Functions

- Functions are the most important kind of value in functional programming
  - Functions are values!
- Mathematically, a function $f$ associates an element of a set $X$ to a unique element of second set $Y$
  - We write $f : X \rightarrow Y$

```haskell
three :: Integer -> Integer
infinity :: Integer
square :: Integer -> Integer
smaller :: (Integer, Integer) -> Integer
```
Currying Functions

• Functional can be applied to a partially resulting in new functions with fewer arguments

\[
\text{smaller} :: (\text{Integer, Integer}) \rightarrow \text{Integer} \\
\text{smaller} (x, y) = \text{if } x \leq y \text{ then } x \text{ else } y
\]

\[
\text{smaller2} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{smaller2} x y = \text{if } x \leq y \text{ then } x \text{ else } y
\]

• The value of the application \(\text{smaller2} \ x\) is a function with type \(\text{Integer} \rightarrow \text{Integer}\)
  – This is known as \textit{currying} a function

Curried Functions

• Curried functions help reduce the number of parentheses
  – Parentheses make the syntax of some functional languages (e.g. Lisp, Scheme) ugly

• Curried functions are useful

\[
\text{twice} :: (\text{Integer} \rightarrow \text{Integer}) \rightarrow (\text{Integer} \rightarrow \text{Integer}) \\
\text{twice } f \ x = f \ (f \ x)
\]

\[
\text{square} :: \text{Integer} \rightarrow \text{Integer} \\
\text{square } x = x \ast x
\]

\[
\text{quad} :: \text{Integer} \rightarrow \text{Integer} \\
\text{quad} = \text{twice square}
\]
Operators

- **Operators** are functions in infix notation rather than prefix notation
  - *E.g.* `3 + 4` rather than `plus 3 4`

- Functions can be used in infix notation using the ‘`f`’ notation
  - *E.g.* `3 `plus` 4`

- Operators can be used in prefix notation using the `(op)` notation
  - *E.g.* `(+) 3 4`

- As any other function, operators can be applied partially using the `sections` notation
  - *E.g.* The type of `(+3)` is `Integer -> Integer`

Operator Associativity

- Operators associate from left to right (*left-associative*) or from right to left (*right-associative*)
  - *E.g.* `->` is right-associative `3 -> 4 -> 5` means `(3 -> 4) -> 5`

- Operator `->` is right-associative
  - `x -> y -> z` means `x -> (y -> z)`

- Exercise: deduce the type of `h` in

\[
\text{h x y = f (g x y)}
\]

```
f :: Integer -> Integer
g :: Integer -> Integer -> Integer
```

Felix Hernandez-Campos
Function Definition

- Functions with no parameters are constants
  - E.g. \( \pi = 3.14 \) has type \( \pi :: \text{Float} \)
- The definition of a function can depend on the value of the parameters
  - Definitions with conditional expressions
  - Definitions with guarded equations

```haskell
smaller :: Integer -> Integer -> Integer
smaller x y = if x <= y then x else y
smaller2 x y
  | x <= y = x
  | y > x  = y
```

Recursive definitions

- Functional languages make extensive use of recursion

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

- What is the result of \( \text{fact} \ -1 \)?
- The following definition is more a

```haskell
fact n
  | n < 0   = error "negative argument for fact"
  | n == 0  = 1
  | otherwise = n * fact (n-1)
```
Local Definitions

- Another useful notation in function definition is a local definition.
- Local declarations are introduced using the keyword where.
- For instance,

\[
f :: \text{Integer} \to \text{Integer} \to \text{Integer}
f \ x \ y
\]
\[
| x \leq 10 = x + a \\
| x > 10 = x - a
\]
where \( a = \text{square } b \)
\( b = y + 1 \)

Types

- The following primitive types are available in Haskell:
  - \text{Bool}
  - \text{Integer}
  - \text{Float}
  - \text{Double}
  - \text{Char}
- Any expression in Haskell has a type:
  - Primitive types
  - Derived types
  - Polymorphic types
Polymorphic Types

- Some functions and operations work with many types
- Polymorphic types are specified using type variables
- For instance, the curry function has a polymorphic type

\[
\text{curry} :: ((a, b) \to c) \to (a \to b \to c)
\]
\[
\text{curry } f \ x \ y = f \ (x, y)
\]

- Type variables can be qualified using type classes

\[
(*) :: \text{Num } a \Rightarrow a \to a 
\]

Lists

- Lists are the workhorse of functional programming
- Lists are denoted as sequences of elements separated by commas and enclosed within square brackets
  - E.g. [1, 2, 3]
- The previous notation is a shorthand for the List data type
  - E.g. 1:2:3:[]
• Functions that operate on lists often have polymorphic types

```
length :: [a] -> Integer
length []     = 0
length (x:xs) = 1 + length xs
```

• In the previous example, the appropriate definition of the function for the specific arguments was chosen using pattern matching.

Example Derivation

```
length :: [a] -> Integer
length []     = 0         \[1\]
length (x:xs) = 1 + length xs \[2\]
```

```
length [1,2,3] = \{ definition (2) \} 
                1 + length [2,3] 
                = \{ definition (2) \} 
                1 + 1 + length [3] 
                = \{ definition of + \} 
                2 + length [3] 
                = \{ definition (2) \} 
                2 + 1 + length [] 
```

```
= \{ definition of + \} 
  3 + length [] 
= \{ definition (1) \} 
  3 + 0 
= \{ definition of + \} 
  3
```
Lists
Other Functions

head :: [a] -> a
head (x:xs) = x

tail :: [a] -> [a]
tail (x:xs) = xs

Data Types

- New data types can be created using the keyword `data`
- Examples

  data Bool = False | True

  data Color = Red | Green | Blue | Violet

  data Point a = Pt a a
Polymorphic Types

- Type constructors have types

\[
\text{Pt} :: \text{a} \rightarrow \text{a} \rightarrow \text{Point}
\]

\[
\text{data Point a = Pt a a}
\]

- Examples

\[
\text{Pt 2.0 3.0 :: Point Float}
\]

\[
\text{Pt 'a' 'b' :: Point Char}
\]

\[
\text{Pt True False :: Point Bool}
\]

Recursive Types

Binary Trees

\[
\text{data Tree a = Leaf a | Branch (Tree a) (Tree a)}
\]

\[
\text{Branch :: Tree a \rightarrow Tree a \rightarrow Tree a}
\]

\[
\text{Leaf :: a \rightarrow Tree a}
\]

- Example function that operates on trees

\[
\text{fringe :: Tree a \rightarrow [a]}
\]

\[
\text{fringe (Leaf x) = [x]}
\]

\[
\text{fringe (Branch left right) = fringe left}
\]

\[
++ \text{ fringe right}
\]
**List Comprehensions**

- Lists can be defined by enumeration using *list comprehensions*
  - Syntax:
    
    ```haskell
    \[ f \ x \mid x \leftarrow xs \]
    \[ (x,y) \mid x \leftarrow xs, y \leftarrow ys \]
    
    - Example
      ```haskell
      quicksort [] = []
      quicksort (x:xs) = quicksort [y | y <- xs, y<x ]
                     ++ [x]
                     ++ quicksort [y | y <- xs, y>=x]
      ```

**Reading Assignment**

  - [http://www.haskell.org/tutorial/](http://www.haskell.org/tutorial/)
  - Read sections 1 and 2