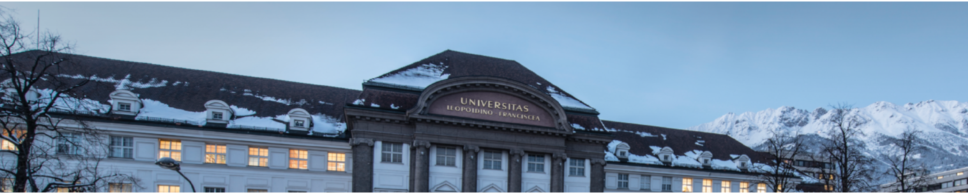
universität innsbruck

WS 2018/2019



Functional Programming

Lecture 2

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- Types and Type Classes
- Lists
- Patterns, Guards, and More
- Higher-Order Functions

Types and Type Classes

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ldots\ \tau$$

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ \ldots\ \ \tau$$

- with type variables α – `a`, `b`, ...

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ \ldots\ \ \tau$$

- with type variables α – `a`, `b`, ...
- type constructors C – `Bool`, `Int`, `[]`, `(,)`, ...

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ldots\ \tau$$

- with type variables α – `a`, `b`, ...
- type constructors C – `Bool`, `Int`, `[]`, `(,)`, ...

List

Pair

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ \dots\ \ \tau$$

- with type variables α – `a`, `b`, ...
- type constructors C – `Bool`, `Int`, `[]`, `(,)`, ...
- function type constructor `->` (special case of previous item)

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ \ldots\ \ \tau$$

- with type variables α – `a`, `b`, ...
- type constructors C – `Bool`, `Int`, `[]`, `(,)`, ...
- function type constructor `->` (special case of previous item)
- `->` associates to the right: $\tau \texttt{ -> } (\tau \texttt{ -> } \tau) = \tau \texttt{ -> } \tau \texttt{ -> } \tau$

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ldots\ \tau$$

- with type variables α – `a`, `b`, ...
- type constructors C – `Bool`, `Int`, `[]`, `(,)`, ...
- function type constructor `->` (special case of previous item)
- `->` associates to the right: $\tau \texttt{ -> } (\tau \texttt{ -> } \tau) = \tau \texttt{ -> } \tau \texttt{ -> } \tau$
- as approximation types may be thought of as collections of values their inhabitants can reduce to, e.g., $\texttt{Bool} = \{\texttt{True}, \texttt{False}\}$, reflects the intuition that every expression of type `Bool` reduces either to `True` or `False` (or diverges) during runtime

Basic Concepts

- types τ are built according to grammar

$$\tau ::= \alpha \mid \tau \texttt{ -> } \tau \mid C\ \tau\ \ \ldots\ \ \tau$$

- with type variables α – `a`, `b`, ...
- type constructors C – `Bool`, `Int`, `[]`, `(,)`, ...
- function type constructor `->` (special case of previous item)
- `->` associates to the right: $\tau \texttt{ -> } (\tau \texttt{ -> } \tau) = \tau \texttt{ -> } \tau \texttt{ -> } \tau$
- as approximation types may be thought of as collections of values their inhabitants can reduce to, e.g., `Bool` $= \{$`True`, `False`$\}$, reflects the intuition that every expression of type `Bool` reduces either to `True` or `False` (or diverges) during runtime
- type signature/constraint e `::` τ means "e is of type τ"

Basic Types

Basic Types

- `Bool` – logical values (`True`, `False`)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)

syntactic sugar for list of characters, e.g., `['a','b','c']`

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)
- `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, . . .)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)
- `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)
- `Integer` – arbitrary-precision integers

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)
- `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)
- `Integer` – arbitrary-precision integers
- `Float` – single-precision floating-point numbers (`-12.34`, `3.14159`)

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)
- `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)
- `Integer` – arbitrary-precision integers
- `Float` – single-precision floating-point numbers (`-12.34`, `3.14159`)
- `Double` – double-precision floating-point numbers

Basic Types

- `Bool` – logical values (`True`, `False`)
- `Char` – single characters (`'a'`, `'\n'`, ...)
- `String` – sequences of characters (`"abc"`, `"1+2=3"`)
- `Int` – fixed-precision integers with at least 29 bits (`-100`, `0`, `999`)
- `Integer` – arbitrary-precision integers
- `Float` – single-precision floating-point numbers (`-12.34`, `3.14159`)
- `Double` – double-precision floating-point numbers

Note – Show Types in GHCi

- `Prelude> :set +t`
- commands may be put inside `~/.ghci` (read on GHCi startup)

List Types

- type of lists with elements of type τ: $[\tau]$
- all elements are of same type
- no restriction on length of list

List Types

- type of lists with elements of type τ: $[\tau]$
- all elements are of same type
- no restriction on length of list

Tuple Types

- type of tuples with elements of types $\tau_1, \ldots, \tau_n$: $(\tau_1, \ldots, \tau_n)$
- length: 2 (pair), 3 (triple), 4 (quadruple), ..., n (n-tuple), ...
- elements may be of different types
- fixed number of elements

List Types

- type of lists with elements of type τ: $[\tau]$
- all elements are of same type
- no restriction on length of list

Tuple Types

- type of tuples with elements of types $\tau_1, \ldots, \tau_n$: $(\tau_1, \ldots, \tau_n)$
- length: 2 (pair), 3 (triple), 4 (quadruple), …, n (n-tuple), …
- elements may be of different types
- fixed number of elements

Examples

```
['a','b','c','d'] :: [Char]
["One","Two","Three"] :: [String]
[['a','b'],['c','d','e']] :: [[Char]]
(False,True) :: (Bool,Bool)
("Yes",True,'a') :: (String,Bool,Char)
```

Function Types

- type of functions from values of type τ_1 to values of type τ_2: τ_1 `->` τ_2
- every function takes single argument and returns single result
- simulating multiple arguments: use tuples

Function Types

- type of functions from values of type τ_1 to values of type τ_2: τ_1 `->` τ_2
- every function takes single argument and returns single result
- simulating multiple arguments: use tuples

Function Types

- type of functions from values of type τ_1 to values of type τ_2: τ_1 -> τ_2
- every function takes single argument and returns single result
- simulating multiple arguments: use tuples

Examples

```
not :: Bool -> Bool
not True = False
not False = True

add :: (Int, Int) -> Int
add (x, y) = x + y
```

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

Anonymous Functions – “Lambda-Abstractions”

- `\x -> ` e is function taking `x` and returning e

Currying

- transform function taking tuple as input into function returning another function as output
- in presence of partial application, curried functions are more versatile than uncurried functions

Example

```
add' :: Int -> (Int -> Int)
add' x y = x + y
-- partial application: successor function
suc = add' 1
```

Anonymous Functions – “Lambda-Abstractions”

- `\x -> `e is function taking `x` and returning e

Example

```
add' = \x -> \y -> x + y
```

Basic Functions

- `Bool`: "conjunction" `&&`, "disjunction" `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`

Basic Functions

- `Bool`: "conjunction" `&&`, "disjunction" `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`
- `(a, b)`: choose first `fst`, choose second `snd`

Basic Functions

- `Bool`: "conjunction" `&&`, "disjunction" `||`, negation `not`, equality `==`, and `otherwise` as alias for `True`
- `(a, b)`: choose first `fst`, choose second `snd`

Examples

```
not True      == False
(False && x) == False
(True || x)  == True
otherwise     == True

fst (x, y) == x
snd (x, y) == y
```

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Realization – Class Constrains

- syntax: e :: C `a` => τ
- meaning: "for every type `a` of class C, the type of e is τ"
  (where τ does contain `a`)

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Realization – Class Constrains

- syntax: e :: C a => τ
- meaning: "for every type a of class C, the type of e is τ" (where τ does contain a)

Example – Addition

- `(+) :: Num a => a -> a -> a`
- "for every type `a` of class `Num`, addition has type `a -> a -> a`"
- since, e.g., `Int` is of class `Num`, we obtain that addition is of type `Int -> Int -> Int`, when used on `Int`s

Overloaded Types

- support standard set of operations
- use same name, independent of actual type

Realization – Class Constrains

- syntax: e :: C a => τ
- meaning: "for every type a of class C, the type of e is τ" (where τ does contain a)

Example – Addition

(op) turns infix op into prefix

- `(+) :: Num a => a -> a -> a`
- "for every type a of class Num, addition has type `a -> a -> a`"
- since, e.g., Int is of class Num, we obtain that addition is of type `Int -> Int -> Int`, when used on Ints

The `Eq` Class – Equality

- specification, one of:

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

The `Eq` Class – Equality

- specification, one of:
```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

The `Ord` Class – Orders

- prerequisite: `Eq`
- specification, one of:
```
compare :: Ord a => a -> a -> Ordering
(<=)    :: Ord a => a -> a -> Bool
```
- where `Ordering` = {`LT`, `EQ`, `GT`}
- additional functions: `(<)`, `(>=)`, `(>)`, `min`, `max`

The `Eq` Class – Equality

- specification, one of:
```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

The `Ord` Class – Orders

- prerequisite: `Eq`
- specification, one of:
```
compare :: Ord a => a -> a -> Ordering
(<=)    :: Ord a => a -> a -> Bool
```
- where `Ordering` = {`LT`, `EQ`, `GT`}
- additional functions: `(<)`, `(>=)`, `(>)`, `min`, `max`

The `Read` Class – "from string"

- useful functions:
```
read :: Read a => String -> a
```

The `Show` Class – "to string"

- specification, one of:

```
show     :: Show a => a -> String
showsPrec:: Show a => Int -> a -> String -> String
```

- additional functions: `showList`

The `Show` Class – "to string"

- specification, one of:

```
show      :: Show a => a -> String
showsPrec:: Show a => Int -> a -> String -> String
```

- additional functions: `showList`

The `Num` Class – Numeric Types

- prerequisites: `Eq` and `Show`
- specification, all of:

```
(+)         :: Num a => a -> a -> a
(*)         :: Num a => a -> a -> a
(-)         :: Num a => a -> a -> a
abs         :: Num a => a -> a
signum      :: Num a => a -> a
fromInteger :: Num a => Integer -> a
```

- additional functions: `negate`

The `Show` Class – "to string"

- specification, one of:

```
show      :: Show a => a -> String
showsPrec:: Show a => Int -> a -> String -> String
```

- additional functions: `showList`

The `Num` Class – Numeric Types

- prerequisites: `Eq` and `Show`
- specification, all of:

```
(+)         :: Num a => a -> a -> a
(*)         :: Num a => a -> a -> a
(-)         :: Num a => a -> a -> a
abs         :: Num a => a -> a
signum      :: Num a => a -> a
fromInteger :: Num a => Integer -> a
```

- additional functions: `negate`

visit: `http://haskell.org` → `Documentation` → `Language Report: Haskell 2010`

Lists

Constructing Lists

- `[a]` ::= `[]` | `a : [a]`
- for given list, exactly two cases: either empty `[]`, or contains at least one element x and a remaining list xs (x : xs)
- `[`x_1`,`x_2`,` ... `,`x_n`]` abbreviates x_1 : (x_2 : ($\cdots$: (x_n : `[]`) $\cdots$))
- `(:)` is right-associative, hence x_1 : (x_2 : xs) = x_1 : x_2 : xs

Constructing Lists

- `[a]` ::= `[]` | `a : [a]`
- for given list, exactly two cases: either empty `[]`, or contains at least one element x and a remaining list xs (x : xs)
- `[`x_1`,`x_2`,` ... `,`x_n`]` abbreviates x_1 : (x_2 : ($\cdots$: (x_n : `[]`) $\cdots$))
- `(:)` is right-associative, hence x_1 : (x_2 : xs) = x_1 : x_2 : xs

Examples

```
1 : (2 : (3 : (4 : []))) == 1 : 2 : 3 : 4 : []
1 : 2 : 3 : 4 : []       == [1,2,3,4]
1 : [2,3,4]              == [1,2,3,4]
```

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

A Polymorphic List Function

- polymorphic means "having many forms"
- definition

```
myReplicate n x =
  if n <= 0 then []
  else x : myReplicate (n - 1) x
```

- `myReplicate` has type `(Ord t, Num t) => t -> a -> [a]`
- can construct lists with elements of arbitrary type `a`, where length is given by some ordered numeric type `t`

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

A Polymorphic List Function

- polymorphic means "having many forms"
- definition

```
myReplicate n x =
  if n <= 0 then []
  else x : myReplicate (n - 1) x
```

- `myReplicate` has type `(Ord t, Num t) => t -> a -> [a]`
- can construct lists with elements of arbitrary type `a`, where length is given by some ordered numeric type `t`

Accessing List Elements – Selectors

- `head :: [a] -> a` – extract first element (fail on empty list)
- `tail :: [a] -> [a]` – drop first element (fail on empty list)

A Polymorphic List Function

- polymorphic means "having many forms"
- definition

```
myReplicate n x =
  if n <= 0 then []
  else x : myReplicate (n - 1) x
```

- `myReplicate` has type `(Ord t, Num t) => t -> a -> [a]`
- can construct lists with elements of arbitrary type `a`, where length is given by some ordered numeric type `t`

Exercise

use equational reasoning to evaluate `myReplicate 2 'c'`

Testing for Emptiness

- `null :: [a] -> Bool` – `True` iff argument is empty list

Testing for Emptiness

- `null :: [a] -> Bool` – `True` iff argument is empty list

Functions on Integer Lists

```
range m n =
  if m > n then []
  else m : range (m + 1) n

mySum xs =
  if null xs then 0
  else head xs + mySum (tail xs)

prod xs =
  if null xs then 1
  else head xs * prod (tail xs)
```

Examples

```
range 1 3 = [1,2,3]
range 3 2 = []

mySum [1,2,3] = 1 + 2 + 3 + 0
mySum [] = 0

prod [1,2,3] = 1 * 2 * 3 * 1
prod [] = 1
```

$$\texttt{mySum (range 1 n)} = \sum_{i=1}^{\texttt{n}} i$$

Patterns, Guards, and More

Patterns

- used to match specific cases

Patterns

- used to match specific cases
- defined by

$\langle pat \rangle$	::=	`_`	wildcard
	|	`x`	variable pattern
	|	`x` @ $\langle pat \rangle$	"as" pattern
	|	[$\langle pat \rangle$, ... , $\langle pat \rangle$]	list pattern
	|	($\langle pat \rangle$, ... , $\langle pat \rangle$)	tuple pattern
	|	`C` $\langle pat \rangle$ ... $\langle pat \rangle$	constructor pattern

Patterns

- used to match specific cases
- defined by

$$\begin{array}{lcll} \langle pat \rangle & ::= & \texttt{\_} & \text{wildcard} \\ & | & \texttt{x} & \text{variable pattern} \\ & | & \texttt{x}\,\texttt{@}\,\langle pat \rangle & \text{``as'' pattern} \\ & | & \texttt{[}\langle pat \rangle\texttt{,}\ldots\texttt{,}\langle pat \rangle\texttt{]} & \text{list pattern} \\ & | & \texttt{(}\langle pat \rangle\texttt{,}\ldots\texttt{,}\langle pat \rangle\texttt{)} & \text{tuple pattern} \\ & | & \texttt{C}\ \langle pat \rangle\ \ldots\ \langle pat \rangle & \text{constructor pattern} \end{array}$$

- _ matches everything and ignores the result

Patterns

- used to match specific cases
- defined by

⟨*pat*⟩	::=	`_`	wildcard
	|	`x`	variable pattern
	|	`x @` ⟨*pat*⟩	"as" pattern
	|	`[`⟨*pat*⟩`, ... ,`⟨*pat*⟩`]`	list pattern
	|	`(`⟨*pat*⟩`, ... ,`⟨*pat*⟩`)`	tuple pattern
	|	`C` ⟨*pat*⟩ ... ⟨*pat*⟩	constructor pattern

- `_` matches everything and ignores the result
- `x` matches everything and binds the result to `x`

Patterns

- used to match specific cases
- defined by

$$\begin{array}{rcll} \langle \mathit{pat} \rangle & ::= & \texttt{\_} & \text{wildcard} \\ & | & \texttt{x} & \text{variable pattern} \\ & | & \texttt{x}\,\texttt{@}\,\langle \mathit{pat} \rangle & \text{``as'' pattern} \\ & | & \texttt{[}\langle \mathit{pat} \rangle\texttt{,} \ldots \texttt{,}\langle \mathit{pat} \rangle\texttt{]} & \text{list pattern} \\ & | & \texttt{(}\langle \mathit{pat} \rangle\texttt{,} \ldots \texttt{,}\langle \mathit{pat} \rangle\texttt{)} & \text{tuple pattern} \\ & | & \texttt{C}\ \langle \mathit{pat} \rangle\ \ldots\ \langle \mathit{pat} \rangle & \text{constructor pattern} \end{array}$$

- `_` matches everything and ignores the result
- `x` matches everything and binds the result to `x`
- `x @` ⟨*pat*⟩ matches the same as ⟨*pat*⟩ and binds result to `x`

Patterns

- used to match specific cases
- defined by

⟨*pat*⟩	::=	`_`	wildcard
	|	`x`	variable pattern
	|	`x @` ⟨*pat*⟩	"as" pattern
	|	`[`⟨*pat*⟩`,` . . . `,`⟨*pat*⟩`]`	list pattern
	|	`(`⟨*pat*⟩`,` . . . `,`⟨*pat*⟩`)`	tuple pattern
	|	`C` ⟨*pat*⟩ . . . ⟨*pat*⟩	constructor pattern

- `_` matches everything and ignores the result
- `x` matches everything and binds the result to `x`
- `x @` ⟨*pat*⟩ matches the same as ⟨*pat*⟩ and binds result to `x`
- constructor patterns match the described application of a data constructor (example constructors are `(:)` and `[]` for lists, `True` and `False` for Boolean values, . . .)

Patterns

- used to match specific cases
- defined by

⟨*pat*⟩	::=	`_`	wildcard
	|	`x`	variable pattern
	|	`x @` ⟨*pat*⟩	"as" pattern
	|	`[`⟨*pat*⟩`,` . . . `,`⟨*pat*⟩`]`	list pattern
	|	`(`⟨*pat*⟩`,` . . . `,`⟨*pat*⟩`)`	tuple pattern
	|	`C` ⟨*pat*⟩ . . . ⟨*pat*⟩	constructor pattern

- `_` matches everything and ignores the result
- `x` matches everything and binds the result to `x`
- `x @` ⟨*pat*⟩ matches the same as ⟨*pat*⟩ and binds result to `x`
- constructor patterns match the described application of a data constructor (example constructors are `(:)` and `[]` for lists, `True` and `False` for Boolean values, . . .)
- patterns may be used in arguments of function definitions and together with the **`case`** construct

The `case` Construct

$$
\begin{aligned}
\texttt{case}\ e\ \texttt{of}\ \langle pat_1\rangle \quad & \texttt{->} \quad e_1 \\
\vdots \qquad & \\
\langle pat_n\rangle \quad & \texttt{->} \quad e_n
\end{aligned}
$$

- checks $\langle pat_1\rangle$ to $\langle pat_n\rangle$ top to bottom
- if $\langle pat_i\rangle$ is first match, e_i is evaluated

The `case` Construct

$$\texttt{case}\ e\ \texttt{of}\ \langle pat_1 \rangle\ \texttt{->}\ e_1$$
$$\vdots$$
$$\langle pat_n \rangle\ \texttt{->}\ e_n$$

- checks $\langle pat_1 \rangle$ to $\langle pat_n \rangle$ top to bottom
- if $\langle pat_i \rangle$ is first match, e_i is evaluated

Example – Pattern Matching

```
mySum [] = ... -- constructor pattern
fst (x, _) = x -- patterns: tuple, variable, wildcard
case xs of [x] -> ... -- patterns: list, variable
           _   -> ... -- wildcard
```

Pattern Guards

- a pattern may be followed by a guard b

Pattern Guards

- a pattern may be followed by a guard b
- syntax: $\langle \textit{pat} \rangle$ `|` b

Pattern Guards

- a pattern may be followed by a guard b
- syntax: $\langle \textit{pat} \rangle$ | b
- where b is a Boolean expression

Pattern Guards

- a pattern may be followed by a guard b
- syntax: $\langle$*pat*$\rangle$ | b
- where b is a Boolean expression

Example

```
f1 (x, _) | x >= 0  = x -- only if x non-negative
f2 (x:xs) | null xs = ... -- same as [x]
```

Refined Definitions

```
myReplicate n x | n <= 0    = []
                | otherwise = x : myReplicate (n - 1) x

range m n | m > n     = []
          | otherwise = m : range (m + 1) n

mySum []     = 0
mySum (x:xs) = x + mySum xs

prod []     = 1
prod (x:xs) = x * prod xs
```

Higher-Order Functions

Definition

a function is of higher-order if it takes functions as arguments

Definition

a function is of higher-order if it takes functions as arguments

Examples

```
twice f x = f (f x) -- apply f twice to x
```

Definition

a function is of higher-order if it takes functions as arguments

Examples

```
twice f x = f (f x) -- apply f twice to x
```

Sections

- abbreviation for partially applied infix operators
- `(x `op`)` abbreviates `(\y -> x `op` y)`
- `(`op` y)` abbreviates `(\x -> x `op` y)`

Definition

a function is of higher-order if it takes functions as arguments

Examples

```
twice f x = f (f x) -- apply f twice to x
```

Sections

- abbreviation for partially applied infix operators
- `(x `op`)` abbreviates `(\y -> x `op` y)`
- `(`op` y)` abbreviates `(\x -> x `op` y)`

Examples

```
ghci> twice (^2) 10
10000
```

Processing Lists – `map`

- possible definition

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

- syntactic sugar `map f xs` = `[f x | x <- xs]`

Processing Lists – map

- possible definition
```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```
- syntactic sugar `map f xs` = `[f x | x <- xs]`

Examples

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]
ghci> import Data.Char
ghci> map isDigit ['a','1','b','2']
[False,True,False,True]
ghci> map reverse ["abc","def","ghi"]
["cba","fed","ihg"]
ghci> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

Processing Lists – `filter`

- possible definition

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
  | p x        = x : filter p xs
  | otherwise = filter p xs
```

- syntactic sugar `filter p xs` = `[x | x <- xs, p x]`

Processing Lists – `filter`

- possible definition

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
  | p x        = x : filter p xs
  | otherwise = filter p xs
```

- syntactic sugar `filter p xs` = `[x | x <- xs, p x]`

Examples

```
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> filter (>5) [1..10]
[6,7,8,9,10]
ghci> filter (/= ' ') "abc def ghi"
"abcdefghi"
```

“Fold Right” – A Very Expressive Function

- possible definition

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []       = b
foldr f b (x:xs) = x `f` (foldr f b xs)
```

- `b` is ‘base value‘
- `f` combining function (binary)
- intuitively `foldr f b` $[x_1, x_2, \ldots, x_n]$

$$= \texttt{foldr f b} \; (x_1 \;\texttt{:}\; (x_2 \;\texttt{:}\; \cdots \; (x_n \;\texttt{:}\; \texttt{[]}) \cdots))$$
$$= (x_1 \;\texttt{`f`}\; (x_2 \;\texttt{`f`}\; \ldots \; (x_n \;\texttt{`f`}\; \texttt{b}\,) \ldots))$$

"Fold Right" – A Very Expressive Function

- possible definition

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []       = b
foldr f b (x:xs) = x `f` (foldr f b xs)
```

- b is 'base value'
- f combining function (binary)
- intuitively foldr f b $[x_1, x_2, \ldots, x_n]$

$$
\begin{aligned}
&= \texttt{foldr f b } (x_1 \ \texttt{:}\ (x_2 \ \texttt{:}\ \cdots \ (x_n \ \texttt{:}\ \texttt{[]})\cdots)) \\
&= (x_1 \ \texttt{`f`}\ (x_2 \ \texttt{`f`}\ \ldots \ (x_n \ \texttt{`f`}\ \texttt{b})\ldots))
\end{aligned}
$$

This Pattern is Very General

- take (+) for f and 0 for b: foldr (+) 0 = sum
- take (*) for f and 1 for b: foldr (*) 1 = product
- take const (+1) for f and 0 for b:
  foldr (const (+1)) 0 = length (where const f _ = f)

"Fold Right" – A Very Expressive Function

- possible definition

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []       = b
foldr f b (x:xs) = x `f` (foldr f b xs)
```

- b is 'base value'
- f combining function (binary)
- intuitively foldr f b [x_1,x_2,...,x_n]

$$= \texttt{foldr f b} \ (x_1 \ : \ (x_2 \ : \ \cdots \ (x_n \ : \ \texttt{[]}) \cdots))$$
$$= (x_1 \ \texttt{`f`} \ (x_2 \ \texttt{`f`} \ \ldots \ (x_n \ \texttt{`f`} \ \texttt{b}) \ldots))$$

This Pattern is Very General

add dummy argument

- take (+) for f and 0 for b: foldr (+) 0 = sum
- take (*) for f and 1 for b: foldr (*) 1 = product
- take const (+1) for f and 0 for b:
  foldr (const (+1)) 0 = length (where const f _ = f)

Homework (for November 9th)

1. Read Chapters 1 and 2 of Real World Haskell.
2. Work through lessons 4 to 5 on `http://tryhaskell.org/`.
3. Give the types (and class constraints) for each of:

```
pair x y          =  (x, y)
tail2 xs          =  tail (tail xs)
triple x          =  x * 3
thrice f x        =  f (f (f x))
mapPair f (x, y)  =  (f x, f y)
idList            =  filter (const True)
```

4. Use equational reasoning to stepwise compute the result of `filter (const False) ["a","b","c"]` on paper.
5. Using `foldr`, give alternative definitions of two of the functions we have seen so far (excluding those that we already defined via `foldr`).
6. Define a function `intercalate :: [a] -> [[a]] -> [a]` such that `intercalate xs xss` inserts the list `xs` between the lists in `xss` and concatenates the result.

   Example:

   `intercalate "; " ["one","two","six"]` = `"one; two; six"`