher order functions

In Haskell, all objects (including functions) are first class citizens. That is,

- all objects can be named,
- all objects can be members of a list/tuple,
- all objects can be passed as arguments to functions,
- all objects can be returned from functions,
- all objects can be the value of some expression

```
twice :: (a -> a) -> (a -> a)
twice f = g
    where g x = f (f x)

compose f g = h
    where h x = f (g x)
```

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)


compose f g x  = f (g x)
  -- or --
compose f g = f.g
```

curried form:	uncurried form:
`compose :: (a->b) -> (c->a) -> c->b`	`compose :: (a->b, c->a) -> c->b`
`compose f g = f.g`	`compose (f,g) = f.g`

- In Haskell, any function takes one argument and returns one value.
  - What if we need more than one argument?

    Uncurried We either present the arguments packed in a tuple, or

    Curried We use partial application: we build a function that takes one argument and that return a function which in turn takes one argument and returns another function which in turn. . .

curried:

```
add :: (Num a) => a -> a -> a
add x y = x + y
-- equivalent to the explicit version
-- add x = g
--     where g y = x + y

incr :: (Num a) => a -> a
incr = add 1
```

uncurried:

```
add :: (Num a) => (a, a) -> a
add (x,y) = x + y

incr :: (Num a) => a -> a
incr x = add (1,x)
```

To curry or not to curry (cont'd)

- Curying is made possible by lexical closures →all the values existing when a particular function is defined will exist when the function is run
- What if we have a curried function and we want an uncurried one (or the other way around)?
  - The following two functions are predefined:

```
curry f = g
    where g a b = f (a,b) -- lexical closure:
                          -- f available inside g
uncurry g = f
    where f (a,b) = g a b
```

To curry or not to curry (cont'd)

- Curying is made possible by lexical closures →all the values existing when a particular function is defined will exist when the function is run
- What if we have a curried function and we want an uncurried one (or the other way around)?
  - The following two functions are predefined:

    ```
    curry f = g
        where g a b = f (a,b) -- lexical closure:
                              -- f available inside g
    uncurry g = f
        where f (a,b) = g a b
    ```

  - Alternatively,

    ```
    curry f a b = f (a,b)
    uncurry g (a,b) = g a b
    ```

  - Note that the two functions are curried themselves. . .

GLOBAL VARIABLES

- Given a nonnegative number $x :: \mathit{Float}$, write a function *mySqrt* that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$
  - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$

Global variables

- Given a nonnegative number $x :: \mathit{Float}$, write a function *mySqrt* that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$
  - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$
  - So we have:

```
mySqrt   :: Float -> Float
mySqrt x = sqrt' x
  where sqrt' y = if good y then y else sqrt' (improve y)
        good y = abs (y*y - x) < eps
        improve y = (y + x/y)/2
        eps = 0.0001
```

  - x is very similar to a global variable in procedural programming

- Given a nonnegative number x :: *Float*, write a function *mySqrt* that computes an approximation of $\sqrt{x}$ with precision $\epsilon = 0.0001$
  - Newton says that, if y_n is an approximation of $\sqrt{x}$, then a better approximation is $y_{n+1} = (y_n + x/y_n)/2$
  - So we have:

```
mySqrt   :: Float -> Float
mySqrt x = sqrt' x
  where sqrt' y = if good y then y else sqrt' (improve y)
        good y = abs (y*y - x) < eps
        improve y = (y + x/y)/2
        eps = 0.0001
```

  - x is very similar to a global variable in procedural programming
  - Even closer to procedural programming:

```
mySqrt   :: Float -> Float
mySqrt x = until done improve x
    where done y = abs (y*y - x)
                   < eps
          improve y = (y + x/y)/2
          eps = 0.0001
```

```
until :: (a -> Bool) ->
         (a -> a) -> a -> a
until p f x
    | p x = x
    | True = until p f (f x)
```

ACCUMULATING RESULTS

1.
```
mystery x = aux x []
    where aux [] ret = ret
          aux (x:xs) ret = aux xs (x:ret)
```

ACCUMULATING RESULTS

1. ```
   mystery x = aux x []
       where aux [] ret = ret
             aux (x:xs) ret = aux xs (x:ret)
   ```
2. ```
   reverse [] = []
   reverse (x:xs) = reverse xs ++ [x]
   ```

- What is the difference between these two implementations?

ACCUMULATING RESULTS

1.
```
mystery x = aux x []
    where aux [] ret = ret
          aux (x:xs) ret = aux xs (x:ret)
```
2.
```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

- What is the difference between these two implementations?
  - An accumulating argument is used for efficiency purposes
  - It basically transforms a general recursion into a tail recursion

Maps

- *map* applies a function to each element in a list

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

- For example:

```
upto m n = if m > n then [] else m: upto (m+1) n
square x = x * x

Prelude> map ((<) 3) [1,2,3,4]
[True,True,False,False]
Prelude> sum (map square (upto 1 10))
385
Prelude>
```

MAPS (CONT'D)

- Intermission: *zip* and *unzip*

```
Prelude> zip [0,1,2,3,4] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> zip [0,1,2,3,4,5,6] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> unzip [(0,'h'),(1,'e'),(2,'l'),(4,'o')]
([0,1,2,4],"helo")
Prelude>
```

- A more complex (and useful) example of *map*:

```
mystery :: (Ord a) => [a] -> Bool
mystery xs =  and (map (uncurry (<=)) (zip xs (tail xs)))
```

- Intermission: *zip* and *unzip*

```
Prelude> zip [0,1,2,3,4] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> zip [0,1,2,3,4,5,6] "hello"
[(0,'h'),(1,'e'),(2,'l'),(3,'l'),(4,'o')]
Prelude> unzip [(0,'h'),(1,'e'),(2,'l'),(4,'o')]
([0,1,2,4],"helo")
Prelude>
```

- A more complex (and useful) example of *map*:

```
mystery :: (Ord a) => [a] -> Bool
mystery xs =  and (map (uncurry (<=)) (zip xs (tail xs)))
```

- This finds whether the argument list is in nondecreasing order

Filters

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

- Example:

```
mystery :: [(String,Int)] -> [String]
mystery xs = map fst (filter ( ((<=) 80) . snd ) xs)
```

Filters

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

- Example:

```
mystery :: [(String,Int)] -> [String]
mystery xs = map fst (filter ( ((<=) 80) . snd ) xs)
```

```
Prelude> mystery [("a",70),("b",80),("c",91),("d",79)]
["b","c"]
```

  - Suppose that the final grades for some course are kept as a list of pairs (student name, grade). This then finds all the students that got an A

FOLDS

$$[l_1, l_2, \ldots, l_n] \xrightarrow{foldr} l_1 \bullet (l_2 \bullet (l_3 \bullet (\cdots \bullet (l_n \bullet \mathbf{id}) \cdots)))$$

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op id [] = id
foldr op id (x:xs) = x `op` (foldr op id xs)
```

$$[l_1, l_2, \ldots, l_n] \xrightarrow{foldl} (\cdots(((\mathbf{id} \bullet l_1) \bullet l_2) \bullet l_3) \bullet \cdots \bullet l_n)$$

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op id [] = id
foldl op id (x:xs) = foldl op (x `op` id) xs
```

- Almost all the interesting functions on lists are or can be implemented using *foldr* or *foldl*:

```
and = foldr (&&) True              concat = foldr (++) []
sum = foldr (+) 0                  length = foldr oneplus 0
map f = foldr ((:).f) []               where oneplus x n = 1 + n
```

List comprehension

- Examples:

```
triples :: Int -> [(Int,Int,Int)]
triples n = [(x,y,z) | x <- [1..n],y <- [1..n],z <- [1..n]]
      -- or [(x,y,z) | x <- [1..n],y <- [x..n],z <- [z..n]]
pyth :: (Int,Int,Int) -> Bool
pyth (x,y,z) = x*x + y*y == z*z
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z) | (x,y,z) <- triples n, pyth (x,y,z)]
```

- General form:

$$[exp | gen_1, gen_2, \ldots, gen_n, guard_1, guard_2, \ldots guard_p]$$

- Quicksort:

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

Polymorphic types

- Some functions have a type definition involving only type names:

```
and :: [Bool] -> Bool
and = foldr (&&) True
```

  - These functions are monomorphic
- It is however useful sometimes to write functions that can work on data of more than one type. These are polymorphic functions

```
length :: [a] -> Int  -- For any type a: length :: [a]->Int
map :: (a -> b) -> [a] -> [b]
```

- Restricted polymorphism: What is the most general type of a function that sorts a list of values, and why?

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

Polymorphic types

- Some functions have a type definition involving only type names:

```
and :: [Bool] -> Bool
and = foldr (&&) True
```

  - These functions are monomorphic

- It is however useful sometimes to write functions that can work on data of more than one type. These are polymorphic functions

```
length :: [a] -> Int  -- For any type a: length :: [a]->Int
map :: (a -> b) -> [a] -> [b]
```

- Restricted polymorphism: What is the most general type of a function that sorts a list of values, and why?

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

  - `qsort ::  (Ord a) => [a] -> [a]`
    ($[a] \rightarrow [a]$ for any type a such that an order is defined over a)

Type synonyms

- A function that adds two polynomials with floating point coefficients:
  `polyAdd :: [Float] -> [Float] -> [Float]`
- `polyAdd :: Poly -> Poly -> Poly` would have been nicer though…
  - This can be done by defining "`Poly`" as a type synonym for `[Float]`:
    `type Poly = [Float]`
  - Type synonyms can also be parameterized:

```
type Stack a = [a]

newstack :: Stack a
newstack = []

push :: a -> Stack a -> Stack a
push x xs = x:xs

aCharStack :: Stack Char
aCharStack = push 'a' (push 'b' newstack)
```

```
Main> aCharStack
"ab"
Main> push 'x' aCharStack
"xab"
Main> :t aCharStack
aCharStack :: Stack Char
```

ALGEBRAIC TYPES

- Remember when we defined functions using induction (aka recursion)?
- Types can be defined in a similar manner (the general form of mathematical induction is called structural induction): Take for example natural numbers:

```
data Nat = Zero | Succ Nat
    deriving Show

-- Operations:
addNat,mulNat :: Nat -> Nat -> Nat
addNat m Zero = m
addNat m (Succ n) = Succ (addNat m n)
mulNat m Zero = Zero
mulNat m (Succ n) = addnat (mulNat m n) m
```

- Again, type definitions can be parameterized:

```
data List a  = Nil | Cons a (List a)
--  data [a] = []  | a : [a]
data BinTree a = Null | Node a (BinTree a) (BinTree a)
```

TYPE CLASSES

- Each type may belong to a type class that define general operations. This also offers a mechanism for overloading
  - Type classes in Haskell are similar with abstract classes in Java

```
data Nat = Zero | Succ Nat
           deriving (Eq,Show)

instance Ord Nat where
    Zero <= x = True
    x <= Zero = False
    (Succ x) <= (Succ y) = x <= y

one,two,three :: Nat
one = Succ Zero
two = Succ one
three = Succ two
```

```
Main> one
Succ Zero
Main> two
Succ (Succ Zero)
Main> three
Succ (Succ (Succ Zero))
Main> one > two
False
Main> one > Zero
True
Main> two < three
True
Main>
```

...The class *Ord* is defined in the standard prelude as follows:

```
class (Eq a) => Ord a where
    compare                 :: a -> a -> Ordering
    (<), (<=), (>=), (>)    :: a -> a -> Bool
    max, min                :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x >= y      = x
              | otherwise   = y
    min x y   | x <= y      = x
              | otherwise   = y
```

Type classes (cont'd)

```haskell
data Nat = Zero | Succ Nat
           deriving (Eq,Ord,Show)

instance Num Nat where
    m + Zero = m
    m + (Succ n) = Succ (m + n)
    m * Zero = Zero
    m * (Succ n) = (m * n) + m

one,two,three :: Nat
one = Succ Zero
two = Succ one
three = Succ two
```

```
Main> one + two
Succ (Succ (Succ Zero))
Main> two * three
Succ (Succ (Succ (Succ
 (Succ (Succ Zero)))))
Main> one + two == three
True
Main> two * three == one
False
Main> three - two

ERROR - Control stack
        overflow
```

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)   :: a -> a -> a
    negate          :: a -> a
    abs, signum     :: a -> a
    fromInteger     :: Integer -> a
    fromInt         :: Int -> a

    -- Minimal complete definition: All, except negate or (-)
    x - y            = x + negate y
    fromInt          = fromIntegral
    negate x         = 0 - x
```

SUBTRACTION

```
instance Num Nat where
    m + Zero = m
    m + (Succ n) = Succ (m + n)
    m * Zero = Zero
    m * (Succ n) = (m * n) + m

    m - Zero = m
    (Succ m) - (Succ n) = m - n
```

```
Nat> one - one
Zero
Nat> two - one
Succ Zero
Nat> two - three

Program error: pattern match failure:
               instNum_v1563_v1577 Nat_Zero one
```

```
class Enum a where
    succ, pred           :: a -> a
    toEnum               :: Int -> a
    fromEnum             :: a -> Int
    enumFrom             :: a -> [a]                -- [n..]
    enumFromThen         :: a -> a -> [a]           -- [n,m..]
    enumFromTo           :: a -> a -> [a]           -- [n..m]
    enumFromThenTo       :: a -> a -> a -> [a]      -- [n,n'..m]

    -- Minimal complete definition: toEnum, fromEnum
    succ                  = toEnum . (1+)       . fromEnum
    pred                  = toEnum . subtract 1 . fromEnum
    enumFrom x            = map toEnum [ fromEnum x ..]
    enumFromTo x y        = map toEnum [ fromEnum x .. fromEnum y ]
    enumFromThen x y      = map toEnum [ fromEnum x, fromEnum y ..]
    enumFromThenTo x y z  = map toEnum [ fromEnum x, fromEnum y ..
                                                     fromEnum z ]
```

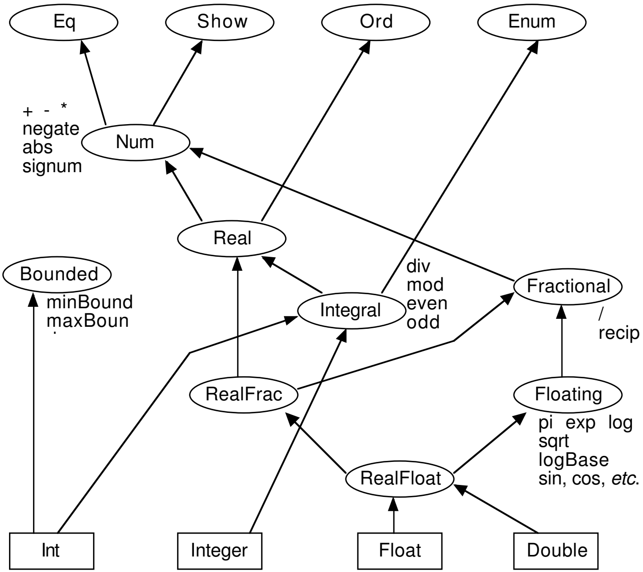
Other interesting type classes (cont'd)

```
class Show a where
    show      :: a -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS

    -- Minimal complete definition: show or showsPrec
    show x          = showsPrec 0 x ""
    showsPrec _ x s = show x ++ s
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
                      where showl []     = showChar ']'
                            showl (x:xs) = showChar ',' .
                                           shows x . showl xs

instance Show Nat where
    show Zero = "0"
    show (Succ n) = "1 + " ++ show n
```

EXAMPLE OF TYPE CLASSES



- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments:
  - Inspect the argument patterns if any:
  - parse definitions top to bottom, right to left:
    - "q : []" $\Longrightarrow$
    - "(f q x)" $\Longrightarrow$
    - "scanl f (f q x) xs" $\Longrightarrow$
    - "q : scanl f (f q x) xs" $\Longrightarrow$
  - So the overall type is

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any:
  - parse definitions top to bottom, right to left:
    - "q : []" $\Longrightarrow$
    - "(f q x)" $\Longrightarrow$
    - "scanl f (f q x) xs" $\Longrightarrow$
    - "q : scanl f (f q x) xs" $\Longrightarrow$
  - So the overall type is

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    - “q : []” $\Longrightarrow$
    - “(f q x)” $\Longrightarrow$
    - “scanl f (f q x) xs” $\Longrightarrow$
    - “q : scanl f (f q x) xs” $\Longrightarrow$
  - So the overall type is

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    - "q : []" $\Longrightarrow$ no extra information
    - "(f q x)" $\Longrightarrow$
    - "scanl f (f q x) xs" $\Longrightarrow$
    - "q : scanl f (f q x) xs" $\Longrightarrow$
  - So the overall type is

Type inference

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    - "q : []" $\Longrightarrow$ no extra information
    - "(f q x)" $\Longrightarrow$ f is a function with 2 arguments: $(\beta \to \gamma \to \eta) \to \beta \to [\gamma] \to \delta$
    - "scanl f (f q x) xs" $\Longrightarrow$
    - "q : scanl f (f q x) xs" $\Longrightarrow$
  - So the overall type is

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    - "q : []" $\Longrightarrow$ no extra information
    - "(f q x)" $\Longrightarrow$ f is a function with 2 arguments: $(\beta \to \gamma \to \eta) \to \beta \to [\gamma] \to \delta$
    - "scanl f (f q x) xs" $\Longrightarrow$ $\beta = \delta$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to \delta$
    - "q : scanl f (f q x) xs" $\Longrightarrow$
  - So the overall type is

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    - "q : []" $\Longrightarrow$ no extra information
    - "(f q x)" $\Longrightarrow$ f is a function with 2 arguments: $(\beta \to \gamma \to \eta) \to \beta \to [\gamma] \to \delta$
    - "scanl f (f q x) xs" $\Longrightarrow$ $\beta = \delta$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to \delta$
    - "q : scanl f (f q x) xs" $\Longrightarrow$ $\delta = [\beta]$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to [\beta]$
  - So the overall type is

Type inference

- Different from type checking; in fact precedes type checking
  - allows the compilers to find the types automatically
- Example:

```
scanl f q [] = q : []
scanl f q (x : xs) = q : scanl f (f q x) xs
```

  - First count the arguments: $\alpha \to \beta \to \gamma \to \delta$
  - Inspect the argument patterns if any: $\alpha \to \beta \to [\gamma] \to \delta$
  - parse definitions top to bottom, right to left:
    - "q : []" $\implies$ no extra information
    - "(f q x)" $\implies$ f is a function with 2 arguments: $(\beta \to \gamma \to \eta) \to \beta \to [\gamma] \to \delta$
    - "scanl f (f q x) xs" $\implies$ $\beta = \delta$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to \delta$
    - "q : scanl f (f q x) xs" $\implies$ $\delta = [\beta]$ i.e. $(\beta \to \gamma \to \beta) \to \beta \to [\gamma] \to [\beta]$
  - So the overall type is

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

- Recall that a Haskell function accepts one argument and returns one result

peanuts	$\rightarrow$	chocolate-covered peanuts
raisins	$\rightarrow$	chocolate-covered raisins
ants	$\rightarrow$	chocolate-covered ants

- Using the lambda calculus, a general “chocolate-covering” function (or rather λ-expression) is described as follows:

$$\lambda x.\text{chocolate-covered } x$$

- Then we can get chocolate-covered ants by applying this function:

$$(\lambda x.\text{chocolate-covered } x) \text{ ants} \quad \rightarrow \quad \text{chocolate-covered ants}$$

The lambda notation (cont'd)

- A general covering function:

$$\lambda y.\lambda x.y\text{-covered } x$$

The result of the application of such a function is itself a function:

$$\begin{array}{lcl} (\lambda y.\lambda x.y\text{-covered } x)\text{ caramel} & \rightarrow & \lambda x.\text{caramel-covered } x \\ ((\lambda y.\lambda x.y\text{-covered } x)\text{ caramel})\text{ ants} & \rightarrow & (\lambda x.\text{caramel-covered } x)\text{ ants} \\ & \rightarrow & \text{caramel-covered ants} \end{array}$$

- Functions can also be parameters to other functions:

$$\lambda f.(f)\text{ ants}$$

$$\begin{array}{ll} ((\lambda f.(f)\text{ ants})\ \lambda x.\text{chocolate-covered})\ x & \\ \rightarrow & (\lambda x.\text{chocolate-covered } x)\text{ ants} \\ \rightarrow & \text{chocolate-covered ants} \end{array}$$

- The lambda calculus is a formal system designed to investigate function definition, function application and recursion
  - Introduced by Alonzo Church and Stephen Kleene in the 1930s
- We start with a countable set of identifiers, e.g., $\{a, b, c, \ldots, x, y, z, x1, x2, \ldots\}$ and we build expressions using the following rules:

$$
\begin{array}{lcll}
\text{LEXPRESSION} & \rightarrow & \text{IDENTIFIER} & \\
\text{LEXPRESSION} & \rightarrow & \lambda\text{IDENTIFIER.LEXPRESSION} & \text{(abstraction)} \\
\text{LEXPRESSION} & \rightarrow & (\text{LEXPRESSION})\text{LEXPRESSION} & \text{(combination)} \\
\text{LEXPRESSION} & \rightarrow & (\text{LEXPRESSION}) &
\end{array}
$$

  - In an expression $\lambda x.E$, x is called a bound variable; a variable that is not bound is a free variable
- Syntactical sugar: Normally, no literal constants exist in lambda calculus. We use, however, literals for clarity
  - Further sugar: HASKELL

- In lambda calculus, an expression $(\lambda x.E)F$ can be reduced to $E[F/x]$
  - $E[F/x]$ stands for the expression E, where F is substituted for all the bound occurrences of x
- In fact, there are three reduction rules:
  - α: $\lambda x.E$ reduces to $\lambda y.E[y/x]$ if y is not free in E (change of variable)
  - β: $(\lambda x.E)F$ reduces to $E[F/x]$ (functional application)
  - γ: $\lambda x.(Fx)$ reduces to F if x is not free in F (extensionality)
- The purpose in life of a Haskell program, given some expression, is to repeatedly apply these reduction rules in order to bring that expression to its "irreducible" form or normal form

- In a Haskell program, we write functions and then apply them
  - Haskell programs are nothing more than collections of λ-expressions, with added sugar for convenience (and diabetes)
- We write a Haskell program by writing λ-expressions and giving names to them:

```
succ x = x + 1

length = foldr onepl 0
    where onepl x n = 1+n

Main> succ 10
11
```

```
succ = \ x -> x + 1

length = foldr (\ x -> \ n -> 1+n) 0
-- shorthand:  (\ x n -> 1+n)

Main> (\ x -> x + 1) 10
11
```

- Another example: `map (\ x -> x+1) [1,2,3]` maps (i.e., applies) the λ-expression $\lambda x.x + 1$ to all the elements of the list, thus producing `[2,3,4]`
- In general, for some expression E, $\lambda x.E$ (in Haskell-speak: `\ x -> E`) denotes the function that maps x to the (value of) E

- More than one order of reduction is usually possible in lambda calculus (and thus in Haskell):

```
square   :: Integer -> Integer
square x = x * x

smaller  :: (Integer, Integer) -> Integer
smaller (x,y) = if x<=y then x else y
```

$\mathit{square}\,(\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. *smaller*) $\mathit{square}\,5$ $\Rightarrow$ (def. *square*) 5×5 $\Rightarrow$ (def. $\times$) 25	$\mathit{square}\,(\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. *square*) $(\mathit{smaller}\,(5,78)) \times (\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. *smaller*) $5 \times (\mathit{smaller}\,(5,78))$ $\Rightarrow$ (def. *smaller*) 5×5 $\Rightarrow$ (def. $\times$) 25

- Sometimes it even matters:

```
three   :: Integer -> Integer
three x = 3

infty :: Integer
infty = infty + 1
```

$\textit{three infty}$ $\Rightarrow$ (def. $\textit{infty}$) $\textit{three}\,(\textit{infty} + 1)$ $\Rightarrow$ (def. $\textit{infty}$) $\textit{three}\,((\textit{infty} + 1) + 1)$ $\Rightarrow$ (def. $\textit{infty}$) $\textit{three}\,(((\textit{infty} + 1) + 1) + 1)$ $\vdots$	$\textit{three infty}$ $\Rightarrow$ (def. $\textit{three}$) 3

LAZY HASKELL

- Haskell uses the second variant, called lazy evaluation (normal order, outermost reduction), as opposed to eager evaluation (applicative order, innermost reduction):

```
Main> three infty
3
```

- Why is good to be lazy:
  - Doesn't hurt: If an irreducible form can be obtained by both kinds of reduction, then the results are guaranteed to be the same
  - More robust: If an irreducible form can be obtained, then lazy evaluation is guaranteed to obtain it
  - Even useful: It is sometimes useful (and, given the lazy evaluation, possible) to work with infinite objects

- `[1 .. 100]` produces the list of numbers between 1 and 100, but what is produced by `[1 .. ]`?

```
Prelude> [1 ..] !! 10
11
Prelude> [1 .. ] !! 12345
12346
Prelude> zip ['a' .. 'g'] [1 ..]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6),('g',7)]
```

  - A stream of prime numbers:

```
primes :: [Integer]
primes = sieve [2 .. ]
  where sieve (x:xs) = x : [n | n <- sieve xs, mod n x /= 0]
      -- alternate:
      -- sieve (x:xs) = x : sieve (filter (\ n -> mod n x /= 0) xs)

Main> take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

- Streams can also be used to improve efficiency (dramatically!)
- Take the Fibonacci numbers:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

  - Complexity?

- Now take them again, using a memo stream:

```
fastfib :: Integer -> Integer
fastfib n = fibList %% n
    where fibList = 1 : 1 : zipWith (+) fibList (tail fibList)
          (x:xs) %% 0 = x
          (x:xs) %% n = xs %% (n - 1)
```

  - Complexity?

- Streams can also be used to improve efficiency (dramatically!)
- Take the Fibonacci numbers:

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

  - Complexity? $O(2^n)$
- Now take them again, using a memo stream:

```
fastfib :: Integer -> Integer
fastfib n = fibList %% n
    where fibList = 1 : 1 : zipWith (+) fibList (tail fibList)
          (x:xs) %% 0 = x
          (x:xs) %% n = xs %% (n - 1)
```

  - Complexity? $O(n)$
- Typical application: dynamic programming

	Functional programming	Ordinary programming
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Write functions that define the problem	Figure out solution
4.	Coffee break	Program solution
5.	Encode problem instance as data	Encode problem instance as data
6.	Apply function to data	Apply program to data
7.	Mathematical analysis	Debug procedural errors