Functional Programming in Haskell

Part I : Basics

Madhavan Mukund

Chennai Mathematical Institute

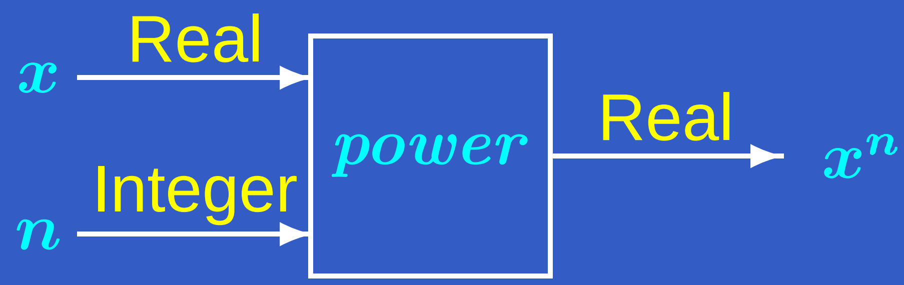
92 G N Chetty Rd, Chennai 600 017, India

`madhavan@cmi.ac.in`

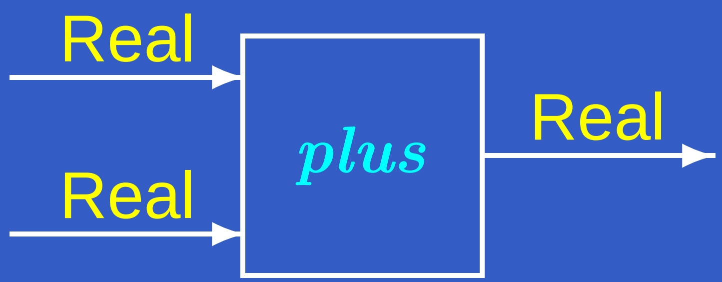
`http://www.cmi.ac.in/~madhavan`

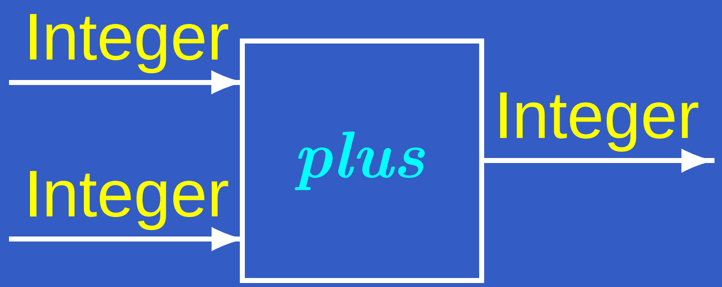
Functions

- Transform inputs to output
- Operate on specific types



- These functions are different





Functional programming

- Program $\Leftrightarrow$ set of function definitions
- Function definition $\Leftrightarrow$ how to “calculate” the value
- Declarative programming
  - Provably correct programs

    Functional program closely follows mathematical definition
  - Rapid prototyping

    Easy to go from specification (what we require) to implementation (working program)

Functional programming in Haskell

- Built-in types `Int`, `Float`, `Bool`, `...`
  with basic operations `+`, `-`, `*`, `/`, ||, &&, `...`

Functional programming in Haskell

- Built-in types `Int`, `Float`, `Bool`, ...
  with basic operations `+`, `-`, `*`, `/`, ||, &&, ...
- Defining a new function (and its type)

```
power :: Float -> Int -> Float
power x n = if (n == 0) then 1.0
            else x * (power x (n-1))
```

Functional programming in Haskell

- Built-in types `Int`, `Float`, `Bool`, . . .
  with basic operations `+`, `-`, `*`, `/`, ||, &&, . . .
- Defining a new function (and its type)

```
power :: Float -> Int -> Float
power x n = if (n == 0) then 1.0
            else x * (power x (n-1))
```

- Multiple arguments are consumed "one at a time"

  Not . . .

```
power :: Float × Int -> Float
power (x,n) = ...
```

Functional programming in Haskell

- Built-in types `Int`, `Float`, `Bool`, `...`
  with basic operations `+`, `-`, `*`, `/`, ||, &&, `...`
- Defining a new function (and its type)

```
power :: Float -> Int -> Float
power x n = if (n == 0) then 1.0
            else x * (power x (n-1))
```

- Multiple arguments are consumed “one at a time”

  Not . . .

```
power :: Float × Int -> Float
power (x,n) = ...
```

- Need not be a “total” function

  What is `power 2.0 -1`?

Ways of defining functions

- Multiple definitions, read top to bottom

Ways of defining functions

- Multiple definitions, read top to bottom
- Definition by cases

```
power ::  Float -> Int -> Float
power x 0  =  1.0
power x n  =  x * (power x (n-1))
```

Ways of defining functions

- Multiple definitions, read top to bottom
- Definition by cases

```
power ::  Float -> Int -> Float
power x 0  =  1.0
power x n  =  x * (power x (n-1))
```

- Implicit “pattern matching” of arguments

```
xor ::  Bool -> Bool -> Bool
xor True True     =  False
xor False False   =  False
xor x y           =  True
```

Ways of defining functions . . .

- Multiple options with conditional guards

```
max ::  Int -> Int -> Int

max i j | (i >= j) = i
        | (i < j)  = j
```

Ways of defining functions . . .

- Multiple options with conditional guards

```
max :: Int -> Int -> Int

max i j | (i >= j) = i
        | (i < j)  = j
```

- Default conditional value — `otherwise`

```
max3 :: Int -> Int -> Int -> Int

max3 i j k | (i >= j) && (i >= k) = i
           | (j >= k)             = j
           | otherwise            = k
```

Ways of defining functions . . .

- Multiple options with conditional guards

```
max :: Int -> Int -> Int

max i j | (i >= j) = i
        | (i < j)  = j
```

- Default conditional value — `otherwise`

```
max3 :: Int -> Int -> Int -> Int

max3 i j k | (i >= j) && (i >= k) = i
           | (j >= k)             = j
           | otherwise            = k
```

- Note: Conditional guards are evaluated top to bottom!

Pairs, triples, . . .

- Can form n-tuples of types

Pairs, triples, . . .

- Can form n-tuples of types
- `(x,y,z) ::  (Float,Float,Float)` represents a point in 3D

Pairs, triples, . . .

- Can form n-tuples of types
- `(x,y,z) :: (Float,Float,Float)` represents a point in 3D
- Can define a function

```
distance3D ::
  (Float,Float,Float) ->
  (Float,Float,Float) -> Float
```

Pairs, triples, ...

- Can form n-tuples of types
- `(x,y,z) :: (Float,Float,Float)` represents a point in 3D
- Can define a function

```
distance3D ::
  (Float,Float,Float) ->
  (Float,Float,Float) -> Float
```

- Functions can return n-tuples

```
maxAndMinOf3 ::
  Int -> Int -> Int -> (Int,Int)
```

Local definitions using `where`

Example: Compute distance between two points in 2D

```
distance ::
  (Float,Float)->(Float,Float)->Float

distance (x1,y1) (x2,y2) =
  sqrt((sqr xdistance) + (sqr ydistance))
  where

    xdistance = x2 - x1
    ydistance = y2 - y1


    sqr ::  Float -> Float
    sqr z      = z*z
```

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

$$\texttt{3 + 5} \leadsto \texttt{8} \qquad \texttt{True || False} \leadsto \texttt{True}$$

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

  `power 3.0 2`

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

  `power 3.0 2`

  $\leadsto$ `3.0 * (power 3.0 (2-1))`

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

  `power 3.0 2`

  $\leadsto$ `3.0 * (power 3.0 (2-1))`

  $\leadsto$ `3.0 * (power 3.0 1)`

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

```
power 3.0 2
 ⤳ 3.0 * (power 3.0 (2-1))
 ⤳ 3.0 * (power 3.0 1)
 ⤳ 3.0 * 3.0 * (power 3.0 (1-1))
```

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

```
power 3.0 2
```

$\leadsto$ `3.0 * (power 3.0 (2-1))`

$\leadsto$ `3.0 * (power 3.0 1)`

$\leadsto$ `3.0 * 3.0 * (power 3.0 (1-1))`

$\leadsto$ `3.0 * 3.0 * (power 3.0 0)`

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

  `power 3.0 2`

  $\leadsto$ `3.0 * (power 3.0 (2-1))`

  $\leadsto$ `3.0 * (power 3.0 1)`

  $\leadsto$ `3.0 * 3.0 * (power 3.0 (1-1))`

  $\leadsto$ `3.0 * 3.0 * (power 3.0 0)`

  $\leadsto$ `3.0 * 3.0 * 1.0`

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

$3 + 5 \leadsto 8$ True || False $\leadsto$ True

- Simplifications based on user defined functions

```
power 3.0 2
~> 3.0 * (power 3.0 (2-1))
~> 3.0 * (power 3.0 1)
~> 3.0 * 3.0 * (power 3.0 (1-1))
~> 3.0 * 3.0 * (power 3.0 0)
~> 3.0 * 3.0 * 1.0
~> 9.0 * 1.0
```

Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

  `3 + 5` $\leadsto$ `8`   `True || False` $\leadsto$ `True`

- Simplifications based on user defined functions

```
power 3.0 2
 ⇝ 3.0 * (power 3.0 (2-1))
 ⇝ 3.0 * (power 3.0 1)
 ⇝ 3.0 * 3.0 * (power 3.0 (1-1))
 ⇝ 3.0 * 3.0 * (power 3.0 0)
 ⇝ 3.0 * 3.0 * 1.0
 ⇝ 9.0 * 1.0  ⇝ 9.0
```

Functions that manipulate functions

- A function with input type a and output type b has type `a -> b`

Functions that manipulate functions

- A function with input type a and output type b has type `a -> b`
- Recall useful convention that all functions read one argument at a time!

Functions that manipulate functions

- A function with input type a and output type b has type `a -> b`
- Recall useful convention that all functions read one argument at a time!
- A function can take another function as argument

Functions that manipulate functions

- A function with input type a and output type b has type `a -> b`
- Recall useful convention that all functions read one argument at a time!
- A function can take another function as argument
- ```
  apply ::  (Int -> Int) -> Int -> Int
  apply f n = f n
  ```
  ```

Functions that manipulate functions

- A function with input type a and output type b has type `a -> b`
- Recall useful convention that all functions read one argument at a time!
- A function can take another function as argument
-
```
apply ::  (Int -> Int) -> Int -> Int
apply f n = f n
```
-
```
twice ::  (Int -> Int) -> Int -> Int
twice f n = f (f n)
```

Functions that manipulate functions

- A function with input type a and output type b has type `a -> b`
- Recall useful convention that all functions read one argument at a time!
- A function can take another function as argument
- ```
  apply ::  (Int -> Int) -> Int -> Int
  apply f n = f n
  ```
- ```
  twice ::  (Int -> Int) -> Int -> Int
  twice f n = f (f n)
  ```
- `twice sqr 7` $\leadsto$ `sqr (sqr 7)` $\leadsto$ `sqr (7*7)` $\cdots$ $\leadsto$ `49*49` $\leadsto$ `2401`

Running Haskell programs

- hugs — A Haskell interpreter
  Available for Linux, Windows, . . .
- ghc — the Glasgow Haskell Compiler
- Look at `http://www.haskell.org`

Collections

- Basic collective type is a list

Collections

- Basic collective type is a list
- All items of a list must be of the same type

  `[Int]` — list of `Int`,

  `[(Float,Float)]` — list of pairs of `Float`

Collections

- Basic collective type is a list
- All items of a list must be of the same type

  `[Int]` — list of `Int`,

  `[(Float,Float)]` — list of pairs of `Float`
- Lists are written as follows:

  `[2,3,1,7]`

  `[(3.0,7.5),(7.6,9.2),(3.3,7.868)]`

Collections

- Basic collective type is a list
- All items of a list must be of the same type
  `[Int]` — list of `Int`,
  `[(Float,Float)]` — list of pairs of `Float`
- Lists are written as follows:
  `[2,3,1,7]`
  `[(3.0,7.5),(7.6,9.2),(3.3,7.868)]`
- Empty list is denoted `[]` (for all types)

Basic operations on lists

- `++` concatenates lists

  `[1,3] ++ [5,7] = [1,3,5,7]`

Basic operations on lists

- `++` concatenates lists

  `[1,3] ++ [5,7] = [1,3,5,7]`

- Unique way of decomposing a nonempty list
  - head : first element of the list
  - tail : the rest (may be empty!) —

```
head [1,3,5,7] = 1
tail [1,3,5,7] = [3,5,7]
```

Basic operations on lists

- ++ concatenates lists

  ```
  [1,3] ++ [5,7] = [1,3,5,7]
  ```

- Unique way of decomposing a nonempty list
  - head : first element of the list
  - tail : the rest (may be empty!) —

    ```
    head [1,3,5,7] = 1
    tail [1,3,5,7] = [3,5,7]
    ```

- Note the types: head is an element, tail is a list

Basic operations on lists

- `++` concatenates lists

  `[1,3] ++ [5,7] = [1,3,5,7]`

- Unique way of decomposing a nonempty list
  - head : first element of the list
  - tail : the rest (may be empty!) —

    ```
    head [1,3,5,7] = 1
    tail [1,3,5,7] = [3,5,7]
    ```

- Note the types: head is an element, tail is a list
- Write `x:l` to denote the list with head `x`, tail `l`

Functions on lists

- Define functions by induction on list structure

Functions on lists

- Define functions by induction on list structure
  - Base case
    Value of `f` on `[]`
  - Step
    Extend value of `f` on `l` to `f` on `x:l`

Functions on lists

- Define functions by induction on list structure
  - Base case
    Value of `f` on `[]`
  - Step
    Extend value of `f` on `l` to `f` on `x:l`
- ```
  length ::  [Int] -> Int
  length []      =  0
  length (x:l)   =  1 + length l
  ```
  ```

Functions on lists

- Define functions by induction on list structure
  - Base case
    Value of `f` on `[]`
  - Step
    Extend value of `f` on `l` to `f` on `x:l`

- ```
  length ::  [Int] -> Int
  length []      = 0
  length (x:l)   = 1 + length l
  ```

- ```
  reverse ::  [Int] -> [Int]
  reverse []      = []
  reverse (x:l)   = (reverse l) ++ [x]
  ```

Some builtin list functions

- `length l, reverse l, sum l, ...`

Some builtin list functions

- `length l`, `reverse l`, `sum l`, ...
- `head l`, `tail l`

Some builtin list functions

- `length l`, `reverse l`, `sum l`, `...`
- `head l`, `tail l`
- Dually, `init l`, `last l`

  `init [1,2,3] = [1,2]`, `last [1,2,3] = 3`

Some builtin list functions

- `length l, reverse l, sum l, ...`
- `head l`, `tail l`
- Dually, `init l`, `last l`

  `init [1,2,3] = [1,2]`, `last [1,2,3] = 3`
- `take n l` — extract the first `n` elements of `l`

  `drop n l` — drop the first `n` elements of `l`

Some builtin list functions

- `length l, reverse l, sum l, ...`
- `head l`, `tail l`
- Dually, `init l`, `last l`

  `init [1,2,3] = [1,2]`, `last [1,2,3] = 3`
- `take n l` — extract the first `n` elements of `l`

  `drop n l` — drop the first `n` elements of `l`
- Shortcut list notation

Some builtin list functions

- `length l, reverse l, sum l, ...`
- `head l`, `tail l`
- Dually, `init l`, `last l`

  `init [1,2,3] = [1,2]`, `last [1,2,3] = 3`
- `take n l` — extract the first `n` elements of `l`

  `drop n l` — drop the first `n` elements of `l`
- Shortcut list notation
  - `[m..n]` abbreviates the list `[m,m+1,...,n]`

    Example `[3..7] = [3,4,5,6,7]`

Some builtin list functions

- `length l, reverse l, sum l, ...`
- `head l`, `tail l`
- Dually, `init l`, `last l`

  `init [1,2,3] = [1,2]`, `last [1,2,3] = 3`
- `take n l` — extract the first `n` elements of `l`

  `drop n l` — drop the first `n` elements of `l`
- Shortcut list notation
  - `[m..n]` abbreviates the list `[m,m+1,...,n]`

    Example `[3..7] = [3,4,5,6,7]`
  - Arithmetic progressions

    `[1,3..8] = [1,3,5,7]`

    `[9,8..5] = [9,8,7,6,5]`

Operating on each element of a list

- `map f l` applies `f` to each item of `l`

Operating on each element of a list

- `map f l` applies `f` to each item of `l`

```
square ::  Int -> Int
square n = n*n
map square [1,2,4,9] ⇝ [2,4,9,81]
```

Operating on each element of a list

- `map f l` applies `f` to each item of `l`

```
square ::  Int -> Int
square n = n*n
map square [1,2,4,9] ⇝ [2,4,9,81]
```

- `filter p l` selects items from `l` that satisfy `p`

Operating on each element of a list

- `map f l` applies `f` to each item of `l`

```
square ::  Int -> Int
square n = n*n
map square [1,2,4,9] ⇝ [2,4,9,81]
```

- `filter p l` selects items from `l` that satisfy `p`

```
even ::  Int -> Bool
even x = (mod x 2 == 0)
filter even [1,2,4,9] ⇝ [2,4]
```

Operating on each element of a list

- `map f l` applies `f` to each item of `l`

```
square :: Int -> Int
square n = n*n
map square [1,2,4,9] ⇝ [2,4,9,81]
```

- `filter p l` selects items from `l` that satisfy `p`

```
even :: Int -> Bool
even x = (mod x 2 == 0)
filter even [1,2,4,9] ⇝ [2,4]
```

- Can compose these functions

Operating on each element of a list

- `map f l` applies `f` to each item of `l`

```
square ::  Int -> Int
square n = n*n
map square [1,2,4,9] ⇝ [2,4,9,81]
```

- `filter p l` selects items from `l` that satisfy `p`

```
even ::  Int -> Bool
even x = (mod x 2 == 0)
filter even [1,2,4,9] ⇝ [2,4]
```

- Can compose these functions

```
map square (filter even [1..10]) ⇝
                         [4,16,36,64,100]
```

List comprehension: New lists from old

- The set of squares of the even numbers between 1 and 10

$$\{x^2 \mid x \in \{1, \ldots, 10\}, even(x)\}$$

List comprehension: New lists from old

- The set of squares of the even numbers between 1 and 10

$$\{x^2 \mid x \in \{1, \ldots, 10\}, even(x)\}$$

- The list of squares of the even numbers between 1 and 10

```
[ square x | x <- [1..10], even x ]
     where even x = (mod x 2 == 0)
           square x = x*x
```

Using list comprehensions ...

- ```
  divisors ::  Int -> [Int]
  divisors n = [ m | m <- [1..n],
                     mod n m == 0 ]
  ```
  ```

Using list comprehensions . . .

- ```
  divisors :: Int -> [Int]
  divisors n = [ m | m <- [1..n],
                     mod n m == 0 ]
  ```

- ```
  prime :: Int -> Bool
  prime n = (divisors n == [1,n])
  ```

Example: Quicksort

- Choose an element of the list as a splitter and create sublists of elements smaller than the splitter and larger than the splitter
- Recursively sort these sublists and combine

Example: Quicksort

- Choose an element of the list as a splitter and create sublists of elements smaller than the splitter and larger than the splitter
- Recursively sort these sublists and combine

```
qsort [] = []
qsort l =
 qsort lower ++ [splitter] ++ qsort upper
 where
  splitter = head l
  lower = [i | i <- tail l, i <= splitter]
  upper = [i | i <- tail l, i > splitter]
```

Polymorphism

- Are

```
length ::  [Int] -> Int
length ::  [Float] -> Int
```

different functions?

Polymorphism

- Are

  ```
  length :: [Int] -> Int
  length :: [Float] -> Int
  ```

  different functions?

- `length` only looks at the “structure” of the list, not “into” individual elements

  For any underlying type `t`, `length :: [t] -> Int`

Polymorphism

- Are

  `length :: [Int] -> Int`

  `length :: [Float] -> Int`

  different functions?

- `length` only looks at the “structure” of the list, not “into” individual elements

  For any underlying type `t`, `length :: [t] -> Int`

- Use `a,b,...` to denote generic types

  So, `length :: [a] -> Int`

Polymorphism

- Are

  ```
  length :: [Int] -> Int
  length :: [Float] -> Int
  ```

  different functions?

- `length` only looks at the "structure" of the list, not "into" individual elements

  For any underlying type `t`, `length :: [t] -> Int`

- Use `a,b,...` to denote generic types

  So, `length :: [a] -> Int`

- Similarly, the most general type of `reverse` is `reverse :: [a] -> [a]`

Polymorphism versus overloading

- "True" polymorphism

  The same computation is performed for different types

Polymorphism versus overloading

- “True” polymorphism

  The same computation is performed for different types

- Overloading

  Same symbol or function name denotes different computations for different types

Polymorphism versus overloading

- “True” polymorphism

  The same computation is performed for different types

- Overloading

  Same symbol or function name denotes different computations for different types

- Example Arithmetic operators

  At bit level, algorithms for `Int + Int` and `Float + Float` are different

Polymorphism versus overloading

- “True” polymorphism

  The same computation is performed for different types

- Overloading

  Same symbol or function name denotes different computations for different types

- Example Arithmetic operators

  At bit level, algorithms for `Int + Int` and `Float + Float` are different

- What about subclass polymorphism in OO programming?

Polymorphism versus overloading in OO

```
class Shape {
}
class Circle extends Shape {
  double size {return pi*radius*radius}
}
class Square extends Shape {
  double size {return side*side}
}
Shape s1 = new Circle; print s1.size();
Shape s2 = new Square; print s2.size();
```

Polymorphism versus overloading in OO

- ```
  class Shape {
  }
  class Circle extends Shape {
    double size {return pi*radius*radius}
  }
  class Square extends Shape {
    double size {return side*side}
  }
  Shape s1 = new Circle; print s1.size();
  Shape s2 = new Square; print s2.size();
  ```
- Implementation of `size` is different!!
  ```

Conditional polymorphism

- What about

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

Conditional polymorphism

- What about
```
member x [] = False
member x (y:l) | (x == y) = True
               | otherwise = member x l
```
- Is `member ::  a -> [a] -> Bool` a valid description of the type?

Conditional polymorphism

- What about
```
member x [] = False
member x (y:l) | (x == y) = True
               | otherwise = member x l
```
- Is `member ::  a -> [a] -> Bool` a valid description of the type?
- What is the value of `member qsort [qsort, mergesort, plus]`?

  Equality of functions cannot be checked effectively

Conditional polymorphism

- What about
```
member x [] = False
member x (y:l) | (x == y) = True
               | otherwise = member x l
```
- Is `member ::  a -> [a] -> Bool` a valid description of the type?
- What is the value of `member qsort [qsort, mergesort, plus]`?

  Equality of functions cannot be checked effectively
- The underlying type should support equality

Type classes

- Haskell organizes types into classes. A type class is a subset of all types.
- The class `Eq` contains all types that support `==` on their elements.

  The "predicate" `Eq a` tells whether or not `a` belongs to `Eq`
- Haskell would type this as

```
member ::  Eq a => a -> [a] -> Bool
```

- Likewise `Ord a` is the set of types that support comparison, so

```
quickSort:  Ord a => [a] -> [a]
```

Examples of declarative programming

Compute all initial segments of a list

Examples of declarative programming

Compute all initial segments of a list

- Initial segments of [ ] are empty

Compute all initial segments of a list

- Initial segments of `[]` are empty
- Initial segments of `x:l` — all initial segments of `l` with x in front, plus the empty segment

Examples of declarative programming

Compute all initial segments of a list

- Initial segments of `[]` are empty
- Initial segments of `x:l` — all initial segments of `l` with x in front, plus the empty segment
- ```
  initial ::  [a] -> [[a]]
  initial [] = [[]]
  initial (x:l) = [[]] ++
                      [ x:z | z <- initial l]
  ```
  ```

Examples of declarative programming . . .

- The empty list [ ] has no permutations

Examples of declarative programming . . .

- The empty list `[]` has no permutations
- Permutations of `x:l`
  "Interleave" `x` through each permutation of `l`

Examples of declarative programming . . .

- The empty list `[]` has no permutations
- Permutations of `x:l`
  "Interleave" `x` through each permutation of `l`
- ```
  interleave ::  a -> [a] -> [[a]]
  interleave x [] = [[x]]
  interleave x (y:l) =
   [x:y:l] ++
   [y:l2 | l2 <- (interleave x l)]

  perms ::  [a] -> [[a]]
  perms [] = [[]]
  perms (x:l) =
    [z | y <- perms l, z <- interleave x y]
  ```
  ```

Second lecture

- User defined datatypes

  Stacks, queues, trees, . . .
- Hiding implementation details using modules
- “Infinite” data structures