Functional Programming – Part 2: Types

PARTIAL APPLICATION

Partial application in Haskell involves passing less than full number of arguments to function that takes multiple arguments.

- Remembering the maxim that: ***Functions are not partial, you can partially apply a function.***
- An important property to remember is that: function application is **left associative** where the partial application is **right associative**

For example:

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

addOne = add 1
```

In this example, `addOne` is the result of partially applying `add`. It is a new function that takes an integer, adds 1 to it and returns that as the result. The important property here is that the `->` operator is right associative, and function application is left associative, meaning the type signature of add actually looks like this:

```
add :: Int -> (Int -> Int)
```

This means that add actually takes one argument and returns a function that takes another argument and returns an Int.

For example:

```
comp2 :: (a -> b) -> (b -> b -> c) -> (a -> a -> c)
comp2 f g = (\x y -> g (f x) (f y))
```

Remembering the maxim that: *Functions are not partial, you can partially apply a function.*

So, is this a partial definition of `comp2'`?

```
comp2' f = (\x y -> add (f x) (f y))
```

Not really, this is the definition of another function. But you can achieve the desired result by partially applying `comp2`, like so:

```
comp2' f = comp2 f add
```

This example shows that it is not enough to just use some of the semantics from other operation as in example this will produce a different definition instead partial application happens when a method is applied to a function in order to retrieve a result which later will also be applied to a function giving us result.

ANONYMOUS FUNCTION

Lambda expression denoted by \x -> x + 1

- Backslash in Haskell's is a way of expressing a Lambda and is supposed to look like a Lambda
- Sometimes is more convenient to include anonymous functions in operations like map and foldl / foldr.

EXAMPLE:

(\x -> x + 1) 4

will return 5.

Here is a standard approach with a named function

```
addOneList lst = map addOne' lst
  where addOne' x = x + 1
```

But here's another way, where we pass the anonymous function into map rather than any named function.

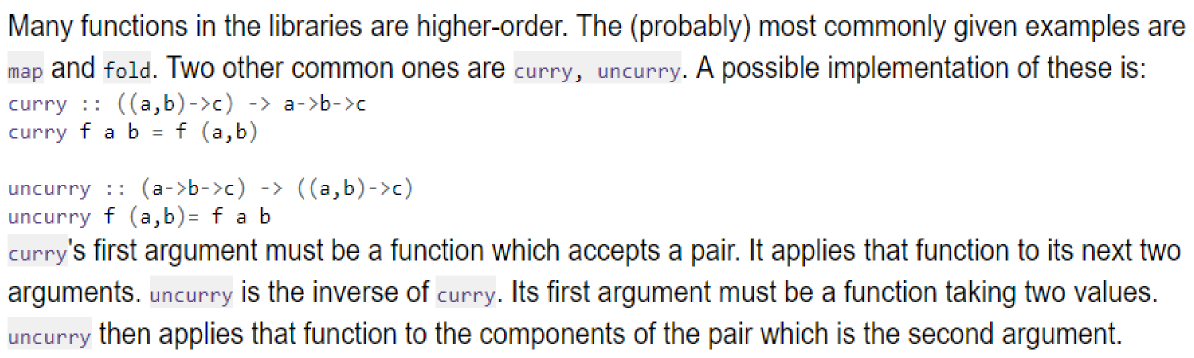
```
addOneList' lst = map (\x -> x + 1) lst
```

STRONGLY TYPED LANGUAGE

Strongly-typed language is one which each variable must be declared with a data type. A variable cannot start off life without knowing the range of values it can hold, and once it is declared, the data type of the variable cannot change.

HIGHER ORDER FUNCTIONS

Definition: *A higher-order function is a function that takes other functions as arguments or returns a function as result.*

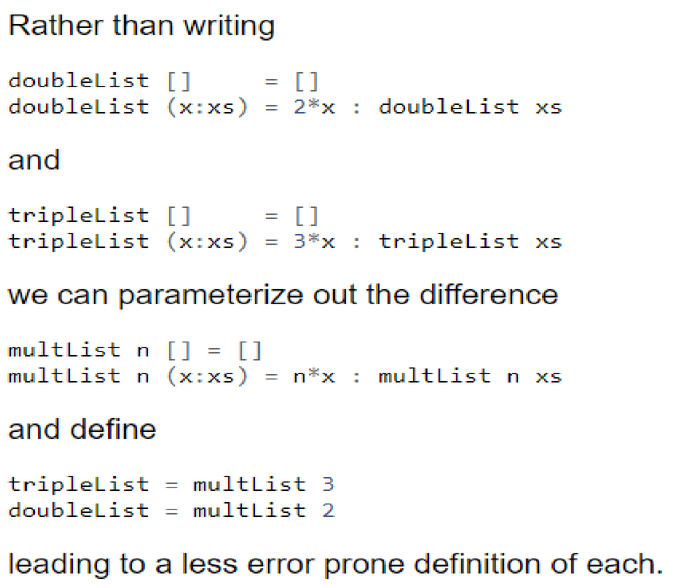


Many functions in the libraries are higher-order. The (probably) most commonly given examples are `map` and `fold`. Two other common ones are `curry, uncurry`. A possible implementation of these is:

```
curry :: ((a,b)->c) -> a->b->c
curry f a b = f (a,b)

uncurry :: (a->b->c) -> ((a,b)->c)
uncurry f (a,b)= f a b
```

`curry`'s first argument must be a function which accepts a pair. It applies that function to its next two arguments. `uncurry` is the inverse of `curry`. Its first argument must be a function taking two values. `uncurry` then applies that function to the components of the pair which is the second argument.



Rather than writing

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

and

```
tripleList []     = []
tripleList (x:xs) = 3*x : tripleList xs
```

we can parameterize out the difference

```
multList n [] = []
multList n (x:xs) = n*x : multList n xs
```

and define

```
tripleList = multList 3
doubleList = multList 2
```

leading to a less error prone definition of each.

INTRODUCTION

- Values are divided into classes, called **types**
- Each **type** is associated with a set of valid operations
- A **type** system enforces this by the construction and use of well-typed expressions
- Types may be given **explicitly** by the programmer or **inferred automatically** by the **type system**
- Typing may be checked before execution (**static typing)** or during execution (**dynamic typing)**

- ***Statically typed languages include*** *C, Java and Haskell – type check DURING COMPILE TIME*
- ***Dynamically typed languages include*** *Python, JavaScript and PHP – type check DURING RUN TIME*

Types system introduces a practical way of programming, it is very useful and important.

ADVANTAGES OF STATIC TYPING

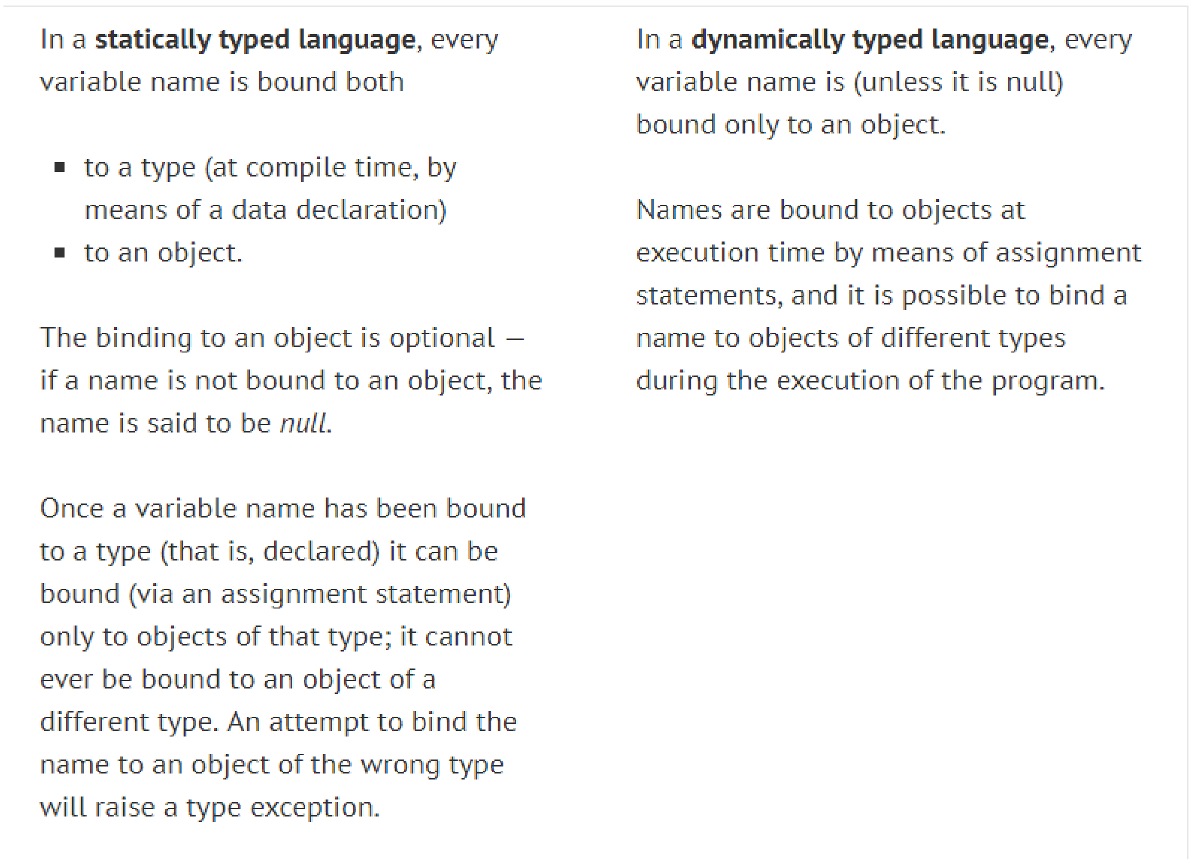
Static typing is carried out at **compile time**. Expressions that cannot be typed are considered erroneous and they are rejected by the compile without evaluation

- A program that is accepted by the type system is not guaranteed to be correct however it is free of type errors at run time. Because they will be pre-determined at compile time.

STATIC TYPING *advantages:*

- Detect possible errors at an early stage and so it helps with the development and maintenance of software.
- In the design and documentation of software by constituting a simple form of specification.

OTHER INFORMATION



In a **statically typed language**, every variable name is bound both

- to a type (at compile time, by means of a data declaration)
- to an object.

The binding to an object is optional — if a name is not bound to an object, the name is said to be *null*.

Once a variable name has been bound to a type (that is, declared) it can be bound (via an assignment statement) only to objects of that type; it cannot ever be bound to an object of a different type. An attempt to bind the name to an object of the wrong type will raise a type exception.

In a **dynamically typed language**, every variable name is (unless it is null) bound only to an object.

Names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program.

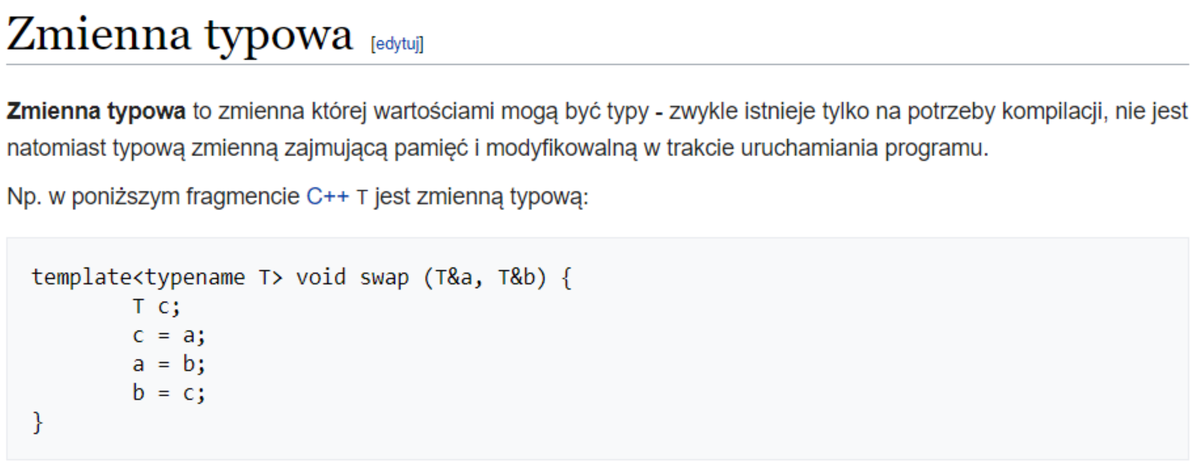
TYPES IN HASKELL

The set of predefined types includes:

- **Primitive data types:** numbers, Booleans, characters, etc…
  E.g. Int, Integer, Float, Double, Bool, Char
- **Constructed types**: list, tuples, etc…
  E.g. [Double] is the type of lists of doubles, (Int, Bool) is a pair of an integer and a Boolean
- **Function types**:
  E.g. Char -> Bool, (Int -> Int) -> Int

- Programmer can also define new types.

TYPE VARIABLE – ZMIENNA TYPOWA



Zmienna typowa [edytuj]

Zmienna typowa to zmienna której wartościami mogą być typy - zwykle istnieje tylko na potrzeby kompilacji, nie jest natomiast typową zmienną zajmującą pamięć i modyfikowalną w trakcie uruchamiania programu.

Np. w poniższym fragmencie C++ T jest zmienną typową:

```
template<typename T> void swap (T&a, T&b) {
        T c;
        c = a;
        a = b;
        b = c;
}
```

POLYMORPHISM

Type system can be

- **MONOMORPHIC:** every expression has at most one type. == 1 TYPE
- **POLYMORPHIC:** expression may have more than one type. > 1 TYPE

This is achieved by introducing *type variables,* which can be instantiated with other types

HASKELL has a POLYMORPHIC type system.

EXAMPLE:

Functional composition *(.)* and *curry* are **polymorphic** functions where *a, b,* and *c* are type variables.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
curry :: ((a, b) -> c) -> (a -> b -> c)
```

POLYMORPHIC TYPES

Formally, the language of polymorphic types is defined as a set of terms *T*, built out of *type variables V,* (a, b, c, …), and *type constructor C,* which are either atomic (e.g. **Int, Bool, Char)** or take other types as arguments (e.g. ->, []).

$$
\begin{array}{lcl}
\mathcal{V} & ::= & \mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots \\
\mathcal{C} & ::= & \mathtt{Int}, \mathtt{Bool}, \mathtt{Char}, \mathtt{->}, \mathtt{[]}, \ldots \\
\mathcal{T} & ::= & \mathcal{V} \mid \mathcal{C}\,(\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n)
\end{array}
$$

A polymorphic type represents the set of its *instances*, obtained by substituting type variables by types.

EXAMPLES OF POLYMORHISM 1.0

Type variables can be instantiated to different types in different context.

EXAMPLE

Let's evaluate quad: It takes an integer and returns integer however it makes call to square twice therefore its type evaluation will be 1st square (Int -> Int) 2nd square (Int -> Int) 3rd quad (Int -> Int) therefore this function takes a Int and returns the same.

Let's evaluate unsquared: First we can see that unsquared makes call to sqrt' and then to square so we can infer the type to be 1st sqrt' (Int -> Float) 2nd we look at the square type (Int -> Int) 3rd the unsquared which have type (Int -> Float)

```
square :: Int -> Int
sqrt' :: Int -> Float

quad :: Int -> Int
quad = square . square

unsquare :: Int -> Float
unsquare = sqrt' . square
```

where quad uses an instance of `(.)` of type
`(Int -> Int) -> (Int -> Int) -> (Int -> Int)`
and unsquare uses an instance of `(.)` of type
`(Int -> Float) -> (Int -> Int) -> (Int -> Float)`

Now looking at the second figure, we can see that in both cases when we used (.) the type is matching with its definition. We can substitute A to Int, B to Int and C to Float

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
curry :: ((a, b) -> c) -> (a -> b -> c)
```

EXAMPLE OF POLYMORHISM 1.1

The error function is also polymorphic

```
error :: String -> a

fact :: Int -> Int
fact n
    | n == 0 = 1
    | n > 0 = n * (fact (n - 1))
    | otherwise = error "negative argument"
```

Here the function error is used with type `String -> Int`

OVERLOADING

AD-HOC POLYMORPHISM – OVERLOADING

Overloading is a related notion where several functions, with different types, share the same name.

Arithmetic operations such as addition can be used both with integers or reals.

But a polymorphic type such as (+) :: a -> a -> a is too general; for example, it would allow addition to be used with characters of type Char.

There are several solutions for this problem:

1. Use different symbols for addition on integers and on reals. For example, + and +. (as in OCaml)
2. Enrich the language of types, for example:
   (+) :: (Int -> Int -> Int) ^ (Float -> Float -> Float)
3. Define a notion of a *type class*, for example:
   `(+) :: Num a => a -> a -> a`
   that is, the function `(+)` has the type `a -> a -> a`
   where `a` is an instance of the type class `Num`.

3rd one is the approach that Haskell is using, by creating type classes, this (+) will only work if all instances of a belong to type class Num.

TYPE CLASSES

A type class can be thought of as an interface. Types that belong to the class must implement functions specified by the class. Common type classes include: **Eq, Ord, Num, Integral, Enum, Floating** and **Show.**

EXAMPLE

- (==) :: Eq a => a -> a -> Bool
- (>) :: Ord a => a -> a -> Bool
- fromIntegral :: (Num b, Integral a) => a -> b
- sqrt :: Floating a => a -> a
- min :: Ord a => a -> a -> a

EXAMPLE

- sqrt (fromIntegral 9) -> 3.0

-> – function type constructor.

Constraints on the type variables on the right => normal type

Week 6

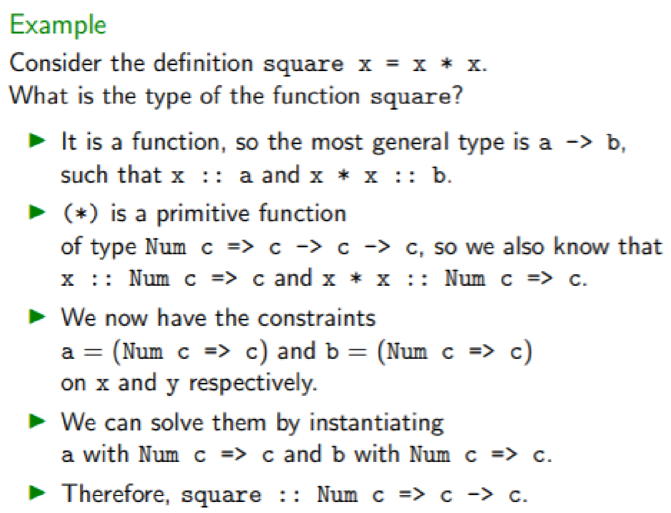
TYPE INFERENCE

Most modern functional languages do not require that the programmer provides the type for the expression used. The compiles is able to **infer** a type, if one exists

- The expression is decomposed into sub-expressions are put together to build the expression generates constraints that their types must satisfy, if necessary by instantiating the type variables present.
- The manner in which the sub-expressions are put together to build the expression generates constraints that their types must satisfy, if necessary by instantiating the type variables present.
- If the constraints cannot be solved, inference fails, and the expression is considered untypeable

Ord is a class of comparable types

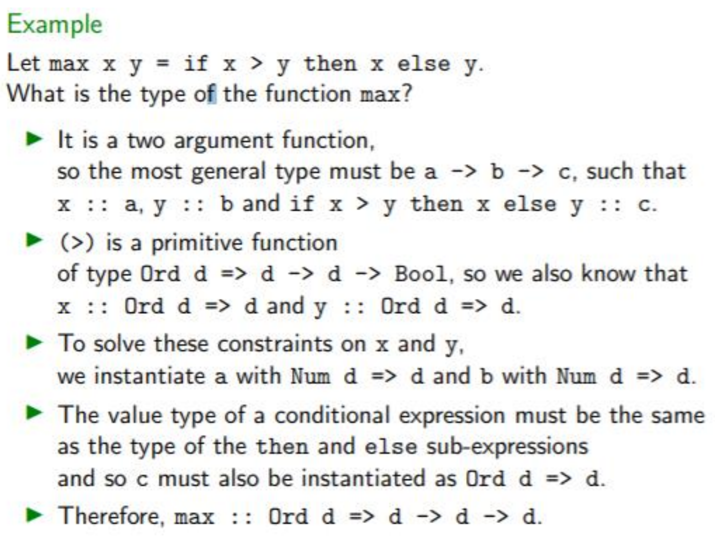
EXAMPLE



Example

Consider the definition `square x = x * x`.
What is the type of the function `square`?

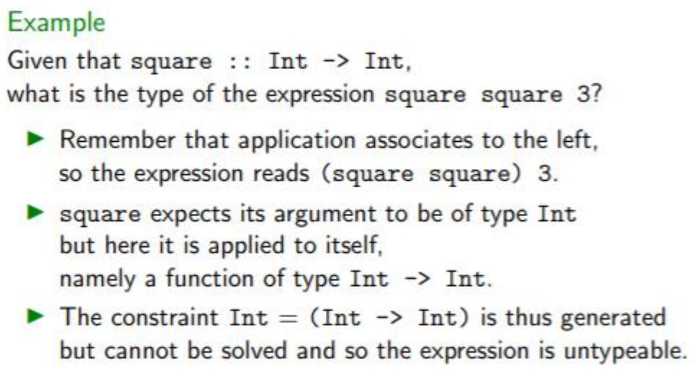
- It is a function, so the most general type is `a -> b`, such that `x :: a` and `x * x :: b`.
- `(*)` is a primitive function of type `Num c => c -> c -> c`, so we also know that `x :: Num c => c` and `x * x :: Num c => c`.
- We now have the constraints a = (`Num c => c`) and b = (`Num c => c`) on `x` and `y` respectively.
- We can solve them by instantiating a with `Num c => c` and b with `Num c => c`.
- Therefore, `square :: Num c => c -> c`.



Example

Let `max x y = if x > y then x else y`.
What is the type of the function `max`?

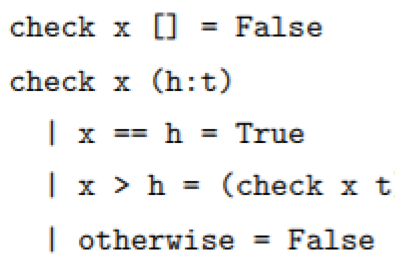
- It is a two argument function, so the most general type must be `a -> b -> c`, such that `x :: a`, `y :: b` and `if x > y then x else y :: c`.
- `(>)` is a primitive function of type `Ord d => d -> d -> Bool`, so we also know that `x :: Ord d => d` and `y :: Ord d => d`.
- To solve these constraints on `x` and `y`, we instantiate a with `Num d => d` and b with `Num d => d`.
- The value type of a conditional expression must be the same as the type of the `then` and `else` sub-expressions and so c must also be instantiated as `Ord d => d`.
- Therefore, `max :: Ord d => d -> d -> d`.



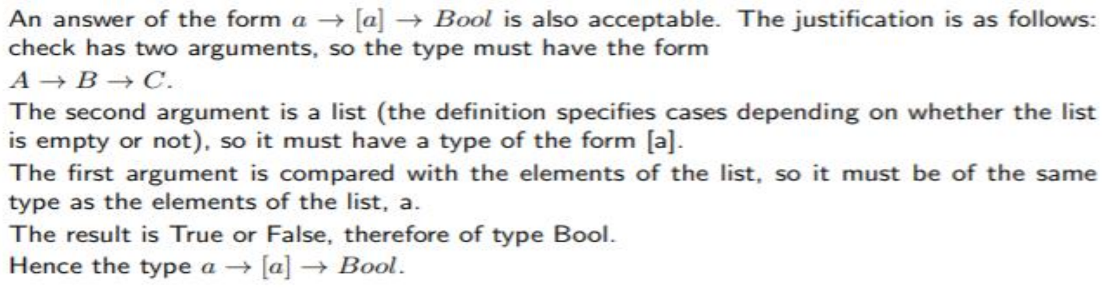
Example

Given that `square :: Int -> Int`,
what is the type of the expression `square square 3`?

- Remember that application associates to the left, so the expression reads `(square square) 3`.
- `square` expects its argument to be of type `Int` but here it is applied to itself, namely a function of type `Int -> Int`.
- The constraint `Int` = (`Int -> Int`) is thus generated but cannot be solved and so the expression is untypeable.



```
check x [] = False
check x (h:t)
  | x == h = True
  | x > h = (check x t)
  | otherwise = False
```

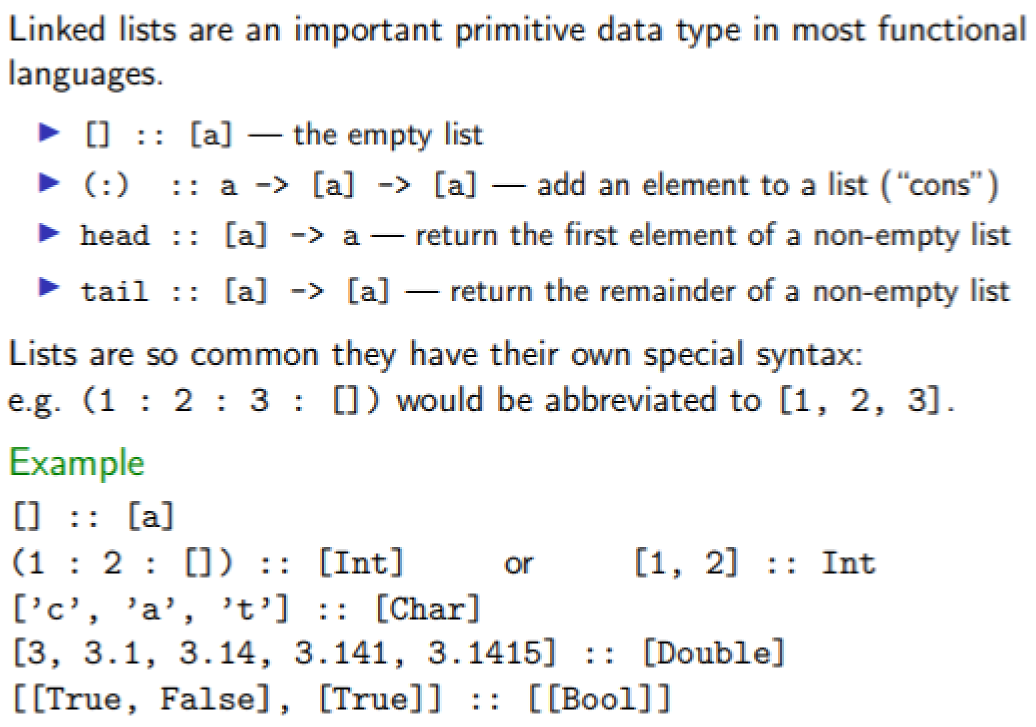


An answer of the form $a \to [a] \to Bool$ is also acceptable. The justification is as follows: check has two arguments, so the type must have the form
$A \to B \to C$.
The second argument is a list (the definition specifies cases depending on whether the list is empty or not), so it must have a type of the form [a].
The first argument is compared with the elements of the list, so it must be of the same type as the elements of the list, a.
The result is True or False, therefore of type Bool.
Hence the type $a \to [a] \to Bool$.

LISTS

xs !! n – nth location in the xs list

.. – make sequence within the range for example 1..20 1 to 20



Linked lists are an important primitive data type in most functional languages.

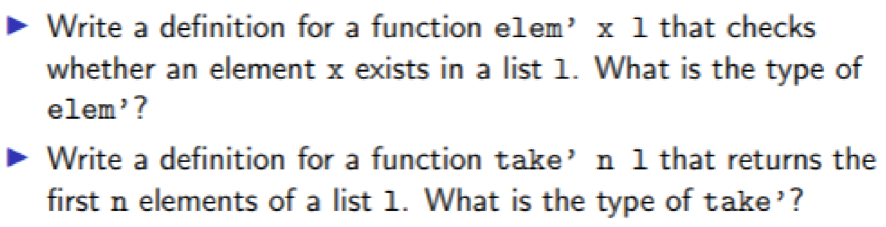
- `[] :: [a]` — the empty list
- `(:)  :: a -> [a] -> [a]` — add an element to a list ("cons")
- `head :: [a] -> a` — return the first element of a non-empty list
- `tail :: [a] -> [a]` — return the remainder of a non-empty list

Lists are so common they have their own special syntax:
e.g. `(1 : 2 : 3 : [])` would be abbreviated to `[1, 2, 3]`.

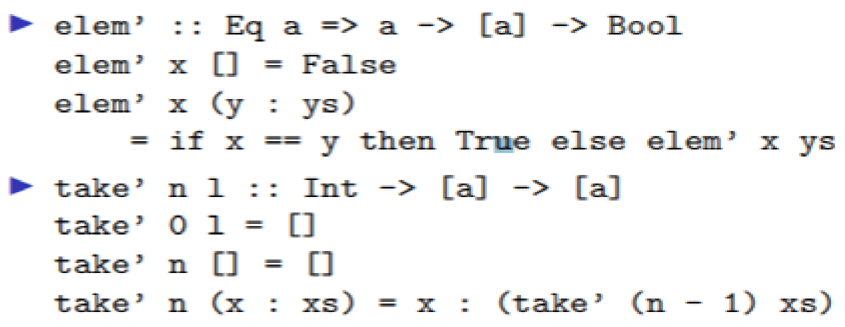
Example

```
[] :: [a]
(1 : 2 : []) :: [Int]      or      [1, 2] :: Int
['c', 'a', 't'] :: [Char]
[3, 3.1, 3.14, 3.141, 3.1415] :: [Double]
[[True, False], [True]] :: [[Bool]]
```

EXERCISES



- Write a definition for a function `elem' x l` that checks whether an element `x` exists in a list `l`. What is the type of `elem'`?
- Write a definition for a function `take' n l` that returns the first `n` elements of a list `l`. What is the type of `take'`?



```
▶ elem' :: Eq a => a -> [a] -> Bool
  elem' x [] = False
  elem' x (y : ys)
      = if x == y then True else elem' x ys
▶ take' n l :: Int -> [a] -> [a]
  take' 0 l = []
  take' n [] = []
  take' n (x : xs) = x : (take' (n - 1) xs)
```

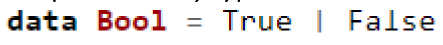
Step back here a moment and take note of the similarities.

- A *data constructor* is a "function" that takes 0 or more *values* and gives you back a new **value**.
- A *type constructor* is a "function" that takes 0 or more *types* and gives you back a new **type**.

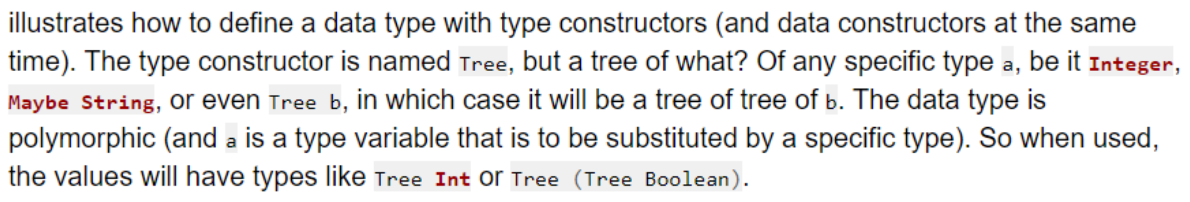
Week 6

TYPE CONSTRUCTOR

A **TYPE CONSTRUCTOR** may have zero or more arguments, if it has zero arguments it is called a nullary type constructor (or simply a **type**). An example of *nullary type constructor* Bool with two nullary data constructors **True** and **False.**



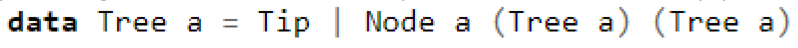
```
data Bool = True | False
```



illustrates how to define a data type with type constructors (and data constructors at the same time). The type constructor is named `Tree`, but a tree of what? Of any specific type `a`, be it `Integer`, `Maybe String`, or even `Tree b`, in which case it will be a tree of tree of `b`. The data type is polymorphic (and `a` is a type variable that is to be substituted by a specific type). So when used, the values will have types like `Tree Int` or `Tree (Tree Boolean)`.

DATA CONSTRUCTOR

A **data constructor (**or **value constructor**) can have zero or more arguments where a data constructor taking zero arguments is called a nullary data constructor or simply a **constant**. For example, Tree:



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

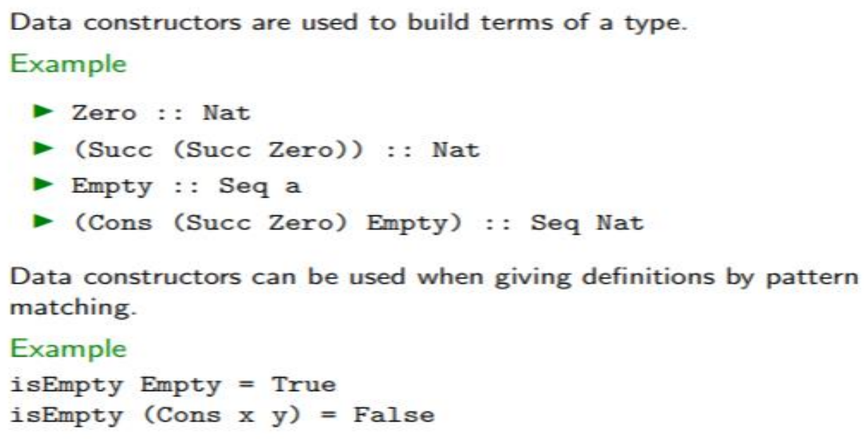
USER DEFINED TYPES

We can define our own new types by declaring the data and type constructors of that type.

Example:

- data Nat = Zero | Succ Nat
  Here, Nat is new recursive type and **Zero** and **Succ** are its data constructors
- data Seq a = Empty | Cons a (Seq a)
  Here, Seq a is a new polymorphic, recursive type, whose elements are built using one of the data constructors **Empty** or **Cons**

- **Nat** and **Seq** are *type constructors*
- **Zero, Succ, Empty** and **Cons** are type constructors.
- **|** separates data constructors



Data constructors are used to build terms of a type.

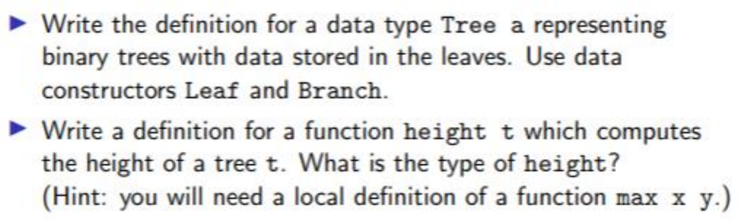
Example

- `Zero :: Nat`
- `(Succ (Succ Zero)) :: Nat`
- `Empty :: Seq a`
- `(Cons (Succ Zero) Empty) :: Seq Nat`

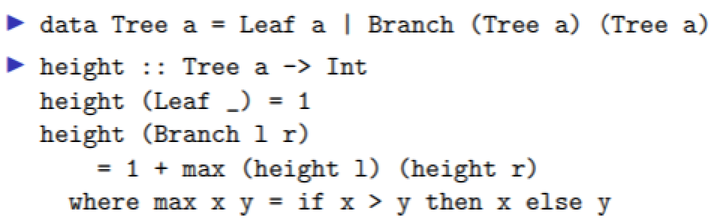
Data constructors can be used when giving definitions by pattern matching.

Example

```
isEmpty Empty = True
isEmpty (Cons x y) = False
```



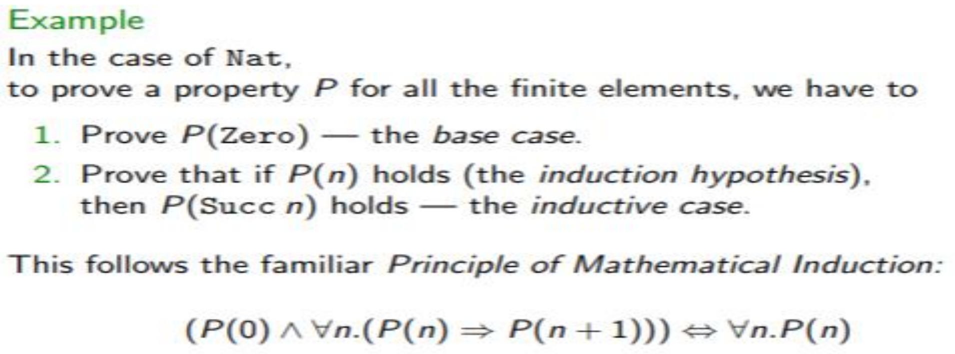
- Write the definition for a data type `Tree a` representing binary trees with data stored in the leaves. Use data constructors `Leaf` and `Branch`.
- Write a definition for a function `height t` which computes the height of a tree `t`. What is the type of `height`? (Hint: you will need a local definition of a function `max x y`.)



- `data Tree a = Leaf a | Branch (Tree a) (Tree a)`
-
```
height :: Tree a -> Int
height (Leaf _) = 1
height (Branch l r)
    = 1 + max (height l) (height r)
  where max x y = if x > y then x else y
```

INDUCTION

To reason with recursive types, we can use the Principle of Structural Induction



Example

In the case of Nat,
to prove a property P for all the finite elements, we have to

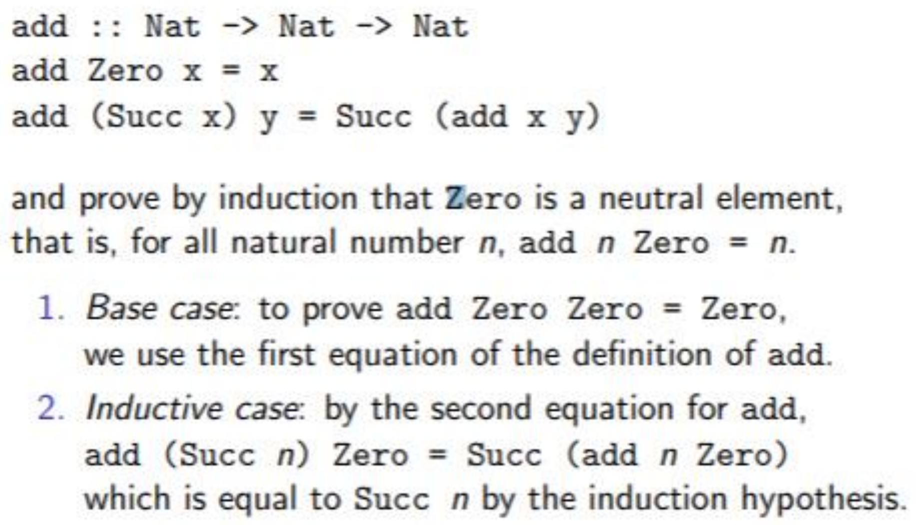
1. Prove $P(\text{Zero})$ — the *base case*.
2. Prove that if $P(n)$ holds (the *induction hypothesis*), then $P(\text{Succ}\ n)$ holds — the *inductive case*.

This follows the familiar *Principle of Mathematical Induction:*

$$(P(0) \land \forall n.(P(n) \Rightarrow P(n+1))) \Leftrightarrow \forall n.P(n)$$

EXAMPLE OF INDUCTION

We can define addition on ***Nat*** *by pattern matching*



```
add :: Nat -> Nat -> Nat
add Zero x = x
add (Succ x) y = Succ (add x y)
```

and prove by induction that Zero is a neutral element,
that is, for all natural number n, add n Zero = n.

1. *Base case*: to prove add Zero Zero = Zero,
   we use the first equation of the definition of add.
2. *Inductive case*: by the second equation for add,
   add (Succ n) Zero = Succ (add n Zero)
   which is equal to Succ n by the induction hypothesis.

We can use similar approach to pattern matching, by proving all the cases. For base cases looking at the base rules like Zero, and prove property for them as a base case, and then we have cases for each recursive data constructor, and we need to prove it inductively.