Revision
Haskell

• **Pure:** Only functions in a mathematical sense of a function can be written in Haskell! Functions have no side-effects.

  Functions are **first-class**, that is, functions are values which can be used in exactly the same ways as any other sort of value.

• **Lazy:** Expressions are not evaluated until their results are needed.

  We can work with infinite data structures: li
Guarded expressions

Allow alternatives (conditions) in functions. Each alternative is called a guard.
Guards are evaluated in order from top to bottom (first guard\(_1\), then guard\(_2\), etc.) until the top-most guard that evaluates to True.
If this guard is guard\(_i\), then the result of \texttt{foo} is result\(_i\)

\[
\text{foo } x_1 \ x_2 \ \ldots \ x_k
\]
\[
| \text{guard}_1 = \text{result}_1 \\
| \text{guard}_2 = \text{result}_2 \\
\ldots
\]
\[
| \text{otherwise} = \text{result}_n
\]
Haskell Type System

Some important features:

• static typing (every Haskell expression has a type, and types are all checked at compile-time; programs with type errors will not even compile)
• type inference
• polymorphic functions (using parametric polymorphism, a function or a data type can be written generically so that it can handle values identically without depending on their type)
• type classes
Algebraic datatypes

\textbf{data} Cutlery = Spoon | Fork | Knife

\textbf{data} Thing = Spoon | Shoe | Orange | Soap | Possum
\textbf{data} Sweet = Chocolate \mid Lollipop \mid Marshmallow

\textbf{data} Sweet = Chocolate \text{ Int } \text{ChocoKind} \\
\mid Lollipop \text{ Int } \text{Color} \\
\mid Marshmallow

\textbf{data} ChocoKind = Dark \mid Milk \mid White
\textbf{data} Color = Red \mid Green \mid Blue \mid Yellow \mid Mix
Deconstructing Types

chocolateKind :: Sweet -> ChocoKind
chocolateKind (Chocolate _ chocokind) = chocokind
chocolateKind _ | = error "Not a chocolate"

• We specify one of the constructor functions and give names to each argument of the constructor. This "giving names" is called is **variables binding**.

• "Binding" is used to assign a variable to each of the values so that we can refer to them on the right side of the function definition.

```
chocolateKind :: Sweet -> ChocoKind
chocolateKind (Chocolate kind _) = kind
chocolateKind _ | = error "Not a chocolate"
```
Case expressions

```haskell
chocolateKind sweet = case sweet of
    Chocolate _ chocokind -> chocokind
    _ -> error "Not a chocolate"
```

Generally:

```haskell
case expression of pattern₁ -> result₁
    pattern₂ -> result₂
    ... patternₙ -> resultₙ
```
Case expressions

describeList :: [a] -> String

describeList xs =
   "The list is " ++ case xs of
      [] -> "empty."
      [_] -> "a singleton list."
     _:_ -> "a longer list."
Tuples

A **fixed** number of values of (possibly different) fixed types **combined** into a single value.

Example: associating prices to shopping items:

("Salt", 139)
("Chips", 25)

type ShopItem = (String, Int)
Generally:

Type \((t_1, t_2, \ldots, t_n)\) describes values \((v_1, v_2, \ldots, v_n)\) where

\[ v_1 :: t_1, \quad v_2 :: t_2, \ldots, \quad v_n :: t_n \]

I.e. \(v_1\) is of type \(t_1\),

\(v_2\) is of type \(t_2\),

\(\ldots\),

\(v_n\) is of type \(t_n\)

Generalizes pairs, triples, quadruples, quintuples, sextuples, etc.
type ShopItem = (String, Int)

name :: ShopItem -> String
name (n, p) = n

price :: ShopItem -> Int
price (n, p) = p
Records

data Dessert = Choco {weight :: Int, kind :: ChocoKind}
  | Lolli {weight :: Int, color :: Color}
  | Marsh

Field names as accessor functions

Prelude> :t weight
weight :: Dessert -> Int

Prelude> :t kind
kind :: Dessert -> ChocoKind

Prelude> :t color
color :: Dessert -> Color
Extracting values from records

`getWeight :: Dessert -> Int`
`getWeight Choco{weight = w} = w`
`getWeight Lolli{weight = w} = w`
`getWeight _ = error "Your dessert is weightless"

`getWeight2 :: Dessert -> Int`
`getWeight2 (Choco w _) = w`
`getWeight2 (Lolli w _) = w`
`getWeight2 _ = error "Your dessert is weightless"

`getWeight3 :: Dessert -> Int`
`getWeight3 dsrt = weight dsrt`
biteDessert :: Dessert -> Dessert
biteDessert dsrt = dsrt { weight = weight dsrt - 1 }

eatDessert :: Dessert -> Dessert
eatDessert dsrt = dsrt { weight = 0 }

The field names for a record
• work as functions to get the value of a field (accessor functions)
• operate as labels in the syntax that uses braces
Fundamental rules for creating good recursive functions

• There must be a base case (or cases)
• Each recursive call must lead towards a base case

Base case – defining a value for which the function is evaluated without recursively calling itself

• We can always take an unbounded recursion with a starting point and transform it into a bounded recursion where the starting point is the base case.
Basic idea behind good recursive algorithms

To solve a problem, solve a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem.
Factorial

A factorial of a non-negative integer $n$, denoted by $n!$, is the product of all positive integers less than or equal to $n$.

$0! = 1$ (by convention)
$1! = 1$
$2! = 2 \times 1$
$3! = 3 \times 2 \times 1$
$4! = 4 \times 3 \times 2 \times 1$
$5! = 5 \times 4 \times 3 \times 2 \times 1$
$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$
$\ldots$

$2! = 2 \times 1!$
$3! = 3 \times 2!$
$4! = 4 \times 3!$
$5! = 5 \times 4!$
$6! = 6 \times 5!$
$\ldots$

$n! = n \times (n-1)!$
fac :: Integer -> Integer
fac n
  | n == 0    = 1
  | n > 0     = n * fac (n-1)
  | otherwise = error "Negative argument"

Function evaluation on input value 4

fac 4
→ 4 * fac 3
→ 4 * (3 * fac 2)
→ 4 * (3 * (2 * fac 1))
→ 4 * (3 * (2 * (1 * fac 0)))
→ 4 * (3 * (2 * (1 * 1)))
→ 4 * (3 * (2 * 1))
→ 4 * (3 * 2)
→ 4 * 6
→ 24
Recursive Datatypes / Lists

The list type is an abstract datatype that represents a countable number of ordered values, where the same value may occur more than once.

In functional programming, list datatypes are recursive data types defined with

- a constructor for creating an empty list
- a constructor for adding an item at the beginning of a list

All objects of recursive types can contain an arbitrary number of nesting of constructors.

Haskell allows us to build 'infinite values' with finite representation. Hence, Haskell lists can be infinite.
Typical Operations on Lists

• Determine the first element (head) of a list
• Determine the elements after the head of the list (tail)
• Compute the length of a list
• Prepend an entity to a list
• Concatenate two lists
• Check whether the list is empty
• Check whether an element belongs to a list
• Return the number of times an element occurs in a list
• Return a prefix of length n of a list
• ...
User-defined Polymorphic Type for List

data List a = Nil | Entry a (List a)
data [a] = [] | a : [a] deriving (Eq, Ord)

Compare with our custom list datatype definition:

data CustomList a = Empty | Cons a (CustomList a)
Haskell Built-in Lists

data  [a] = [] | a : [a]  deriving (Eq, Ord)

Compare with our custom list datatype definition:

data List a = Nil | Entry a (List a)
Every list is either the **empty** list [], or a non-empty list.

If a list is non-empty it can be written in the form x:xs, where x is the first item of the list (the **head**) and xs is the rest of the list (the **tail**).

The operator : is the list constructor, called “cons” (from Lisp).

The infix operator (:) :: a->[a]->[a] is **right-associative**: x:y:zs == x:(y:zs)
Examples

4:\[2,3\] == \[4,2,3\]
    where 4 is the head and \[2,3\] is the tail.

3:\[] == \[3\]
2:3:\[] == \[2,3\]
4:2:3:\[] == \[4,2,3\]

4:2:3:\[] == 4:(2:(3:\[]))
List Patterns

A constructor pattern over lists will either be \([]\) or have the form \((p:ps)\):

• A list matches \([]\) if and only if it is empty

• A list matches \((p:ps)\) if and only if it is non-empty and its head matches \(p\) and its tail matches \(ps\)
# Examples of Built-in Lists

<table>
<thead>
<tr>
<th>Empty list</th>
<th>Haskell Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>[t]</td>
<td></td>
</tr>
<tr>
<td>List consisting of a single element 1</td>
<td>1 : [ ]</td>
<td>Num a =&gt; [a]</td>
</tr>
<tr>
<td>List consisting of a single element ‘a’</td>
<td>‘a’ : [ ]</td>
<td>[Char]</td>
</tr>
<tr>
<td>List consisting of a single element “hello”</td>
<td>“hello” : [ ]</td>
<td>[[Char]]</td>
</tr>
<tr>
<td>List consisting of an element 3 followed by an element 2.3</td>
<td>3 : (2.3 : [])</td>
<td>Fractional a =&gt; [a]</td>
</tr>
</tbody>
</table>
List Comprehensions

Describe the elements of one list in terms of another list:

\[
[ 2 * n \mid n \leftarrow [2,4,7] ] \quad \text{is} \quad [4,8,14]
\]

\[
[\text{even } n \mid n \leftarrow [2,4,7] ] \quad \text{is} \quad [\text{True,True,}\text{False}]
\]

\[
[ 2 * n \mid n \leftarrow [2,4,7], \text{ even } n, n > 3 ]
\]

\quad \text{is} \quad [8]
Higher Order Functions

In Haskell functions are themselves data that can be:

1. applied to fewer than all of their arguments to return a new partially applied function
2. passed as arguments to or returned from functions, which are then called higher order functions
3. expressed as literal values (expressions) without naming them in the form of lambda abstractions
4. combined using operators
Curried Functions

- Functions in Haskell take their arguments one at a time: this is called *currying* after Haskell Curry of λ-calculus fame.

- Currying supports partial application, and makes Haskell syntax clean.
• Every function in Haskell takes exactly one argument.
• If this application yields a function, then this function may be applied to a further argument, and so on.
• Putting a space between two things is simply function application.
• The space is sort of like an operator and it has the highest precedence.

For example

\[
\text{multiply} :: \text{Int} \to \text{Int} \to \text{Int}
\]

is in fact:

\[
\text{multiply} :: \text{Int} \to (\text{Int} \to \text{Int})
\]

So, we have:

\[
\text{multiply } 2 :: \text{Int} \to \text{Int}
\]
In general:

\[ f \ e_1 \ e_2 \ldots \ e_k \]
\[ t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \]

are shorthands for:

\[ (((\ldots((f \ e_1) e_2)\ldots) e_k) \]
\[ t_1 \rightarrow (t_2 \rightarrow (\ldots(t_n \rightarrow t)\ldots)\]

i.e., function application is **left-associative**

\[ \rightarrow \text{ is right-associative} \]
multiplyThree :: Int -> Int -> Int -> Int
multiplyThree x y z :: x * y * z

multiplyThree 1 3 5 is actually
(((multiplyThree 1) 3) 5)

(multiplyThree 1) :: Int -> Int -> Int
(((multiplyThree 1) 3) 5) :: Int
multiplyThree :: Int -> (Int -> (Int -> Int))
multiplyThree x y z :: x * y * z

multiplyThree 1 3 5
(((multiplyThree 1) 3) 5)

(multiplyThree 1) :: Int -> (Int -> Int)
((multiplyThree 1) 3) :: Int -> Int
(((multiplyThree 1) 3) 5) :: Int
Generic HOF

• *Transform every element of a list in some way* — **map**
  • convert all characters of a string to upper case
  • double every number in the list

• *Combine the elements of a list using some operator* — **fold/reduce**
  • sum the numeric elements of a list

We use **functions as arguments** (e.g., convert, double, sum) to **generic** functions over lists.
Don’t Repeat Yourself

• nothing should be duplicated: every idea, algorithm, and piece of data should occur exactly once in your code.

• taking similar pieces of code and factoring out their commonality is known as the process of abstraction.

• Haskell is very good at abstraction due to features such as
  • parametric polymorphism
  • higher-order functions
  • type classes
map

map :: (a -> b) -> [a] -> [b]

base Prelude, base Data.List

map f xs is the list obtained by applying f to each element of xs, i.e.,

> map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
> map f [x1, x2, ...] == [f x1, f x2, ...]
**filter**

\[
\texttt{filter :: (a -> \text{Bool}) \to [a] \to [a]}
\]

base Prelude, base Data.List

- filter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

\[
> \text{filter \ p \ xs} = [x \mid x \leftarrow \text{xs}, p \ x]
\]
foldr

\[
\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b
\]

base Prelude, base Data.List

foldr, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

\[
> \text{foldr } f \ z \ [x_1, x_2, \ldots, x_n] \equiv x_1 \mathbin{\bigcirc} f \ (x_2 \mathbin{\bigcirc} f \ \ldots \ (x_n \mathbin{\bigcirc} f \ z)\ldots)
\]
foldl

\[ \textbf{foldl} :: (a \to b \to a) \to a \to [b] \to a \]

\textit{base Prelude, base Data.List}

- foldl, applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

\[ > \text{foldl } f \ z \ [x_1, x_2, \ldots, x_n] = (\ldots((z \ f \ x_1) \ f \ x_2) \ f \ldots) \ f \ x_n \]

The list must be finite.

It folds the list up from the left side.
iterate

iterate :: (a -> a) -> a -> [a]

base Prelude, base Data.List

iterate f x returns an infinite list of repeated applications of f to x:

> iterate f x == [x, f x, f (f x), ...]
Anonymous Functions

• An anonymous function is a function without a name.

• Anonymous functions are often used in a context where a functional expression is used only once in a higher order function.

• Often anonymous functions make code easier and faster to comprehend.

• Sometimes it is more convenient to use anonymous functions. This is often the case when using \texttt{map} and \texttt{foldl} / \texttt{foldr}. 
addOne :: Enum a => a -> a
addOne x = x + 1

can be rewritten as

\( x \rightarrow x + 1 \)
ghci> (\x -> x + 1) 2
3

ghci> (\x -> x + 1) 100
101

ghci> map (\x -> x + 1) [1..5]
[2,3,4,5,6]
sumNums :: Num a => a -> a -> a
sumNums = x + y

can be rewritten as

\x y -> x + y

ghci> foldr (\x y -> x + y) 0 [1..5]
15
Eta Conversion

\[ x \rightarrow x + 1 \]

is equivalent to

\( (+1) \)
addOneList lst = map addOne lst
    where
    addOne x = x + 1

addOneList' lst = map (\x -> x + 1) lst

addOneList'' = map (+1)
ghci> map (+1)[1..5]
[2,3,4,5,6]