Lecture 3 – Types and Classes

Violet Ka I Pun

violet@ifi.uio.no

Types

Types

- A type represents a number of values

Types

- A type represents a number of values
- Examples:
  - integers `...,-2,-1,0,1,2,...`, type `Int`
  - pairs of integers, type `(Int,Int)`
  - lists of pair of integers, type `[(Int,Int)]`
  - functions from integers to integers, type `Int -> Int`
  - user-defined types as `data Mybool = Usann | Sann`

Types

- A type represents a number of values
- Examples:
  - integers `...,-2,-1,0,1,2,...`, type `Int`
  - pairs of integers, type `(Int,Int)`
  - lists of pair of integers, type `[(Int,Int)]`
  - functions from integers to integers, type `Int -> Int`
  - user-defined types as `data Mybool = Usann | Sann`
- Each Haskell expression has a type: `<expr> :: <type>`

Types

- A type represents a number of values
- Examples:
  - integers `...,-2,-1,0,1,2,...`, type `Int`
  - pairs of integers, type `(Int,Int)`
  - lists of pair of integers, type `[(Int,Int)]`
  - functions from integers to integers, type `Int -> Int`
  - user-defined types as `data Mybool = Usann | Sann`
- Each Haskell expression has a type: <`expr`> `::` <`type`>

  > `:type expr` (or `:t expr`) returns the type in GHCi

Types

- A type represents a number of values
- Examples:
  - integers `...,-2,-1,0,1,2,...`, type `Int`
  - pairs of integers, type `(Int,Int)`
  - lists of pair of integers, type `[(Int,Int)]`
  - functions from integers to integers, type `Int -> Int`
  - user-defined types as `data Mybool = Usann | Sann`
- Each Haskell expression has a type: `<expr> :: <type>`

  $>$ `:type expr` (or `:t expr`) returns the type in GHCi
- Type does not change during evaluation

Type Errors

Apply a function to one or more arguments of the wrong type, e.g.,

```
> 1 + False
ERROR
```

Type Errors

Apply a function to one or more arguments of the wrong type, e.g.,

```
> 1 + False
ERROR
```

- + is supposed to add two numbers, but `False` is a logical value

Type Errors

Apply a function to one or more arguments of the wrong type, e.g.,

```
> 1 + False
ERROR
```

- + is supposed to add two numbers, but `False` is a logical value

All type errors are found at compile time

- makes programs safer and faster
  by removing the need for type checks at run time.

Basic types

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`

Basic types

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
- boolean values: `True,False`, ... type `Bool`

Basic types

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
- boolean values: `True,False`, ... type `Bool`
- symbols: `...,'A',...'z',...`, ... type `Char`

Basic types

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
- boolean values: `True,False`, ... type `Bool`
- symbols: `...,'A',...'z',...`, ... type `Char`
- strings: `‘‘abc’’`, ... type `String`

Basic types

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
- boolean values: `True,False`, ... type `Bool`
- symbols: `...,'A',...'z',...`, ... type `Char`
- strings: `''abc''`, ... type `String`
- fractions: `1.2, 1.234` .... type `Float`

Basic types, each with a set of operators

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
  `+, -, *, div, mod`
- boolean values: `True,False`, ... type `Bool`
- symbols: `...,'A',...'z',...`, ... type `Char`
- strings: `''abc''`, ... type `String`
- fractions: `1.2, 1.234` .... type `Float`

Basic types, each with a set of operators

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
  `+, -, *, div, mod`
- boolean values: `True,False`, ... type `Bool`
  `not, ||, &&, or, and`
- symbols: `...,'A',...'z',...`, ... type `Char`
- strings: `''abc''`, ... type `String`
- fractions: `1.2, 1.234` .... type `Float`

Basic types, each with a set of operators

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
  `+, -, *, div, mod`
- boolean values: `True,False`, ... type `Bool`
  `not, ||, &&, or, and`
- symbols: `...,'A',...'z',...`, ... type `Char`
  `toUpper, toLower` in Data.Char
- strings: `‘‘abc’’`, ... type `String`
- fractions: `1.2, 1.234` .... type `Float`

Basic types, each with a set of operators

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
  `+, -, *, div, mod`
- boolean values: `True,False`, ... type `Bool`
  `not, ||, &&, or, and`
- symbols: `...,'A',...'z',...`, ... type `Char`
  `toUpper, toLower` in Data.Char
- strings: ``‘‘abc’’``, ... type `String`
  `++, head, tail`
- fractions: `1.2, 1.234` .... type `Float`

Basic types, each with a set of operators

- integers: `...,-2,-1,0,1,2,...`, ... type `Int` and `Integer`
  `+, -, *, div, mod`
- boolean values: `True,False`, ... type `Bool`
  `not, ||, &&, or, and`
- symbols: `...,'A',...'z',...`, ... type `Char`
  `toUpper, toLower` in Data.Char
- strings: `''abc''`, ... type `String`
  `++, head, tail`
- fractions: `1.2, 1.234` .... type `Float`
  `+, -, *, /`

More operators

Basic types come also with operators that return values of another types

- `Float`:
  `==,<,... :: (Float,Float) -> Bool`
- `Int`:
  `==,<,... :: (Int, Int) -> Bool`
- `String`:
  `length :: [a] -> Int`,
  `==,<,...:: (String,String) -> Bool`

More operators

Basic types come also with operators that return values of another types

- `Float`:
  `==,<,... :: (Float,Float) -> Bool`
- `Int`:
  `==,<,... :: (Int, Int) -> Bool`
- `String`:
  `length :: [a] -> Int`,
  `==,<,...:: (String,String) -> Bool`
- `Char`:
  `ord :: Char -> Int`, `chr :: Int -> Char`,
  `isDigit :: Char -> Bool`

Types and Type-operators

- Lists: `[typeE]`
- Product (Cartesian): `(typeL,typeR)`
- Functions: `typeK -> typeM`

Types and Type-operators

- Lists: `[typeE]`
- Product (Cartesian): `(typeL,typeR)`
- Functions: `typeK -> typeM`
- Syntax for types:
  `T ::= name | [T] | ({T,} T, T) | T->T | (T->T)`

Types and Type-operators

- Lists: `[typeE]`
- Product (Cartesian): `(typeL,typeR)`
- Functions: `typeK -> typeM`
- Syntax for types:
  `T ::= name | [T] | ({T,} T, T) | T->T | (T->T)`
- Unique AST for `a -> b -> c` ?

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
```

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
```

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
           [1,2,3]  ::  [Int]
```

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
           [1,2,3]  ::  [Int]
     [3.5,1.4,7.8]  ::  [Float]
```

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
           [1,2,3]  ::  [Int]
     [3.5,1.4,7.8]  ::  [Float]
         [5,6,9.2]  ::  [Float]
```

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
           [1,2,3]  ::  [Int]
     [3.5,1.4,7.8]  ::  [Float]
         [5,6,9.2]  ::  [Float]
```

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
           [1,2,3]  ::  [Int]
   [3.5,1.4,7.8]    ::  [Float]
       [5,6,9.2]    ::  [Float]
```

- The type of the elements are unrestricted

Lists

A sequence of values of the same type

```
[False,True,False]  ::  [Bool]
 ['a','b','c','d']  ::  [Char]
           [1,2,3]  ::  [Int]
     [3.5,1.4,7.8]  ::  [Float]
        [5,6,9.2]   ::  [Float]
```

- The type of the elements are unrestricted
- What is the size of a list of type `[Int]`?

Lists

- Given `typeE` we have a type of lists `[typeE]`

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and

  for `x::typeE, l::[typeE]`, becomes `x:l :: [typeE]`

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE, l::[typeE]`, becomes `x:l :: [typeE]`

[1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE, l::[typeE]`, becomes `x:l :: [typeE]`
  [1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]
- **Destruction**: `head (x:l) = x`   `tail (x:l) = l`

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE, l::[typeE]`, becomes `x:l :: [typeE]`
  [1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]
- **Destruction**: `head (x:l) = x`   `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE`, `l::[typeE]`, becomes `x:l :: [typeE]`
  [1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]
- **Destruction**: `head (x:l) = x`   `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)
- Try: **let** e=1 **in** `[e]==[1]`, **let** e=12 **in** `[e]==[1,2]`

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE`, `l::[typeE]`, becomes `x:l :: [typeE]`
  $[1, 1.2] = 1 : 1.2 : [] :: (\text{Fractional t}) => [\ \text{t}\ ]$
- **Destruction**: `head (x:l) = x` `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)
- Try: **let** `e=1` **in** `[e]==[1]`, **let** `e=12` **in** `[e]==[1,2]`
- Alternative notation
  `[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]`

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE`, `l::[typeE]`, becomes `x:l :: [typeE]`
  $[1, 1.2] = 1 : 1.2 : [] :: (\text{Fractional } t) => [\ t\ ]$
- **Destruction**: `head (x:l) = x`   `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)
- Try: **let** `e=1` **in** `[e]==[1]`, **let** `e=12` **in** `[e]==[1,2]`
- Alternative notation
  `[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]`
- Equality: if `typeE` has ==,
  then `[typeE]` has ==,
  namely: same length and element-wise equality

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE, l::[typeE]`, becomes `x:l :: [typeE]`
  [1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]
- **Destruction**: `head (x:l) = x` `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)
- Try: **let** e=1 **in** `[e]==[1]`, **let** e=12 **in** `[e]==[1,2]`
- Alternative notation
  `[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]`
- Equality: if `typeE` has `==`,
  then `[typeE]` has `==`,
  namely: same length and element-wise equality
- If `typeE` is an instance of `Show`, so is `[typeE]`.

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE, l::[typeE]`, becomes `x:l :: [typeE]`
  [1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]
- **Destruction**: `head (x:l) = x` `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)
- Try: **let** e=1 **in** `[e]==[1]`, **let** e=12 **in** `[e]==[1,2]`
- Alternative notation
  `[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]`
- Equality: if `typeE` has `==`,
  then `[typeE]` has `==`,
  namely: same length and element-wise equality
- If `typeE` is an instance of `Show`, so is `[typeE]`.
- If `typeE` is an instance of `Ord`, so is `[typeE]`.

Lists

- Given `typeE` we have a type of lists `[typeE]`
- **Construction**: `[]` and
  for `x::typeE`, `l::[typeE]`, becomes `x:l :: [typeE]`
  [1, 1.2] = 1 : 1.2 : [] :: (Fractional t) => [t]
- **Destruction**: `head (x:l) = x` `tail (x:l) = l`
- Length function: `length : [a] -> Int` (polymorphic!)
- Try: **let** e=1 **in** `[e]==[1]`, **let** e=12 **in** `[e]==[1,2]`
- Alternative notation
  `[1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]`
- Equality: if `typeE` has `==`,
  then `[typeE]` has `==`,
  namely: same length and element-wise equality
- If `typeE` is an instance of `Show`, so is `[typeE]`.
- If `typeE` is an instance of `Ord`, so is `[typeE]`.

`Show` and `Ord` are type classes

Cartesian Product (pairs/tuples)

A sequence of values of different types:

Cartesian Product (pairs/tuples)

A sequence of values of different types:

```
(False,True)  ::  (Bool,Bool)
```
(pair)

Cartesian Product (pairs/tuples)

A sequence of values of different types:

```
      (False,True)  ::  (Bool,Bool)            (pair)
 (``Yes'','a',True)  ::  (String,Char,Bool)    (triple)
```

Cartesian Product (pairs/tuples)

A sequence of values of different types:

```
       (False,True)  ::  (Bool,Bool)              (pair)
  (``Yes'','a',True)  ::  (String,Char,Bool)      (triple)
```

$(\texttt{t}_1,\ \texttt{t}_2,\ \ldots\texttt{t}_n)$ (n-tuple)

Cartesian Product (pairs/tuples)

A sequence of values of different types:

`(False,True)`	`::`	`(Bool,Bool)`	(pair)
`(''Yes'','a',True)`	`::`	`(String,Char,Bool)`	(triple)
$(t_1,\ t_2,\ \ldots t_n)$			(n-tuple)
`(t)`			not permitted

A sequence of values of different types:

```
      (False,True)  ::  (Bool,Bool)          (pair)
(``Yes'','a',True)  ::  (String,Char,Bool)   (triple)
             (t1, t2, ...tn)                 (n-tuple)
                 (t)                         not permitted
```

- The number of components in a tuple is called its arity

Cartesian Product (pairs/tuples)

A sequence of values of different types:

```
       (False,True)  ::  (Bool,Bool)            (pair)
(''Yes'','a',True)  ::  (String,Char,Bool)     (triple)
```

$(\mathtt{t}_1,\ \mathtt{t}_2,\ \ldots\mathtt{t}_n)$ (n-tuple)

`(t)` not permitted

- The number of components in a tuple is called its arity
- The type of the components is unrestricted, e.g.:

Cartesian Product (pairs/tuples)

A sequence of values of different types:

`(False,True)`	`::`	`(Bool,Bool)`	(pair)
`(''Yes'','a',True)`	`::`	`(String,Char,Bool)`	(triple)
	$(t_1, t_2, \ldots t_n)$		(n-tuple)
	`(t)`		not permitted

- The number of components in a tuple is called its arity
- The type of the components is unrestricted, e.g.:
  - `('a',(False,'b')) :: (Char,(Bool,Char))`
  - `(True,['a','b']) :: (Bool,[Char])`

A sequence of values of different types:

```
      (False,True)  ::  (Bool,Bool)            (pair)
(‘‘Yes’’,’a’,True)  ::  (String,Char,Bool)     (triple)
```

$(\mathtt{t}_1,\ \mathtt{t}_2,\ \ldots \mathtt{t}_n)$ (n-tuple)

`(t)` not permitted

- The number of components in a tuple is called its arity
- The type of the components is unrestricted, e.g.:
  - `(’a’,(False,’b’)) :: (Char,(Bool,Char))`
  - `(True,[’a’,’b’]) :: (Bool,[Char])`

- What is the size of a tuple of type `(Char,(Bool,Char))`

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,
  become `(x,y) :: (typeL,typeR)`

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,
  become `(x,y) :: (typeL,typeR)`
  (1, 1.2) :: (Num t, Fractional t1) => (t, t1)

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,
  become `(x,y) :: (typeL,typeR)`

  (1, 1.2) :: (Num t, Fractional t1) => (t, t1)
- **Destruction**: `fst (x,y) = x` `snd (x,y) = y`

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,
  become `(x,y) :: (typeL,typeR)`
  (1, 1.2) :: (Num t, Fractional t1) => (t, t1)
- **Destruction**: `fst (x,y) = x` `snd (x,y) = y`
- Equality: if `typeL` and `typeR` have `==`,
  then `(typeL,typeR)` has `==`,
  namely: `(x,y) == (x',y') iff x==x' and y==y'`

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR,`
  become `(x,y) :: (typeL,typeR)`
  (1, 1.2) :: (Num t, Fractional t1) => (t, t1)
- **Destruction**: `fst (x,y) = x`   `snd (x,y) = y`
- Equality: if `typeL` and `typeR` have `==`,
  then `(typeL,typeR)` has `==`,
  namely: `(x,y) == (x',y') iff x==x' and y==y'`
  - If `typeL` and `typeR` are both instances of `Show`,
    so is `(typeL,typeR)`.

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,
  become `(x,y) :: (typeL,typeR)`
  (1, 1.2) :: (Num t, Fractional t1) => (t, t1)
- **Destruction**: `fst (x,y) = x`   `snd (x,y) = y`
- Equality: if `typeL` and `typeR` have `==`,
  then `(typeL,typeR)` has `==`,
  namely: `(x,y) == (x',y') iff x==x' and y==y'`
  - If `typeL` and `typeR` are both instances of `Show`, so is `(typeL,typeR)`.
  - If `typeL` and `typeR` are both instances of `Ord`, so is `(typeL,typeR)`.

Cartesian Product (pairs/tuples)

- Given `typeL` and `typeR`, we have a type of pairs `(typeL,typeR)`
- **Construction**: for `x :: typeL, y :: typeR`,
  become `(x,y) :: (typeL,typeR)`
  (1, 1.2) :: (Num t, Fractional t1) => (t, t1)
- **Destruction**: `fst (x,y) = x`   `snd (x,y) = y`
- Equality: if `typeL` and `typeR` have ==,
  then `(typeL,typeR)` has ==,
  namely: `(x,y) == (x',y') iff x==x' and y==y'`
  - If `typeL` and `typeR` are both instances of `Show`,
    so is `(typeL,typeR)`.
  - If `typeL` and `typeR` are both instances of `Ord`,
    so is `(typeL,typeR)`.

`Show` and `Ord` are type classes

Examples – value :: type

Examples – value :: type

1

Examples – value :: type

$$1 \quad :: \quad \mathsf{Int}$$

Examples – value :: type

1 :: Int   Num t => t

Examples – value :: type

1	::	Int	Num t => t
Int, Float are types		Num, Fract are	type classes
		“1” is overloaded	

Examples – value :: type

$$\begin{array}{rcl} 1 & :: & \text{Int} \\ 2.0 + 1.2 & :: & \text{Float} \end{array}$$

Num t => t

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]			

Examples – value :: type

value		type	
1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]

head [1, 1.2]

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0			

Examples – value :: type

1	::	Int	Num t => t
$2.0 + 1.2$	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]	::	[(String,Num)]	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]	::	[(String,Num)]	
1 + x			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]	::	[(String,Num)]	
1 + x	::	error	

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]	::	[(String,Num)]	
1 + x	::	error	
(1 +)			

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]	::	[(String,Num)]	
1 + x	::	error	
(1 +)	::		

Examples – value :: type

1	::	Int	Num t => t
2.0 + 1.2	::	Float	Fractional t => t
[1, 1.2]	::	[Float]	Fractional t => [t]
head [1, 1.2] =1.0	::		Fractional t => t
-(-1)	::	Int	Num t => t
- - 1	::	error	
[1,2,3]	::	[Int]	Num t => [t]
'a'	::	Char	
'a' : 'b' : []	::	[Char]	
"abc"	::	[Char]	i.e., String
["ab", "ab", "cd"]	::	[[Char]]	i.e., [String]
("ab", 12)	::	(String, Int)	Num t => (String, t)
["ab", 12]	::	error – list elements must have same type : [t]	
[("ab",12), ("cd", 24)]	::	[(String,Num)]	
1 + x	::	error	
(1 +)	::	Int -> Int	Num a => a -> a

A function:

- A mapping from values of one type to values of another type

```
not  ::  Bool -> Bool
```

Function Types

A function:

- A mapping from values of one type to values of another type

```
 not  ::  Bool -> Bool
even  ::  Int -> Bool
```

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK`, `<expr> :: typeM`

  `<name> x = <expr>`

  – e.g.: `double x = x+x`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`

  `<name> x = <expr>`

  – e.g.: `double x = x+x` or

  – e.g.: `\ x -> x+x`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`

  `<name> x = <expr>`

  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression

  – e.g.: `\ x -> x+x` :: Num t => t -> t

  – e.g.: `\ x -> \ y -> x+y`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`

  `<name> x = <expr>`

  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> g x + 1

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`.............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  NOT lambda expression

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f ::

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`

  `<name> x = <expr>`

  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression

  – e.g.: `\ x -> x+x` :: Num t => t -> t

  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1

  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1

  f :: Num a => (t -> a) -> t -> a

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: **?**

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: ?
- **Destruction**: function application, `<function> <expr>`:
  – `double 2` `(\ x -> x+x) 2`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>` .... anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a   g :: ?
- **Destruction**: function application, `<function> <expr>`:
  – `double 2`   `(\ x -> x+x) 2 == 4`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: **?**
- **Destruction**: function application, `<function> <expr>`:
  – `double 2` `(\ x -> x+x) 2 == 4`
  – `f (\y -> y*y)`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: ?
- **Destruction**: function application, `<function> <expr>`:
  – `double 2` `(\ x -> x+x) 2 == 4`
  – `f (\y -> y*y)` `\x -> ((\y -> y*y) x)+1`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: **?**
- **Destruction**: function application, `<function> <expr>`:
  – `double 2` `(\ x -> x+x) 2 == 4`
  – `f (\y -> y*y)` `\x -> ((\y -> y*y) x)+1`
  `== \x -> x*x + 1`

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>` ....anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: **?**
- **Destruction**: function application, `<function> <expr>`:
  – `double 2` `(\ x -> x+x) 2 == 4`
  – `f (\y -> y*y)` `\x -> ((\y -> y*y) x)+1`
  `== \x -> x*x + 1`
- Equality in principle impossible

Function Types

- Given `typeK` and `typeM`, we have a type `typeK -> typeM`
- **Construction**: for `x :: typeK, <expr> :: typeM`
  `<name> x = <expr>`
  – e.g.: `double x = x+x` or
- `\ x -> <expr>`............anonymous function / lambda expression
  – e.g.: `\ x -> x+x` :: Num t => t -> t
  – e.g.: `\ x -> \ y -> x+y` :: Num t => t -> t -> t
- f g = \ x -> (g x) + 1
  f = \ g -> \ x -> (g x) + 1 , or f = \ g x -> (g x) + 1
  f :: Num a => (t -> a) -> t -> a g :: ?
- **Destruction**: function application, `<function> <expr>`:
  – `double 2` `(\ x -> x+x) 2 == 4`
  – `f (\y -> y*y)` `\x -> ((\y -> y*y) x)+1`
  `== \x -> x*x + 1`
- Equality in principle impossible
- Show in principle impossible

Curring (Haskell Curry)

prefix
div :: Integral t => t -> t -> t
div 7 2 == 3

prefix		infix
div :: Integral t => t -> t -> t	$\gg$	`` `div` `` can be written in infix notat.
div 7 2 == 3		``7 `div` 2 == 3``

Curring (Haskell Curry)

prefix		
div :: Integral t => t -> t -> t	$\gg$	`` `div` `` can be written in infix notat.
div 7 2 == 3		``7 `div` 2 == 3``

+ :: (Int, Int) -> Int
3 + 4 == + (3,4) == 7

Curring (Haskell Curry)

prefix		
div :: Integral t => t -> t -> t	$\gg$	`‘div‘` can be written in infix notat.
div 7 2 == 3		`7 ‘div‘ 2 == 3`

(+) :: Int -> Int -> Int	$\ll$	+ :: (Int, Int) -> Int
(+) 3 4 == 7		3 + 4 == + (3,4) == 7

Curring (Haskell Curry)

prefix		
div :: Integral t => t -> t -> t	$\gg$	`` `div` `` can be written in infix notat.
div 7 2 == 3		`` 7 `div` 2 == 3 ``

(+) :: Int -> Int -> Int	$\ll$	+ :: (Int, Int) -> Int
(+) 3 4 == 7		3 + 4 == + (3,4) == 7
(−) :: Int -> Int -> Int	$\ll$	− :: (Int, Int) -> Int

Curring (Haskell Curry)

prefix		
div :: Integral t => t -> t -> t	$\gg$	`‘div‘` can be written in infix notat.
div 7 2 == 3		`7 ‘div‘ 2 == 3`
(+) :: Int -> Int -> Int	$\ll$	+ :: (Int, Int) -> Int
(+) 3 4 == 7		3 + 4 == + (3,4) == 7
(−) :: Int -> Int -> Int	$\ll$	− :: (Int, Int) -> Int

Two types are isomorphic:

A -> (B -> C) $\simeq$ (A, B) -> C

Curring (Haskell Curry)

	prefix		
	div :: Integral t => t -> t -> t	$\gg$	`div` can be written in infix notat.
	div 7 2 == 3		`` 7 `div` 2 == 3 ``

(+) :: Int -> Int -> Int	$\ll$	+ :: (Int, Int) -> Int
(+) 3 4 == 7		3 + 4 == + (3,4) == 7
(−) :: Int -> Int -> Int	$\ll$	− :: (Int, Int) -> Int

Two types are isomorphic:

A -> (B -> C)	$\simeq$	(A, B) -> C
(f) :: A -> (B -> C)	$\ll$	f :: (A, B) -> C

Curring (Haskell Curry)

prefix		
div :: Integral t => t -> t -> t	$\gg$	`` `div` `` can be written in infix notat.
div 7 2 == 3		`` 7 `div` 2 == 3 ``

(+) :: Int -> Int -> Int	$\ll$	+ :: (Int, Int) -> Int
(+) 3 4 == 7		3 + 4 == + (3,4) == 7
(−) :: Int -> Int -> Int	$\ll$	− :: (Int, Int) -> Int

Two types are isomorphic:

A -> (B -> C)	$\simeq$	(A, B) -> C
(f) :: A -> (B -> C)	$\ll$	f :: (A, B) -> C
(f) a b	=	f(a,b)

Curring (Haskell Curry)

prefix		
div :: Integral t => t -> t -> t	$\gg$	`` `div` `` can be written in infix notat.
div 7 2 == 3		``7 `div` 2 == 3``
(+) :: Int -> Int -> Int	$\ll$	+ :: (Int, Int) -> Int
(+) 3 4 == 7		3 + 4 == + (3,4) == 7
(−) :: Int -> Int -> Int	$\ll$	− :: (Int, Int) -> Int

Two types are isomorphic:

A -> (B -> C)	$\simeq$	(A, B) -> C
(f) :: A -> (B -> C)	$\ll$	f :: (A, B) -> C
(f) a b	=	f(a,b)
(+2) = (+) 2 :: Int -> Int		"section"
(+2) 1 = 3, etc.		

Currying

Introduced by Gottlob Frege, developed by Moses Schönfinkel

Currying

Introduced by Gottlob Frege, developed by Moses Schönfinkel

Haskell Brooks Curry

Three programming languages named after him:

- Haskell
- Brook
- Curry

Basic types, each with a set of operators

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `+, -, *, div, mod`
- boolean values `True,False`, ... :: `Bool`
  `not, ||, &&, or, and`
- symbols `...,'A',...'z',...`, ... :: `Char`
  `toUpper, toLower, isDigit (Bool)` in Data.Char
- strings like `''abc''`, ... :: `String`
  `++, head, tail`
- fractions ... 1.2, 3.5 ... . :: `Float`
  `+, -, *, /`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `(+),(-),(*),(div) ::`
- boolean values `True,False`, ... :: `Bool`
- symbols `...,'A',...'z',...`, ... :: `Char`
- strings like `''abc''`, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
- symbols `...,'A',...'z',...`, ... :: `Char`
- strings like ``‘‘abc’’``, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` .... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not ::` `or, and ::`
- symbols `...,'A',...'z',...,` .... :: `Char`
- strings like `''abc''`, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and ::`
- symbols `...,'A',...'z',...`, ... :: `Char`
- strings like ``‘‘abc’’``, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` .... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
- symbols `...,'A',...'z',...,` .... :: `Char`
- strings like `‘‘abc’’`, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) ::`
- symbols `...,'A',...'z',...`, ... :: `Char`
- strings like `''abc''`, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...`, ... :: `Char`
- strings like ``‘‘abc’’``, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...`, ... :: `Char`
  `toUpper ::` `isDigit ::`
- strings like `''abc''`, ... :: `String`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...`, ...:: `Int`

  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ...:: `Bool`

  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`

  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...`, ...:: `Char`

  `toUpper :: Char -> Char; isDigit ::`
- strings like `''abc''`, ...:: `String`
- fractions ... 1.2, 3.5 ... .:: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` . . . . :: `Int`

  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`

  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`

  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...,` . . . . :: `Char`

  `toUpper :: Char -> Char; isDigit :: Char -> Bool`
- strings like ``‘‘abc’’``, ... :: `String`

- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` .... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...,` .... :: `Char`
  `toUpper :: Char -> Char; isDigit :: Char -> Bool`
- strings like ``‘‘abc’’``, ... :: `String`
  `head ::` `(++) ::`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` .... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...,` .... :: `Char`
  `toUpper :: Char -> Char; isDigit :: Char -> Bool`
- strings like ``abc'', ... :: `String`
  `head :: [a] -> a; (++) ::`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` .... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...,` .... :: `Char`
  `toUpper :: Char -> Char; isDigit :: Char -> Bool`
- strings like ``‘‘abc’’``, ... :: `String`
  `head :: [a] -> a; (++) :: [a] -> [a] -> [a]`
- fractions ... 1.2, 3.5 ... . :: `Float`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` . . . . :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...,` . . . . :: `Char`
  `toUpper :: Char -> Char; isDigit :: Char -> Bool`
- strings like ``‘‘abc’’``, ... :: `String`
  `head :: [a] -> a; (++) :: [a] -> [a] -> [a]`
- fractions ... 1.2, 3.5 ... . :: `Float`
  `(/) ::`

Basic types, what is the type of the operators?

- integers `...,-2,-1,0,1,2,...,` .... :: `Int`
  `(+),(-),(*),(div) :: Num a => a -> a -> a`
- boolean values `True,False`, ... :: `Bool`
  `not :: Bool -> Bool; or, and :: [Bool] -> Bool`
  `(||),(&&) :: Bool -> Bool -> Bool`
- symbols `...,'A',...'z',...`, ... :: `Char`
  `toUpper :: Char -> Char; isDigit :: Char -> Bool`
- strings like ``abc'', ... :: `String`
  `head :: [a] -> a; (++) :: [a] -> [a] -> [a]`
- fractions ... 1.2, 3.5 ... . :: `Float`
  `(/) :: Fractional a => a -> a -> a`

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`
- Eq, with `==,/= :: a -> a -> Bool`

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`
- Eq, with `==,/= :: a -> a -> Bool`
- Num, (+), (-), (*), abs, negate...,
  – a subclass of `Eq` and `Show` with subclasses like

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`
- Eq, with `==,/= :: a -> a -> Bool`
- Num, (+), (-), (*), abs, negate...,
  – a subclass of Eq and Show with subclasses like
    - Integral

Types are (most often) instances of type "classes"

- `Show`, with an operation `show :: a -> String`
  `Read`, with an operation `read :: String -> a`
- `Eq`, with `==,/= :: a -> a -> Bool`
- `Num`, (+), (-), (*), abs, negate...,
  – a subclass of `Eq` and `Show` with subclasses like
    - `Integral`
    - `Fractional`

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`
- Eq, with `==,/= :: a -> a -> Bool`
- Num, (+), (-), (*), abs, negate…,
  – a subclass of Eq and Show with subclasses like
    - Integral
    - Fractional
    - …

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`
- Eq, with `==,/= :: a -> a -> Bool`
- Num, (+), (-), (*), abs, negate...,
  – a subclass of Eq and Show with subclasses like
    - Integral
    - Fractional
    - ...
- Ord, a subclass of Eq, with instances (types) that are totally ordered; operations like
  `(<),(<=),(>),(>=) :: a -> a -> Bool`
  `min,max :: a -> a -> a`

Types are (most often) instances of type "classes"

- Show, with an operation `show :: a -> String`
  Read, with an operation `read :: String -> a`
- Eq, with `==,/= :: a -> a -> Bool`
- Num, (+), (-), (*), abs, negate...,
  – a subclass of Eq and Show with subclasses like
    - Integral
    - Fractional
    - ...
- Ord, a subclass of Eq, with instances (types) that are totally ordered; operations like
  `(<),(<=),(>),(>=) :: a -> a -> Bool`
  `min,max :: a -> a -> a`
- ...

Types are (most often) instances of type "classes"

- `Show`, with an operation `show :: a -> String`
  `Read`, with an operation `read :: String -> a`
- `Eq`, with `==,/= :: a -> a -> Bool`
- `Num`, (+), (-), (*), abs, negate...,
  – a subclass of `Eq` and `Show` with subclasses like
  - `Integral`
  - `Fractional`
  - ...
- `Ord`, a subclass of `Eq`, with instances (types) that are totally ordered; operations like
  `(<),(<=),(>),(>=) :: a -> a -> Bool`
  `min,max :: a -> a -> a`
- ...
- All basic types are instances of `Eq`, `Ord`, `Show` and `Read`.
  Also, lists and tuples are of such types, if the types of their elements/components are so.

Type classes

- class Eq t where

```
    (==), (/=) :: t -> t -> Bool
```

Type classes

- class Eq t where

```
    (==), (/=) :: t -> t -> Bool
    x /= y = not(x == y)
```

Type classes

- class Eq t where
  (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
- classes can have subclasses
  class Eq t => Ord t where

Type classes

- class Eq t where
```
    (==), (/=) :: t -> t -> Bool
    x /= y = not(x == y)
```
- classes can have subclasses
```
  class Eq t => Ord t where
    (>), (<), (>=), (<=) :: t -> t -> Bool
```

- class Eq t where
```
    (==), (/=) :: t -> t -> Bool
    x /= y = not(x == y)
```
- classes can have subclasses
```
class Eq t => Ord t where
    (>), (<), (>=), (<=) :: t -> t -> Bool
    min, max             :: t -> t -> t
    ...
```

Type classes

- class Eq t where

```
    (==), (/=) :: t -> t -> Bool
    x /= y = not(x == y)
```

- classes can have subclasses

```
class Eq t => Ord t where
    (>), (<), (>=), (<=) :: t -> t -> Bool
    min, max             :: t -> t -> t
    ...
```

- types declared with data and newtype can be instances of classes

Type classes

- class Eq t where
  ```
  (==), (/=) :: t -> t -> Bool
  x /= y = not(x == y)
  ```
- classes can have subclasses
  ```
  class Eq t => Ord t where
      (>), (<), (>=), (<=) :: t -> t -> Bool
      min, max             :: t -> t -> t
      ...
  ```
- types declared with data and newtype can be instances of classes

i) instance Ord Bool where
```
    False < True = True
```

Type classes

- class Eq t where

```
    (==), (/=) :: t -> t -> Bool
    x /= y = not(x == y)
```

- classes can have subclasses

```
class Eq t => Ord t where
    (>), (<), (>=), (<=) :: t -> t -> Bool
    min, max             :: t -> t -> t
    ...
```

- types declared with data and newtype can be instances of classes

i) instance Ord Bool where

```
    False < True = True
```

ii) data Bool = False | True deriving (Eq, Ord)

Type classes

- class Eq t where

```
    (==), (/=) :: t -> t -> Bool
    x /= y = not(x == y)
```

- classes can have subclasses

```
class Eq t => Ord t where
    (>), (<), (>=), (<=) :: t -> t -> Bool
    min, max             :: t -> t -> t
    ...
```

- types declared with data and newtype can be instances of classes

i) instance Ord Bool where

```
    False < True = True
```

ii) data Bool = False | True deriving (Eq, Ord)

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`

  `newtype Nat = Na Int`

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`

  `newtype Nat = Na Int`

```
data BTreInt = Leaf Int | Node Int Int
```

Leaf and Node are constructors

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`
  `newtype Nat = Na Int`

`data BTreInt = Leaf Int | Node Int Int`

`Leaf` and `Node` are constructors

(unique constructor name: `... | Node Int` – error)

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`
  `newtype Nat = Na Int`

```
data BTreInt = Leaf Int | Node Int Int
```

`Leaf` and `Node` are constructors

(unique constructor name: `... | Node Int` – error)

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`
  `newtype Nat = Na Int`

```
data BTreInt = Leaf Int | Node Int Int
```

`Leaf` and `Node` are constructors

(unique constructor name: `... | Node Int` – error)

- Names of types and contructors must start with capital letters!

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`
  `newtype Nat = Na Int`

  the type can be declared as an instance of a class deriving (Eq, Ord)
  instance Eq Direction where...

  `data BTreInt = Leaf Int | Node Int Int`
  `Leaf` and `Node` are constructors
  (unique constructor name: `... | Node Int` – error)

- Names of types and contructors must start with capital letters!

User-defined types

- **Type synonyms**: `type Pos = (Int,Int)`
- **New data types**: `data Direction = Ne | Se | Sw | Nw`
  `newtype Nat = Na Int`

  the type can be declared as an instance of a class deriving (Eq, Ord)
  instance Eq Direction where...

  `data BTreInt = Leaf Int | Node Int Int`
  `Leaf` and `Node` are constructors
  (unique constructor name: `... | Node Int` – error)

- Names of types and contructors must start with capital letters!

- For simple types, functions which are inherited through `deriving`

  – from `Show, Read, Eq, Ord, Enum, Bounded` –

  are automatically derived by Haskell. However, one can redefine them, if desired.

Type synonyms – `type`

```
type Pos = (Int,Int)
```

Type synonyms – `type`

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)`

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]`

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  ```
  type Dict = [(Key, Value)]
  ```
  ```
  find n d =
  ```

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  ```
  type Dict = [(Key, Value)]
  ```
  ```
  find n d =      [ v' | (k',v') <- d, k' == n ]
  ```

Type synonyms – type

```haskell
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:

```haskell
type Dict = [(Key, Value)]

find n d =       [ v' | (k',v') <- d, k' == n ]
```

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  ```
  type Dict = [(Key, Value)]                instance Eq Key

  find n d =      [ v' | (k',v') <- d, k' == n ]
  ```

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:

  `type Dict = [(Key, Value)]` instance `Eq Key`

  ```
  find n d = head [ v' | (k',v') <- d, k' == n ]
  ```

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]` instance `Eq Key`

  `find n d = head [ v' | (k',v') <- d, k' == n ]`

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]` instance `Eq Key`

  `find n d = head [ v' | (k',v') <- d, k' == n ]`

– parametrise the type (with type variables)

  `type Dict k v = [(k,v)]`

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]` instance `Eq Key`

  `find n d = head [ v' | (k',v') <- d, k' == n ]`

– parametrise the type (with type variables)

  `type Dict k v = [(k,v)]` ← `k, v` are type parameters

Type synonyms – type

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]` instance `Eq Key`

  `find n d = head [ v' | (k',v') <- d, k' == n ]`

– parametrise the type (with type variables)

  `type Dict k v = [(k,v)]` ← `k, v` are type parameters

- `find ::`

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]` instance `Eq Key`

  `find n d = head [ v' | (k',v') <- d, k' == n ]`

– parametrise the type (with type variables)

  `type Dict k v = [(k,v)]` ← `k, v` are type parameters

- `find ::          k -> Dict k v -> v`

Type synonyms – `type`

```
type Pos = (Int,Int)
```

Parametrised types:

- `type Anypair a = (a,a)` ← a is a type parameter
- Given the types `Key` and `Value`, we can declare a new type:
  `type Dict = [(Key, Value)]` instance `Eq Key`

  `find n d = head [ v' | (k',v') <- d, k' == n ]`

– parametrise the type (with type variables)

  `type Dict k v = [(k,v)]` ← `k, v` are type parameters

- `find :: Eq k => k -> Dict k v -> v`

User-defined data types: data

```
data Direction = North | South | East | West
```

User-defined data types: data

```
data Direction = North | South | East | West
```

– own constructors of data type requires keyword data...

User-defined data types: data

- data Fig = Circ Float | Rect Float Float

User-defined data types: data

- data $\mathrm{Fig} = \mathrm{Circ}$ Float $|$ Rect Float Float
- data contructors are functions of the type, i.e.,

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig
  Rect :: Float -> Float -> Fig

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig

  Rect :: Float -> Float -> Fig

> but they do not have any defining equations, because Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig

  Rect :: Float -> Float -> Fig

$>$ but they do not have any defining equations, because
Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig

  Rect :: Float -> Float -> Fig

> but they do not have any defining equations, because
Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

  square n = Rect n n

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig

  Rect :: Float -> Float -> Fig

> but they do not have any defining equations, because
> Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

  square n = Rect n n

  area (Rect n m) = n*m

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig

  Rect :: Float -> Float -> Fig

> but they do not have any defining equations, because
  Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

  square n = Rect n n

  area (Rect n m) = n*m

  area (Circ n) = pi * n^ 2

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig
  Rect :: Float -> Float -> Fig

> but they do not have any defining equations, because
  Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

  square n = Rect n n

  area (Rect n m) = n*m

  area (Circ n) = pi * n^ 2

- Circ/Rect can informally be understood as "subtypes" of Fig

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig
  Rect :: Float -> Float -> Fig

> but they do not have any defining equations, because
  Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

  square n = Rect n n

  area (Rect n m) = n*m

  area (Circ n) = pi * n^ 2

- Circ/Rect can informally be understood as "subtypes" of Fig
- data Circ = Ci Int
  data Fig = Circ | ...

User-defined data types: data

- data Fig = Circ Float | Rect Float Float
- data contructors are functions of the type, i.e.,

  Circ :: Float -> Fig

  Rect :: Float -> Float -> Fig

\> but they do not have any defining equations, because
  Rect 2 3 is already a completely evaluated value = Rect 2 3 :: Fig

- Functions can now be defined by using constructors as patterns:

  square n = Rect n n

  area (Rect n m) = n*m

  area (Circ n) = pi * n^ 2

- Circ/Rect can informally be understood as “subtypes” of Fig
- data Circ = Ci Int

  data Fig = Constr Circ | ...

Example

- data Move = Up | Down | Left | Right

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos
  mv (x,y) Left = (x–1,y)   mv (x,y) Right = (x+1,y)

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos
  mv (x,y) Left = (x–1,y)   mv (x,y) Right = (x+1,y)
  mv (x,y) Up = (x,y+1)   mv (x,y) Down = (x,y–1)

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos
  mv (x,y) Left = (x–1,y)   mv (x,y) Right = (x+1,y)
  mv (x,y) Up = (x,y+1)   mv (x,y) Down = (x,y–1)
- mvs :: Pos -> [Move] -> Pos

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos
  mv (x,y) Left = (x–1,y)   mv (x,y) Right = (x+1,y)
  mv (x,y) Up = (x,y+1)   mv (x,y) Down = (x,y–1)
- mvs :: Pos -> [Move] -> Pos
  mvs p [] = p

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos
  mv (x,y) Left = (x–1,y) mv (x,y) Right = (x+1,y)
  mv (x,y) Up = (x,y+1) mv (x,y) Down = (x,y–1)
- mvs :: Pos -> [Move] -> Pos
  mvs p [] = p
  mvs p (r:rs) =

Example

- data Move = Up | Down | Left | Right
  deriving (Eq, Ord, Read, Show)
- type Pos = (Int, Int)
- mv :: Pos -> Move -> Pos
  mv (x,y) Left = (x–1,y)   mv (x,y) Right = (x+1,y)
  mv (x,y) Up = (x,y+1)   mv (x,y) Down = (x,y–1)
- mvs :: Pos -> [Move] -> Pos
  mvs p [] = p
  mvs p (r:rs) = mvs (mv p r) rs

User-defined recursive data types

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x

  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x

  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r   case syntax!

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r case syntax!
- insert x (Leaf y) = ?

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x

  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r   case syntax!

  insert x (Leaf y) = if x<y then

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x

  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r case syntax!

insert x (Leaf y) = if x<y then Node (Leaf x) y Emp

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x

  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r case syntax!

  insert x (Leaf y) = if x<y then Node (Leaf x) y Emp

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r   case syntax!

  insert x (Leaf y) = if x<y then Node (Leaf x) y Emp
  else Node Emp y (Leaf x)

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r   case syntax!
- insert x Emp = Leaf x
  insert x (Leaf y) = if x<y then Node (Leaf x) y Emp
  else Node Emp y (Leaf x)

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r   case syntax!
- insert x Emp = Leaf x
  insert x (Leaf y) = if x<y then Node (Leaf x) y Emp
  else Node Emp y (Leaf x)
  insert x (Node l y r) =

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r   case syntax!
- insert x Emp = Leaf x
  insert x (Leaf y) = if x<y then Node (Leaf x) y Emp
  else Node Emp y (Leaf x)
  insert x (Node l y r) = if x<y then Node (insert x l) y r

- data Tree = Leaf Int | Node Tree Int Tree | Emp
  search: left subtrees have only smaller nodes than the root
  right subtrees have no smaller nodes than the root
- list (Leaf y) = [y]
  list (Node l y r) = list l ++ [y] ++ list r . . . . sorted list!
- find x (Leaf y) = **if** x==y **then** y **else** error "No " ++ show x
  find x (Leaf x) = x . . . . – illegal: unique variables in patterns
  find x (Node l n r) | x==n = n
  | x<n = find x l
  | otherwise = find x r case syntax!
- insert x Emp = Leaf x
  insert x (Leaf y) = if x<y then Node (Leaf x) y Emp
  else Node Emp y (Leaf x)
  insert x (Node l y r) = if x<y then Node (insert x l) y r
  else Node l y (insert x r)

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|$ height(left subtree) – height(right subtree) $| \leq 1$

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|$ height(left subtree) – height(right subtree) $| \leq 1$
  – in any subtree

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  | height(left subtree) – height(right subtree) | $\leq$ 1
  – in any subtree
- height Emp = 0
  height (Leaf y) = 1
  height (Node l a r) = 1 + max (height l) (height r)

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  | height(left subtree) – height(right subtree) | $\leq$ 1
  – in any subtree
- height Emp = 0
  height (Leaf y) = 1
  height (Node l a r) = 1 + max (height l) (height r)
- balanced Emp = True
  balanced (Leaf y) = True

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  | height(left subtree) – height(right subtree) | $\leq$ 1
  – in any subtree
- height Emp = 0
  height (Leaf y) = 1
  height (Node l a r) = 1 + max (height l) (height r)
- balanced Emp = True
  balanced (Leaf y) = True
  balanced (Node l y r) =
  abs((height l) – (height r)) <= 1

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  | height(left subtree) – height(right subtree) | $\leq$ 1
  – in any subtree
- height Emp = 0
  height (Leaf y) = 1
  height (Node l a r) = 1 + max (height l) (height r)
- balanced Emp = True
  balanced (Leaf y) = True
  balanced (Node l y r) =
  [abs((height l) – (height r)) <= 1 , balanced(l), balanced(r)]

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  | height(left subtree) – height(right subtree) | $\leq$ 1
  – in any subtree
- height Emp = 0
  height (Leaf y) = 1
  height (Node l a r) = 1 + max (height l) (height r)
- balanced Emp = True
  balanced (Leaf y) = True
  balanced (Node l y r) =
  and [abs((height l) – (height r)) <= 1 , balanced(l), balanced(r)]

  `and :: [ Bool ] -> Bool`

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|$ #(leaves in the left tree) – #(leaves in right subtree) $| \leq 1$

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|\ \#$(leaves in the left tree) – $\#$(leaves in right subtree) $|\ \leq 1$
  – in any subtree

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|$ #(leaves in the left tree) – #(leaves in right subtree) $| \leq 1$
  – in any subtree
- leaf Emp = 0
  leaf (Leaf y) = 1
  leaf (Node l a r) = leaf(l) + leaf(r)

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|$ #(leaves in the left tree) – #(leaves in right subtree) $| \leq 1$
  – in any subtree
- leaf Emp = 0
  leaf (Leaf y) = 1
  leaf (Node l a r) = leaf(l) + leaf(r)
- balanced Emp = True
  balanced (Leaf y) = True

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  $|\ \#(\text{leaves in the left tree}) - \#(\text{leaves in right subtree})\ | \leq 1$
  – in any subtree
- leaf Emp = 0
  leaf (Leaf y) = 1
  leaf (Node l a r) = leaf(l) + leaf(r)
- balanced Emp = True
  balanced (Leaf y) = True
  balanced (Node l y r) =
  abs(leaf(l) – leaf(r)) <= 1

User-defined recursive data types

- data Tree $=$ Leaf Int $|$ Node Tree Int Tree $|$ Emp
- balanced binary tree?
  $|\ \#$(leaves in the left tree) – $\#$(leaves in right subtree) $| \leq 1$
  – in any subtree
- leaf Emp $= 0$
  leaf (Leaf y) $= 1$
  leaf (Node l a r) $=$ leaf(l) $+$ leaf(r)
- balanced Emp $=$ True
  balanced (Leaf y) $=$ True
  balanced (Node l y r) $=$
  [abs(leaf(l) – leaf(r)) $<=$ 1 , balanced(l), balanced(r)]

User-defined recursive data types

- data Tree = Leaf Int | Node Tree Int Tree | Emp
- balanced binary tree?
  | #(leaves in the left tree) – #(leaves in right subtree) | $\leq$ 1
  – in any subtree
- leaf Emp = 0
  leaf (Leaf y) = 1
  leaf (Node l a r) = leaf(l) + leaf(r)
- balanced Emp = True
  balanced (Leaf y) = True
  balanced (Node l y r) =
  and [abs(leaf(l) – leaf(r)) <= 1 , balanced(l), balanced(r)]

```
and :: [ Bool ] -> Bool
```

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) ....data only at leaves

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) ....data only at leaves
= Leaf | Node (Tree t) t (Tree t) ....can only at internal nodes

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t]

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t]

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) ....data only at leaves
= Leaf | Node (Tree t) t (Tree t) ....can only at internal nodes
= Node t [Tree t] .....several children (a leaf has empty child-list)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t] . . . . . several children (a leaf has empty child-list)

- left (Leaf x) =

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) .... data only at leaves
= Leaf | Node (Tree t) t (Tree t) .... can only at internal nodes
= Node t [Tree t] ..... several children (a leaf has empty child-list)

- left (Leaf x) = (???)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t] . . . . . several children (a leaf has empty child-list)

- left (Leaf x) = Nothing

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) ....data only at leaves
= Leaf | Node (Tree t) t (Tree t) ....can only at internal nodes
= Node t [Tree t] .....several children (a leaf has empty child-list)

- left (Leaf x) = Nothing
  left (Node l x r) = Just l

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) ....data only at leaves
= Leaf | Node (Tree t) t (Tree t) ....can only at internal nodes
= Node t [Tree t] ..... several children (a leaf has empty child-list)

- left (Leaf x) = Nothing
  left (Node l x r) = Just l
- data Maybe t = Nothing | Just t (`import Data.Maybe`)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t] . . . . . several children (a leaf has empty child-list)

- left (Leaf x) = Nothing
  left (Node l x r) = Just l
- data Maybe t = Nothing | Just t (`import Data.Maybe`)
  for partial functions – which returns values of type t

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) ....data only at leaves
= Leaf | Node (Tree t) t (Tree t) ....can only at internal nodes
= Node t [Tree t] ..... several children (a leaf has empty child-list)

- left (Leaf x) = Nothing
  left (Node l x r) = Just l
- data Maybe t = Nothing | Just t (`import Data.Maybe`)
  for partial functions – which returns values of type t
  phead [] = Nothing
  phead (x:xs) = Just x

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t] . . . . . several children (a leaf has empty child-list)

- left (Leaf x) = Nothing
  left (Node l x r) = Just l
- data Maybe t = Nothing | Just t (`import Data.Maybe`)
  for partial functions – which returns values of type t
  phead [] = Nothing
  phead (x:xs) = Just x :: Maybe t (the type of x)

User-defined parametrised data types

- data Tree = Leaf Int | Node Tree Int Tree
- data Tree t = Leaf t | Node (Tree t) t (Tree t)

Tree t = Leaf t | Node (Tree t) (Tree t) . . . . data only at leaves
= Leaf | Node (Tree t) t (Tree t) . . . . can only at internal nodes
= Node t [Tree t] . . . . . several children (a leaf has empty child-list)

- left (Leaf x) = Nothing
  left (Node l x r) = Just l
- data Maybe t = Nothing | Just t (`import Data.Maybe`)
  for partial functions – which returns values of type t
  phead [] = Nothing
  phead (x:xs) = Just x :: Maybe t (the type of x)
  Sometimes, requires cast (back) to get the correct type of the correct result, e.g,.
  fromJust :: Maybe t -> t
  fromJust (Just x) = x
  which is often required when Maybe-values can be returned

User-defined data types – newtype

Creates a new type in a similar way as `data`,

User-defined data types – newtype

Creates a new type in a similar way as `data`,
but has only a single constructor

```
newtype Nat = N Int
```

User-defined data types – newtype

Creates a new type in a similar way as `data`,
but has only a single constructor

```
newtype Nat = N Int
```

- `type Nat = Int`
  - `Nat` and `Int` are the same in this case

User-defined data types – newtype

Creates a new type in a similar way as `data`,
but has only a single constructor

`newtype Nat = N Int`

- `type Nat = Int`
  - `Nat` and `Int` are the same in this case
- `data Nat = N Int`
  - Almost the same, but the constructor is removed after type checking

User-defined data types – newtype

Creates a new type in a similar way as `data`,
but has only a single constructor

```
newtype Nat = N Int
```

- `type Nat = Int`
  - `Nat` and `Int` are the same in this case
- `data Nat = N Int`
  - Almost the same, but the constructor is removed after type checking
- Improve type safety, maintain performance

User-defined data types – newtype