Recursive Datatypes and Lists

Types and constructors

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

Interpretation:

"Here is a new type Suit. This type has four possible values: Spades, Hearts, Diamonds and Clubs."

Types and constructors

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

This definition introduces five things:

- The type Suit
- The constructors

```
Spades    :: Suit
Hearts    :: Suit
Diamonds  :: Suit
Clubs     :: Suit
```

Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

Interpretation:

"Here is a new type Rank. Values of this type have five possible possible forms: Numeric n, Jack, Queen, King or Ace, where n is a value of type Integer"

Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

This definition introduces six things:

- The type Rank
- The constructors

```
Numeric   :: ???
Jack      :: ???
Queen     :: ???
King      :: ???
Ace       :: ???
```

Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

This definition introduces six things:

- The type Rank
- The constructors

Numeric	:: Integer → Rank
Jack	:: ???
Queen	:: ???
King	:: ???
Ace	:: ???

Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

This definition introduces six things:

- The type Rank
- The constructors

Numeric	:: Integer → Rank
Jack	:: Rank
Queen	:: Rank
King	:: Rank
Ace	:: Rank

Types and constructors

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

Type

Constructor

Type

Types and constructors

```
data Card = Card Rank Suit
```

Interpretation:

"Here is a new type Card. Values of this type have the form Card r s, where r and s are values of type Rank and Suit respectively."

Types and constructors

```
data Card = Card Rank Suit
```

This definition introduces two things:

- The type Card
- The constructor

  Card :: ???

Types and constructors

```
data Card = Card Rank Suit
```

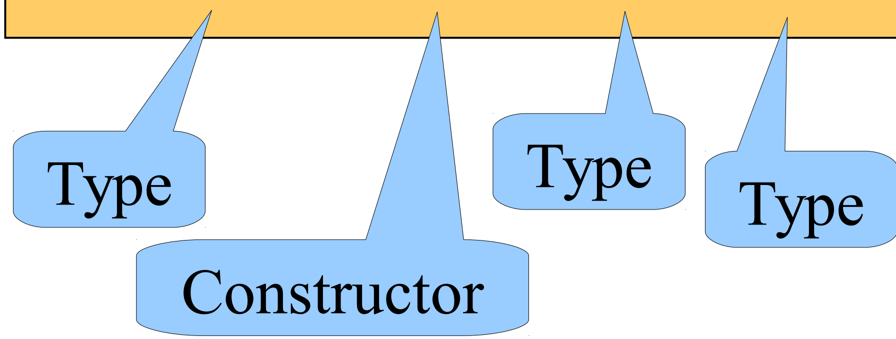
This definition introduces two things:

- The type Card
- The constructor

  Card :: Rank → Suit → Card

Types and constructors

```
data Card = Card Rank Suit
```



Types and constructors

```
data Hand = Empty | Add Card Hand
```

Interpretation:

"Here is a new type Hand. Values of this type have two possible forms: Empty or Add c h where c and h are of type Card and Hand respectively."

Types and constructors

```
data Hand = Empty | Add Card Hand
```

Alternative interpretation:

“A hand is either empty or consists of a card on top of a smaller hand.”

Types and constructors

```
data Hand = Empty | Add Card Hand
```

This definition introduces three things:

- The type Hand
- The constructors

```
Empty    :: ???
Add      :: ???
```

Types and constructors

```
data Hand = Empty | Add Card Hand
```

This definition introduces three things:

- The type Hand
- The constructors

```
Empty     :: Hand
Add       :: ???
```

Types and constructors

```
data Hand = Empty | Add Card Hand
```

This definition introduces three things:

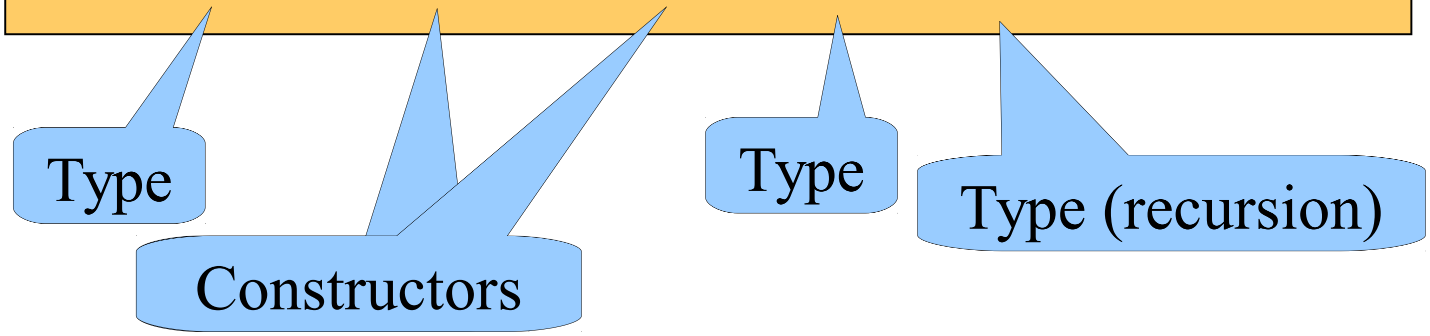
- The type Hand
- The constructors

  Empty :: Hand

  Add :: Card → Hand → Hand

Types and constructors

```
data Hand = Empty | Add Card Hand
```



Pattern matching

Define functions by stating their results for all possible forms of the input

```
size :: Hand → Integer
```

Pattern matching

Define functions by stating their results for all possible forms of the input

```
size :: Hand → Integer
size Empty              = 0
size (Add card hand)    = 1 + size hand
```

Interpretation:

"If the argument is Empty, then the result is 0. If the argument consists of a card card on top of a hand hand, then the result is 1 + the size of the rest of the hand."

Pattern matching

```
size :: Hand → Integer
size Empty                  = 0
size (Add card hand)        = 1 + size hand
```

Patterns have two purposes:

1. Distinguish between forms of the input (e.g. Empty and Add)
2. Give names to parts of the input (In the definition of size, card is the first card in the argument, and hand is the rest of the hand.)

Pattern matching

```
size :: Hand → Integer
size Empty                  = 0
size (Add kort resten)      = 1 + size resten
```

Variables can have arbitrary names

Construction/destruction

When used in an expression (RHS), Add *constructs* a hand:

```
aHand :: Hand
aHand = Add c1 (Add c2 Empty)
```

When used in a pattern (LHS), Add *destructs* a hand:

```
size (Add card hand) = …
```

Lists
– how they work

Lists of arbitrary type

```
data List = Empty | Add ?? List
```

- Can we generalize the Hand type to lists with elements of arbitrary type?
- What to put on the place of the ??

Lists of arbitrary type

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

A parameterized type

Constructors:

```
Empty    :: ???
Add      :: ???
```

Lists of arbitrary type

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

A parameterized type

Constructors:

```
Empty    :: List a
Add      :: ???
```

Lists of arbitrary type

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

A parameterized type

Constructors:

Empty	:: List a
Add	:: a → List a → List a

Constructing lists

- List containing the numbers 1, 2 and 3:

```
myList1 :: List Integer
myList1 = Add 1 (Add 2 (Add 3 Empty))
```

- List containing the strings “apa”, “hund”:

```
myList2 :: List String
myList2 = Add “apa” (Add “hund” Empty)
```

Constructing lists

- Cannot mix elements of different types:

  myList3 = Add True (Add “bil” Empty)

  Error: Couldn't match expected type ‘Bool’ with actual type ‘[Char]’

Lists of arbitrary type

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

A parameterized type

Constructors:

Empty	:: List a
Add	:: a → List a → List a

Built-in lists

The List type is just for demonstration, Haskell has an *equivalent* built-in list type that should be used instead:

List a ≈ [a]

Built-in lists

```
data [a] = [] | (:) a [a]
```

Not a legal definition, but the built-in lists are *conceptually* defined like this

Constructors:

[] :: [a]

(:) :: a → [a] → [a]

Built-in lists

Instead of

```
Add 1 (Add 2 (Add 3 Empty))
```

we can use built-in lists and write

```
(:) 1 ((:) 2 ((:) 3 []))
```

or, equivalently

```
1 : 2 : 3 : []
```

or, equivalently

```
[1,2,3]
```

Special syntax for the built-in lists

Some list operations

- From the Data.List module (also in the Prelude):

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

take           :: Int -> [a] -> [a]
  -- (take n) picks the first n elements

(++)           :: [a] -> [a] -> [a]
  -- append a list after another

replicate      :: Int -> a -> [a]
  -- make a list by replicating an element
```

Some list operations

```
*Main> reverse [1,2,3]
[3,2,1]

*Main> take 4 [1..10]
[1,2,3,4]

*Main> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]

*Main> replicate 5 2
[2,2,2,2,2]
```

Strings are lists of characters

```
type String = [Char]

Prelude> 'g' : "apa"
"gapa"

Prelude> "flyg" ++ "plan"
"flygplan"

Prelude> ['A','p','a']
"Apa"
```

Type synonym definition

More on Types

- Functions can have "general" types:
  - *polymorphism*
  - reverse :: [a] → [a]
  - (:)   :: a → [a] → [a]
- Sometimes, these types can be restricted
  - Ord a => … for comparisons (<, <=, >, >=, …)
  - Eq a => … for equality (==, /=)
  - Num a => … for numeric operations (+, -, *, …)

Do's and Don'ts

```
isBig :: Integer → Bool
isBig n | n > 9999  = True
        | otherwise = False
```

guards and boolean results

```
isBig :: Integer → Bool
isBig n = n > 9999
```

Do's and Don'ts

```
resultIsSmall :: Integer → Bool
resultIsSmall n = isSmall (f n) == True
```

comparison with a boolean constant

```
resultIsSmall :: Integer → Bool
resultIsSmall n = isSmall (f n)
```

Do’s and Don’ts

```
resultIsBig :: Integer → Bool
resultIsBig n = isSmall (f n) == False
```

comparison with a boolean constant

```
resultIsBig :: Integer → Bool
resultIsBig n = not (isSmall (f n))
```

nd Don’ts

Do not make unnecessary case distinctions

necessary case distinction?

```
fun1 :: [Integer] → Bool
fun1 []       = False
fun1 (x:xs) = length (x:xs) == 10
```

repeated code

```
fun1 :: [Integer] → Bool
fun1 xs = length xs == 10
```

…s and Don’ts

Make the base case as simple as possible

```
…n2 :: […ger] → Integer
fun2 [x]      = calc x
fun2 (x:xs) = calc x + fun2 xs
```

right base case ?

repeated code

```
fun2 :: [Integer] → Integer
fun2 []       = 0
fun2 (x:xs) = calc x + fun2 xs
```