Lecture 1 — Functional Programming

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

CO2008 Functional Programming *(In a nutshell)*

- **Write a program** to add up the first `n` square numbers:

$$\texttt{sumofsqs n} = 0 + \texttt{1}^2 + \texttt{2}^2 + \ldots + \texttt{(n-1)}^2 + \texttt{n}^2$$

- Clear **Haskell** solution:

```
sumSquares :: Int -> Int
sumSquares 0 = 0
sumSquares n = n*n + sumSquares (n-1)
```

- Less clear, possibly incorrect **Java** solution:

```
public int sumSquares(int n){
private int s,i;
s=1; i=1;
while (i<n) { i = i+1;s = s+i*i; } }
```

- Key ideas (functions, types and recursion) lead to clear and succinct programs.

  It is good to have Haskell on your CV!

- **From Imperative to Functional Programming:**
  - What is imperative programming?
  - What is functional programming?
- **Key Ideas in Functional Programming:**
  - **Types:** Provide the data for our programs
  - **Functions:** These are our programs!
- **Advantages:**
  - Haskell code is typically short
  - Haskell code is close to the algorithms used

- **Problem:** write a program to add up the first `n` square numbers

$$\texttt{ssquares n} = 0 + \texttt{1}^2 + \texttt{2}^2 + \ldots + \texttt{(n-1)}^2 + \texttt{n}^2$$

- **Program:** We could write the following in Java

```
public int ssquares(int n){
private int s,i;
s=0; i=0;
      while (i<n) {i = i+1;s = s+i*i;}
}
```

- **Execution:** We may visualize running the program as follows

(Stack) Memory				(Stack) Memory
`s = ??`	$\longrightarrow$	`ssquares 4`	$\longrightarrow$	`s = 30`
`i = ??`	$\longrightarrow$		$\longrightarrow$	`i = 4`

- **Key Idea:** Imperative programs transform the memory

- **Functional Content:** What the program achieves
  - Programs take some input values and return an output value
  - `ssquares` takes a number and returns the sum of the squares up to and including that number
- **Implementational Content:** How the program does it
  - Imperative programs transform the memory using variable declarations and assignment statements
  - `ssquares` uses variables `i` and `s` to represent locations in memory. The program transforms the memory until `s` contains the correct number.

- **Motivation:** Problems arise as programs contain two aspects:
  - High-level algorithms and low-level implementational features
  - Humans are good at the former but not the latter
- **Idea:** The idea of functional programming is to
  - Concentrate on the functional (I/O) behaviour of programs
  - Leave memory management to the language implementation
- **Summary:** Functional languages are more abstract and avoid low level detail.

- **Types:** First we give the type of summing-squares

```
hssquares ::  Int -> Int
```

- **Functions:** Our program is a function

```
hssquares 0 = 0
hssquares n = n*n + hssquares (n-1)
```

- **Evaluation:** Run the program by expanding definitions

```
hssquares 3  ⇒  3*3+ hssquares 2
             ⇒  9 + (2*2 + hssquares 1)
             ⇒  9 + (2*2 + (1*1 + hssquares 0)
             ⇒  9 + (4 + (1+0))  ⇒  14
```

- **Comment:** No mention of memory in the code.

- **Motivation:** Recall from CO1003/4 that types model data.
- **Integers**: `Int` is the Haskell type $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- **String**: `String` is the Haskell type of lists of characters.
- **Complex Datatypes**: Can be made from the basic types, eg lists of integers.
- **Built in Operations ("Functions on types"):**
  - Arithmetic Operations: `+ * - div mod abs`
  - Ordering Operations: `> >= == /= <= <`

- Arithmetic Operations: `+ * - div mod abs`

- `div` is the (Euclidean) division on integers. It is the process of division of two integers, which produces a quotient and a remainder. `div` outputs the quotient. ``17 `div` 3 = 5``. (back quote `)

- `mod` is the *modulo* operation on integers. It finds the remainder after division of one number by another (sometimes called modulus). ``5 `mod` 3 = 2, 8 `mod` 2 = 0``.

- `abs` is the *absolute value* function on integers. It assigns x to a positive integer x and $-x$ to a negative integer x.
  `abs (-1) = abs (1) = 1`

- **Intuition:** Recall a function (or if you prefer method) $f : a \to b$ between sets associates to every input-value a unique output-value

$$x \in a \longrightarrow \boxed{\text{Function } f} \xrightarrow{?} y \in b$$

- **Example:** The *square* and *cube* functions are written

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

```
cube :: Int -> Int
cube x = x * square x
```

- **In General:** In Haskell, functions are defined as follows

```
⟨ function-name⟩ :: ⟨ input type⟩->⟨ output type⟩
⟨ function-name⟩ ⟨ variable⟩ = ⟨ expression⟩
```

- **Intuition:** A function f with n inputs is written `f::a1->...-> an-> a`

$$\begin{array}{lc} x_1 \in a_1 & \longrightarrow \\ x_2 \in a_2 & \longrightarrow \\ \quad\vdots & \vdots \\ x_n \in a_n & \longrightarrow \end{array} \boxed{\text{Function } f} \xrightarrow{?} y \in a$$

- **Example:** The "distance" between two integers

```
diff ::  Int -> Int -> Int
diff x y = abs (x - y)
```

- **In General:**

⟨ `function-name`⟩ :: ⟨ `type 1`⟩`->`...`->`⟨ `type n`⟩`->`⟨ `output-type`⟩

⟨ `function-name`⟩ ⟨ `variable 1`⟩...⟨ `variable n`⟩ $=$ ⟨ `expression`⟩

- **Motivation:** Get the *result/output* of a function by *applying* it to an *argument/input*
  - Write the function name followed by the input
- **In General:** Application is governed by the typing rule
  - If `f` is a function of type `a->b`, and `e` is an expression of type `a`,
  - then `f e` is the result of applying `f` to `e` and has type `b`
- **Key Idea:** Expressions are fragments of code built by applying functions to arguments.

```
square 4            square (3 + 1)     square 3 + 1
cube (square 2)     diff 6 7           square 2.2(?)
```

- **More Expressions:** Use **back quotes** ‘ to turn functions into infix operations and **brackets** () to turn infix operations into functions

```
5 * 4        (*) 5 4      mod 13 4       13 `mod` 4
5-(3*4)      (5-3)*4      7 >= (3*3)     5 * (-1)
```

- **Precedence:** Usual rules of precedence and bracketing apply

- **Example of Evaluation:**

```
cube(square3) ⇒ (square 3) * square (square 3)
              ⇒ (3*3) * ((square 3) * (square 3))
              ⇒ 9 * ((3*3) * (3*3))
              ⇒ (9 * (9*9))
              ⇒ 729
```

- The final outcome of an evalution is called a *value*

Summary — Comparing Functional and Imperative Programs

- **Difference 1:** Level of Abstraction
  - Imperative Programs include low level memory details
  - Functional Programs describe only high-level algorithms

- **Difference 2:** How execution works
  - Imperative Programming based upon memory transformation
  - Functional Programming based upon expression evaluation

- **Difference 3:** Type systems
  - Type systems play a key role in functional programming

- **Types:** A type is a collection of data values
- **Functions:** Transform inputs to outputs
  - We build complex expressions by defining functions and applying them to other expressions
  - The simplest (evaluated) expressions are (data) values
- **Evaluation:** Calculates the result of applying a function to an input
  - Expressions can be evaluated to values by hand or by GHCi (the interpreter of the Glasgow Haskell Compiler)
- **Now:** Go and look at the first practical!

Lecture 2 — More Types and Functions

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **New Types:** Today we shall learn about the following types
  - The type of booleans: `Bool`
  - The type of characters: `Char`
  - The type of strings: `String`
  - The type of fractions: `Float`
- **New Functions and Expressions:** And also about the following functions
  - Conditional expressions and guarded functions
  - Error handling and local declarations

- **Values of** `Bool` **:** Contains two values — `True, False`

- **Logical Operations:** Various built in functions

```
&&  ::  Bool -> Bool -> Bool
||  ::  Bool -> Bool -> Bool
not ::  Bool -> Bool
```

- **Example:** Define the exclusive-OR function which takes as input two booleans and returns `True` just in case they are different

```
exOr ::  Bool -> Bool -> Bool
```

- **Example:** Maximum of two numbers

```
maxi ::  Int -> Int -> Int
maxi n m = if n>=m then n else m
```

- **Example:** Testing if an integer is `0`

```
isZero ::  Int -> Bool
isZero x = if (x == 0) then True else False
```

- **Conditionals:** A *conditional expression* has the form

```
if b then e1 else e2
```

  where

  – `b` is an expression of type `Bool`

  – `e1` and `e2` are expressions with the same type

- **Example:** `doubleMax` returns double the maximum of its inputs

```
doubleMax ::  Int -> Int -> Int
doubleMax x y
    | x >= y  =  2*x
    | x < y   =  2*y
```

- **Definition:** A guarded function is of the form

```
⟨function-name⟩ :: ⟨type 1⟩->...->⟨type n⟩ ->⟨output type⟩

⟨function-name⟩ ⟨var 1⟩...⟨var n⟩
      | ⟨guard 1⟩  =  ⟨expression 1⟩
      | ...        =  ...
      | ⟨guard m⟩  =  ⟨expression m⟩
```

where `⟨guard 1⟩`,...,`⟨guard m⟩` :: `Bool`

The `Char` *type*

- **Elements of** `Char` **:** Letters, digits and special characters

- **Forming elements of** `Char` **:** Single right quotes (apostrophes) form characters:

```
'd' :: Char    '3' :: Char
```

- **Functions:** Characters have codes and conversion functions

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

  Erratum: use *import Data.Char* or *:module Data.Char* to import these goodies!

- **Examples:** Try them out! (Also try isDigit and digitToInt)

```
offset :: Int
offset = ord 'A' - ord 'a'

capitalize :: Char -> Char
capitalize ch = chr (ord ch + offset)
```

```
isLower :: Char -> Bool
isLower x = ('a' <= x) && (x <= 'z')
```

- **Elements of `String`:** Lists of characters

- **Forming elements of `String`:** Double quotes form strings

  `"Newcastle Utd"` `"1a"`

- **Special Strings:** Newline and Tab characters

  `"Super \n Alan"` `"1\t2\t3"` `putStr( "Super \n Alan")`

- **Combining Strings:** Strings can be combined by `++`

  `"Super " ++ "Alan " ++ "Shearer" = "Super Alan Shearer"`

- **Example:** `duplicate` gives two copies of a string

- **Elements of `Float` :** Contains decimals, eg `-21.3, 23.1e-2`

- **Built in Functions:** Arithmetic, Ordering, Trigonometric

- **Conversions:** Functions between `Int` and `String`

```
ceiling, floor, round   ::  Float -> Int
fromIntegral            ::  Int -> Float
show                    ::  Float -> String
read                    ::  String -> Float
```

- **Overloading:** Overloading is when values/functions belong to several types

```
2 ::  Int       show ::  Int -> String
2 ::  Float     show ::  Float -> String
```

- **Motivation:** Informative error messages for run-time errors

- **Example:** Dividing by zero will cause a run-time error

```
myDiv :: Float -> Float -> Float
myDiv x y = x/y
```

- **Solution:** Use an `error` message in a guarded definition

```
myDiv :: Float -> Float -> Float
myDiv x y
    | y /= 0      = x/y
    | otherwise   = error "Attempt to divide by 0"
```

- **Execution:** If we try to divide by `0` we get

```
Prelude> mydiv 5 0
Program error:  Attempt to divide by 0
```

- **Motivation:** Functions will often depend on other functions
- **Example :** Summing the squares of two numbers

```
sq ::  Int -> Int
sq x = x * x

sumSquares ::  Int -> Int -> Int
sumSquares x y = sq x + sq y
```

- **Problem:** Such definitions clutter the top-level environment
- **Answer:** Local definitions allow auxiliary functions

```
sumSquares2 ::  Int -> Int -> Int
sumSquares2 x y =  sq x + sq y
                   where sq z = z * z
```

- **Quadratic Equations:** The solutions of $ax^2 + bx + c = 0$ are

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- **Types:** Our program will have type

```
roots ::  Float -> Float -> Float -> String
```

- **Guards:** There are 4 cases to check so use a guarded definition

```
roots a b c
   | a == 0           =   ....
   | b*b-4*a*c < 0    =   ....
   | b*b-4*a*c == 0   =   ....
   | otherwise        =   ....
```

- **Code:** Now we can add in the answers

```
roots a b c
    | a == 0            = error "Not a quadratic eqn"
    | b*b-4*a*c < 0     = "No roots."
    | b*b-4*a*c == 0    = "One root: " ++ show (-b/2*a)
    | otherwise         = "Two roots: " ++
                          show ((-b + sqrt (b*b-4*a*c))/2*a)
                          ++ "and" ++
                          show ((-b - sqrt (b*b-4*a*c))/2*a)
```

- **Problem:** This program uses several expressions repeatedly
  - Being cluttered, the program is hard to read
  - Similarly the program is hard to understand
  - Repeated evaluation of the same expression is inefficient

- **Local decs:** Expressions used repeatedly are made local

```
roots a b c
   | a == 0      = error "Not a quadratic eqn"
   | disc < 0    = "No roots."
   | disc == 0   = "One root: " ++ show centre
   | otherwise   = "Two roots: " ++
                    show (centre + offset) ++ "and" ++
                    show (centre - offset)
     where
                 disc = b*b-4*a*c
                 offset = (sqrt disc) / 2*a
                 centre = -b/2*a
```

- **Types:** We have learned about Haskell's basic types. For each type we learned
  - Its basic values (elements)
  - Its built in functions
- **Expressions:** How to write expressions involving
  - Conditional expressions and Guarded functions
  - Error Handling and Local Declarations

Lecture 3 — New Types from Old

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Building New Types:** Today we will learn about the following compound types
  - Pairs
  - Tuples
  - Type Synonyms
- **Describing Types:** As with basic types, for each type we want to know
  - What are the values of the type
  - What expressions can we write and how to evaluate them

From simple data values to complex data values

- **Motivation:** Data for programs modelled by values of a type
- **Problem:** Single values in basic types too simple for real data
- **Example:** A point on a plane can be specified by
  - A number for the x-coordinate and another for the y-coordinate
- **Example:** A person's complete name could be specified by
  - A string for the first name and another for the second name
- **Example:** The performance of a football team could be
  - A string for the team and a number for the points

- **Examples:** For instance
  - The expression `(5,3)` has type `(Int, Int)`
  - The name `("Alan","Shearer")` has type `(String, String)`
  - The performance `("Newcastle", 22)` has type `(String,Int)`
- **Question:** What are the values of a pair type?
- **Answer:** A pair type contains pairs of values, ie
  - If `e1` has type `a` and `e2` has type `b`
  - Then `(e1,e2)` has type `(a,b)`

- **Types:** Pair types can be used as input and/or output types

- **Examples:** The built in functions `fst` and `snd` are vital

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

winUpdate ::  (String,Int) -> (String,Int)
winUpdate (x,y) = (x,y+3)

movePoint ::  Int -> Int -> (Int,Int) -> (Int,Int)
movePoint m n (x,y) = (x+m,y+n)
```

- **Key Idea:** If input is a pair-type, use $(\langle \texttt{var1} \rangle, \langle \texttt{var2} \rangle)$ in definition

- **Key Idea:** If output is a pair-type, result is often $(\langle \texttt{exp1} \rangle, \langle \texttt{exp2} \rangle)$

- **Motivation:** Some data consists of more than two parts

- **Example:** Person on a mailing list

  – Specified by name, telephone number, and age

  – A person p on the list can have type `(String, Int, Int)`

- **Idea:** Generalise pairs of types to collections of types

- **Type Rule:** Given types `a1,...,an`, then `(a1,...,an)` is a type

- **Expression Formation:** Given expressions `e1::a1, ..., en::an`, then

  `(e1,...,en)` :: `(a1,...,an)`

- **Example 1:** Write a function to test if a customer is an adult

```
isAdult :: (String,Int,Int) -> Bool

isAdult (name, tel, age) = (age >= 18)
```

- **Example 2:** Write a function to update the telephone number

```
updateMove :: (String,Int,Int) -> Int -> (String,Int,Int)
```

- **Example 3:** Write a function to update age after a birthday

```
updateAge :: (String,Int,Int) -> (String,Int,Int)
```

- **Definition:** Functions now have the form

```
<function-name> :: <type 1> -> ... -> <type n> -> <out-type>

<function-name> <pat 1> ... <pat n> = <exp n>
```

- **Patterns:** Patterns are
  - Variables `x`: Use for any type
  - Constants `0, True, ‘‘cherry’’`: Definition by cases
  - Tuples `(x,..,z)`: If the argument has a tuple-type
  - Wildcards _: If the output doesn’t use the input

- **In general:** Use several lines and mix patterns.

- **Example:** Using values and wildcards

```
isZero :: Int -> Bool
isZero 0 = True
isZero _ = False
```

- **Example:** Using tuples and multiple arguments

```
expand :: Int -> (Int,Int) -> (Int,Int,Int)
expand n (x,y) = (n, n*x, n*y)
```

- **Example:** Days in the month

```
days :: String -> Int -> Int
days "January"  year  = 31
days "February" year  = if isLeap year then 29 else 28
days "March"    year  = 31
 .....
```

- **Motivation:** More descriptive names for particular types.
- **Definition:** Type synonyms are declared with the keyword `type`.

```
type Team = String
type Goals = Int
type Match = ((Team,Goals), (Team,Goals))

numu ::  Match
numu = (("Newcastle", 4), ("Manchester Utd", 3))
```

- **Functions:** Types of functions are more descriptive, same code

```
homeTeam ::  Match -> Team
totalGoals ::  Match -> Goals
```

- **Tuples:** Collections of data from other types
- **Pairs:** Pairs, triples etc are examples of tuples
- **Type synonyms:** Make programs easier to understand
- **Pattern Matching:** General description of functions covering definition by cases, tuples etc.
- **Pitfall!** What is the difference between

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

addTwo ::  Int -> Int -> Int
addTwo x y = x + y
```

Lecture 4 — List Types

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Lists:** What are lists?
  - Forming list types
  - Forming elements of list types
- **Functions over lists:** Some old friends, some new friends
  - Functions from CO1003/4: `cons, append, head, tail`
  - Some new functions: `map, filter`
- **Clarity:** Unlike Java, Haskell treatment of lists is clear
  - No list iterators!

- **Example 1:** `[3, 5, 14] :: [Int]` and `[3, 4+1, double 7] :: [Int]`
- **Example 3:** `['d','t','g']` :: `[Char]`
- **Example 4:** `[['d'], ['d','t'], ['d','t','g']] :: [[Char]]`
- **Example 5:** `[double, square, cube] :: [Int -> Int]`
- **Empty List:** The empty list is `[]` and belongs to all list types
- **List Expressions:** Lists are written using square brackets `[...]`
  - If `e1`, . . . , `en` are expressions of type `a`
  - Then `[e1, ..., en]` is an expression of type `[a]`

- **Cons:** The cons function `:` adds an element to a list

```
: :: a -> [a] -> [a]

1       : [2,3,4]    = [1,2,3,4]
addone  : [square]   = [addone, square]
'a'     : ['b', 'z'] = ['a', 'b', 'z']
```

- **Append:** Append joins two lists together

```
++ ::  [a] -> [a] -> [a]

[True, True] ++ [False]  = [True, True, False]
[1,2] ++ ([3] ++ [4,5])  = [1,2,3,4,5]
([1,2] ++ [3]) ++ [4,5]  = [1,2,3,4,5]
[] ++ [54.6, 67.5]       = [54.6, 67.5]
[6,5] ++ (4 :  [7,3])    = [6,5,4,7,3]
```

- **Head and Tail:** Head gives the first element of a list, tail the remainder

```
head [double, square] = double
head ([5,6]++[6,7])    = 5

tail [double, square] = [square]
tail ([5,6]++[6,7])    = [6,6,7]
```

- **Length and Sum:** The length of a list and the sum of a list of integers

```
length (tail [1,2,3]) = 2
sum [1+4,8,45] = 58
```

- **Sequences:** The list of integers from `1` to `n` is written

```
[1 .. n]
```

`String` *is actually a list type*

- **Note:** The type `String` is a type synonym for `[Char]`.

- Hence we can use list notation on strings: eg.

-

```
head "abcdef" = 'a'

tail "abcdef" = "bcdef"

"Fer"++"-Jan" = "Fer-Jan"

"abcd"= 'a':"bcd"
```

- **Map:** Map is a function which has two inputs.
  - The first input is a function eg `f`
  - The second is a list eg `[e1,e1,e3]`

  The output is the list obtained by applying the function to every element of the input list eg `[f e1, f e2, f e3]`

- **Filter:** Filter is a function which has two inputs.
  - The first is a *test*, ie a function returning a `Bool`.
  - The second is a list

  The output is the list of elements of the input list which the function maps to `True`, ie those elements which pass the test.

- **Even Numbers:** The even numbers less than or equal to `n`
  - `evens::Int->[Int]`

- **Solution 1** — Using `filter`.

```
evens2 ::  Int -> [Int]
evens2 n = filter isEven [1 ..  n]
               where isEven x = (x `mod` 2 == 0)
```

- **Solution 2** — Using `map`

- **Types:** We have looked at list types
  - What list types and list expressions looks like
  - What built in functions are available
- **New Functions:**
  - Map: Apply a function to every member of a list
  - Filter: Delete those that don't satisfy a property or test
- **Algorithms:** Develop an algorithm by asking
  - Can I solve this problem by applying a function to every member of a list or by deleting certain elements.

Lecture 5 — List Comprehensions

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Recall Map:** Map is a function which has two inputs.

```
map add2 [2, 5, 6] = [4, 7, 8]
```

- **Recall Filter:** Filter is a function which has two inputs.

```
filter isEven [2, 3, 4, 5, 6, 7] = [2, 4, 6]
```

  Both Map and Filter essentially construct new lists from old lists.

- **List comprehension:** An alternative way of constructing lists
  - Definition of list comprehension
  - Comparison with `map` and `filter`

- **Example 1:** If `xs = [2,4,7]` then

```
[ 2*x | x <- xs ] = [4,8,14]
```

- **Example 2:** If `isEven :: Int->Bool` tests for even-ness

```
[ isEven x | x <- xs ] = [True,True,False]
```

- **In General:** (Simple) list comprehensions are of the form

  [⟨`exp`⟩ | ⟨`variable`⟩ `<-` ⟨`list-exp`⟩]

- **Evaluation:** The meaning of a list comprehension is
  - Take each element of `list-exp`, evaluate the expression `exp` for each element and return the results in a list.

- **Example 1:** A function which doubles a list's elements

```
double :: [Int] -> [Int]
```

- **Example 2:** A function which tags an integer with its evenness

```
isEvenList :: [Int] -> [(Int,Bool)]
```

- **Example 3:** A function to add pairs of numbers

```
addpairs :: [(Int,Int)] -> [Int]
```

- **In general:** `map f l = [f x | x <- l]`

- **Intuition:** List Comprehension can also select elements from a list

- **Example:** We can select the even numbers in a list

```
[ e | e <- l, isEven e]
```

- **Example:** Selecting names beginning with `A`

```
names ::  [String] -> [String]
names l = [ e | e <- l , head e == 'A' ]
```

- **Example:** Combining selection and applying functions

```
doubleEven ::  [Int] -> [Int]
doubleEven l =[ 2*e | e <- l , isEven e ]
```

- **In General:** These list comprehensions are of the form

  [⟨`exp`⟩ | ⟨`variable`⟩ `<-` ⟨`list-exp`⟩ , ⟨`test`⟩]

- **Example:** In fact, we can use several tests — if `l = [2,5,8,10]`

```
[ 2*e | e <- l , isEven e , e>3 ] = [16,20]
```

- **Key Example:** Cartesian product of two lists is a list of all pairs, such that for each pair, the first component comes from the first list and the second component from the second list.

```
[ (x,y) | x<-[1,2,3], y<-['a','b','c'] ]
                                = [(1,'a'), (1,'b') ...  ]

league ::  [Team]
games = [ (t1,t2) | t1 <- league, t2 <- league, t1 /= t2]
```

- **Problem:** Given a list remove all duplicate entries

- **Algorithm:** Given a list,
  - Keep first element
  - Delete all occurrences of the first element
  - Repeat the process on the tail

- **Code:**

- **List Types:** We have looked at list types
  - What list types and list expressions looks like
  - What built in functions are available

- **List comprehensions:** Like `filter` and `map`. They allow us to
  - Select elements of a list
  - Delete those that dont satisfy certain properties
  - Apply a function to each element of the remainder

Lecture 6 — Recursion over Natural Numbers

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Recursion:** General features of recursion
  - What is a recursive function?
  - How do we write recursive functions?
  - How do we evaluate recursive functions?

- **Recursion over Natural Numbers:** Special features
  - How can we guarantee evaluation works?
  - Recursion using patterns.
  - Avoiding negative input.

- **Example:** Adding up the first n squares

$$\texttt{hssquares n} = \texttt{1}^2 \ + \ \ldots \ + \ \texttt{(n-1)}^2 \ + \ \texttt{n}^2$$

- **Types:** First we give the type of summing-squares

```
hssquares ::  Int -> Int
```

- **Definitions:** Our program is a function

```
hssquares 0 = 0
hssquares n = n*n + hssquares (n-1)
```

- **Key Idea:** `hssquares` is recursive as its definition contains `hssquares` in a right-hand side – the function name "recurs".

- **Definition:** A function is *recursive* if the name recurs in its definition.

- **Intuition:** You will have seen recursion in action before
  - Imperative procedures which call themselves
  - Divide-and-conquer algorithms

- **Why Recursion:** Recursive definitions tend to be
  - Shorter, more understandable and easier to prove correct
  - Compare with a non-recursive solution

```
nrssquares n = n * (n+0.5) * (n+1)/3
```

- **Example 1:** Let's calculate `hssquares 4`

```
hssquares 4  ⇒  4*4 + hssquares 3
             ⇒  16 + (3*3 + hssquares 2)
                ...
             ⇒  16 + (9 + ..  (1 + hssquares 0))
             ⇒  16 + (9 + ...  (1 + 0))      ⇒ 30
```

- **Example 2:** Here is a non-terminating function

```
mydouble n  =  n + mydouble (n/2)
mydouble 4  ⇒  4 + mydouble 2
            ⇒  4 + 2 + mydouble 1
            ⇒  4 + 2 + 1 + mydouble 0.5 ⇒ ......
```

- **Question:** Will evaluation stop?

Problems with Recursion

- **Questions:** There are some outstanding problems

  1. Is `hssquares` defined for every number?
  2. Does an evaluation of a recursive function always terminate?
  3. What happens if `hssquares` is applied to a negative number?
  4. Are these recursive definitions sensible: `f n = f n`, `g n = g (n+1)`

- **Answers:** Here are the answers

  1. Yes: The variable pattern matches every input.
  2. Not always: See examples.
  3. Trouble: Evaluation doesn't terminate.
  4. No: Why not?

- **Motivation:** Restrict definitions to get better behaviour

- **Idea:** Many functions defined by three cases
  - A non-recursive call selected by the pattern `0`
  - A recursive call selected by the pattern `n`
  - The error case deals with negative input

- **Example** Our program now looks like

```
hssquares2 0  =  0
hssquares2 n
 | n>0        =  n*n + hssquares (n-1)
 | n<0        =  if n < 0 error ``Negative input''
```

- **Example 1:** `star` uses recursion over `Int` to return a string

```
star    :: Int -> String
star 0  =  []
star n
 | n>0  =  '*' :  star (n-1)
 | n<0  =  error ``Negative input''
```

- **Example 2:** `power` is recursive in its second argument

```
power      :: Float -> Int -> Float
power x 0  =  1
power x n
 | n>0     =  x * power x (n-1)
 | n<0     =  error ``Negative input''
```

- **In General:** Use the following style of definition

```
⟨function-name⟩ 0  =  ⟨exp 1⟩
⟨function-name⟩ n
 | n>0             =  ⟨exp 2⟩
 | n<0             =  error⟨message⟩
```

  where

`⟨exp 1⟩`	does not contain	`⟨function-name⟩`
`⟨exp 2⟩`	may contain	`⟨function-name⟩` applied to `n-1`

- **Evaluation:** Termination guaranteed!
  - If the input evaluates to `0`, evaluate `⟨exp 1⟩`
  - If not, if the input is greater than `0`, evaluate `⟨exp 2⟩`
  - If neither, return the error message

- **Problem:** Produce a table for `perf :: Int -> (String, Int)` where `perf 1 = ("Arsenal",4)` etc.

- **Stage 1:** We need some headings and then the actual table

```
printTable :: Int -> IO()

printTable numberTeams = putStr(header ++ rows numberTeams)
                         where
                         header = "Team\t Points\n"
```

- **Stage 2:** Convert each "row" to a string, recursively.

```
rows     :: Int -> String
rows 0   =  .....
rows n
 | n >0  =  .....
 | n <0  =  .....
```

- **Base Case:** If we want no entries, then just return `[]`

```
rows 0 = []
```

- **Recursive Case:** Convert `n`-rows by
  - recursively converting the first `n-1`-rows, and
  - adding on the `n`-th row

- **Code:** Code for the recursive call

```
perf :: Int -> (String,Int)
perf 1 = ("Arsenal",4)
perf 2 = ("Notts",5)
perf 3 = ("Chelsea",7)
perf n = error "perf out of range"

rows :: Int -> String
rows n
  | n=0 = []
  | n>1 = rows (n-1) ++ fst(perf n) ++ "\t\t "
                 ++  show(snd(perf n)) ++ "\n"
  | n < 0 = error"rows out of range"

printTable :: Int -> IO()
printTable numberTeams = putStr(header ++ rows numberTeams)
                         where
                         header = "Team\t\t Points\n"
```

- **Recursion:** Allows new functions to be written.
  - Advantages: Clarity, brevity, tractability
  - Disadvantages: Evaluation may not stop
- **Primitive Recursion:** Avoids bad behaviour of some recursive functions
  - The value at `0` is non-recursive
  - Each recursive call uses a smaller input
  - An error-clause catches negative inputs
- **Algorithm:** Ask yourself, what needs to be done to the recursive call to get the answer.

Lecture 7 — Recursion over Lists

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Lists:** Another look at lists
  - Lists are a recursive structure
  - Every list can be formed by `[]` and `:`
- **List Recursion:** Primitive recursion for Lists
  - How do we write primitive recursive functions
  - Examples — `++, length, head, tail, take, drop, zip`
- **Avoiding Recursion?:** List comprehensions revisited

- **Question:** This lecture is about the following question
  - We know what a recursive function over `Int` is
  - What is a recursive function over lists?

- **Answer:** In general, the answer is the same as before
  - A recursive function mentions itself in its definition
  - Evaluating the function may reintroduce the function
  - Hopefully this will stop at the answer

- **Recall:** The two basic operations concerning lists
  - The empty list `[]`
  - The cons operator `(:)  ::  a -> [a] -> [a]`
- **Key Idea:** Every list is either empty, or of the form `x:xs`

  `[2,3,7] = 2:3:7:[]  [True, False] = True:False:[]`
- **Recursion:** Define recursive functions using the scheme
  - Non-recursive call: Define the function on the empty list `[]`
  - Recursive call: Define the function on `(x:xs)` by using the function only on `xs`

- **Example 1:** Doubling every element of an integer list

```
double ::  [Int] -> [Int]
double [] = []
double (x:xs) = (2*x) :  double xs
```

- **Example 2:** Selecting the even members of a list

```
onlyEvens          ::  [Int] -> [Int]
onlyEvens []       =   []
onlyEvens (x:xs)   =   if isEven x then x:rest else rest
                       where rest = onlyEvens xs
```

- **Example 3:** Flattening some lists

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

- **Definition:** Primitive Recursive List Functions are given by

$$\begin{array}{lll} \langle\texttt{function-name}\rangle\ \texttt{[]} & = & \langle\texttt{expression 1}\rangle \\ \langle\texttt{function-name}\rangle\ \texttt{(x:xs)} & = & \langle\texttt{expression 2}\rangle \end{array}$$

where

$\langle$`expression 1`$\rangle$	does not contain	$\langle$`function-name`$\rangle$
$\langle$`expression 2`$\rangle$	may contain expressions	$\langle$`function-name`$\rangle$ `xs`

- **Compare:** Very similar to recursion over `Int`

$$\begin{array}{lll} \langle\texttt{function-name}\rangle\ \texttt{0} & = & \langle\texttt{expression 1}\rangle \\ \langle\texttt{function-name}\rangle\ \texttt{n} & = & \langle\texttt{expression 2}\rangle \end{array}$$

where

$\langle$`expression 1`$\rangle$	does not contain	$\langle$`function-name`$\rangle$
$\langle$`expression 2`$\rangle$	may contain expressions	$\langle$`function-name`$\rangle$ `(n-1)`

- **Example 4:** Append is defined recursively

```
append :: [a] -> [a] -> [a]
```

- **Example 5:** Testing if an integer is an element of a list

```
member :: Int -> [Int] -> Bool
```

- **Example 6:** Reversing a list

```
reverse :: [a] -> [a]
```

- **Mapping:** Applying a function to every member of the list

```
double [2,3,72,1]  =  [2*2, 2*3, 2*72, 2*1]
isEven [2,3,72,1]  =  [True, False, True, False]
```

- **Filtering:** Selecting particular elements

```
onlyEvens [2,3,72,1] = [2,72]
```

- **Taking Lists Apart:** `head, tail, take, drop`

- **Combining Lists:** `zip`

- **Folding:** Combining the elements of the list

```
sumList [2,3,7,2,1]  =  2 + 3 + 7 + 2 + 1
```

- **Recall:** List comprehensions look like

  [⟨`exp`⟩ | ⟨`variable`⟩ `<-` ⟨`list-exp`⟩ , ⟨`test`⟩]

- **Intuition:** Roughly speaking this means
  - Take each element of the list ⟨`list-exp`⟩
  - Check they satisfy ⟨`test`⟩
  - Form a list by applying ⟨`exp`⟩ to those that do

- **Idea:** Equivalent to filtering and then mapping. As these are recursive, so are list comprehensions although the recursion is hidden

- **List Recursion:** Lists are recursive data structures
  - Hence, functions over lists tend to be recursive
  - But, as before, general recursion is badly behaved

- **Primitive List Recursion:** Similar to natural numbers
  - A non-recursive call using the pattern `[]`
  - A recursive call using the pattern `(x:xs)`

- **List comprehension:** An alternative way of doing some recursion

Lecture 8 — More Complex Recursion

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Problem:** Our restrictions on recursive functions are too severe
- **Solution:** New definitional formats which keep termination
  - Using new patterns
  - Generalising the recursion scheme
- **Examples:** Applications to integers and lists
- **Sorting Algorithms:** What is a sorting algorithm?
  - Insertion Sort, Quicksort and Mergesort

- **Recall:** Our primitive recursive functions follow the scheme
  - **Base Case:** Define the function non-recursively at `0`
  - **Inductive Case:** Define the function at positive `n` in terms of the function at `(n-1)`

$$
\begin{array}{l}
\langle\texttt{function-name}\rangle\ \texttt{0=}\ \langle\texttt{exp 1}\rangle \\
\langle\texttt{function-name}\rangle\ \texttt{n} \\
\quad \mid\ \texttt{n>0}\ \ \texttt{=}\ \ \langle\texttt{exp 2}\rangle \\
\quad \mid\ \texttt{n<0}\ \ \texttt{=}\ \ \texttt{error}\langle\texttt{message}\rangle
\end{array}
$$

  where

⟨`expression 1`⟩	does not contain	⟨`function-name`⟩
⟨`expression 2`⟩	may contain	⟨`function-name`⟩ applied to `n`

- But some functions do not fit this scheme and require more complex patterns

- **Example:** The first Fibonacci numbers are `0,1`. For each subsequent Fibonacci number, add the previous two together

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

- **Problem:** The following does not terminate on input 1

```
fib 0 = 0
fib n = fib (n-1) + fib (n-2)
```

- **Solution:** Second base case!

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

To be more precise add error message in case $(n < 0)$.
Otherwise the funtion will not terminate at negative input.

- **Recall:** Our primitive recursive functions follow the pattern
  - **Base Case:** Defines the function at `[]` non-recursively
  - **Inductive Case:** Defines the function at `(x:xs)` in terms of the function at `xs`

```
⟨function-name⟩ []     = ⟨exp 1⟩
⟨function-name⟩ (x:xs) = ⟨exp 2⟩
```

where

⟨`expression 1`⟩	does not contain	⟨`function-name`⟩
⟨`expression 2`⟩	may contain	⟨`function-name`⟩ applied to `xs`

- **Motivation:** As with integers, some functions don't fit this shape

- **Recall:** With integers, we used more general patterns.

- **Idea:** Use `(x:(y:xs))` pattern to access first two elements

- **Example:** We want a function to delete every second element

```
delete [2,3,5,7,9,5,7] = [2,5,9,7]
```

- **Solution:** Here is the code

```
delete ::  [a] -> [a]
delete [] = []
delete [x] = [x]
delete (x:(y:xs)) = x :  delete xs
```

- **Example:** To delete every third element use pattern `(x:(y:(z:xs)))`

- **Example 1:** Summing pairs in a list of pairs

```
sumPairs ::  [(Int,Int)] -> Int
```

- **Example 2:** Unzipping lists `unZip ::  [(a,b)] -> ([a],[b])`

- **Problem:** A sorting algorithm rearranges a list in order

```
sort [2,7,13,5,0,4] = [0,2,4,5,7,13]
```

- **Recursion:** Such algorithms usually recursively sort a smaller list

- **Insertsort Alg:** To sort a list, sort the tail recursively, and then insert the head

- **Code:**

```
insort ::  [Int] -> [Int]
insort [] = []
insort (x:xs) = insert x (insort xs)
```

where `insert` puts the number `x` in the correct place

- **Patterns:** Insert takes two arguments, number and list
  - The recursion for `insert` doesn't depend on the number
  - The recursion for `insert` does depend on whether the list is empty or not — use the `[]` and `(x:xs)` patterns

- **Code:** Here is the final code

```
insert ::  Int -> [Int] -> [Int]
insert n [] = [n]
insert n (x:xs)
    | n <= x     = n:x:xs
    | otherwise  = x:(insert n xs)
```

- **Quicksort Alg:** Given a list `l` and a number `n` in the list

$$\begin{aligned} \text{sort } \mathtt{l} \quad = \quad & \text{sort those elements less than } \mathtt{n} \ ++ \\ & \text{number of occurrences of } \mathtt{n} \ ++ \\ & \text{sort those elements greater than } \mathtt{n} \end{aligned}$$

- **Code:** The algorithm may be coded

```
qsort ::  [Int] -> [Int]
qsort [] = []
qsort (x:xs)  =   qsort (less x xs) ++
                  occs x (x:xs) ++
                  qsort (more x xs)
```

where `less, occs, more` are auxiliary functions

Alternative: write `x: occs x xs` instead of `occs x (x:xs)`

- **Problem:** The auxiliary functions can be specified
  - `less` takes a number and a list and returns those elements of the list less than the number
  - `occs` takes a number and a list and returns the occurrences of the number in the list
  - `more` takes a number and a list and returns those elements of the list more than the number

- **Code:** Using list comprehensions gives short code

```
less, occs, more ::  Int -> [Int] -> [Int]
less n xs = [x | x <- xs, x < n]
occs n xs = [x | x <- xs, x == n]
more n xs = [x | x <- xs, x > n]
```

- **Mergesort Alg:** Split the list in half, recursively sort each half and merge the results

- **Code:** Overall function reflects the algorithm

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort ys) (msort ws)
           where (ys,ws) = (take l xs, drop l xs)
                           where l = length xs `div` 2
```

  where merge combines two sorted lists

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) = if x<y then x :  merge xs (y:ys)
                              else y :  merge (x:xs) ys
```

- **Recursion Schemes:** We've generalised the recursion schemes to allow more functions to be written
  - More general patterns
  - Recursive calls to ANY smaller value
- **Examples:** Applied them to recursion over integers and lists
- **Sorting Algorithms:** We've put these ideas into practice by defining three sorting algorithms
  - Insertion Sort
  - QuickSort
  - MergeSort

Lecture 8.5 — Interlude JavaScript

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- Clickable links to background information
  - `https://en.wikipedia.org/wiki/Node.js`
  - `https://en.wikipedia.org/wiki/JavaScript`
- The official JavaScript language specification (June 2017)

  `https://www.ecma-international.org/ecma-262/8.0/`
- pointers to syntax and language definition
  - `https://javascript.info/`
  - `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
  - `https://www.w3schools.com/js/`

- Several methods:
  - Use the developer console of Chrome browser (press F12) Try to type `alert("I'm JavaScript!");` in the console
  - From terminal (linux): `js file.js`
  - From within an html file `https://www.w3schools.com/js/`

- ```
  function fibonacci(num){
  _ var a = 1, b = 0, temp;
  _ while (num >= 0){
  _ _ temp = a;
  _ _ a = a + b;
  _ _ b = temp;
  _ _ num--;
  _ }
  _ return b;
  }
  ```
  ```

- ```
  function fibonacci(num) {
  _ if (num <= 1) return 1;
  _ return fibonacci(num - 1) + fibonacci(num - 2);
  }
  ```

  Note the hidden double base case!
  ```

- Become familiar with the following
  - `JavaScript Fundamentals`
  - `Functions` in `JavaScript Fundamentals`
  - `Arrays` in `Data Types`
  - `Advanced working with functions` in `The JavaScript language`

    This explains difference between iterative thinking (for loops) and recursive thinking.

    Recommendation: "Recursion gives usually shorter code."

- `Arrays` in `Data Types`

- Beware of differences in seemingly analogous array concepts in Haskell - JavaScript

  – H: all elements of array must have same type; J: an array can store elements of any type.

  – nth element of list: H `list !! n`; J `list[n]`

  – length of list: H `length list`; J `list.length`

  – head of list: H `head list`; J `list.shift()`

  – add new element to list: H `a:list`; J `list.unshift(a)` (use alert to show the change)

- last of list: H `last list`; J `list.unshift()`

- Challenge: try to code the Phonebook question of Worksheet2 in JavaScript!

- Challenge: try to code the sorting algorithms in JavaScript!

Lecture 9 — Higher Order Functions

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Motivation:** Why do we want higher order functions

- **Definition:** What is a higher order function

- **Examples:**

  – Mapping: Applying a function to every member of a list

  – Filtering: Selecting elements of a list satisfying a property

- **Application:** Higher order sorting algorithms

- **Example 1:** A function to double the elements of a list

```
doubleList ::  [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = (2*x) :  doubleList xs
```

- **Example 2:** A function to square the elements of a list

```
squareList ::  [Int] -> [Int]
squareList [] = []
squareList (x:xs) = (x*x) :  squareList xs
```

- **Example 3:** A function to increment the elements of a list

```
incList ::  [Int] -> [Int]
incList [] = []
incList (x:xs) = (x+1) :  incList xs
```

- **Problem:** Three separate definitions despite a clear pattern

- **Intuition:** Examples apply a function to each member of a list

```
function ::  Int -> Int

functionList ::  [Int] -> [Int]
functionList [] = []
functionList (x:xs) = (function x) :  functionList xs
```

  where in our previous examples `function` is

  `double`   `square`   `inc`

- **Key Idea:** Make auxiliary function `function` an input

- **The Idea Coded:**

```
map f [] = []
map f (x:xs) = (fx) :  map f xs
```

- **Advantages:** There are several advantages
  - Shortens code as previous examples are given by

```
doubleList xs  =  map double xs
squareList xs  =  map square xs
   incList xs  =  map inc xs
```

  - Captures the algorithmic content and is easier to understand
  - Easier code-modification and code re-use

- **Question:** What is the type of `map`?
  - First argument is a function
  - Second argument is a list whose elements have the same type and the input of the function.
  - Result is a list whose elements are the output type of the function.
- **Answer:** So overall type is `map :: (a -> b) -> [a] -> [b]`
- **Definition:** A function is higher-order if an input is a function.
- **Another Example:** Type of `filter` is

```
filter :: (a -> Bool) -> [a] -> [a]
```

- **Idea:** Recall our implementation of *quicksort*

```
qsort ::  Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort less ++ occs ++ qsort more
               where
               less = [e | e<-xs, e<x]
               occs = x :  [e | e<-xs, e==x]
               more = [e | e<-xs, e>x]
```

- **Polymorphism:** Quicksort requires an order on the elements:
  - The output list depends upon the order on the elements
  - This requirement is reflected in type class information `Ord a`
  - Don't worry about type classes as they are beyond this course

- **Example:** Games tables might have type `[(Team,Points)]`

- **Problem:** How can we order the table?

```
Arsenal  16
AVilla   16
Derby    10
Birm.     4
...
```

- **Solution:** Write a new function for this problem

```
tSort [] = []
tSort (x:xs) = tSort less ++ [x] ++ tSort more
               where more = [e| e<-xs, snd e > snd x]
                     less = [e| e<-xs, snd e < snd x]
```

- What did we assume here?

- **Motivation:** But what if we want other orders, eg
  - Sort teams in order of names, not points
  - Sort on points, but if two teams have the same points, compare names
- **Key Idea:** Make the comparison a parameter of quicksort

```
qsortCp ::  (a -> a -> Bool) -> [a] -> [a]
qsortCp ord [] = []
qsortCp ord (x:xs) = qsortCp ord less ++ occs ++ qsortCp ord more
                 where less = [ e | e <- xs, ord e x]
                       occs = x :  [ e | e <- xs, e == x]
                       more = [ e | e <- xs, ord x e]
```

- **Key Idea:** To use a higher order sorting algorithm, use the required order to define the function to *sort by*

- **Example 1:** To sort by names

```
ord (t, p) (t', p') = t < t'
```

- **Example 2:** To sort by points and then names

```
ord (t, p) (t', p') = (p < p') || (p == p' && t < t')
```

- What should we assume about `ord`?

- **Higher Order Functions:** Functions which takes functions as input
  - Facilitates code reuse and more abstract code
  - Many list functions are either `map`, `filter` or `fold`

- **HO Sorting:** An application of higher order functions to sorting
  - Produces more powerful sorting
  - Order of resulting list determined by a function
  - Lexicographic order allows us to try one order and then another

Lecture 10 — (Parametric) Polymorphism

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

- **Motivation:** Some examples leading to polymorphism
- **Definition:** What is *parametric* polymorphism?
  - What is a polymorphic type?
  - What is a polymorphic function?
  - Polymorphism and higher order functions
  - Applying polymorphic functions to polymorphic expressions

- **Example:** Let us define the length of a list of integers

```
mylength ::  [Int] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

- **Problem:** We want to evaluate the length of a list of characters

```
Prelude> mylength ['a', 'g']
ERROR: Type error in application
*** expression :  mylength ['a','g']
*** term :  ['a','g']
*** type :  [Char]
*** does not match :  [Int]
```

- **Solution:** Define a new length function for lists of characters
  ... but this is not very efficient!

- **Better Solution:** The algorithm's input depends on the list type, but not on the type of integers.

- **Idea:** An alternative approach to typing `mylength`
  - There is one input and one output: `mylength :: a -> b`
  - The output is an integer: `mylength :: a -> Int`
  - The input is a list: `mylength :: [c] -> Int`
  - There is nothing more to infer from the code of `mylength` so

    ```
    mylength :: [c] -> Int
    ```

    This is an efficient function - works at all list types!

- **Types**: Now we will deal with the following types:
  - Basic, built in types: `Int`, `Char`, `Bool`, `String`, `Float`
  - Type variables representing any type: `a, b, c, ...`
  - Types built with type constructors: `[]`, `->`, `(,)`

    ```
    [Int]  a->a  a->b  a->Bool  (String,a->a)  [a->Bool]
    ```

  - Type synonyms: `type <type-name> = <type-expression>`

    ```
    type Point = (Int,Int)
    type Line = (Point,Point)
    type Test = a->Bool
    ```

- **Polymorphism** is the ability to appear in different forms

- **Definition:** A type is *parametric polymorphic* iff it contains type variables (that is, type parameters).

- **Definition:** A function is *parametric polymorphic* iff it can be called on different types of input, and it is implemented by (code for) a single algorithm

- **Definition:** A function is *overloaded* iff it can be called on different types of input, and for each type of input, the function is implemented by (code for) a particular algorithm.

- **Examples:** Of overloading are the arithmetic operators: integer and floating-point addition.

- **Key Idea:** Expressions have many types
  - Amongst these is a *principle* type

- **Example:** What is the type of `id x = x`
  - `id` sends an integer to an integer. So `id :: Int -> Int`
  - `id` sends a list of type `a` to a list of type `a`. So `id::[a]->[a]`
  - `id` sends an expression of type b to an expression of type b. So `id::b->b`

- **Principle Type:** The last type includes the previous two – why?
  - In fact the principal type of `id` is `id::b->b` – why?

- **Example 1:** What is the type of `map`

```
map f [] = []
map f (x:xs) = f x :  map f xs
```

- **Example 2:** What is the type of `filter`

```
filter f [] = []
filter f (x:xs) = if f x then x:filter f xs else filter f xs
```

- **Example 1:** What is the type of `map`

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

- **Example 2:** What is the type of `filter`

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

- **Example 3:** What is the type of `iterate`

```
iterate f 0 x = x
iterate f n x = f (iterate f (n-1) x)
```

- **Example 3:** What is the type of `iterate`

```
iterate ::  (a->a)->Int->a->a
iterate f 0 x = x
iterate f n x = f (iterate f (n-1) x)
```

Example

```
iterate (2*) 4 1 = (2*(2*(2*(2* (1))))) = 16
```

In general

```
iterate f n x
```

is

$$\mathtt{f}^n\ \mathtt{(x)}$$

- **Previously:** The typing of applications of expressions:
  - If `exp1` is an expression with type `a -> b`
  - And `exp2` is an expression with type `a`
  - Then `exp1 exp2` has type `b`

- **Problem:** How does this apply to polymorphic functions?

```
length         :: [c] -> Int
[2,4,5]        :: [Int]
length [2,4,5] :: Int
```

- **Key Idea:** Argument type can be an *instance* of input type

- **Recall:** Two facts about expressions containing variables
  - Variables stand for arbitrary elements of a particular type
  - *Instances* of the expression are obtained by substituting expressions for variables

- **Key Idea:** (Parametric) polymorphic types are defined in the same way:
  - Type-expressions may contain type-variables
  - *Instances* of type-expressions are obtained by substituting types for type-variables

- **Example:** `[Int]` is an instance of `[c]` – substitute `Int` for `c`

- **Monomorphic:** Can a function be applied to an argument?
  - If the function's input type is the same type as its argument

$$\frac{\texttt{f::a->b} \quad \texttt{x::a}}{\texttt{f x ::} \quad \texttt{b}}$$

- **Polymorphically:** Can a function be applied to an argument?
  - If the function's input type is *unifiable* with argument's type

$$\frac{\texttt{f::a->b} \quad \texttt{x::c} \quad \theta \text{ unifies } \texttt{a,c}}{\texttt{f x ::} \ \theta\texttt{b}}$$

where θ maps type variables to types

- **Example:** In the `length` example, set θ`c=Int`

- **Past Paper:** Assume `f` is a function with principle type

  ```
  f::([a],[b])->Int->[(b,a)]
  ```

  Do the following expressions type check? State **Yes** or **No** with a brief reason and, if **Yes**, what is the principal type of the expression?

  1. `f (3,3) 2`
  2. `f ([],[]) 5`
  3. `f ([tail,head], []) 3`
  4. `f ([True,False], ['x'])`

- **Polymorphism:**
  - Saves on code — one function (algorithm) has many types
  - This implements our algorithmic intuition
- **Type Checking:** Expressions and functions have many types including a principle one
  - Polymorphic functions are applied to expressions whose type is an instance of the type of the input of the function

Lecture 10 — Algebraic Types

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

Recall the types in Haskell we have seen so far.

- basic types like `Int` , `Float` , `Char` , `Bool`
- compound types:
  - tuple types like `(Int, String)`
  - list types like `[Int]`
  - function types like `Int -> Int`
  - type synonyms like `type Word = String`
  - polymorphic types like `(a->b) -> [a] ->[b]`
  - polymorphic types and classes like `Ord a => [a] -> [a]`

There are other types which are difficult to model using the types seen so far.

Examples include:

- The type of months: January, February, .., December.
- The type of geometric shapes whose elements are either circles or rectangles.
- The type of trees.

All these types can be modelled by Haskell **algebraic types**.

Algebraic Types I: Enumerated Types

Enumerated types are types which are defined by enumerating the elements.

Example. Temperatures are either hot or cold.

```
data Temp = Cold | Hot
```

The type `Temp` has two members `Cold` and `Hot`, such that

```
Cold :: Temp
Hot :: Temp
```

`Cold` and `Hot` are called **CONSTRUCTORS**.

Example.

```
data Season = Spring | Summer | Autumn | Winter
```

The type `Season` has four members (four constructors) which are:

`Spring`, `Summer`, `Autumn` and `Winter`.

This means that

```
Spring :: Season
Summer :: Season
Autumm :: Season
Winter :: Season
```

```
data Month = January|February|March|April|May|June|July|
             August|September|October|November|December
```

Example. Checking if a month is in summer.

```
isSummer ::  Month -> Bool
isSummer June = True
isSummer July = True
isSummer August = True
isSummer September = True
isSummer _ = False
```

To define functions over enumerated types we use **pattern matching**.

ENUMERATED TYPES.

$$\texttt{data}\ \langle\texttt{type-name}\rangle = \langle\texttt{constructor 1}\rangle \mid \ldots \mid \langle\texttt{constructor n}\rangle$$

FUNCTIONS ON ENUMERATED TYPES.

$$\langle\texttt{function-name}\rangle :: \langle\texttt{type-name}\rangle - > \langle\texttt{out-type}\rangle$$
$$\langle\texttt{function-name}\rangle\langle\texttt{pat}\rangle = \langle\texttt{exp}\rangle$$

Rule for names: the name of the type and the names of the constructors begin with capital letters. Name of functions begin with lower case.

Example. A geometric shape is either a circle or a rectangle.

```
data Shape = Circle Float | Rectangle Float Float
```

There are two ways of building an element of `Shape`:

1. Supplying the radius of a circle:

   ```
   Circle 3.9 ::  Shape
   ```

2. Giving the sides of a rectangle:

   ```
   Rectangle 3.5 13.5 ::  Shape
   ```

Key idea. Incorporate *extra type information* in type definition.

Example.

```
data Shape = Circle Float | Rectangle Float Float
```

Elements of type `Shape` are *constructed* by applying `Shape` and `Circle` to certain arguments.

```
Circle :: Float -> Shape
Rectangle :: Float -> Float -> Shape
```

`Circle` and `Shape` are called
CONSTRUCTOR FUNCTIONS or just **CONSTRUCTORS**.

The general form of such a definition looks like

$$
\begin{array}{lcl}
\texttt{data}\ \langle\texttt{type-name}\rangle & = & \langle\texttt{Cons-1}\rangle\ \langle\texttt{type}\rangle \ldots \langle\texttt{type}\rangle \\
 & | & \langle\texttt{Cons-2}\rangle\ \langle\texttt{type}\rangle \ldots \langle\texttt{type}\rangle \\
 & .. & \ldots \\
 & | & \langle\texttt{Cons-r}\rangle\ \langle\texttt{type}\rangle \ldots \langle\texttt{type}\rangle
\end{array}
$$

Here the **constructor (functions)** are:

$$\texttt{Cons-i ::}\ \ \texttt{type} \rightarrow \ldots \rightarrow \texttt{type} \rightarrow \texttt{type-name}$$

Example. Testing if a shape is a rectangle

```
isRect ::  Shape -> Bool
isRect (Circle x) = False
isRect (Rectangle h w) = True
```

Key Idea. Again, use *pattern matching* to define functions. Now, *supply variables* for the constructor functions.

Example. Area of a shape is given by

```
area ::  Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle h w) = h * w
```

Alternative definition:

```
area ::  Shape -> Float
area x = case x of
            Circle r -> pi * r * r
            Rectangle h w -> h * w
```

Example.

```
data Temp = Cold | Hot
```

PROBLEM. If you type `Hot` to the prompt, you get an error!!!

```
Main > Hot
ERROR - Cannot find "show" function for:
*** Expression :  Hot
*** Of type :  Temp
```

SOLUTION. Add the clause `deriving` after the type definition.

```
data Temp  =  Cold|Hot
              deriving Show
```

What does `deriving` mean? What is `Show`?

Class of types	Operators defined over the types belonging to the class
`Eq`	equality and inequality
`Ord`	order between elements
`Enum`	operations like `[n..m]`
`Show`	operations that turn elements into textual form

If we want to define a non-standard equality on the type `Temp`, we have to make the type an `instance` of the class `Eq`.

```
instance Eq Temp where
 Cold  ==  Cold  =  True
   _   ==   _    =  False
```

If we want the standard definition of equality, Haskell generates it automatically.

```
data Temp  =  Cold|Hot
              deriving Eq
```

Example.

```
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq,Ord,Enum,Show)
```

Haskell automatically generates definitions of equality, ordering, enumeration and text functions.

`Spring == Spring` evaluates to `True`.

`[Summer .. Winter]` gives the list `[Summer, Autumn, Winter]`.

Example. Months of the year:

```
data Month  =  January|February|March|April|May|June|July|
               August|September|October|November|December
               deriving (Eq,Ord,Enum,Show)
```

Example.

```
data Shape  =  Circle Float | Rectangle Float Float
               deriving (Eq,Ord,Show)
```

We cannot enumerate shapes. Being in `Enum` can only be derived for enumerated types.

Algebraic Types. Algebraic types allow

- Choice in the sorts of elements that make up the type
- Elements can contain parameters which belong to other types

Functions.

1. Functions are defined by pattern matching.
2. Functions of built-in classes can be derived automatically by Haskell.

Lecture 11 — Recursive Algebraic Types

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

Recursive algebraic types are types which are described in terms of themselves.

Example. A type `Exp` for arithmetic expressions defined by

```
data Exp = Num Int | Add Exp Exp | Mul Exp Exp
           deriving Show
```

Key Idea: `Exp` also appears on the righthand side of the definition.

Informal expression	*Haskell representation*
5	`Num 5`
$5 + 21$	`Add (Num 5) (Num 21)`
$8 * 10$	`Mul (Num 8) (Num 10)`
$(4 * (2 + 5))$	`Mul (Num 4) (Add (Num 2) (Num 15))`

Elements of `Exp` are either

1. integer expressions:

```
Num 5 :: Exp
```

2. or a combination of expressions using the arithmetic operations:

```
Add (Num 5) (Num 7) :: Exp
Mul (Num 5) (Num 7) :: Exp
```

To build an element of type `Elem` we use a combination of the following three **constructor functions**:

```
Num ::  Int -> Exp
Add ::  Exp -> Exp -> Exp
Mul ::  Exp -> Exp -> Exp
```

Example. Lists of integers can be modelled by the type:

```
data NList = Nil | Cons Int NList
             deriving Show
```

Instances of elements of type `NList` are:

```
Nil
Cons 12 Nil
Cons 17 (Cons 12 Nil)
```

The constructors are:

```
Nil :: NList
Cons :: Int -> NList ->NList
```

Example. Trees of integers can be modelled by the type:

```
data NTree  =  NNil | NNode Int NTree NTree
               deriving Show
```

Instances of elements of type `NTree` are:

`NNil`

`NNode 12 NNil NNil`

`NNode 17 (NNode 12 NNil NNil) (NNode 22 NNil NNil)`

The constructors are:

`NNil :: NTree`

`NNode :: Int -> NTree -> NTree -> NTree`

Recall: functions over algebraic types are defined by pattern matching with one clause for each constructor

```
data Shape = Circle Float | Rectangle Float Float
```

```
perim ::  Shape -> Float
perim (Circle x) = 2 * 3.14 * x
perim (Rectangle h w) = 2 * (h + w)
```

Example. Evaluation of expressions

```
eval ::  Exp -> Int
eval (Num x) = x
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Now our functions can be *recursive*. The form of the recursive definition follows the recursion on the type definition.

There are two parts:

1. *Non-recursive or base cases*. The value of `Num x` is `x`.

2. *Recursive cases*. The value of an expression is calculated in terms of the values of its subexpresions `e1` and `e2`.

Example. Sum the nodes of a tree:

```
sumTree ::  NTree -> Int
sumTree NNil = 0
sumTree (NNode n t1 t2) = n+ sumTree t1 + sumTree t2
```

Example. Left subtree:

```
leftTree ::  NTree -> NTree
leftTree NNil = NNil
leftTree (NNode n t1 t2) = t1
```

This is not a recursive definition.

Example. The depth of a tree:

```
depth::  NTree -> Int
depth NNil = 0
depth (NNode n t1 t2) = 1 + max (depth t1) (depth t2)
```

Example. Find out how many times a number `p` occurs in a tree:

```
occurs::  NTree -> Int -> Int
occurs NNil p = 0
occurs (NNode n t1 t2) p
   |n==p = 1 + occurs t1 p + occurs t2 p
   |otherwise = occurs t1 p + occurs t2 p
```

Types. Algebraic types can be recursive, eg trees can contain subtrees.

Functions.

- As before, define functions on each constructor.
- There is a natural way of definining recursive functions on recursive types.

Representations: How to represent elements of a data structure as expressions of a Haskell datatype.

Lecture 12 —
Polymorphic Algebraic Types

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

Example. Recall the definition of trees of integers:

```
data NTree  =  NNil | NNode Int NTree NTree
               deriving Show
```

Example. Trees of strings can be modelled by the type:

```
data STree  =  SNil | SNode String STree STree
               deriving Show
```

There is nothing special about numbers or strings. All these types of trees have the same structure, shape or *form*. We will define *polymorphic* trees.

Trees that carry elements of an arbitrary type at the nodes:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
              deriving Show
```

where `a` is a type variable, i.e. `a` ranges over types.

Key ideas: • The type `Tree a` is parametric on the type `a`.

• We have to instantiate `a` to get a particular type of trees:

`a`	`Tree a`	description of the type
`Int`	`Tree Int`	trees of integers
`String`	`Tree String`	trees of strings
`[Int]`	`Tree [Int]`	trees of lists of integers

• We get a family of types: `Tree Int`, `Tree String`, etc.

Examples.

1. `Nil :: Tree a` for any type `a`

2. An element of `Tree Int` is:

   `Node 12 Nil Nil :: Tree Int`

3. An element of `Tree String` is:

   `Node "Leicester" Nil Nil :: Tree String`

4. An element of `Tree [Int]` is:

   `Node [1,2] (Node [2,5,1] Nil Nil) Nil :: Tree [Int]`

The built-in type of lists can be given by the definition:

```
data List a  =  NilList | Cons a (List a)
                deriving Show
```

where we use the following notation:

`Haskell notation`	`our definition`
`[a]`	`List a`
`[]`	`NilList`
`:`	`Cons`

The type `[a]` is parametric on `a`. By instantiating `a`, we get a family of types: `[Int]`, `[String]`, `[[Int]]`, etc.

Example. The depth of a tree

```
depth::  Tree a -> Int
depth Nil = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

The function `depth` is *polymorphic*: it has a common definition for all the family of trees.

Example. Find out how many times a number `p` occurs in a tree:

```
occurs::  Eq a => Tree a -> a -> Int
occurs Nil p = 0
occurs (Node n t1 t2) p
   |n==p = 1 + occurs t1 p + occurs t2 p
   |otherwise = occurs t1 p + occurs t2 p
```

The type of this polymorphic function is conditional: it has the condition that the type `a` should belong to the class `Eq`.

Example. The function `mapTree` is similar to the function `map` over lists:

```
mapTree ::  (a->b) -> Tree a -> Tree b
mapTree f Nil = Nil
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1)
(mapTree f t2)
```

Key ideas.

• it is *higher order*, i.e. it is a function that has an argument `f` which is a function too.

• it is *polymorphic*: it has a common definition with type `(a->b) -> Tree a -> Tree b` for all types `a` and `b`.

Example. Consider the tree:

```
Node 3 (Node 7 Nil Nil) (Node 2 (Node 3 Nil Nil) Nil)
```

What do we have to do to get to the second occurrence of `3`? First go to the right and then go to the left.

Type Definition: Encode paths as follows

- A direction is either left or right: `data Dir = L | R`
- A path is a list of directions: `type Path = [Dir]`

Example: So the path to the second occurrence of `3` is represented by the list `[R,L]`.

Example 1: Is a path valid for a particular tree?

```
isPath ::  Path -> Tree a -> Bool
isPath [] (Node n t1 t2) = True
isPath (L:p) (Node n t1 t2) = isPath p t1
isPath (R:p) (Node n t1 t2) = isPath p t2
isPath _ _ = False
```

For instance, the path `[L]` is not a valid path for the tree `Node 3 Nil (Node 2 Nil Nil)` because there is nothing in the left subtree.

Example 2: What data is stored in a tree at the end of the path?

```
extract ::  Path -> Tree a -> a

extract [] (Node n t1 t2) = n
extract (L:p) (Node n t1 t2) = extract p t1
extract (R:p) (Node n t1 t2) = extract p t2
extract _ _ = error "There is no data at this path"
```

We can extract the number `3` at the end of the path `[R,L]` from `Node 3 (Node 7 Nil Nil) (Node 2 (Node 3 Nil Nil) Nil)`

Example. Trees which store data in all the nodes. A Tree is

- A leaf storing data
- A node storing a left subtree, some data, and a right subtree

```
data Tree1 a = Leaf a | Node1 (Tree1 a) a (Tree1 a)
```

Example. Trees which may store no data at a leaf

- The empty tree storing no data
- A leaf storing data
- A node storing a left subtree, some data, and a right subtree

```
data Tree2 a = ND | Leaf a | Node2 (Tree2 a) a (Tree2 a)
```

Key Idea: To define a type, first work out the constructors and their types

Types: Algebraic types can be

- Polymorphic, eg trees can store different forms of data
- Polymorphic algebraic types have a type parameter, eg `Tree Int` or `Tree String`, not `Tree`

Lecture 13 — Errors

Fer-Jan de Vries

Department of Informatics
University of Leicester

January 21, 2021

How should a program deal with a situation which ought not to occur?

Examples: divide by zero, head of an empty list, etc.

We will discuss four approaches for handling errors:

1. The `error` function.
2. Dummy values.
3. Auxiliary functions and the `error` function.
4. The `Error` types (the nicest solution).

The `error` function stops the computation and prints a message.

Example. Recall the `cost` function:

```
type NumCars = Int
type DailyCost = Float
cost ::  NumCars -> DailyCost
cost n
    | n < 0 = error "Car production is always positive"
    | 0 <= n && n <= 1000 = 6*m + 1000
   | otherwise = 4*m + 450
 where m = fromIntegral n
```

Problem. We may loose useful information for stopping the computation.

Suppose we have a database with a daily record of the number of cars produced by a factory:

```
recordcars = [ 1000, -25000, 230000, -20000, 45000, 30000]
```

`map cost recordcars` evaluates to

```
[7000.0,
Program error:  Car production is always positive
```

Problem. The computation stops in the 2nd value and we loose the rest of the information.

Example. Recall this notion of trees:

```
data Tree2 a = ND | Leaf a | Node2 (Tree2 a) a (Tree2 a)
```

Adding an element at one `ND`

```
addData ::  a -> Tree2 a -> Tree2 a
addData x ND = Leaf x
addData x (Leaf y) = error ‘‘Cannot add Data’’
addData x (Node2 t1 y t2) = Node2 (addData x t1) y (addData x t2)
```

Wrong! This doesn’t work, because it

- replaces all `ND`’s with the data
- crashes as soon as we hit a leaf

We want a mechanism that explains when the program worked al right

Instead of giving an error message, we can choose to give a particular value (called *dummy value*) in the error case.

```
cost n
 | n < 0 = -1
 | 0 <= n && n <= 1000 = 6*m + 1000
 | otherwise = 4*m + 450
 where m = fromIntegral n
```

This works if the cost is expected to be always positive.

Drawback. In many cases there is no way to tell when an error has occurred. For instance, imagine a cost function that substracts 1000 instead of adding it.

To solve the problem of adding data in a tree, we can define an auxiliary function. We first test whether there is some `ND` in the tree or not:

```
isND ::  Tree2 a -> Bool

isND ND = True
isND (Leaf x) = False
isND (Node2 t1 x t2) = (isND t1) || (isND t2)
```

Then, we write a function that combines the auxiliary function and the `error` function:

```
addData ::  a -> Tree2 a -> Tree2 a
addData x ND = Leaf x
addData x (Leaf y) = error "There are no ND’s"
addData x (Node2 t1 y t2)
 | (isND t1) = Node2 (addData x t1) y t2
 | (isND t2) = Node2 t1 y (addData x t2)
 | otherwise = error "There are no ND’s"
```

We will see a more efficient way of solving this problem.

We return an error *value* as a result. For this, we define a polymorphic type:

```
data Err a = Ok a | Fail
```

We return a value that contains the following information:

- the program worked and returns a certain value of type `a` or
- the program didn't work

Key Idea. Functions now return error types. Compare with Java's try–blocks

Example. Redefining `cost`

```
cost n
 | n < 0 = Fail
 | 0 <= n && n <= 1000 = Ok(6*m + 1000)
 | otherwise = Ok(4*m + 450)
 where m = fromIntegral n
```

Consider the database

`recordcars =[ 1000, -25000, 230000, -20000, 45000, 30000]`.

Now, `map cost recordcars` evaluates to

`[Ok 7000.0,Fail,Ok 920450.0,Fail,Ok 180450.0,Ok 120450.0]`.

Advantage. We can see all production costs: the correct and the incorrect ones.

Example. Remove the `n`-th element from a list (there is no zero-th element)

```
remove ::  Int -> [a] -> Err [a]

remove 1 (x:xs) = Ok xs
remove n (x:xs) = case remove (n-1) xs of
                    Fail -> Fail
                    Ok zs -> Ok (x:zs)
remove n _ = Fail
```

Note the use of case. We have two cases: either `remove n xs` evaluates to `Fail` or it evaluates to an expression of the form `OK zs`.

Example. Redefining `addData` efficiently:

```
addData ::  a -> Tree2 a -> Err (Tree2 a)

addData x ND                =  Ok (Leaf x)
addData x (Leaf y)          =  Fail
addData x (Node2 t1 y t2)   =  case (addData x t1) of
                                 Ok t -> Ok (Node2 t y t2)
                                 Fail -> case (addData x t2) of
                                    Ok t -> Ok (Node2 t1 y t)
                                    Fail -> Fail
```

Advantage. We do not need an auxiliary function like `isND`. The test for `ND`'s is incorporated into the function `addData`. This version of `addData` is *more efficient*.

- **Example:** What element occurs at a path in a tree

```
lookup :: Path -> NTree a -> Err a
```

- Try to write your own code for `lookup`.

- **Final comment:** instead of defining our own type `Err a` we could have used the standard type `Maybe a`.

- **Example:** What element occurs at a path in a tree even if the path points at something outside the tree...

```
data NTree = NNil | NNode Int NTree NTree deriving Show

lookup ::  Path -> NTree a -> Err a
lookup [] (NNode a t1 t2) = Ok a
lookup (L:path) (NNode a t1 t2) = case (lookup path t1) of
                  Fail -> Fail
                  Ok b -> Ok b
lookup (R:path) (NNode a t1 t2) = case (lookup path t2) of
                  Fail -> Fail
                  Ok b -> Ok b
lookup _ _ = Fail
```

Today You Should Have Learned

- **Trees:** Different varieties of Trees
  - Is there data stored in nodes?
  - How many subtrees does a node have?
- **Paths:** Do you understand what paths are?
  - Can you write functions using paths?
- **Errors:** Using error types as exception handlers
  - Java has `try`, `catch` blocks etc
- **Practical:** Combines all three of these types