Lecture 2 – Programming in Haskell

Violet Ka I Pun

violet@ifi.uio.no

What is it?

What is it?

Generally speaking

- Supports and encourages programming in a functional style

Functional Language

What is it?

Generally speaking

- Supports and encourages programming in a functional style
- Functional programming

  – the basic method of computation is

  the application of functions to arguments

Functional Programming

Principles:

- Based on the evaluation of expressions

Functional Programming

Principles:

- Based on the evaluation of expressions
- Result depends only on the result of expressions

Functional Programming

Principles:

- Based on the evaluation of expressions
- Result depends only on the result of expressions

  no side-effects, not assignment to variables

Functional Programming

Principles:

- Based on the evaluation of expressions
- Result depends only on the result of expressions
  no side-effects, not assignment to variables
- Implicit memory-management (including 'garbage collection')

Functional Programming

Principles:

- Based on the evaluation of expressions
- Result depends only on the result of expressions
  no side-effects, not assignment to variables
- Implicit memory-management (including ‘garbage collection’)
- Functions are first-class objects

Background

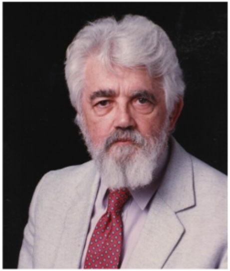
1930s:

Alonzo Church

Developed the *lambda calculus*, a simple but powerful theory of **functions**

Background

1950s:



John McCarthy

Developed *Lisp*

- The first functional language
- Some influences from the lambda calculus
- Variable assignments

FP language families

- LISP (1958), untyped, dialects:
    - Common Lisp
    - Scheme
- ML (1978), typed, dialects:
    - CAML, OCAML,
    - SMLNJ
    - Haskell

Useful resources

- Haskell homepage
  - http://www.haskell.org
- Learn in 10 minutes
  - http://www.haskell.org/haskellwiki/Learn_Haskell_in_10_minutes
- Real World Haskell
  - http://www.realwordlhaskell.org
- Programming in Haskell, slides
  - http://www.cs.nott.ac.uk/~gmh/book.html

Why FP?

- Very concise code (but, of course, Turing-complete)

Why FP?

- Very concise code (but, of course, Turing-complete)

Example: summing the integers 1 to 10

Why FP?

- Very concise code (but, of course, Turing-complete)

Example: summing the integers 1 to 10

Java

```
int total = 0;
for (int i = 1; i | 10; i++)
   total = total + i;
```

Why FP?

- Very concise code (but, of course, Turing-complete)

Example: summing the integers 1 to 10

Java

```
int total = 0;
for (int i = 1; i | 10; i++)
   total = total + i;
```

Haskell

```
sum [1..10]
```

Why FP?

- Very concise code (but, of course, Turing-complete)

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)
  - Polymorphism (reuse of algorithms)

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)
  - Polymorphism (reuse of algorithms)
  - Higher-order (functions in and out)

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)
  - Polymorphism (reuse of algorithms)
  - Higher-order (functions in and out)
- Important implementation techniques:

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)
  - Polymorphism (reuse of algorithms)
  - Higher-order (functions in and out)
- Important implementation techniques:
  - Local and global variables (stack)

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)
  - Polymorphism (reuse of algorithms)
  - Higher-order (functions in and out)
- Important implementation techniques:
  - Local and global variables (stack)
  - Type checking, type derivation

Why FP?

- Very concise code (but, of course, Turing-complete)
- Clearly formal semantics
- Important concepts within programming:
  - Recursion (data structures and algorithms)
  - Types (explicit and implicit)
  - Polymorphism (reuse of algorithms)
  - Higher-order (functions in and out)
- Important implementation techniques:
  - Local and global variables (stack)
  - Type checking, type derivation
  - 'garbage collection'

Useful things (ghci/Hugs)

Programs are written in single file (text editor), and
the file is loaded (from ghci/Hugs) with
`:load filename.hs`, or
`:l filename` (can use `:cd` *dir* to change to source file directory)

-- the file is **type-checked**, and
afterwards all functions can be called (no `Main`)

Useful things (ghci/Hugs)

Programs are written in single file (text editor), and
the file is loaded (from ghci/Hugs) with
`:load filename.hs`, or
`:l filename` (can use `:cd` *dir* to change to source file directory)

`--` the file is **type-checked**, and
afterwards all functions can be called (no `Main`)

Useful things (ghci/Hugs)

Programs are written in single file (text editor), and
the file is loaded (from ghci/Hugs) with
`:load filename.hs`, or
`:l filename` (can use `:cd` *dir* to change to source file directory)

-- the file is **type-checked**, and
afterwards all functions can be called (no `Main`)

`:reload` – load the program after editing
`:r`

Useful things (ghci/Hugs)

Programs are written in single file (text editor), and
the file is loaded (from ghci/Hugs) with
`:load filename.hs`, or
`:l filename` (can use `:cd` *dir* to change to source file directory)

`--` the file is **type-checked**, and
afterwards all functions can be called (no `Main`)

`:reload` – load the program after editing
`:r`

`:quit` – exit GHCI/Hugs
`:q`

Useful things (ghci/Hugs)

Programs are written in single file (text editor), and
the file is loaded (from ghci/Hugs) with
`:load filename.hs`, or
`:l filename` (can use `:cd` *`dir`* to change to source file directory)

-- the file is **type-checked**, and
afterwards all functions can be called (no `Main`)

`:reload` – load the program after editing
`:r`

`:quit` – exit GHCI/Hugs
`:q`

ghci is included in the Haskell platform.
Available from `www.haskell.org/platform`

Useful things (ghci/Hugs)

Programs are written in single file (text editor), and
the file is loaded (from ghci/Hugs) with
`:load filename.hs`, or
`:l filename`   (can use `:cd` *`dir`* to change to source file directory)

`--` the file is **type-checked**, and
afterwards all functions can be called (no `Main`)

`:reload` – load the program after editing
`:r`

`:quit` – exit GHCI/Hugs
`:q`

ghci is included in the Haskell platform.
Available from `www.haskell.org/platform`

Try Haskell: `http://tryhaskell.org/`

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

`inc n = n + 1` .... – try “inc 13”

Prog —> function-name argument-list **=** expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

```
inc n = n + 1
two = 2
```

`inc n = n + 1` .... – try "inc 13"

`two = 2` .... – try "inc two"

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

```
inc n = n + 1
```
.... – try "inc 13"

```
two = 2
```
.... – try "inc two"

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

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

```
inc n = n + 1
```
. . . . – try "inc 13"

```
two = 2
```
. . . . – try "inc two"

```
fst (x,y) = x
```
. . . . – Obs: what do the parentheses mean?

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

```
inc n = n + 1
```
. . . . – try "inc 13"

```
two = 2
```
. . . . – try "inc two"

```
fst (x,y) = x
```
. . . . – Obs: what do the parentheses mean?

```
snd x y = x
```

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

```
inc n = n + 1
```
.... – try "inc 13"

```
two = 2
```
.... – try "inc two"

```
fst (x,y) = x
```
.... – Obs: what do the parentheses mean?

```
snd x y = x
plus x y = x + y
```

Examples of functions

Prog —> function-name argument-list = expression { Prog }

Write in a file (text editor) and the file is loaded (from ghci/Hugs) with

`:load filename.hs`, or
`:l filename`.

```
inc n = n + 1
two = 2
fst (x,y) = x
snd x y = x
plus x y = x + y
suc = plus 1
```

inc n = n + 1 .... – try "inc 13"

two = 2 .... – try "inc two"

fst (x,y) = x .... – Obs: what do the parentheses mean?

suc = plus 1 .... – try "suc two"

Examples of functions

Prog —> function-name argument-list = expression { Prog }

One can use let
to define functions and variables "local" in
this session of ghci

- let `inc n = n + 1` . . . . – try "inc 13"
- let `two = 2` . . . . – try "inc two"
- let `fst (x,y) = x` . . . . – Obs: what do the parentheses mean?
- let `snd x y = x`
- let `plus x y = x + y`
- let `suc = plus 1` . . . . – try "suc two"

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`

Examples of local bindings (in expression with ‘in’)

- `let x = 2 in x*x + 1`
- `let x = 2 in let y = -1 in x*y + 1`
  `let x = 2 ; y = -1 in x*y + 1`

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`
  let `x = 2 ; y = -1` **in** `x*y + 1`
  let `x = 2` **in** `x*y + 1`
  where y = -1 .... error

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`
  let `x = 2 ; y = -1` **in** `x*y + 1`
  let `x = 2` **in** `x*y + 1`
  where y = -1 .... error
- **let** `y = x*x + x + 1` **where** `x = 2` .... ghci!
- **let** `x = 2 ; x = -1` **in** `x` .... error
- **let** `x = 2` **in let** `x = -1` **in** `x`

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`
  let `x = 2 ; y = -1` **in** `x*y + 1`
  let `x = 2` **in** `x*y + 1`
  where y = -1 . . . . error
- **let** `y = x*x + x + 1` **where** `x = 2` . . . . ghci!
- **let** `x = 2 ; x = -1` **in** `x` . . . . error
- **let** `x = 2` **in let** `x = -1` **in** `x` . . . . –1

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`
  let `x = 2 ; y = -1` **in** `x*y + 1`
  let `x = 2` **in** `x*y + 1`
  where y = -1 . . . . error
- **let** `y = x*x + x + 1` **where** `x = 2` . . . . ghci!
- **let** `x = 2 ; x = -1` **in** `x` . . . . error
- **let** `x = 2` **in let** `x = -1` **in** `x` . . . . –1
- **let** `x = 2 ; z = -1` **in** `x*y + z + 1` . . . . error
- `f 2` . . . . error

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- `let x = 2 in let y = -1 in x*y + 1`
  `let x = 2 ; y = -1 in x*y + 1`
  `let x = 2 in x*y + 1`
  `where y = -1` .... error
- `let y = x*x + x + 1 where x = 2` .... ghci!
- `let x = 2 ; x = -1 in x` .... error
- `let x = 2 in let x = -1 in x` .... –1
- `let x = 2 ; z = -1 in x*y + z + 1` .... error
- `f 2` .... error
- `let x = 2 ; y =x*x in y*y`

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`
  let `x = 2 ; y = -1` **in** `x*y + 1`
  let `x = 2` **in** `x*y + 1`
  where y = -1 . . . . error
- **let** `y = x*x + x + 1` **where** `x = 2` . . . . ghci!
- **let** `x = 2 ; x = -1` **in** `x` . . . . error
- **let** `x = 2` **in let** `x = -1` **in** `x` . . . . –1
- **let** `x = 2 ; z = -1` **in** `x*y + z + 1` . . . . error
- `f 2` . . . . error
- **let** `x = 2 ; y =x*x` **in** `y*y` 16

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- `let x = 2 in let y = -1 in x*y + 1`
  `let x = 2 ; y = -1 in x*y + 1`
  `let x = 2 in x*y + 1`
  `where y = -1` . . . . error
- `let y = x*x + x + 1 where x = 2` . . . . ghci!
- `let x = 2 ; x = -1 in x` . . . . error
- `let x = 2 in let x = -1 in x` . . . . –1
- `let x = 2 ; z = -1 in x*y + z + 1` . . . . error
- `f 2` . . . . error
- `let x = 2 ; y =x*x in y*y` 16
- `let y =x*x ; x = 2 in y*y`

Examples of local bindings (in expression with 'in')

- `let x = 2 in x*x + 1`
- **let** `x = 2` **in let** `y = -1` **in** `x*y + 1`
  let `x = 2 ; y = -1` **in** `x*y + 1`
  let `x = 2` **in** `x*y + 1`
  where y = -1 . . . . error
- **let** `y = x*x + x + 1` **where** `x = 2` . . . . ghci!
- **let** `x = 2 ; x = -1` **in** `x` . . . . error
- **let** `x = 2` **in let** `x = -1` **in** `x` . . . . –1
- **let** `x = 2 ; z = -1` **in** `x*y + z + 1` . . . . error
- `f 2` . . . . error
- **let** `x = 2 ; y =x*x` **in** `y*y` 16
- **let** `y =x*x ; x = 2` **in** `y*y` . . . . 16 – lazy (!)

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

```
fst (suc two, 2+2)
```

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

```
fst (suc two, 2+2)
```

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

```
fst (suc two, 2+2)
```

$= \text{suc two}$

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

```
fst (suc two, 2+2)
```

$$= \text{suc two}$$
$$= \text{plus 1 two}$$

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

```
fst (suc two, 2+2)
```

$$
\begin{aligned}
&= \text{suc two}\\
&= \text{plus 1 two}\\
&= 1 + \text{two}
\end{aligned}
$$

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$
\begin{aligned}
&= \text{suc two} \\
&= \text{plus } 1 \text{ two} \\
&= 1 + \text{two} \\
&= 1 + 2
\end{aligned}
$$

Evaluation

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$\begin{aligned} &= \text{suc two} \\ &= \text{plus } 1 \text{ two} \\ &= 1 + \text{two} \\ &= 1 + 2 \\ &= 3 \end{aligned}$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

```
fst (suc two, 2+2)
```

$$
\begin{aligned}
&= \text{suc two} \\
&= \text{plus } 1 \text{ two} \\
&= 1 + \text{two} \\
&= 1 + 2 \\
&= 3
\end{aligned}
$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)` eager, "call-by-value", "inside-out"

$$
\begin{aligned}
&= \text{suc two} \\
&= \text{plus } 1 \text{ two} \\
&= 1 + \text{two} \\
&= 1 + 2 \\
&= 3
\end{aligned}
$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$
\begin{aligned}
&= \text{suc two} \\
&= \text{plus } 1 \text{ two} \\
&= 1 + \text{two} \\
&= 1 + 2 \\
&= 3
\end{aligned}
$$

eager, "call-by-value", "inside-out"

$$= \text{fst } (\text{suc } 2,\ 2+2)$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$
\begin{aligned}
&= \text{suc two}\\
&= \text{plus } 1 \text{ two}\\
&= 1 + \text{two}\\
&= 1 + 2\\
&= 3
\end{aligned}
$$

eager, "call-by-value", "inside-out"

$$
\begin{aligned}
&= \text{fst } (\text{suc } 2,\ 2{+}2)\\
&= \text{fst } (\text{plus } 1\ 2,\ 2{+}2)
\end{aligned}
$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

= suc two
= plus 1 two
= 1 + two
= 1 + 2
= 3

eager, "call-by-value", "inside-out"

= fst (suc 2, 2+2)
= fst (plus 1 2, 2+2)
= fst (1+2, 2+2)

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$
\begin{aligned}
&= \text{suc two} \\
&= \text{plus 1 two} \\
&= 1 + \text{two} \\
&= 1 + 2 \\
&= 3
\end{aligned}
$$

eager, "call-by-value", "inside-out"

$$
\begin{aligned}
&= \text{fst (suc 2, 2+2)} \\
&= \text{fst (plus 1 2, 2+2)} \\
&= \text{fst (1+2, 2+2)} \\
&= \text{fst (3, 2+2)}
\end{aligned}
$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$\begin{aligned} &= \text{suc two} \\ &= \text{plus 1 two} \\ &= 1 + \text{two} \\ &= 1 + 2 \\ &= 3 \end{aligned}$$

eager, "call-by-value", "inside-out"

$$\begin{aligned} &= \text{fst (suc 2, 2+2)} \\ &= \text{fst (plus 1 2, 2+2)} \\ &= \text{fst (1+2, 2+2)} \\ &= \text{fst (3, 2+2)} \\ &= \text{fst (3, 4)} \end{aligned}$$

Evaluation – lazy: "call by need", "outside-in"

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

= suc two
= plus 1 two
= 1 + two
= 1 + 2
= 3

eager, "call-by-value", "inside-out"

= fst (suc 2, 2+2)
= fst (plus 1 2, 2+2)
= fst (1+2, 2+2)
= fst (3, 2+2)
= fst (3, 4)
= 3

Evaluation – lazy: “call by need”, “outside-in”

- `two = 2`
- `fst (x,y) = x`
- `plus x y = x + y`
- `suc = plus 1`

`fst (suc two, 2+2)`

$$
\begin{aligned}
&= \text{suc two}\\
&= \text{plus 1 two}\\
&= 1 + \text{two}\\
&= 1 + 2\\
&= 3
\end{aligned}
$$

eager, “call-by-value”, “inside-out”

$$
\begin{aligned}
&= \text{fst (suc 2, 2+2)}\\
&= \text{fst (plus 1 2, 2+2)}\\
&= \text{fst (1+2, 2+2)}\\
&= \text{fst (3, 2+2)}\\
&= \text{fst (3, 4)}\\
&= 3
\end{aligned}
$$

Useful things (Haskell)

– function and variable names start with small letters

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! $\quad$ f a b $\neq$ f(a,b)

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

```
f :: A -> B -> ?
```

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $-\ -1$ is incorrect

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \; -1$ is incorrect $-(-1)$ is correct

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \ -1$ is incorrect $-(-1)$ is correct

f a + b written in maths f(a) + b

– function application has higher precedence

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--$ 1 is comment $-\ -1$ is incorrect $-(-1)$ is correct

f a + b written in maths f(a) + b

– function application has higher precedence

f a b + c*d written in maths

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--$ 1 is comment $-\ -1$ is incorrect $-(-1)$ is correct

f a + b written in maths f(a) + b

– function application has higher precedence

f a b + c*d written in maths f(a,b) + c*d

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \; -1$ is incorrect $-(-1)$ is correct

f a + b written in maths f(a) + b

– function application has higher precedence

f a b + c*d written in maths f(a,b) + c*d

f a (g b) written in maths

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--$ 1 is comment $-\ -1$ is incorrect $-(-1)$ is correct

f a + b written in maths f(a) + b

– function application has higher precedence

f a b + c*d written in maths f(a,b) + c*d

f a (g b) written in maths f(a, g(b))

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \ -1$ is incorrect $-(-1)$ is correct

f a + b written in maths f(a) + b

– function application has higher precedence

f a b + c*d written in maths f(a,b) + c*d

f a (g b) written in maths f(a, g(b))

f a g b written in maths

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \ -1$ is incorrect $-(-1)$ is correct

f a + b	written in maths	f(a) + b
	– function application has higher precedence	
f a b + c*d	written in maths	f(a,b) + c*d
f a (g b)	written in maths	f(a, g(b))
f a g b	written in maths	f(a, g, b)

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \; -1$ is incorrect $-(-1)$ is correct

f a + b	written in maths	f(a) + b
	– function application has higher precedence	
f a b + c*d	written in maths	f(a,b) + c*d
f a (g b)	written in maths	f(a, g(b))
f a g b	written in maths	f(a, g, b)

binary f x y = ..., can be called as infix: a ‘f‘ b (with backquotes).

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \ -1$ is incorrect $-(-1)$ is correct

f a + b	written in maths	f(a) + b
	– function application has higher precedence	
f a b + c*d	written in maths	f(a,b) + c*d
f a (g b)	written in maths	f(a, g(b))
f a g b	written in maths	f(a, g, b)

binary f x y = ..., can be called as infix: a ‘f‘ b (with backquotes).

-- comment in a line

Useful things (Haskell)

– function and variable names start with small letters
– Names of data-constructors start with capital letters (True,False...)

Parentheses!!! f a b $\neq$ f(a,b)

assume a :: A and b :: B

`f :: A -> B -> ?` `f :: (A, B) -> ?`

$--1$ is comment $- \ -1$ is incorrect $-(-1)$ is correct

f a + b	written in maths	f(a) + b
	– function application has higher precedence	
f a b + c*d	written in maths	f(a,b) + c*d
f a (g b)	written in maths	f(a, g(b))
f a g b	written in maths	f(a, g, b)

binary f x y = ..., can be called as infix: a ‘f‘ b (with backquotes).

-- comment in a line

{- start of a comment that ends with -}

Some convention...

1. function and argument name MUST start with small letters
   – list arguments have, as a convention, _s at the end: x vs. xs

Some convention...

1. function and argument name MUST start with small letters
   – list arguments have, as a convention, _s at the end: x vs. xs
2. Definitions MUST start in same column – indentation counts!

f = ...	f = ...
g = ...	g = ...
h = ...	h = ...
OK	ERROR

Some convention...

1. function and argument name MUST start with small letters
   – list arguments have, as a convention, _s at the end: x vs. xs
2. Definitions MUST start in same column – indentation counts!

f = ... g = ... h = ...	f = ... g = ... h = ...
OK	ERROR

3. Indentation CAN be used for grouping:

a = b + c
      where
        b = 2
        c = 3
d = a * 2
	a = b + c
	      where
	        { b = 2 ; c = 3 }
	
	d = a * 2

implicit	explicit

Some convention...

1. function and argument name MUST start with small letters
   – list arguments have, as a convention, _s at the end: x vs. xs
2. Definitions MUST start in same column – indentation counts!

f = ...	f = ...
g = ...	g = ...
h = ...	h = ...
OK	ERROR

3. Indentation CAN be used for grouping:

```
a  =  b + c                 a  =  b + c
      where                       where
        b = 2                       { b = 2 ; c = 3 }
        c = 3
d  =  a * 2                 d  =  a * 2
```

implicit	explicit with {_}, not (_)

No loops...

$$\text{fac n} = \prod_{i=1}^{i=n} i$$

No loops...

$$\text{fac n} = \prod_{i=1}^{i=n} i$$

fac n $=$

No loops...

$$\text{fac n} = \prod_{i=1}^{i=n} i$$

fac n =
  res := 1

No loops...

$$\text{fac n} = \prod_{i=1}^{i=n} i$$

fac n =
  res := 1
  for (i:=1, i++, n)

No loops...

$$\text{fac n} = \prod_{i=1}^{i=n} i$$

```
fac n =
  res := 1
  for (i:=1, i++, n)
     res := res * n
```

No loops...

$$\text{fac n} = \prod_{i=1}^{i=n} i$$

```
fac n =
  res := 1
  for (i:=1, i++, n)
     res := res * n
  return res
```

...Only recursion

- `fac n = if n <= 0 then 1 else n * fac (n-1)`

. . . Only recursion

- `fac n = if n <= 0 then 1 else n * fac (n-1)`
- `fac1 n = product [1..n]`

...Only recursion

- `fac n = if n <= 0 then 1 else n * fac (n-1)`
- `fac1 n = product [1..n]`
- ```
  fac2 0 = 1
  fac2 n = n * fac2(n-1)
  ```
  ```

...Only recursion

- `fac n =` **if** `n <= 0` **then** `1` **else** `n * fac (n-1)`
- `fac1 n =` **product** `[1..n]`
- ```
  fac2 0 = 1
  fac2 n = n * fac2(n-1)
  ```
- ```
  fib n = if n <= 1 then 1
          else fib (n-1) + fib (n-2)
  ```

...Only recursion

- `fac n = if n <= 0 then 1 else n * fac (n-1)`
- `fac1 n = product [1..n]`
- ```
  fac2 0 = 1
  fac2 n = n * fac2(n-1)
  ```
- ```
  fib n = if n <= 1 then 1
          else fib (n-1) + fib (n-2)
  ```
- ```
  fib' n = if n <= 1 then (1,1) else
             let (x,y) = fib' (n-1) in (x+y,x)
  ```
  ```

. . . Only recursion

- `fac n = ` **if** ` n <= 0 ` **then** ` 1 ` **else** ` n * fac (n-1)`
- `fac1 n = ` **product** ` [1..n]`
- ```
  fac2 0 = 1
  fac2 n = n * fac2(n-1)
  ```
- ```
  fib n = if n <= 1 then 1
          else fib (n-1) + fib (n-2)
  ```
- ```
  fib' n = if n <= 1 then (1,1) else
             let (x,y) = fib' (n-1) in (x+y,x)
  ```
- Important difference between the last two: running time
  ```

Examples: lists and patterns

empty list [], or x:xs

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if [] ==** xs **then** 0 **else head** xs + sm1 (**tail** xs)

Examples: lists and patterns

empty list [], or x:xs $\quad$ [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs

Examples: lists and patterns

empty list [], or x:xs $\quad$ [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs $\quad$ show: sm2 [x] = x

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys =

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) =

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]

Examples: lists and patterns

empty list [], or x:xs [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x] built in as reverse

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]   built in as reverse
- bub [] = []
  bub [x] = [x]

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]   built in as reverse
- bub [] = []
  bub [x] = [x]
  bub (x:y:xs) =

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]   built in as reverse
- bub [] = []
  bub [x] = [x]
  bub (x:y:xs) = **if** x<y **then** x:bub(y:xs)

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]   built in as reverse
- bub [] = []
  bub [x] = [x]
  bub (x:y:xs) = **if** x<y **then** x:bub(y:xs) **else** y:bub(x:xs)

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]   built in as reverse
- bub [] = []
  bub [x] = [x]
  bub (x:y:xs) = **if** x<y **then** x:bub(y:xs) **else** y:bub(x:xs)

  bubs [] = []

Examples: lists and patterns

empty list [], or x:xs [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x] built in as reverse
- bub [] = []
  bub [x] = [x]
  bub (x:y:xs) = **if** x<y **then** x:bub(y:xs) **else** y:bub(x:xs)

  bubs [] = []
  bubs (x:xs) = bub(x:bubs(xs))

Examples: lists and patterns

empty list [], or x:xs   [1,2,3] = 1:[2,3] = 1:2:[3] = 1:2:3:[]

- sm1 xs = **if** **null** xs **then** 0 **else** **head** xs + sm1 (**tail** xs)
- sm2 [] = 0
  sm2 (x:xs) = x + sm2 xs   show: sm2 [x] = x
- append [] ys = ys
  append (x:xs) ys = x:append xs ys   is built in as: xs **++** ys
- rev [] = []
  rev (x:xs) = rev(xs) ++ [x]   built in as reverse
- bub [] = []
  bub [x] = [x]
  bub (x:y:xs) = **if** x<y **then** x:bub(y:xs) **else** y:bub(x:xs)

  bubs [] = []
  bubs (x:xs) = bub(x:bubs(xs))

Some list functions

- the type 'list a' is written as [a]

Some list functions

- the type ‘list a’ is written as [a]
- head :: [a] $\rightarrow$ a　　　　head [1,2,3,4,5] $=$ 1

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] $=$ 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] $=$ 5

Some list functions

- the type ‘list a’ is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] $=$ 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] $=$ 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] $=$ [2,3,4,5]

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] $=$ 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] $=$ 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] $=$ [2,3,4,5]
- $++$ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]$++$[4,5] $=$ [1,2,3,4,5]

Some list functions

- the type 'list a' is written as [a]
- head :: $[a] \to a$ — head [1,2,3,4,5] $= 1$
- last :: $[a] \to a$ — last [1,2,3,4,5] $= 5$
- tail :: $[a] \to [a]$ — tail [1,2,3,4,5] $=$ [2,3,4,5]
- $++$ :: $[a] \to [a] \to [a]$ — [1,2,3]$++$[4,5] $=$ [1,2,3,4,5]
- length :: $[a] \to$ Int — length [1,2,3,4,5] $= 5$

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- $++$ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]$++$[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- ++ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]++[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]
- take :: Int $\rightarrow$ [a] $\rightarrow$ [a] — take 2 [1,2,3,4,5] = [1,2]

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- ++ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]++[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]
- take :: Int $\rightarrow$ [a] $\rightarrow$ [a] — take 2 [1,2,3,4,5] = [1,2]
- drop :: Int $\rightarrow$ [a] $\rightarrow$ [a] — drop 2 [1,2,3,4,5] = [3,4,5]

Some list functions

- the type ‘list a’ is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- ++ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]++[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]
- take :: Int $\rightarrow$ [a] $\rightarrow$ [a] — take 2 [1,2,3,4,5] = [1,2]
- drop :: Int $\rightarrow$ [a] $\rightarrow$ [a] — drop 2 [1,2,3,4,5] = [3,4,5]
- !! :: [a] $\rightarrow$ Int $\rightarrow$ a — [5,6,7,8] !! 2 = 7

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\to$ a — head [1,2,3,4,5] $=$ 1
- last :: [a] $\to$ a — last [1,2,3,4,5] $=$ 5
- tail :: [a] $\to$ [a] — tail [1,2,3,4,5] $=$ [2,3,4,5]
- $++$ :: [a] $\to$ [a] $\to$ [a] — [1,2,3]$++$[4,5] $=$ [1,2,3,4,5]
- length :: [a] $\to$ Int — length [1,2,3,4,5] $=$ 5
- reverse :: [a] $\to$ [a] — reverse [1,2,3,4,5] $=$ [5,4,3,2,1]
- take :: Int $\to$ [a] $\to$ [a] — take 2 [1,2,3,4,5] $=$ [1,2]
- drop :: Int $\to$ [a] $\to$ [a] — drop 2 [1,2,3,4,5] $=$ [3,4,5]
- !! :: [a] $\to$ Int $\to$ a — [5,6,7,8] !! 2 $=$ 7

  list !! length list

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- ++ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]++[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]
- take :: Int $\rightarrow$ [a] $\rightarrow$ [a] — take 2 [1,2,3,4,5] = [1,2]
- drop :: Int $\rightarrow$ [a] $\rightarrow$ [a] — drop 2 [1,2,3,4,5] = [3,4,5]
- !! :: [a] $\rightarrow$ Int $\rightarrow$ a — [5,6,7,8] !! 2 = 7

  list !! length list — *** Exception. Prelude(!!): Index too large

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- ++ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]++[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]
- take :: Int $\rightarrow$ [a] $\rightarrow$ [a] — take 2 [1,2,3,4,5] = [1,2]
- drop :: Int $\rightarrow$ [a] $\rightarrow$ [a] — drop 2 [1,2,3,4,5] = [3,4,5]
- !! :: [a] $\rightarrow$ Int $\rightarrow$ a — [5,6,7,8] !! 2 = 7

  list !! length list — *** Exception. Prelude(!!): Index too large
- sum :: Num a => [a] $\rightarrow$ a — sum [1,2,3,4] = 10

Some list functions

- the type 'list a' is written as [a]
- head :: [a] $\rightarrow$ a — head [1,2,3,4,5] = 1
- last :: [a] $\rightarrow$ a — last [1,2,3,4,5] = 5
- tail :: [a] $\rightarrow$ [a] — tail [1,2,3,4,5] = [2,3,4,5]
- ++ :: [a] $\rightarrow$ [a] $\rightarrow$ [a] — [1,2,3]++[4,5] = [1,2,3,4,5]
- length :: [a] $\rightarrow$ Int — length [1,2,3,4,5] = 5
- reverse :: [a] $\rightarrow$ [a] — reverse [1,2,3,4,5] = [5,4,3,2,1]
- take :: Int $\rightarrow$ [a] $\rightarrow$ [a] — take 2 [1,2,3,4,5] = [1,2]
- drop :: Int $\rightarrow$ [a] $\rightarrow$ [a] — drop 2 [1,2,3,4,5] = [3,4,5]
- !! :: [a] $\rightarrow$ Int $\rightarrow$ a — [5,6,7,8] !! 2 = 7

  list !! length list — <u>*** Exception. Prelude(!!): Index too large</u>
- sum :: Num a => [a] $\rightarrow$ a — sum [1,2,3,4] = 10
- product :: Num a => [a] $\rightarrow$ a — product [1,2,3,4] = 24

Examples

- last $[x] = x$

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

$>$ ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

$>$ take 5 ones

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

\> ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

\> take 5 ones
[1,1,1,1,1]

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

$>$ ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

$>$ take 5 ones
[1,1,1,1,1]

- find the smallest number in a non-empty list:

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

> take 5 ones
  [1,1,1,1,1]

- find the smallest number in a non-empty list:
  mm [x] = x

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

> take 5 ones
[1,1,1,1,1]

- find the smallest number in a non-empty list:
  mm [x] = x
  mm (x:y:xs) =

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

$>$ ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

$>$ take 5 ones
[1,1,1,1,1]

- find the smallest number in a non-empty list:
  mm [x] = x
  mm (x:y:xs) = **let** z = mm(y:xs) **in**

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

> take 5 ones
[1,1,1,1,1]

- find the smallest number in a non-empty list:
  mm [x] = x
  mm (x:y:xs) = **let** z = mm(y:xs) **in if** x < z **then** x **else** z

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

> take 5 ones
[1,1,1,1,1]

- find the smallest number in a non-empty list:
  mm [x] = x
  mm (x:y:xs) = **let** z = mm(y:xs) **in if** x < z **then** x **else** z

> mm [] = ???

Examples

- last [x] = x
  last (x:y:ys) = last (y:ys)
- ones = 1:ones

> ones
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...

> take 5 ones
  [1,1,1,1,1]

- find the smallest number in a non-empty list:
  mm [x] = x
  mm (x:y:xs) = **let** z = mm(y:xs) **in if** x < z **then** x **else** z

> mm [] = ??? ***Exception: filename : (lnr) : Non-exhaustive patterns in function mm

Examples: list division

- divide the list into 2 (almost) even parts

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = []

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = []

  split xs =

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = []

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = []   – x ‘div‘ y = $\lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = [] – x ‘div‘ y $= \lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

Examples: list division

- divide the list into 2 (almost) even parts

split [] = ([],[])   -- x `div` y = $\lfloor \frac{x}{y} \rfloor$

split xs = **let** h = (length xs) `div` 2 **in** (take h xs, drop h xs)

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = ([],[]) – x ‘div‘ y = $\lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

- divide the list into two: one with odd and another with even indices

  di [1,2,3,4,5,6] = ([1,3,5],[2,4,6])

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = ([],[]) – x ‘div‘ y = $\lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

- divide the list into two: one with odd and another with even indices

  di [1,2,3,4,5,6] = ([1,3,5],[2,4,6])

– di [] = ([],[])

  di (x:xs) =

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = ([],[]) – x ‘div‘ y = $\lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

- divide the list into two: one with odd and another with even indices

  di [1,2,3,4,5,6] = ([1,3,5],[2,4,6])

– di [] = ([],[])

  di (x:xs) = **let** (a,b) = di2 xs **in** (x:a,b)

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = ([],[]) – x ‘div‘ y = $\lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

- divide the list into two: one with odd and another with even indices

  di [1,2,3,4,5,6] = ([1,3,5],[2,4,6])

– di [] = ([],[])
  di (x:xs) = **let** (a,b) = di2 xs **in** (x:a,b)

– di2 [] = ([],[])
  di2 (x:xs) =

Examples: list division

- divide the list into 2 (almost) even parts

  split [] = ([],[]) — x ‘div‘ y = $\lfloor \frac{x}{y} \rfloor$

  split xs = **let** h = (length xs) ‘div‘ 2 **in** (take h xs, drop h xs)

- divide the list into two: one with odd and another with even indices

  di [1,2,3,4,5,6] = ([1,3,5],[2,4,6])

– di [] = ([],[])
  di (x:xs) = **let** (a,b) = di2 xs **in** (x:a,b)

– di2 [] = ([],[])
  di2 (x:xs) = **let** (a,b) = di xs **in** (a,x:b)

Examples

1. merge two sorted lists into a sorted list:

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys $=$ ys

   merge xs [] $=$ xs

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
           if x < y **then**

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
           if x < y **then** x:(merge xs (y:ys)) **else**

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
   if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =

   if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. mergesort:

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
   if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. mergesort:

   ms [] = []
   ms [x] = [x]

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
   if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. mergesort:

   ms [] = []
   ms [x] = [x]

   ms xs = **let**

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
       if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. mergesort:

   ms [] = []
   ms [x] = [x]

   ms xs = **let**

Examples

1. merge two sorted lists into a sorted list:
   merge [] ys = ys
   merge xs [] = xs
   merge (x:xs) (y:ys) =
       if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. mergesort:
   ms [] = []
   ms [x] = [x]
   ms xs = **let** l = (length xs) ‘div‘ 2 ;
           a = ms (take l xs) ;
           b = ms (drop l xs)
       in merge a b

Examples

1. merge two sorted lists into a sorted list:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
        if x < y then x:(merge xs (y:ys)) else y:(merge (x:xs) ys)
```

2. mergesort:

```
ms [] = []
ms [x] = [x]
ms xs = let l = (length xs) `div` 2 ;
              a = ms (take l xs) ;
              b = ms (drop l xs)
           in merge a b
```

Examples

1. merge two sorted lists into a sorted list:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
      if x < y then x:(merge xs (y:ys)) else y:(merge (x:xs) ys)
```

2. mergesort:

```
ms [] = []
ms [x] = [x]
ms xs = let l = (length xs) `div` 2 ;
            a = ms (take l xs) ;
            b = ms (drop l xs)
        in merge a b
```

Examples

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =

           if x < y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. mergesort:

   ms [] = []
   ms [x] = [x]

   ms xs = **let** (aa,bb) = split xs   – split:: [a] -> ([a],[a])

split [] = ([], [])
split xs = let h = (length xs) ‘div‘ 2 in (take h xs, drop h xs)

Examples

1. merge two sorted lists into a sorted list:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
        if x < y then x:(merge xs (y:ys)) else y:(merge (x:xs) ys)
```

2. mergesort:

```
ms [] = []
ms [x] = [x]
ms xs = let  (aa,bb) = split xs              – split:: [a] -> ([a],[a])
           in merge (ms aa) (ms bb)
```

```
split [] = ( [], [] )
split xs = let h = (length xs) ‘div‘ 2 in ( take h xs, drop h xs )
```

Example - quicksort

qs [] = []

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. less x ls = for i in [0..length(ls)-1] ...

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
   less x (y:ys) =

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then**

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) =

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = less x xs ++ [x] ++ grtr x xs

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = qs (less x xs) ++ [x] ++ qs (grtr x xs)

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = qs (less x xs) ++ [x] ++ qs (grtr x xs)

C. less x ys = [y | y <– ys, y < x]

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = qs (less x xs) ++ [x] ++ qs (grtr x xs)

C. less x ys = [y | y <- ys, y < x]   generator

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = qs (less x xs) ++ [x] ++ qs (grtr x xs)

C. less x ys = [y | y <- ys, y < x] guard

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
   less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
   grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = qs (less x xs) ++ [x] ++ qs (grtr x xs)

C. less x ys = [y | y <– ys, y < x]

qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >=x]

Example - quicksort

qs [] = []

? find smaller/larger (than pivot) elements from the list:

A. ~~less x ls = for i in [0..length(ls)-1] ...~~

B. less x [] = []
less x (y:ys) = **if** (x > y) **then** (y:less x ys) **else** (less x ys)

B+. grtr x [] = []
grtr x (y:ys) = **if** (x<=y) **then** y:grtr x ys **else** grtr x ys

B. qs (x:xs) = qs (less x xs) ++ [x] ++ qs (grtr x xs)

C. less x ys = [y | y <– ys, y < x]

qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >=x]

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

Is the result sorted?

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

Is the result sorted?

- sorted [] = True

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

Is the result sorted?

- sorted [] = True

  sorted [x] = True

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

Is the result sorted?

- sorted [] = True

  sorted [x] = True

  sorted (x:y:xs) = x <= y && sorted (y:xs)

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

Is the result sorted?

- sorted [] = True

  sorted [x] = True

  sorted (x:y:xs) = x <= y && sorted (y:xs)

> quickCheck (\li -> sorted (qs li))

- import Test.QuickCheck – at the top of the program

```
qs (x:z) = qs [ y | y <- z, y < x ] ++ [x] ++ qs [ y | y <- z, y >= x ]
```

Is the result sorted?

- sorted [] = True

```
sorted [] = True
sorted [x] = True
sorted (x:y:xs) = x <= y && sorted (y:xs)
```

```
> quickCheck (\li -> sorted (qs li))
+++ OK, passed 100 tests.
```

QuickCheck – quicksort

- import Test.QuickCheck – at the top of the program

  qs (x:z) = qs [y | y <– z, y < x] ++ [x] ++ qs [y | y <– z, y >= x]

Is the result sorted?

- sorted [] = True

  sorted [x] = True

  sorted (x:y:xs) = x <= y && sorted (y:xs)

> quickCheck (\li -> sorted (qs li))

+++ OK, passed 100 tests.

- quickCheck generates random test data for most of the types (in particular, for basic types like Int, [Int], Char, String)

QuickCheck – mergesort

1. merge two sorted lists into a sorted list:

QuickCheck – mergesort

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

QuickCheck – mergesort

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

QuickCheck – mergesort

1. merge two sorted lists into a sorted list:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
        if x<y then x:(merge xs (y:ys)) else y:(merge (x:xs) ys)
```

QuickCheck – mergesort

1. merge two sorted lists into a sorted list:

   merge [] ys = ys

   merge xs [] = xs

   merge (x:xs) (y:ys) =
   if x<y **then** x:(merge xs (y:ys)) **else** y:(merge (x:xs) ys)

2. split a list into two parts:
   split [] = ([], [])
   split xs = let h = (length xs) 'div' 2 in (take h xs, drop h xs)

QuickCheck – mergesort

1. merge two sorted lists into a sorted list:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
        if x<y then x:(merge xs (y:ys)) else y:(merge (x:xs) ys)
```

2. split a list into two parts:

```
split [] = ( [], [] )
split xs = let h = (length xs) `div` 2 in ( take h xs, drop h xs )
```

3. mergesort:

```
ms [] = []
ms [x] = [x]
ms xs = let (aa,bb) = split xs in merge (ms aa) (ms bb)
```

quickCheck – mergesort. . .

4. quickCheck (\li -> sorted (ms li))

4. quickCheck (\li -> sorted (ms li))
5. properties can be collected in the file – name should start with prop_

4. quickCheck (\li -> sorted (ms li))
5. properties can be collected in the file – name should start with prop_

```
prop_merge x y =
      if (sorted x && sorted y) then sorted (merge x y)
                                else True
         where types = x::[Int]
```

4. quickCheck (\li -> sorted (ms li))
5. properties can be collected in the file – name should start with prop_

```
prop_merge x y =
        if (sorted x && sorted y) then sorted (merge x y)
                  else True
           where types = x::[Int]
```

```
> quickCheck prop_merge
```