



Functional Programming

SS 2025

Torsten Grust
University of Tübingen

Administrivia


Slot	Time	Room
Lectures	Thursday, 10:15–11:45	A301 (Sand 13)
Tutorials	Tuesday, 12:15–13:45	F122 (Sand 6/7)

- Lecture: notes on slides and board + live Haskell coding
- **! Sign up for the FP Discord**  (once there, please use command `/verify` first): <https://db.cs.uni-tuebingen.de/discord>
- These slides and Haskell code are downloadable on **GitHub** . See Discord for repo URL and instructions on how to use `git`.

Tutorials + Exercises





- Held weekly (organized by Tim Fischer + Björn Bamberg).
- Tutorials start on **Tuesday, April 29, 2025**.
- New exercise sheets with a handful of problems every Tuesday, hand in by next Tuesday evening.
 - Exercises will mostly make you read/write Haskell **≡** code.
- We use `git` to distribute exercise sheets and collect your solutions (see Discord for details on how to `git pull`, `commit`, and `push`).

Grading + Exam

- We will distribute, collect, and grade **weekly assignments** (Tuesday→Tuesday) via GitHub .
- Score $\geq \frac{2}{3}$ of assignment points to be admitted to end-term exam.
- **No teams**¹, everyone hands in their individual solutions.
- **Final exam on Thursday, July 24, 2025, 10:00–12:00.**
 - Room *tbd*.
 - You may bring a double-sided DIN A4 cheat sheet of notes.

¹ We will switch to teams of two should the number of active attendees be larger than expected.

Install Haskell! ➤

- Multi-platform installer for Haskell: **GHcup**
 - <https://www.haskell.org/ghcup/>
 - Available for Win (WSL2) , macOS , Linux , FreeBSD 
- Haskell compiler **ghc** (Glorious Glasgow Haskell Compiler)
 - Current version 9.12.2, any recent version 9.x should be OK
- Includes Haskell REPL **ghci**
 - To **ghci**'s configuration (usually in `~/.ghci`), add the following switch:

```
:seti -XMonomorphismRestriction
```

Further Reading

- Bird: “Thinking Functionally with Haskell”, Cambridge University Press 2015
- Allen: “Haskell Programming From First Principles”, Gumroad 2016, <http://haskellbook.com>
- FP Complete: “The School of Haskell” at <http://www.schoolofhaskell.com>
- Diehl: “What I Wish I Knew When Learning Haskell” at <https://smunix.github.io/dev.stephendiehl.com/hask>

Functional Programming (*FP*)

A programming language is a medium for **expressing ideas** (not to get a computer perform operations). Thus programs must be written for people to read, and only incidentally for machines to execute.

- Computational model in FP: **reduction** (replace **expressions** by their **value**).

Functional Programming (FP)

In FP, expressions are formed by **applying functions to values**.

1. **Functions as in maths:** $x = y \Rightarrow f(x) = f(y)$
2. **Functions are values** just like numbers or text.

	FP	Imperative
program construction	function application + composition	statement sequencing
execution	reduction (evaluation)	state changes
semantics	λ calculus	Turing machines

- Absence of explicit machine control generally leads to concise, often quite elegant, programs. Focus remains on problem. Programs tends to be easier to reason about.

Example


$n \in \mathbb{N}$, $n \geq 2$ is a **prime number** iff the set of non-trivial factors of n is empty:

$$n \text{ is prime} \Leftrightarrow \{ m \mid m \in \{ 2, \dots, n-1 \}, n \bmod m = 0 \} = \emptyset$$

Haskell Ramp-Up: Function Application and Composition

- Read \equiv as “denotes the same value as”.
- **Apply** f to value e : f_e (“apply”, space $_$ as invisible binary operator, juxtaposition, Haskell speak: `infixl 10 _`)
 - $_$ has max precedence (10): $f\ e_1 + e_2 \equiv (f\ e_1) + e_2$
 - $_$ associates to the left (l): $g\ f\ e \equiv (g\ f)\ e$
- Function **composition**:
 - $g\ (f\ e)$
 - Operator $.$ (“after”): $(g\ .\ f)\ e$ ($.$ = \circ as in $g\ \circ\ f$)
 - Alternative “apply” operator $\$$ (lowest precedence, associates to the right, `infixr 0 \$`): $g\ \$\ f\ \$\ e \equiv g\ (f\ e)$

Infix vs. Prefix Operators

- Prefix application of binary infix operator \otimes :
 $(\otimes) e_1 e_2 \equiv e_1 \otimes e_2$
 - Example: $(\&\&) \text{ True False} \equiv \text{False}$
- Infix application of binary function f :
 $e_1 \text{ `f` } e_2 \equiv f e_1 e_2$ (```: backtick)
 - $x \text{ `elem` } [1,2,3]$
 - $n \text{ `mod` } 2$
- User-defined infix operators, built from symbols
`!#$%&*+ /<=>?@\^|~:.`
 -  Identifiers starting with `:` reserved to denote value constructors of algebraic data types.

Values and Types

- Read `::` as “*has type*”.

Any Haskell value `e` has a type `t` (`e :: t`) that is determined at **compile time**. The `::` type assignment is either given explicitly or inferred by the compiler.

Basic Built-In Haskell Types

Type	Description	Values
<code>Int</code>	64-bit integers $[-2^{63} .. 2^{63}-1]$	<code>0</code> , <code>1</code> , <code>(-42)</code>
<code>Integer</code>	arbitrary-precision integers	<code>0</code> , <code>10^100</code>
<code>Float</code>	single-precision floating points	<code>0.1</code> , <code>1e02</code>
<code>Double</code>	double-precision floating points	<code>0.42</code> , <code>1e-2</code>
<code>Char</code>	Unicode characters	<code>'x'</code> , <code>'\t'</code> , <code>'λ'</code> , <code>'\8710'</code> , <code>'\^G'</code>
<code>Bool</code>	Booleans	<code>True</code> , <code>False</code>
<code>()</code>	“unit” (single-value type)	<code>()</code> (C: <code>void</code>)

Type Constructors

- **Type constructors** build new types from existing types.
- Let a , b , ... denote arbitrary types (type variables):

Type Constructor	Description	Values
(a, b)	pairs of values of types a , b	$(1, \text{True}) :: (\text{Integer}, \text{Bool})$
(a_1, a_2, \dots, a_n)	n -tuples	
$[a]$	lists of values of type a	$[\text{True}, \text{False}] :: [\text{Bool}]$, $[] :: [a]$
$\text{Maybe } a$	optional value of type a	$\text{Just } 42 :: \text{Maybe Integer}$, $\text{Nothing} :: \text{Maybe } a$
$\text{Either } a \ b$	choice between a and b	$\text{Left 'x'} :: \text{Either Char } b$, $\text{Right pi} :: \text{Either } a \ \text{Double}$
$\text{IO } a$	I/O action that returns value of type a (once performed)	$\text{print } 42 :: \text{IO } ()$, $\text{getChar} :: \text{IO Char}$
$a \rightarrow b$	function from type a to b	$\text{isLetter} :: \text{Char} \rightarrow \text{Bool}$

Currying

- Recall:

1. $e_1 ++ e_2 \equiv (++) e_1 e_2$
2. $(++) e_1 e_2 \equiv ((++) e_1) e_2$

- Function application happens one argument at a time (**currying**, *Haskell B. Curry*)
- Type of n -ary function: $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow b$. Type constructor \rightarrow associates to the right, thus read as:

$$a_1 \rightarrow (a_2 \rightarrow (\dots \rightarrow (a_n \rightarrow b)\dots))$$

- Enables **partial application**: “Give me a value of type a_1 , I'll give you a $(n-1)$ -ary function of type $a_2 \rightarrow \dots \rightarrow a_n \rightarrow b$ ”.

Defining Values (and thus: Functions)

- **= binds** names to values, value names must not start with **A-Z** (idiomatic Haskell style: **camelCase**).
- Define constant (0-ary function) **c**, value of **c** is that of expression **e**: **c = e**
- Define **n**-ary function **f**, arguments **x_i** and **f** may occur in **e** (no **letrec** needed): **f x₁ x₂ ... x_n = e**
- Haskell program:
set of top-level bindings (order immaterial, **no** rebinding!)
- Good style: give type assignments for top-level bindings:

```
f :: a1 -> a2 -> b  
f x1 x2 = e
```


Guards

- **Guards** (introduced by `|`) are multi-way conditional expressions:

```
f x1 x2 ... xn
  | q1 = e1
  | q2 = e2
  | q3 = e3
  ...
```

- Guards `qi` (expressions of type `Bool`) evaluated top to bottom, first `True` guard wins. Syntactic sugar: `otherwise` \equiv `True`.
- Compare to math notation:

$$fac\ n = \begin{cases} 1 & , \text{ if } n \leq 1 \\ n * fac\ (n-1) & , \text{ otherwise} \end{cases}$$

- $$f \ x_1 \ x_2 \ \dots \ x_n \ \Big| \ \begin{array}{l} p_1 \vartriangleleft = e_1 \vartriangleleft \\ p_2 \vartriangleleft = e_2 \vartriangleleft \\ \dots \end{array} \quad \text{-- } \vartriangleleft: g_i \text{ in scope}$$
- where
- $$g_1 = \dots \vartriangleleft$$
- $$g_2 = \dots \vartriangleleft$$

- ```
let g_1 = ... \vartriangleleft
 g_2 = ... \vartriangleleft
in e \vartriangleleft -- \vartriangleleft : g_i in scope
```

## Layout (Two-Dimensional Syntax)

---

- The Haskell compiler applies these transformation rules to the program source before compilation begins:
  1. The first token *after* a `where/let` and the first token of a top-level definition define the upper-left corner `┌` of a box.
  2. The first token *left* of the box closes the box `└` (“*offside rule*”).
  3. Insert `{` before the box.
  4. Insert `}` after the box.
  5. Insert `;` before a line that starts at left box border.

## Layout (Example)

---

### 1. Original source:

```
let y = a * b
 f x = (x + y) / y
in f c + f d
```

### 2. Make box visible:

```
let
 y = a * b
 f x = (x + y) / y

in f c + f d -- offside: in
```

### 3. After source transformation:

```
let {y = a * b
 ;f x = (x + y) / y}
in f c + f d
```

## Lists — The Go-to Container Data Structure in FP

---

- Recursive definition of lists:

- `[]` is a list (*nil*), type: `[] :: [a]`
- `x : xs` is a list, if `x :: a` and `xs :: [a]`.  
           ↑   ↑  
       head tail

“*cons*”: `(:) :: a -> [a] -> [a]` with infixr 5 :

- Abbreviate long chains of *cons* using `[...]`:

`3:(2:(1:[]))`  $\equiv$  `3:2:1:[]`  $\equiv$  `[3,2,1]` ( $\equiv$  `3:[2,1]`)

## Lists

---

- Law (⚠ `head`, `tail` are *partial functions*):

$\forall xs \neq []: \text{head } xs : \text{tail } xs \equiv xs$

- Type `String` is a synonym for type `[Char]` (“list of characters”).
- Introduce your own **type synonyms** via (type names: **Uppercase**):

```
type $t_1 = t_2$
```

## Lists

---

- Sequences (of enumerable elements):

$[x..y] \equiv \text{enumFromTo } x \ y$

$[x, x+1, x+2, \dots, y]$

$[x, s..y] \equiv \text{enumFromThenTo } x \ s \ y$

$[x, x+\Delta, x+2*\Delta, \dots, y]$   
**where**  $\Delta = s-x$

$[x..] \equiv \text{enumFrom } x$

$[x, x+1, x+2, \dots]$

$[x, s..] \equiv \text{enumFromThen } x \ s$

$[x, x+\Delta, x+2*\Delta, \dots]$

## Pattern Matching

---

- The idiomatic Haskell way to **define a function by cases**:

```
f :: a1 -> ... -> ak -> b
f p11 ... p1k = e1
f p21 ... p2k = e2
 ⋮
f pn1 ... pnk = en
```


- We have  $e_i :: b$  for all  $i \in \{1, \dots, n\}$
- On a call  $f\ x_1\ x_2\ \dots\ x_k$ , each  $x_i$  is **matched** against **patterns**  $p_{1i}, \dots, p_{ni}$  in order. Result is  $e_r$  if the  $r$ th branch is the first in which all patterns match.



## Pattern Matching

---

| Pattern                      | Matches if...                                            | Bindings in $e_r$                        |
|------------------------------|----------------------------------------------------------|------------------------------------------|
| constant $c$                 | $x_i == c$                                               |                                          |
| variable $v$                 | always                                                   | $v = x_i$                                |
| wildcard $_$                 | always                                                   |                                          |
| tuple<br>$(p_1, \dots, p_m)$ | components of $x_i$ match component<br>patterns $p$      | those bound by component<br>patterns $p$ |
| $[]$                         | $x_i == []$                                              |                                          |
| $p_1:p_2$                    | head $x_i$ matches $p_1$ and tail $x_i$<br>matches $p_2$ | those bound by $p_1$ and $p_2$           |
| $v@p$                        | $p$ matches                                              | those bound by $p$ and $v = x_i$         |

-  In a pattern, a variable may only occur once (*patterns are linear*).

## Pattern Matching in Expressions (**case**)

---

- Pattern matching may be used in any expression (not just in function definitions): **case** expressions.

Matches against patterns  $p_i$  as well as guards  $q_{ij}$  may be used together:

|                                 |  |                             |
|---------------------------------|--|-----------------------------|
| <b>case</b> $e$ <b>of</b> $p_1$ |  | $q_{11} \rightarrow e_{11}$ |
|                                 |  | $q_{12} \rightarrow e_{12}$ |
|                                 |  | $\vdots$                    |
| $p_n$                           |  | $q_{n1} \rightarrow e_{n1}$ |
|                                 |  | $q_{n2} \rightarrow e_{n2}$ |