

Aside: Functional vs. Imperative „Thinking“

value/data directed	state directed
<pre>sumSquares :: Int -> Int sumSquares 0 = 0 sumSquares n = n*n + sumSquares (n-1) sumSquares' :: Int -> Int sumSquares' = sum (map square [1..n])</pre>	<pre>s := 0; i := 0; while i < n do begin i := i+1; s := i*i + s; end</pre>

Result is specified **explicit**.

Result is implicit calculated during an iterative update of the global state.

1

Aside: Higher order functions as control structures

```
while :: (a -> bool) -> (a -> a) -> a -> a
while cond body state
  | cond state = while cond body (body state)
  | otherwise = state
```

`map` is a kind of `for` loop on lists.

- Higher order functions can be viewed as control structures that can be defined by the user.
- Typechecker can look „inside“ these control structures (more errors can be found)

2

Algebraic Datatypes

Algebraic data types in Haskell

Definitions follow the `data` keyword.

The range of definitions is broad, covering:

- enumerated types: "here are the elements of this type", e.g. `Bool`;
- alternatives: a `Shape` is either a circle **or** a square;
- composites: a `Rectangle` has a height **and** width;
- dynamics: lists, ...
- structures like trees: e.g. represent programs.

4

Polymorphism etc.

We will be able to define **generic** or **polymorphic** types, like the built-in list type `[a]`.

The system can **derive** a whole lot of definitions over **data** type for free if we ask it to.

We look at **data** types through a series of examples.

5

Enumerated types

```
data Temp = Cold | Hot
data Season = Spring | Summer | Autumn | Winter
```

These definitions introduce two new types

- `Temp` has two members, `Cold` and `Hot`,
- `Season` has four members, `Spring`, etc.

The elements (`Cold` etc.) begin with capitals. The different alternatives are separated by `|`.

6

Defining functions over these types

Use **pattern matching** to give definitions over these types.

To model the English weather

```
weather :: Season -> Temp
weather Summer = Hot
weather _      = Cold
```

This is a **wildcard** pattern.
It matches anything.

7

Other examples ...

```
show :: Season -> String
show Spring = "Spring"
show Summer = "Summer"
show Autumn = "Autumn"
show Winter = "Winter"
```

But we don't want to do this ... instead write

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

8

Deriving

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

This automatically gives definitions for all the functions in the **classes** (interfaces) listed. E.g.

```
show :: Season -> String           from Show
(==) :: Season -> Season -> Bool  from Eq
e.g. [Summer .. Winter]          from Enum
```

If in doubt, include the **deriving** part.

9

The Boolean type

The Booleans in Haskell are defined in this way

```
data Bool = False | True
          deriving (Show,Eq,Ord,Enum,Read)
```

that helps to explain why **False** and **True** have to begin with capital letters.

10

Adding data ...

The next example has just one alternative ... but this time it contains some data:

```
data People = Person String Int
```

Members of this type all look like this

```
Person "Madonna" 41
Person "Margaret" 83
Person "Electric Aunt Jemima" 77
```

Like a tuple type ... but say 'Person' explicitly.

11

How to form a member of People?

A member of **People** is formed simply by writing **Person** applied to a **String** and an **Int**, as in

```
Person "Ronnie" 14
```

Person is a **constructor function** for this type:

```
Person :: String -> Int -> People
```

It is used like any other function ... but we can also use it in patterns...

12

Defining functions over `People`

```
showPerson :: People -> String
showPerson (Person st n) = st ++ " -- " ++ show n

showPerson (Person "Ronnie" 14)
-> "Ronnie" ++ " -- " ++ show 14
-> "Ronnie" ++ " -- " ++ "14"
-> "Ronnie -- 14"
```

Pattern matching is used here to **extract the components** of the data value.

13

Algebraic types v. tuples

```
data People = Person String Int
type People = (String,Int)
```

- The data explicitly labelled `Person`.
- Can't accidentally treat something else as a person.
- You'll see `Person` in any error messages.
- More succinct.
- Can reuse polymorphic functions defined over pairs, e.g. `fst`, `snd`.

14

Alternatives and data

These types really become useful when we combine alternatives with data...

Example: a `shape` is either

- a `circle`, which has a radius, or
- a `rectangle`, which has a height and a width.

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

15

The `Shape` type

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

The type contains two different kinds of data:

- circles, like

```
Circle 2.3
Circle 31.0
```

- and rectangles, like

```
Rectangle 21.3 4.6
Rectangle 3 4
```

16

Functions over `Shape`

```
isRound :: Shape -> Bool
isRound (Circle _) = True
isRound (Rectangle _ _) = False
```

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

```
makeSquare :: Float -> Shape
makeSquare x = Rectangle x x
```

17

Maybe there's an element ...

Suppose we try to do something, but it might not produce a result. Examples

- Take the head of a list - what if it's empty?
- Divide two numbers - what if divisor is zero?

A type to represent that maybe there's an element of type `a`, or perhaps nothing.

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

18

The Maybe type

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

This defines a polymorphic type, so defines `Maybe Bool`, `Maybe Int`, and so forth.

```
sHead :: [a] -> Maybe a
sHead (x:xs) = Just x
sHead []     = Nothing
```

When `sHead` is used, need to check whether there's an answer returned...

19

Functions over Maybe

```
checkShow :: Maybe a -> String
```

```
checkShow (Just x)
  = "The datum is" ++ show x
```

```
checkShow Nothing
  = "No data"
```

20

data: The general format

```
data t = C1 t1 ... tn | ... | Cn ...
```

Declares **constructors**

```
C1 :: t1 -> ... -> tn -> t
...
Cn :: ... -> t
```

21

Pattern matching

A **pattern** is any term consisting only of

- **constructors** (incl. tuples)
- **variables** (incl. `_`)
- **literals**

Example: `Just(Just True, "123") :: Maybe(Maybe Bool, String)`

22

The case construct

Example:

```
fac n = case n of
  0 -> 1
  n -> n * fac(n-1)
```

General form:

```
case expr of
  pattern -> expr
  ...
  pattern -> expr
```

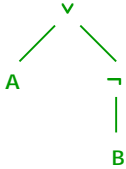
23

Recursive data types

Modelling: logical formulas

Formulae of propositional logic:

D $(A \vee \neg B)$ $(A \wedge (B \wedge (\neg C \vee D)))$



25

Logical formulas as data

- An atomic proposition: A, B, C, \dots is a formula.
- If X, Y are formulas, so are $X \wedge Y, X \vee Y, \neg X$.

```
data Atom = A | B | C | D | E | F
```

```
data Form = Atm Atom |
           And Form Form |
           Or Form Form |
           Not Form
```

26

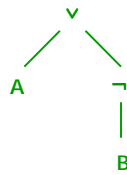
Examples of logical formulas

$(A \vee \neg B)$

Or (Atm A) (Not (Atm B))

```
data Atom = A | B | C | D | E | F
```

```
data Form = Atm Atom |
           And Form Form |
           Or Form Form |
           Not Form
```



27

Functions over Form

How many logical operators are there in a formula?

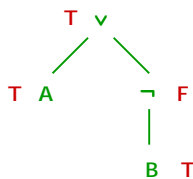
```
countOps :: Form -> Int
```

```
countOps (Atm _) = 0
countOps (And x y) = 1 + countOps x + countOps y
countOps (Or x y) = 1 + countOps x + countOps y
countOps (Not x) = 1 + countOps x
```

28

Evaluating logical formulas

$(A \vee \neg B)$ when A and B both true?



29

Implementing the evaluator ...

```
eval :: Form -> Bool
```

```
eval (And x y) = eval x && eval y
eval (Or x y) = eval x || eval y
eval (Not x) = not (eval x)
eval (Atm a) = ???
```

We need to get the values of the atomic formulas from somewhere ...

The symbols \wedge etc. get interpreted by the corresponding operations $\&\&$ etc. over `Bool`.

30

Valuations

```
type Valu = [(Atom,Bool)]
```

Example: `valu1 = [(A,True),(B,True)]`

```
lookup :: Atom -> Valu -> Bool
```

```
lookup a ((x,b):val)
  | a==x      = b
  | otherwise  = lookup a val
```

```
lookup a [] = error "lookup"++ show a ++"failed"
```

31

The Form evaluator in full

```
eval :: Valu -> Form -> Bool
```

```
eval v (And x y) = eval v x && eval v y
eval v (Or x y)  = eval v x || eval v y
eval v (Not x)   = not (eval v x)
eval v (Atm a)   = lookup a v
```

```
eval valu1 (Or (Atm A) (Not (Atm B)))
~> ... ~> True
```

32

Example: lists, revisited

A list is either empty, `[]` ...
or consists of an element stuck on the front of
another list, `(x:xs)`.

We can define a list type as a `data` type:

```
data List a = Empty | Cons a (List a)
```

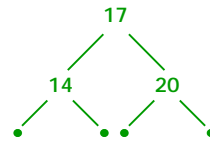
this is a recursive definition.

`Empty` represents `[]` and `(Cons x xs)` is `(x:xs)`.

33

Trees of numbers

A tree is either a leaf, or is a node, which contains a
number and two subtrees:



34

Trees of numbers

A tree is either a leaf, or is a node, which contains a
number and two subtrees:

```
data NTree = NilT |
            Node Int NTree NTree
```

```
Node 17 (Node 14 NilT NilT)
        (Node 20 NilT NilT)
```

35

Example: depth of a tree

```
data NTree = NilT | Node Int NTree NTree
```

```
depth :: NTree -> Int
```

An empty tree has depth zero ...

```
depth NilT = 0
```

otherwise, find the deeper of the two sub-trees and
add one to that depth ...

```
depth (Node n t1 t2)
  = 1 + max (depth t1) (depth t2)
```

36

Example: add the values in a tree

```
data NTree = NilT | Node Int NTree NTree
```

```
sumT :: NTree -> Int
```

An empty tree has sum zero ...

```
sumT NilT = 0
```

otherwise, sum the two sub-trees and them to the value at the node ...

```
sumT (Node n t1 t2)
  = n + sumT t1 + sumT t2
```

37

Example: extract the values

```
data NTree = NilT | Node Int NTree NTree
```

```
flatten :: NTree -> [Int]
```

An empty tree has no elements ...

```
flatten NilT = []
```

otherwise, flatten the two sub-trees and include the value at the node ...

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

38

Example: does a value occur?

```
data NTree = NilT | Node Int NTree NTree
```

```
elemT :: Int -> NTree -> Bool
```

An empty tree has no elements ...

```
elemT x NilT = False
```

otherwise, check the value at the node and in the two sub-trees ...

```
elemT x (Node n t1 t2)
  = n==x || elemT x t1 || elemT x t2
```

39

Polymorphic trees

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

Functions defined earlier still work; types changed:

```
depth    :: Tree a -> Int
```

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

```
sumT     :: Tree Int -> Int
```

40