

# A general introduction to Functional Programming using Haskell

Matteo Rossi  
Dipartimento di Elettronica e Informazione  
Politecnico di Milano  
rossi@elet.polimi.it

# Functional programming in a nutshell

- In mathematics, a function  $f: D \rightarrow R$  takes an argument  $d \in D$  and returns a value  $r \in R$ 
  - that's all it does, it does not “affect” anything else
  - it has no side effects
- The absence of side effects is the key idea behind functional programming
- In fact, one could program in a “functional” way also in classic imperative languages such as C, but functional programming languages enforce it
  - at least, “purely functional” PLs enforce it; most FPLs allow mixing functional style and imperative style
    - in some cases side effects can simplify programming considerably
- In these introductory lectures we present functional programming using Haskell as reference language
  - one of the “pure” FPLs available
- Introductory reference text:  
*Programming in Haskell*  
G. Hutton  
Cambridge University Press, 2007

# First examples in Haskell

- Key mechanism in FPL: function definition

```
double x = x + x
```

- Evaluation through function application:

```
double 3
= { applying double }
  3+3
= { applying + }
  6
```

- Absence of side effects means that evaluation order does not affect the result

- Compare

```
double (double 2)
= { applying the inner double }
  double (2 + 2)
= { applying + }
  double 4
= { applying double }
  4+4
= { applying + }
  8
```

with

```
double (double 2)
= { applying the outer double }
  double 2 + double 2
= { applying the first double }
  (2 + 2) + double 2
= { applying the first + }
  4 + double 2
= { applying double }
  4 + (2 + 2)
= { applying the second + }
  4+4
= { applying + }
  8
```

## First examples (2)

- A typical fragment of imperative programming:

```
count := 0
total := 0
repeat
    count := count + 1
    total := total + count
until
    count = n
```

- In purely functional programming there is no notion of “variable”, whose value changes as the execution progresses nor, in fact, of “loop”
- The basic mechanism for repetition in FP is recursion
- Compare with the definition of function `sum_up_to` given by the following equation:

```
sum_up_to n = if n == 0 then 0 else
              n + sum_up_to (n-1)
```

- An alternative definition:

```
sum_up_to2 n = sum_seq [1..n]
where
    sum_seq [] = 0
    sum_seq (x:xs) = x + sum_seq xs
```

- `sum_up_to2` is a function that takes a value `n` as input
- it is computed by applying function `sum_seq` to the sequence of values from 1 to `n`
- `sum_seq` is defined through the two equations that follow the **where** keyword; it is a function that is applied to sequences of values
  - if the sequence to which `sum_seq` is applied is empty, the result is 0
  - otherwise, the result of `sum_seq` is given by adding the first element of the sequence to the sum of the other elements

# Quicksort in Haskell

- ```
qsort [] = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                   smaller = [a | a <- xs, a <= x ]
                   larger  = [b | b <- xs, b > x ]
```
- The two equations define that quicksort is a function that is applied to sequences of values and that:
  - if `qsort` is applied to an empty sequence, the sequence is already sorted
  - otherwise, let us call `x` the first element of the sequence, and `xs` the rest of the sequence; then, if `smaller` is the subsequence of `xs` that contains all elements no bigger than `x`, and `larger` is the subsequence of `xs` that contains all elements bigger than `x`, the sorted sequence is given by concatenating `smaller`, `x` and `larger`
- Example of execution:

```
qsort [3, 5, 1, 4, 2]
=   { applying qsort }
qsort [1, 2] ++ [3] ++ qsort [5, 4]
=   { applying qsort }
(qsort [] ++ [1] ++ qsort [2]) ++ [3]
++ (qsort [4] ++ [5] ++ qsort [])
=   { applying qsort, since qsort [x] = [x] }
([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
=   { applying ++ }
[1, 2] ++ [3] ++ [4, 5]
=   { applying ++ }
[1, 2, 3, 4, 5]
```

## Function application: Haskell syntax

- In mathematics, if  $f$  is a function that takes two arguments and  $a, b, c, d$  are values, we write:

$f(a, b) + cd$

- In Haskell the same is written as:

`f a b + c*d`

- function application has higher priority than other operators, hence the line above is equivalent to  
`(f a b) + c*d`

# Types in Haskell

- Haskell is statically typed
- we write `v :: T` to state that value `v` has type `T`
  - for example

```
False :: Bool
not  :: Bool -> Bool
not False :: Bool
```

    - `not` is a function that takes a `Bool` as argument and returns a `Bool`
- Some basic types (mostly self-explanatory):
  - `Bool`
  - `Char`
  - `String`
  - `Int`
  - `Integer`
    - `Integer` represents integer numbers with *arbitrary precision*
  - `Float`

# Lists

- Lists: sequences of elements of the same type, e.g.:  
`[False, True, False] :: [Bool]`  
`['a', 'b', 'c', 'd'] :: [Char]`  
`["One", "Two", "Three"] :: [String]`
- The empty list: `[]`
- Lists of lists, e.g.:  
`[['a', 'b'], ['c', 'd', 'e']] :: [[Char]]`
- Operations on lists (i.e., basic functions that take lists as arguments):
  - `length xs`
    - number of elements in list `xs`
  - `head xs`
    - first element of list `xs`
  - `tail xs`
    - all elements of list `xs` except first
  - `xs!!n`
    - n-th element of list `xs` (starting from 0)
  - `take n xs`
    - list made of first n elements of list `xs`
  - `drop n xs`
    - list obtained by removing first n elements of list `xs`
  - `xs ++ ys`
    - list obtained by appending list `ys` after list `xs`
  - `reverse xs`
    - list obtained by reversing the elements of list `xs`



# Tuples

- Tuple: finite sequence of elements of possibly different type, e.g.:  
`(False, True) :: (Bool, Bool)`  
`(False, 'a', True) :: (Bool, Char, Bool)`  
`("Yes", True, 'a') :: (String, Bool, Char)`
- empty tuple: `()`
- More examples:  
`('a', (False, 'b')) :: (Char, (Bool, Char))`  
`(['a', 'b'], [False, True]) :: ([Char], [Bool])`  
`[('a', False), ('b', True)] :: [(Char, Bool)]`

# Function types

- Function: mapping from values of a certain type to values of another type, e.g.:

```
not      :: Bool -> Bool
isDigit :: Char -> Bool
```

- Using tuples and lists no more than one argument is needed:

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

```
zeroto :: Int -> [Int]
zeroto n = [0..n]
```

- Another way of dealing with multiple arguments: functions that return functions, e.g.:

```
add' :: Int -> (Int -> Int)
add' x y = x + y
```

- add' is a function that takes an `Int` as argument, and returns a function that, given another `Int`, returns an `Int`

- It works also for more than two arguments:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

- Functions such as `add'` and `mult` that take arguments one at a time are called *curried*

# Curried functions and partial application

- Curried functions lend themselves to *partial application*
  - this occurs when not all arguments are supplied to the function application
  - the result of partially applying arguments to a curried function is another function, e.g.:

```
add' 1 :: Int -> Int
```

    - we could also define

```
inc :: Int -> Int
inc = add' 1
```

then, the result of `inc 10` is 11
- Note that the function arrow `->` associates to the right, i.e.

```
Int -> Int -> Int -> Int
```

means

```
Int -> (Int -> (Int -> Int))
```
- Function application, instead, associates to the left, i.e.

```
mult x y z
```

means

```
((mult x) y) z
```

# Polymorphic types

- `length` is a function that can be applied to lists of different types of elements:  
`length [1, 3, 5, 7]`  
`length ["Yes", "No"]`  
`length [ isDigit , isLower , isUpper ]`
- In fact, the type of `length` is *polymorphic* (and `length` is a polymorphic function) as it contains a *type variable*:  
`length :: [a] -> Int`
  - `a` is the type variable
    - type variables *must* start with a lowercase letter
- Other examples of polymorphic functions:  
`fst :: (a, b) -> a`  
`head :: [a] -> a`  
`take :: Int -> [a] -> [a]`  
`zip :: [a] -> [b] -> [(a, b)]`  
`id :: a -> a`

# Type classes, overloaded types, methods

- Type class = a collection of types
  - for example, class Num contains any numeric types (e.g., Int, Float)
- If  $a$  is a type variable and  $C$  is a type class,  $C\ a$  is a class constraint
  - it states that type  $a$  must be an instance of type class  $C$
- Overloaded type: a type that includes a class constraint, e.g.,  
 $3 :: Num\ a \Rightarrow a$ 
  - $3$  is a constant that is defined for any numeric type  $a$
- Also,  $+$  is an overloaded function:  
 $(+) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$ 
  - $(+)$  is a (curried) function that can be applied to any pairs of values that belong to an instance of type class Num
- Other examples of overloaded functions:  
 $(-) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$   
 $(*) :: Num\ a \Rightarrow a \rightarrow a \rightarrow a$   
 $negate :: Num\ a \Rightarrow a \rightarrow a$   
 $abs :: Num\ a \Rightarrow a \rightarrow a$   
 $signum :: Num\ a \Rightarrow a \rightarrow a$
- In general, a type class defines *methods*, i.e., overloaded functions that can be applied to values of instances of the type class
- For example, type class Eq contains types that can be compared for equality and inequality; as such, it defines the following 2 methods:  
 $(==) :: a \rightarrow a \rightarrow Bool$   
 $(/=) :: a \rightarrow a \rightarrow Bool$ 
  - Bool, Char, String, Int, Integer, Float are all instances of Eq; so are list and tuple types, if their element types are instances of Eq

## Class declaration

- A new class is declared using the `class` keyword:

```
class Eq a where
```

```
  (==), (=) :: a -> a -> Bool
```

```
  x /= y = not (x == y)
```

- this declaration contains a default definition for method `/=`, so an instance of `Eq` must only define method `==`

- Example o instance of class `Eq`:

```
instance Eq Bool where
```

```
  False == False = True
```

```
  True == True = True
```

```
  _ == _ = False
```

- Classes can be extended to form new classes:

```
class Eq a => Ord a where
```

```
  (<), (<=), (>), (>=) :: a -> a -> Bool
```

```
  min, max :: a -> a -> a
```

```
  min x y | x <= y = x
```

```
          | otherwise = y
```

```
  max x y | x ≤ y = y
```

```
          | otherwise = x
```

- To define an instance of `Ord`, we need to provide the definition of methods `<`, `<=`, `>` and `>=`:

```
instance Ord Bool where
```

```
  False < True = True
```

```
  _ < _ = False
```

```
  b <= c = (b < c) || (b == c)
```

```
  b > c = c < b
```

```
  b >= c = c <= b
```

# Other basic classes

- **Ord** – ordered types
  - it contains instances of class `Eq`, whose values in addition are totally (linearly) ordered
  - it provides the following six methods:
    - `(<)` :: `a -> a -> Bool`
    - `(<=)` :: `a -> a -> Bool`
    - `(>)` :: `a -> a -> Bool`
    - `(>=)` :: `a -> a -> Bool`
    - `min` :: `a -> a -> a`
    - `max` :: `a -> a -> a`
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` are instances of `Ord`; so are list and tuple types, if their elements...
- **Show** – showable types
  - types whose values can be converted into strings of characters using the following method
    - `show` :: `a -> String`
    - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` are instances of `Show`; so are list and tuple types, if their elements...
- **Read** – readable types
  - types whose values can be obtained from strings of characters using the following method
    - `read` :: `String -> a`
    - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` are instances of `Read`; so are list and tuple types, if their elements...

## Other basic classes (cont.)

- **Num** – numeric types
  - types that are instances of both `Eq` and `Show`, whose values are also numeric, hence methods `(+)`, `(-)`, `(*)`, `negate`, `abs`, `signum` can be applied to them
    - examples of instances: `Int`, `Integer`, `Float`
- **Integral** – integral types
  - `Num` plus the following methods:  
`div :: a -> a -> a`  
`mod :: a -> a -> a`
    - examples of instances: `Int`, `Integer`
- **Fractional** – fractional types
  - `Num`, plus the following methods:  
`(/) :: a -> a -> a`  
`recip :: a -> a`
    - examples of instances: `Float`



## Mechanisms to define functions

- From previous functions:

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

- Using conditional expressions

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
           if n == 0 then 0 else 1
```

- the else branch is *mandatory*

- Using guarded equations:

```
signum n | n < 0      = -1
         | n == 0    = 0
         | otherwise = 1
```

- guards are evaluated in the order in which they are written
- `otherwise` (which is optional) is a synonym for `True`

# Pattern matching

- Example:

```
not :: Bool -> Bool
not False = True
not True  = False
```

- Example of *wildcard* pattern, which matches any value:

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_   && _     = False
```

- patterns are matched according to the order in which they are written

- An alternative definition for &&:

```
True  && b = b
False && _ = False
```

- if the first argument is `True`, then the value of `&&` coincides with the value of its second argument; if the first argument is `False`, the value of `&&` is `False` no matter the value of the second argument

- A tuple of patterns is itself a pattern, for example:

```
fst :: (a, b) -> a
fst (x, _) = x
```

```
snd :: (a, b) -> b
snd (_, y) = y
```

- Also, a list of patterns is itself a pattern:

```
first_of_3_a :: [Char] -> Bool
first_of_3_a ['a', _, _] = True
first_of_3_a _ = False
```

- `first_of_3_a` evaluates to `True` if its argument matches a list of 3 elements, in which the first one is `'a'`; if the argument matches anything else, `first_of_3_a` evaluates to `False`

# List patterns

- List patterns can be written using the *constructor operator* for lists ( : )
  - any list is constructed using : starting from the empty list [ ]
    - for example, the list [ 10 , 20 , 30 ] is constructed as follows:  
10 : ( 20 : ( 30 : [ ] ) )
      - essentially, [ 10 , 20 , 30 ] is an abbreviation for  
10 : ( 20 : ( 30 : [ ] ) )
    - : associates to the right, so 10 : ( 20 : ( 30 : [ ] ) ) is the same as 10 : 20 : 30 : [ ]
- Then, **x** : **xs** matches any list that has at least one element, **x**, and a (possibly empty) tail **xs**
  - if we are not interested in the value of the tail, we can write the pattern as **x** : \_

- Examples of definitions of functions on lists:

```
first_a :: [Char] -> Bool
first_a ('a' : _) = True
first_a _         = False
```

```
null :: [a] -> Bool
null []      = True
null (_:_)  = False
```

```
head :: [a] -> a
head (x : _) = x
```

```
tail :: [a] -> [a]
tail (_ : xs) = xs
```

# Lambda expressions

- A lambda expression is used to define an anonymous function
- It is made of:
  - a pattern for each argument of the function
  - a body, which defines how the result is computed from the values of the arguments
  - Examples:
    - $\backslash x \rightarrow x+x$
    - $\backslash (x, y) \rightarrow x+y$
    - $\backslash (x:xs) \rightarrow x^2$ 
      - then, if we evaluate the expression  $(\backslash (x:xs) \rightarrow x^2) [4,0,10]$  we obtain 16 as result
- The meaning of curried function definitions can be given in terms of lambda expressions
  - for example, the meaning of definition  $\text{add } x \ y = x + y$  is  $\text{add} = \backslash x \rightarrow (\backslash y \rightarrow x + y)$
- Lambda expressions are very useful to define functions that are used only once
  - this often occurs for functions that are used as parameters of other functions (more on this later), e.g.:  
 $\text{odds } n = \text{map } (\backslash x \rightarrow x*2+1) [0 \dots n-1]$ 
    - `odds` is a function that takes a number `n` as argument, and returns the list of the first `n` odd numbers
      - e.g., the result of `odds 3` is `[1,3,5]`
    - `map` is a function that takes two arguments, a function and a list, and returns the list obtained by applying the function to the elements of the list
      - e.g., the result of `map (\x -> x*2+1)[0,1,2]` is `[0*2+1, 1*2+1, 2*2+1]`, i.e., `[1,3,5]`

# List comprehensions

- Comprehension: building a set from an existing one, e.g.  
 $\{x^2 \mid x \in \{1..5\}\}$
- A similar notation exists in Haskell to build lists from existing ones, e.g.:  
`[x^2 | x <- [1..5]]`
  - the result is `[1, 4, 9, 16, 25]`
  - `x <- [1..5]` is a *generator*
- there can be more than one generator (separated by commas):  
`[(x,y) | x <- [1,2,3], y <- [4,5]]`
  - the result is `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`
- generators can depend on one another, e.g.:  
`concat :: [[a]] -> [a]`  
`concat xss = [ x | xs <- xss, x <- xs ]`
- *guards* can be used to filter out undesired results produced by earlier generators, e.g.:  
`factors :: Int -> [Int]`  
`factors n = [x | x <- [1..n], n `mod` x == 0]`
- Another example:  
`primes_lt :: Int -> [Int]`  
`primes_lt n = [x | x <- [1..n],  
                  (\y -> factors y /= [1,y]) x]`
- The `zip` function produces a new list by pairing successive elements from existing lists until one (or both) are exhausted:  
`zip ['a', 'b', 'c'][1,2,3,4]`  
results in `[('a',1),('b',2),('c',3)]`
- Example:  
`positions :: Eq a => a -> [a] -> [Int]`  
`positions x xs = let n = length xs-1 in  
                  [i | (x', i) <- zip xs [0..n],  
                          x' == x]`

## Some examples of recursion

- Recursion is one of the key mechanisms to define functions
- The principles are the same as for recursion in imperative languages: a recursive function is defined by re-applying itself to a “smaller argument”
- Examples of recursive functions on lists:

```
product :: Num a => [a] -> a
product []          = 1
product (x:xs)     = x*product xs
```

```
length :: [a] -> Int
length []          = 0
length (_ : xs)   = 1 + length xs
```

```
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

```
isort :: Ord a => [a] -> [a]
isort []          = []
isort (x : xs)   = insert x (isort xs)
```

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _          = []
zip _ []         = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
drop :: Int -> [a] -> [a]
drop 0 xs          = xs
drop (n+1) []     = []
drop (n+1) (_:xs) = drop n xs
```

# Higher-order functions

- In FPLs functions can both
  - return functions
  - take functions as arguments
- Curried functions are an example of the first kind:  
`add :: Int -> (Int -> Int)`  
`add = \x -> (\y -> x+y)`
  - `add` is a function that takes an `Int` as argument, and returns a function that, given an `Int`, returns an `Int`
    - `add 7` is a function that sums 7 to the `Int` passed as argument
- In FPLs a function can also take another function as argument:  
`twice :: (a -> a) -> a -> a`  
`twice f x = f (f x)`
  - function `twice` takes a function `f` and a value `x` as arguments, and applies `f` twice (first to `x`, then to the result of `f x`)
  - for example, the result of `twice (2*) 4` is 16, while the result of `twice reverse [2,10,1]` is `[2,10,1]`

# map and filter

- higher-order functions are often used to perform operations on lists
- A typical example (which we have already encountered):  
`map :: (a -> b) -> [a] -> [b]`  
`map f xs = [f x | x <- xs]`
  - e.g., the result of  
`map reverse ["abc", "def", "ghi"]`  
is `["cba", "fed", "ihg"]`
  - the result of `map (map (+1)) [[1, 2, 3], [4, 5]]` is  
`[[2,3,4],[5,6]]`
    - `map (+1)` returns a function that takes a list of numbers as argument, and returns it with values incremented by 1
- another example:  
`filter :: (a -> Bool) -> [a] -> [a]`  
`filter p xs = [x | x <- xs, p x]`
  - e.g., the result of `filter even [1..10]` is `[2,4,6,8,10]`
- an example of combination of `map` and `filter`:  
`sumsqreven :: Integral a => [a] -> a`  
`sumsqreven xs = sum (map (^2) (filter even xs))`
  - it returns the sum of the squares of the even elements of list `xs`



# foldr

- A pattern often used to define a function `f` that applies an operator  $\oplus$  to the values of a list (with `v` some value):

```
f [] = v
```

```
f (x:xs) = x  $\oplus$  f xs
```

- For example:

```
product [] = 1
```

```
product (x : xs) = x * product xs
```

```
and [] = True
```

```
and (x:xs) = x && and xs
```

- The (higher-order) `foldr` function captures this pattern:

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

```
foldr f v [] = v
```

```
foldr f v (x : xs) = f x (foldr f v xs)
```

- Defining functions `product` and `and` in terms of `foldr`:

```
product = foldr (*) 1
```

```
and      = foldr (&&) True
```

- For example, if we apply `foldr (*) 1` to list

```
1:(2:(3:(4:[]))) we obtain 1*(2*(3*(4*1)))
```

- another example of using `foldr` to define a function:

```
length = foldr (\_ v -> 1+v) 0
```

- In general, the behavior of `foldr` is:

```
foldr ( $\oplus$ ) v [x0,x1,...,xn] = x0 $\oplus$ (x1 $\oplus$ (... (xn $\oplus$ v) ...))
```

- essentially, `foldr` corresponds to the application to the elements of a list of an operator that associates to the *right* (hence, *foldr*)

# foldl

- (higher-order) function `foldl` is the dual of `foldr`: it corresponds to the application of an operator that associates to the *left*
- It corresponds to the pattern:  
 $f\ v\ [] = v$   
 $f\ v\ (x:xs) = f\ (v \oplus x)\ xs$ 
  - argument `v` works as an accumulator, which evolves by applying operator  $\oplus$  with the value of the head of the list
- This is captured by the following definition:  
 $foldl :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$   
 $foldl\ f\ v\ [] = v$   
 $foldl\ f\ v\ (x:xs) = foldl\ f\ (f\ v\ x)\ xs$
- In general, its behavior is:  
 $foldl\ (\oplus)\ v\ [x_0, x_1, \dots, x_n] = (\dots((v \oplus x_0) \oplus x_1) \dots) \oplus x_n$
- Examples of definitions using `foldl`:  
 $product = foldl\ (*)\ 1$   
 $and = foldl\ (\&\&)\ True$ 
  - `product` and `and` can be defined either with `foldr` or with `foldl` since they are both *associative*
- Another example:  
 $reverse = foldl\ (\backslash xs\ x \rightarrow x:xs)\ []$ 
  - for example the result of `reverse [1, 2, 3]` is  $3 : (2 : (1 : []))$

# Composition of functions

- The (higher-order) composition operator `.` takes two functions `f` and `g` as arguments, and applies `f` to the result obtained by applying `g` to its argument
  - the type of the argument of `g` must be the same as the type of the result of `f`
- In other words:  
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$   
 $f . g = \lambda x \rightarrow f (g x)$
- The composition operator can be used to make some definitions more concise: instead of  
`odd n = not (even n)`  
`twice f x = f (f x)`  
`sumsqreven xs = sum (map (^2) (filter even xs))`  
we can define:  
`odd = not . even`  
`twice = f.f`  
`sumsqreven = sum.map (^2).filter even`
  - the last definition works because composition is associative:  
 $f . (g . h) = (f . g) . h$
- the identity function `id = \x -> x` is the unit for `.`, i.e., for any function `f` we have `f.id = id.f`
- We exploit `id` and `foldr` to define a composition of lists of functions:  
`compose :: [a -> a] -> a -> a`  
`compose = foldr (.) id`

# Type declarations

- The simplest way to declare a type is as a synonym of an existing type:

```
type String = [Char]
type Pos = (Int, Int)
type Board = [Pos]
```

- Type parameters are admitted:

```
type Assoc k v = [(k, v)]
```

- Assoc represents, through a list, a lookup table of <key,value> pairs, where the keys are of type k, and the values of type v
- a function that, given a key and a lookup table, returns the value associated with the key:

```
find :: Eq k => k -> Assoc k v -> v
find k t = head[v | (k', v) <- t, k == k']
```

- In Haskell one can also declare entirely new types, through a *data* declaration, e.g.:

```
data Bool = False | True
```

- New types can be used in functions:

```
data Move = Left | Right | Up | Down
```

```
move :: Move -> Pos -> Pos
```

```
move Left (x,y) = (x-1, y)
```

```
move Right (x,y) = (x+1, y)
```

```
move Up (x,y) = (x, y+1)
```

```
move Down (x,y) = (x, y-1)
```

```
moves :: [Move] -> Pos -> Pos
```

```
moves [] p = p
```

```
moves (m : ms) p = moves ms (move m p)
```

## Type declarations with parameters

- Constructors in data declarations can have parameters:  
`data Shape = Circle Float | Rect Float Float`

- Examples of functions on type Shape:

```
square :: Float -> Shape
square n = Rect n n
```

```
area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rect x y) = x * y
```

- notice that we can use pattern matching with the constructors

- `Circle` and `Rect` are constructor functions: their results are values of type `Shape`

- `Circle 1.0` is a value onto itself, of type `Shape`, it is not evaluated any further

- data declarations can also have (type) parameters, e.g.:

```
data Maybe a = Nothing | Just a
```

- type `Maybe` represents optional values (i.e., values that may fail): if the value is undefined, then its value is `Nothing`, otherwise it is `Just v` (with `v` a value of type `a`)

- example of use of type `Maybe`: “safe” functions that, in case of errors, simply return a `Nothing` value:

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

```
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

# Recursive types

- Types defined through data declarations can be recursive:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

- functions on type Tree:

```
occurs :: Eq a => a -> Tree a -> Bool
```

```
occurs v (Leaf x) = v == x
```

```
occurs v (Node t1 x t2) = v == x ||  
                           occurs v t1 ||  
                           occurs v t2
```

```
flatten :: Tree a -> [a]
```

```
flatten (Leaf v) = [v]
```

```
flatten (Node t1 v t2) = flatten t1 ++ [v]  
                        ++ flatten t2
```

# Extended example: tautology checker

- (subset of) Propositional logic:
  - constants: False, True
  - propositional variables: A, B, C, ... Z
  - connectives:  $\neg$  (not),  $\wedge$  (and),  $\Rightarrow$  (imply)
  - parentheses: (, )
    - examples of formulas:
      - $A \wedge \neg A$
      - $(A \wedge B) \Rightarrow A$
      - $A \Rightarrow (A \wedge B)$
    - tautology: a formula that is true, no matter the values of the propositional variable
      - for example,  $(A \wedge B) \Rightarrow A$  is a tautology

- We declare a type for propositions:

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

- The value of a proposition depends on the values assigned to its variables; we use a type `Subst` to represent possible assignments of values to variables, through a lookup table:  
**type** Subst = Assoc Char Bool
  - example of assignment: [(‘A’, False), (‘B’, True)]

## Tautology checker (cont.)

- We define a function that, given a proposition and an assignment of values to variables, returns the value of the proposition in that assignment:

```
eval :: Subst -> Prop -> Bool
eval _ (Const b) = b
eval s (Var v) = find v s
eval s (Not p) = not (eval s p)
eval s (And p1 p2) = eval s p1 && eval s p2
eval s (Imply p1 p2) = eval s p1 <= eval s p2
```

- We define a function that returns all variables in a proposition:

```
vars :: Prop -> [Char]
vars (Const b) = []
vars (Var v) = [v]
vars (Not p) = vars p
vars (And p1 p2) = vars p1 ++ vars p2
vars (Imply p1 p2) = vars p1 ++ vars p2
```

- A function that removes duplicates from a list:

```
rmDups :: Eq a => [a] -> [a]
rmDups [] = []
rmDups (x:xs) = x : rmDups (filter (/= x) xs)
```



## Tautology checker (end)

- A function that, given a number  $n$ , returns all possible combinations of Booleans of length  $n$ :  

```
bools :: Int -> [[Bool]]
bools 0    = []
bools n+1 = (map (False:) bss)++(map (True:) bss)
            where bss = bools n
```

  - e.g., the result of `bools 2` is  
`[[False,False],[False,True],[True,False],[True,True]]`
- A function that generates all possible assignments to the variables of a proposition:  

```
subst :: Prop -> [Subst]
subst p = map (zip vs) (bools (length vs))
          where vs = rmdups (vars p)
```
- finally, the desired function:  

```
isTaut :: Prop -> Bool
isTaut p = and [eval s p | s <- (subst p)]
```
- In fact the result of  

```
isTaut (ImPLY (And (Var 'A') (Var 'B')) (Var 'A'))
```

is `True`