Software System Design and Implementation

Functional Programming

Gabriele Keller

The University of New South Wales School of Computer Science and Engineering Sydney, Australia



COMP3141 16s1

Course software

How to install Haskell (see 'Course Software' on COMP3141 website)

If you're using a Mac and want the Haskell for Mac IDE, please reply to my email



What is functional programming?



SWIFT AGDA Common properties of functional languages

- Functions are the main device to structure programs
- Based on the lambda calculus
- The use of higher-order functions is encouraged
- Side effects are used in a disciplined manner (pure functions)
- Sophisticated type systems (though, the Lisp family is dynamically typed)
 MAPLE
 SCHEME





Functional programming with Haskell



Haskell

- Broad-spectrum programming language
- Widely used with over 2500 open-source libraries and tools
- Haskell is a principled language
 - Purely functional
 - Strictly isolating side effects
 - Strongly typed with a surprisingly expressive type system

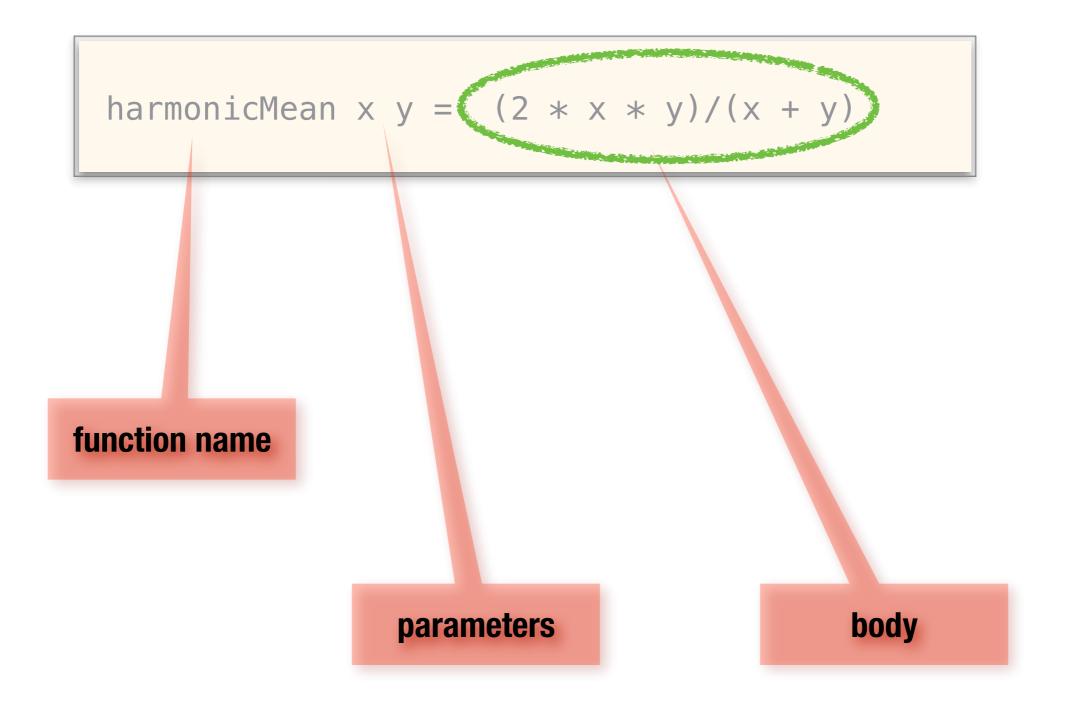
http://haskell.org/





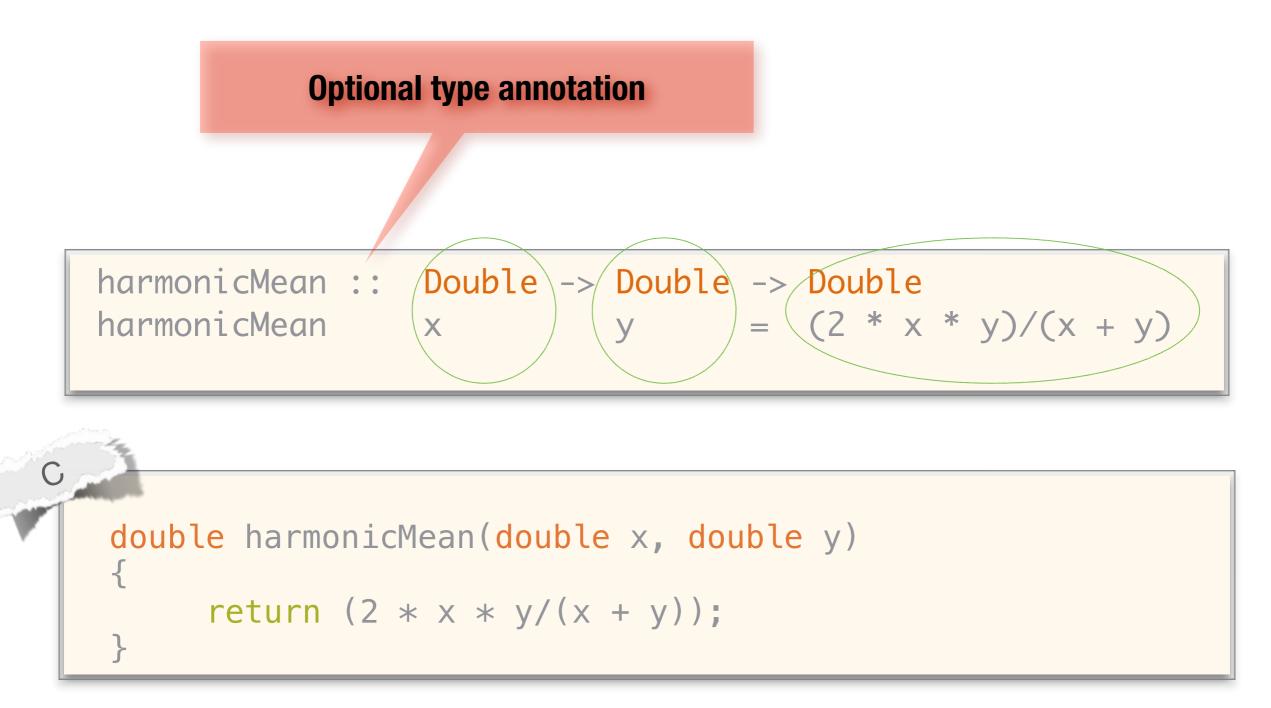
Haskell B. Curry

Simple Functions in Haskell

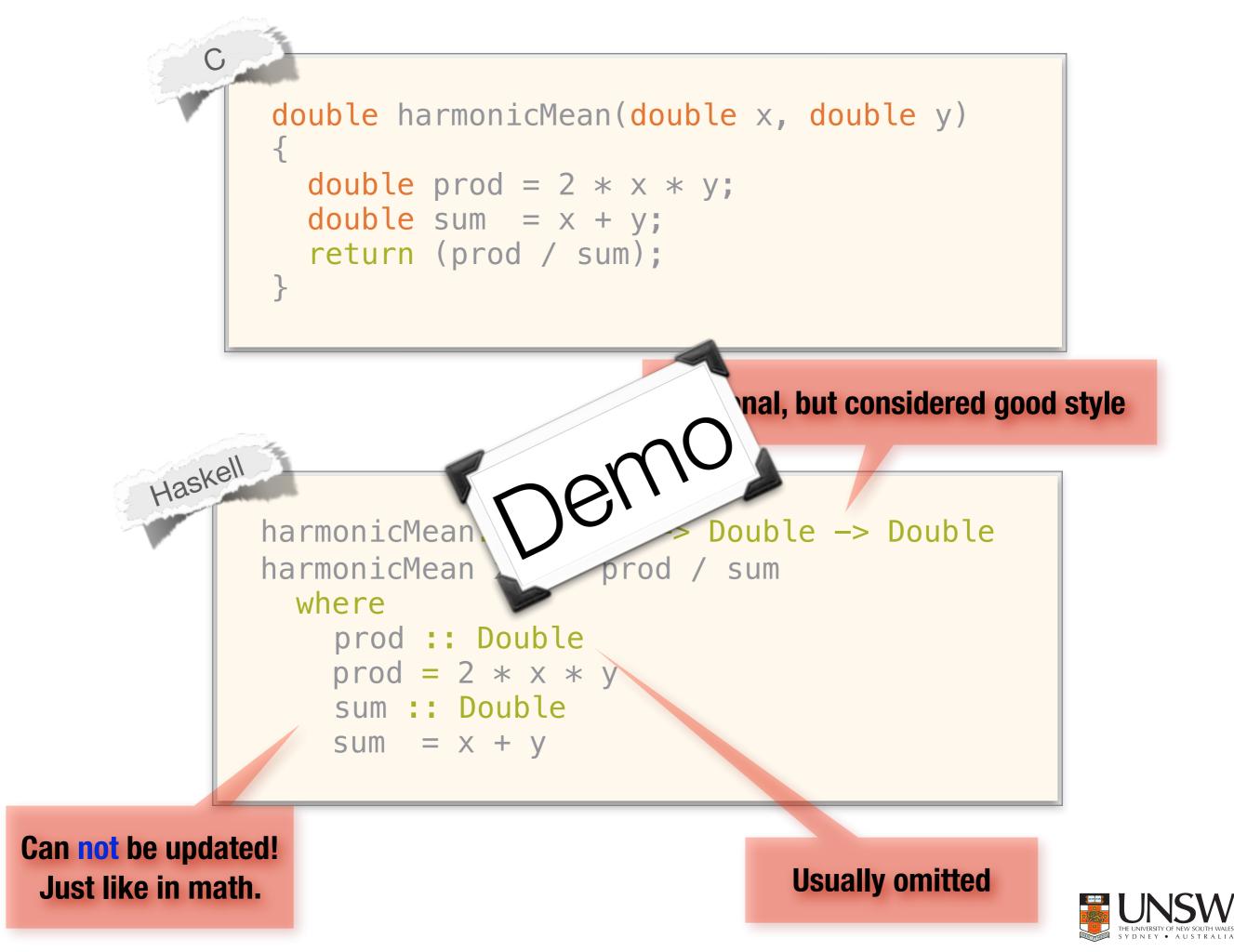




Functions in Haskell

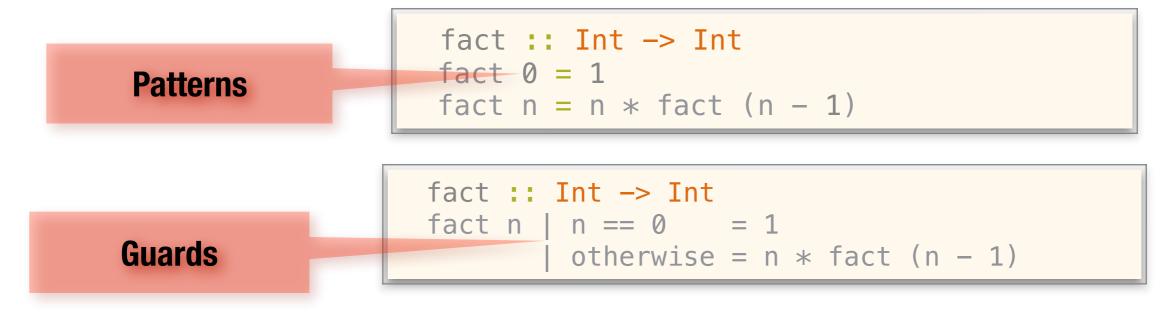






Simple functions

Selection between multiple equations is by pattern matching or guards





• Constants are simply functions with no arguments

theAnswer :: Int
theAnswer = 2 * 21



"-> " is right associative, that is

```
harmonicMean :: Double -> Double -> Double
```

is the same as

```
harmonicMean :: Double -> (Double -> Double)
```

can be interpreted as a function that takes one Double value, and returns a new function as result

harmonicMean :: Double -> Double -> Double
harmonicMean x y = (2 * x * y)/(x + y)

janesFinal :: Double -> Double
janesFinal examMark = harmonicMean 89 examMark



"-> " is right associative, that is

```
harmonicMean :: Double -> Double -> Double
```

is the same as

```
harmonicMean :: Double -> (Double -> Double)
```

can be interpreted as a function that takes one Double value, and returns a new function as result

harmonicMean :: Double -> Double -> Double
harmonicMean x y = (2 * x * y)/(x + y)

janesFinal :: Double -> Double
janesFinal examMark = harmonicMean 89 examMark

```
jebsFinal :: Double -> Double
jebsFinal = harmonicMean 75
```



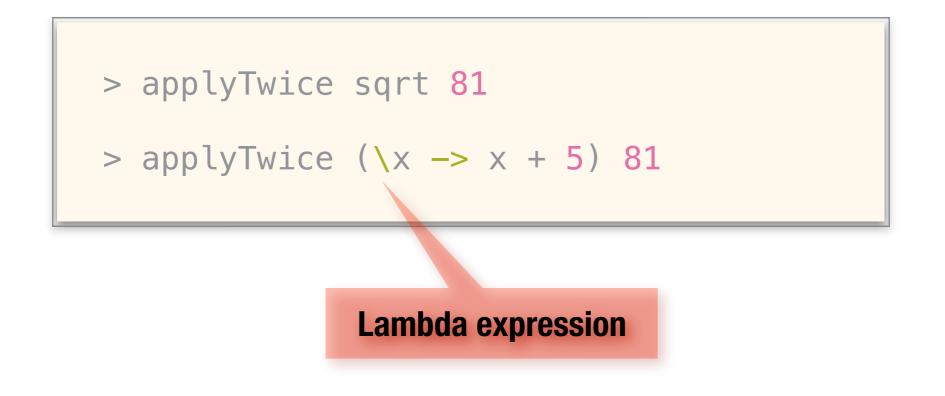
Type inference

- Compiler can work out (nearly) all types by itself
 - You can prototype, leaving out the types, and add them later
- Type signatures are documentation
 - The compiler makes sure the documentation is in sync with the code
- Type signatures catch bugs early compiler complains if they are wrong!





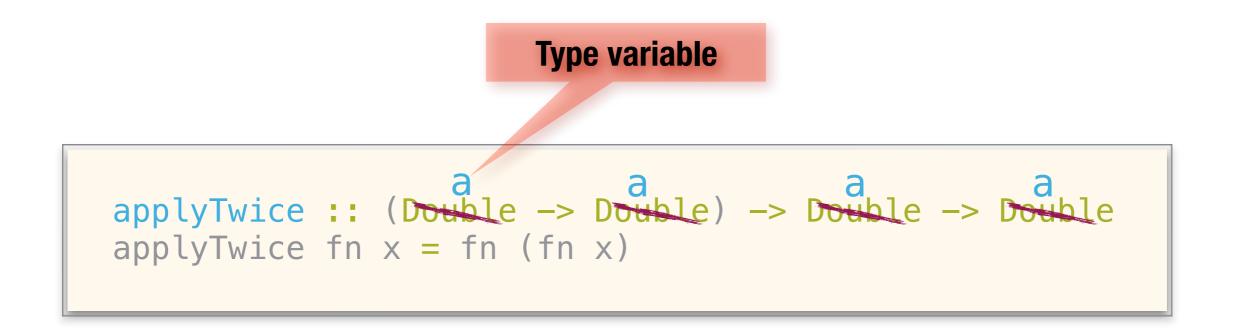
Higher-order functions





Parametric polymorphism

- Polymorphism has a different meaning in FP than in OO
- Java's generics correspond to parametric polymorphism





Types

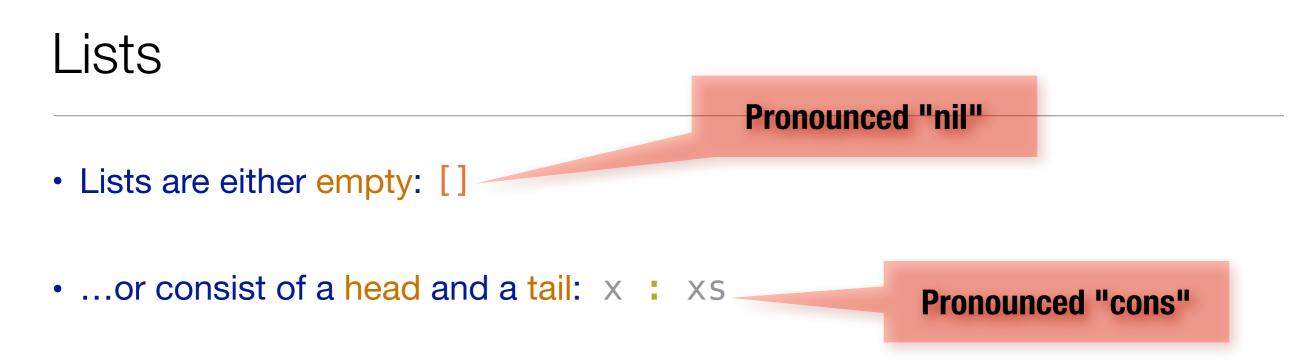


Pre-defined types

- We already saw function types: a -> b
- We also saw elementary types: Int, Float, Double, Char, and so on
- Tuples group multiple types: (), (a, b), (a, b, c), and so on

harmonicMeanT ::(Double, Double) \rightarrow DoubleharmonicMeanT(x, y)= (2 * x * y)/(x + y)

harmonicMeanT :: (Double, Double) -> Double harmonicMeanT pxy = (2 * (fst pxy) * (snd pxy))/(fst pxy) + (snd pxy)) fst :: (a, b) -> a - functions defined in Prelude snd :: (a, b) -> b



- Lists are homogenous all elements in one list have the same type
- List are parametric different lists may contain elements of different type



Some operations on lists

- Length of a list
- Concatenating two lists
- Reversing the elements of a list
- Mapping a function over a list





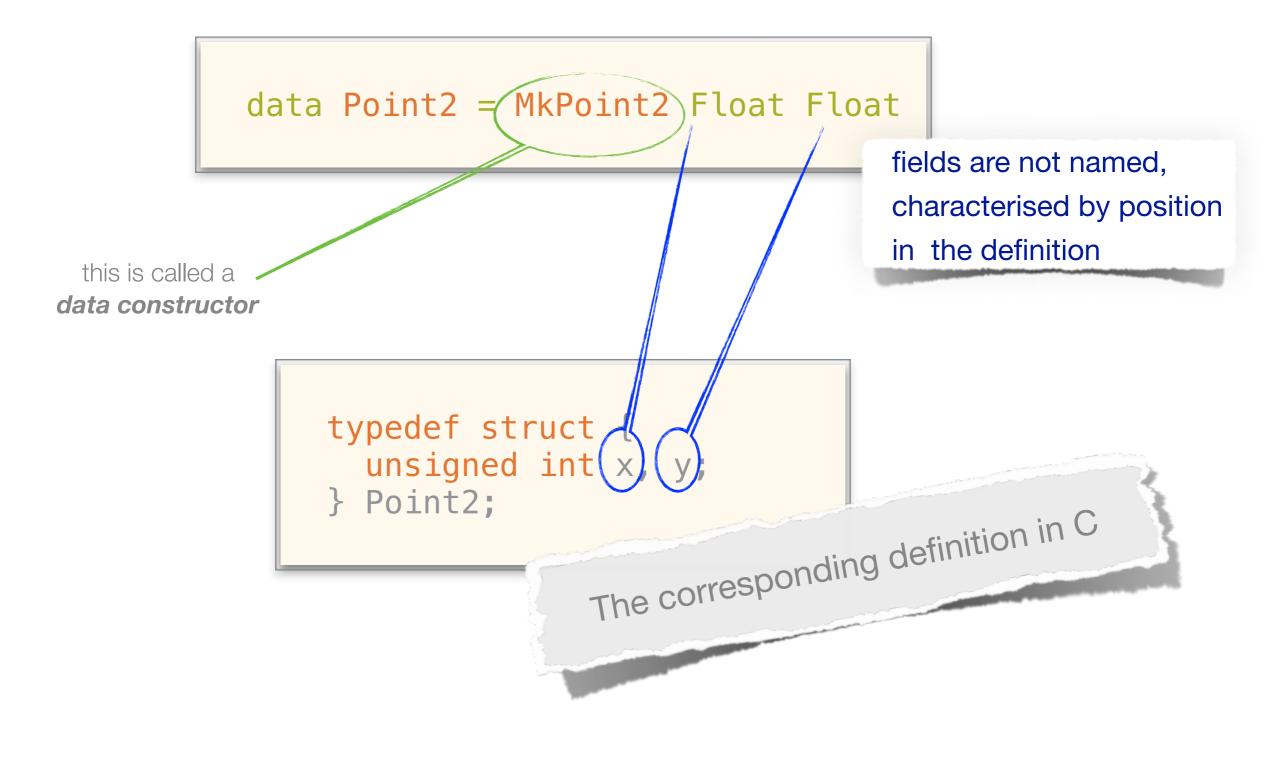
User-defined types

• Type synonyms (typedefs in C)

- Algebraic data types
 - Combination of structs and unions
 - together with pointers in C



• Data types can be like structs in C (we call those data types product types)





```
data Point2 = MkPoint2 Float Float
```

```
point :: Point2
point = MkPoint2 1.3 2.45
```

```
typedef struct {
   unsigned int x, y;
} Point2;

Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
```



```
data Point2 = MkPoint2
  { xPoint :: Float
  , yPoint :: Float
  }

point :: Point2
point = MkPoint2 1.3 2.45
- or
point = MkPoint2 {yPoint = 2.45, xPoint = 1.3}
```

```
typedef struct {
   unsigned int x, y;
} Point2;

Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
```



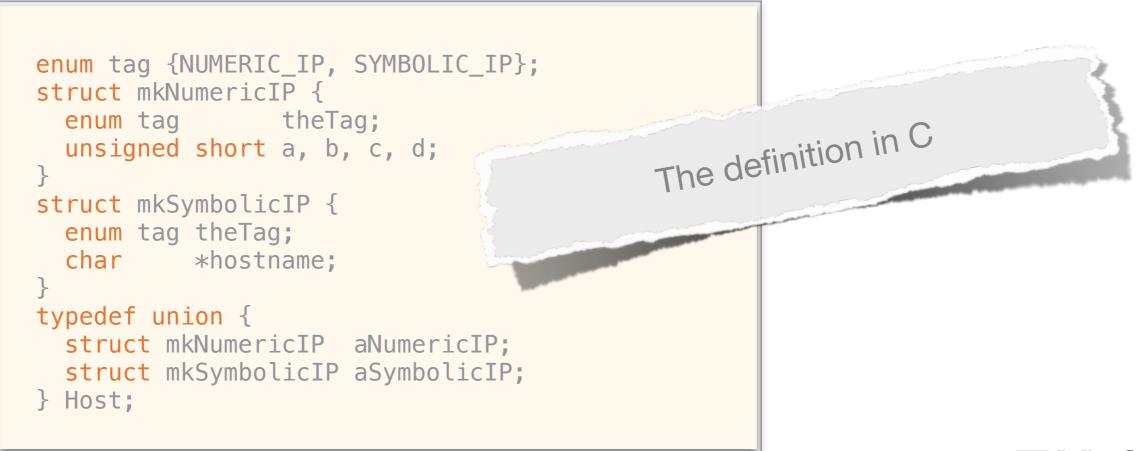
```
data Point2 = MkPoint2
  { xPoint :: Float
  , yPoint :: Float
  }

distance :: Point2 -> Point2 -> Float
distance (MkPoint2 x1 y1) (MkPoint2 x2 y2) =
  sqrt ((x2 - x1)^2 + (y2 - y1)^2)
```



- Problem: define a type to model hostnames, which can be either symbolic (string) or numeric address (4 integer values)
- Data types can be like unions in C (we call those data types sum types)

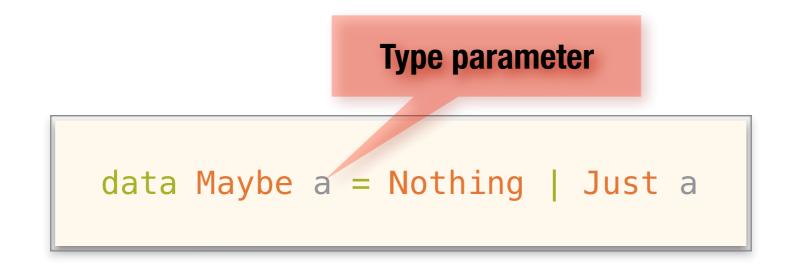






Product-Sum Types

- We call Haskell's data types also product-sum types
- They can be recursive as well
- In contrast to data types in C, but much like generics in Java and C#, Haskell data types can be parameterised





Identifiers in Haskell

- Alphanumeric with underscores (_) and prime symbols (')
- Case matters

Functions & variables	lower case	map, pi, (+), (++)
Data constructors	Upper case	True, Nothing, (:)
Type variables	lower case	a, b, c, eltType
Type constructors	Upper case	Int, Bool, IO



A larger example: Fractal trees

