Functional Programming

Wolfram Kahl

kahl@mcmaster.ca

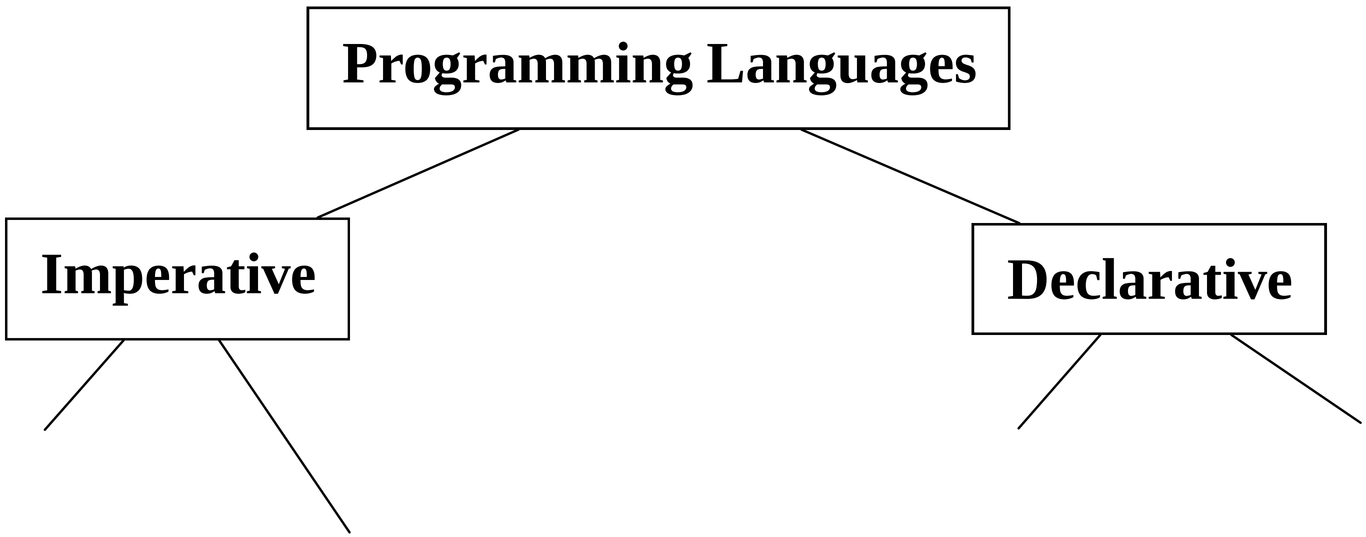
Department of Computing and Software
McMaster University

What Kinds of Programming Languages are There?

What Kinds of Programming Languages are There?

Imperative — "telling the machine what to **do**"

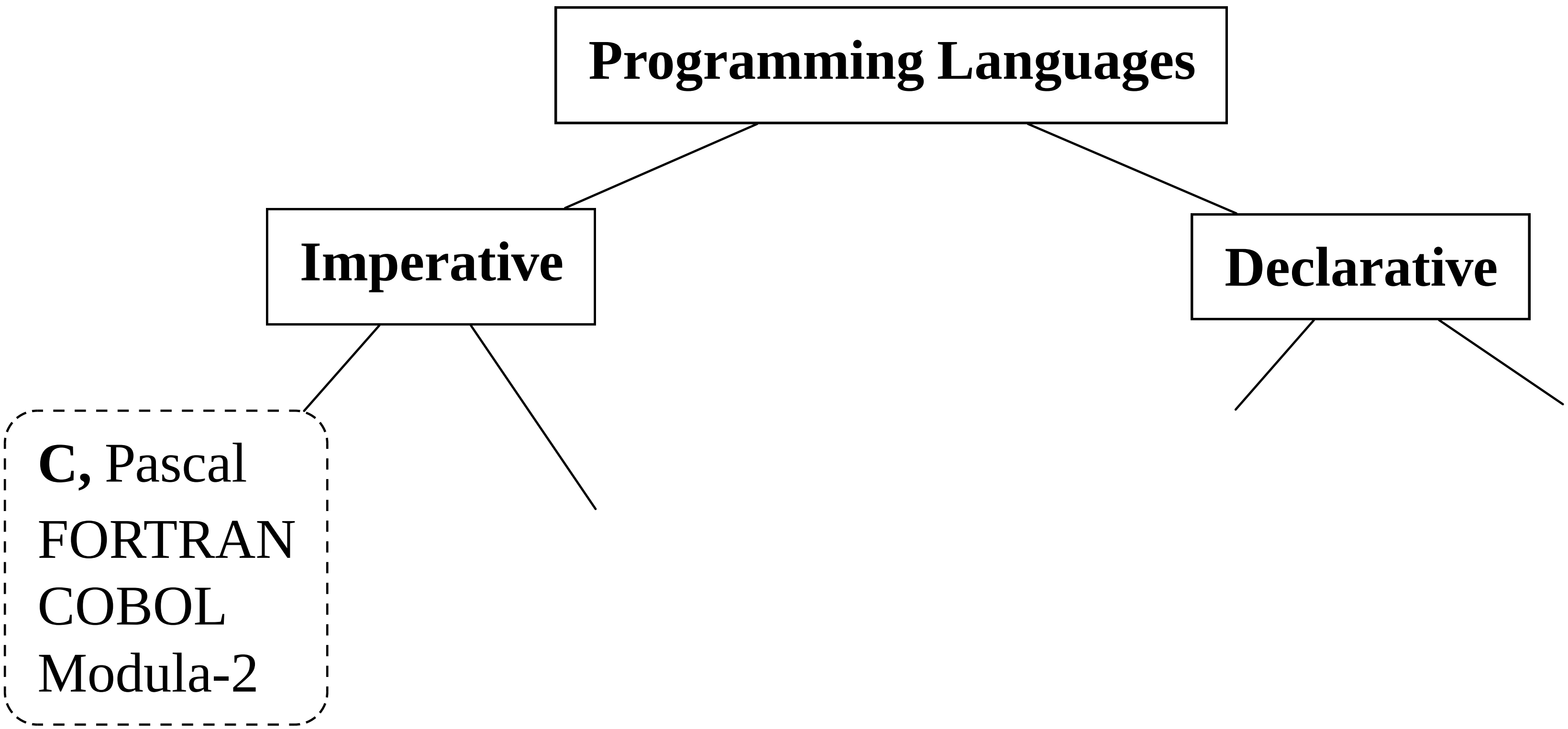
Declarative — "telling the machine what to **achieve**"



What Kinds of Programming Languages are There?

Imperative — "telling the machine what to **do**"

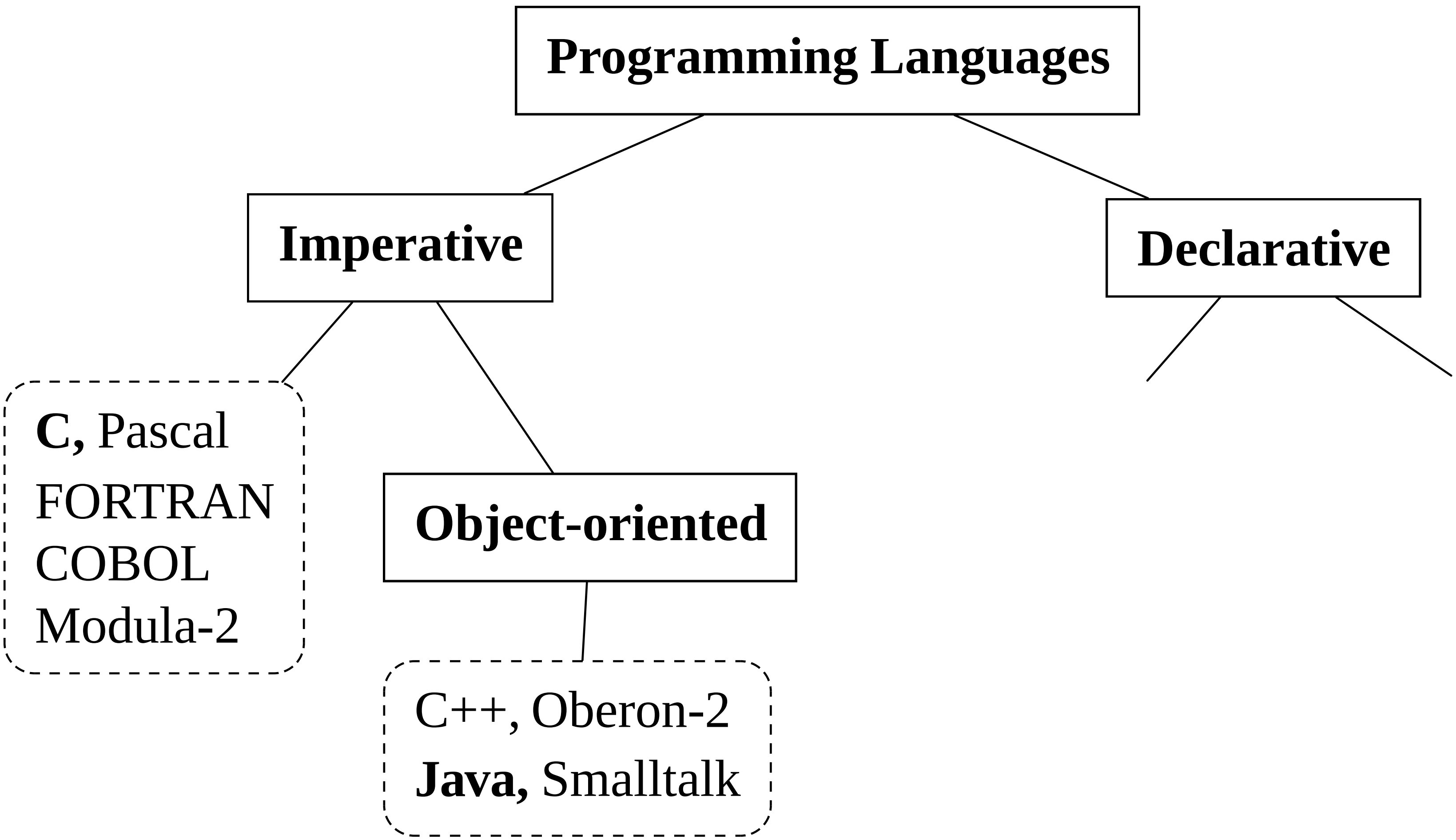
Declarative — "telling the machine what to **achieve**"



What Kinds of Programming Languages are There?

Imperative — "telling the machine what to **do**"

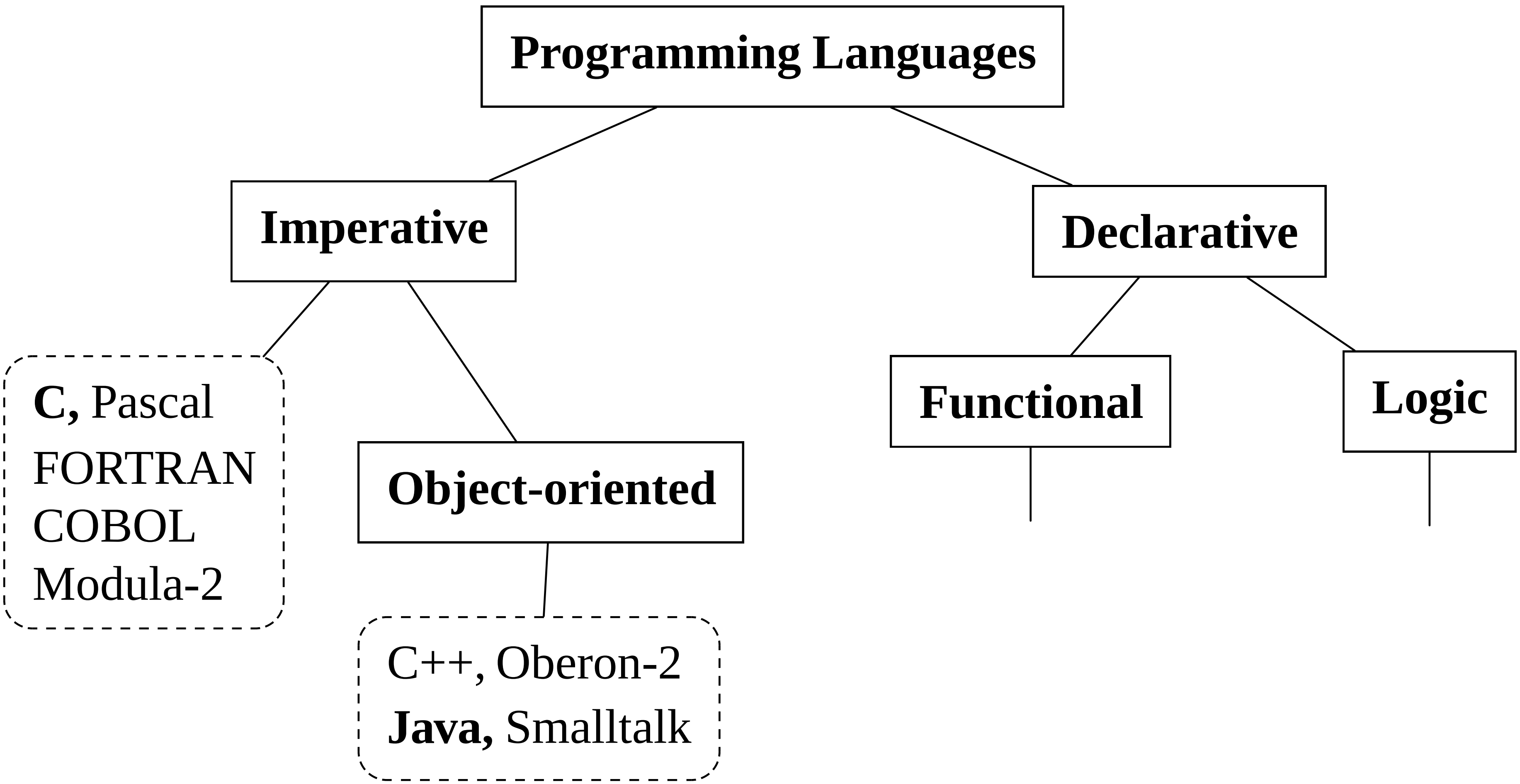
Declarative — "telling the machine what to **achieve**"



What Kinds of Programming Languages are There?

Imperative — “telling the machine what to **do**”

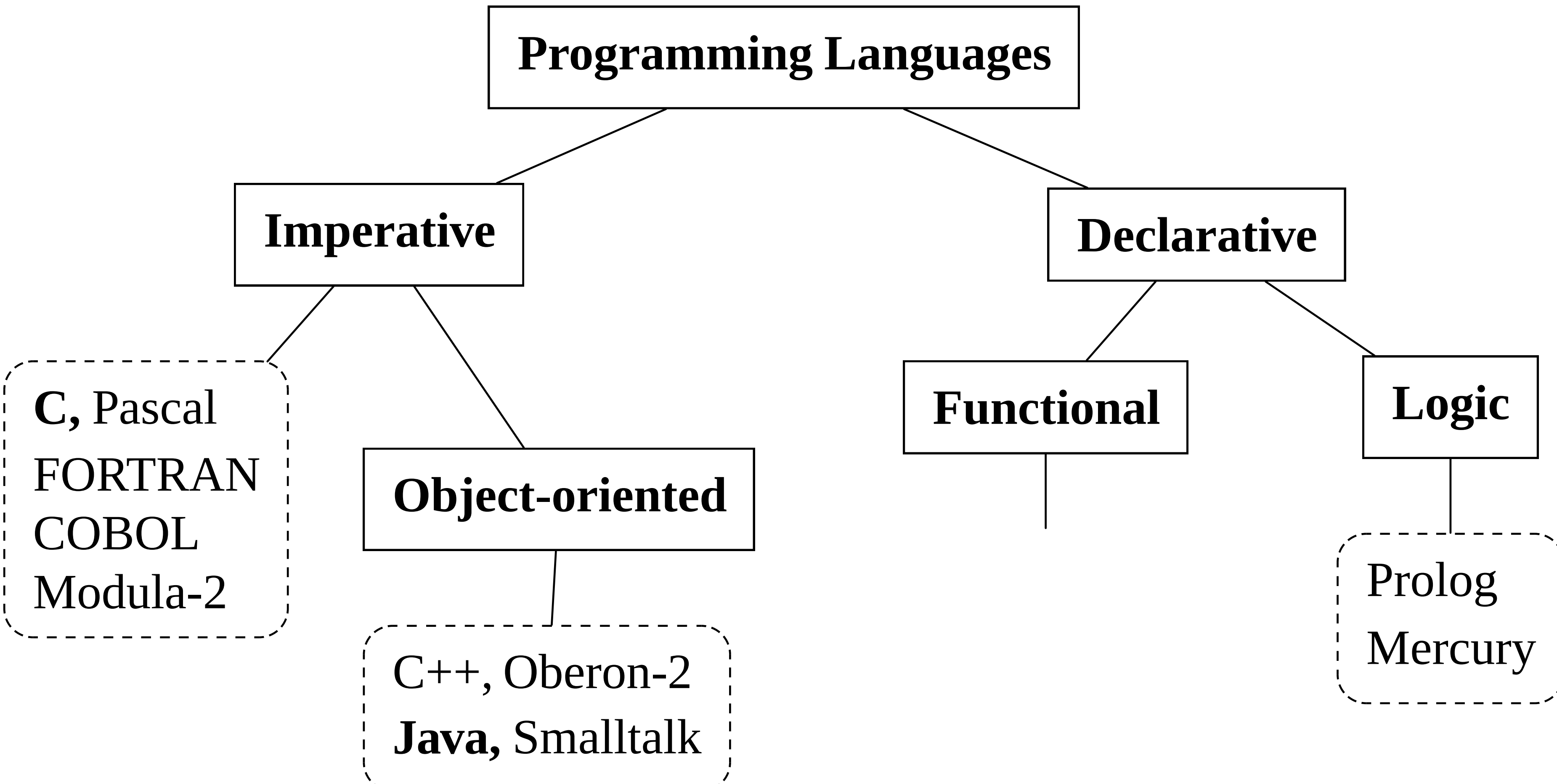
Declarative — “telling the machine what to **achieve**”



What Kinds of Programming Languages are There?

Imperative — “telling the machine what to **do**”

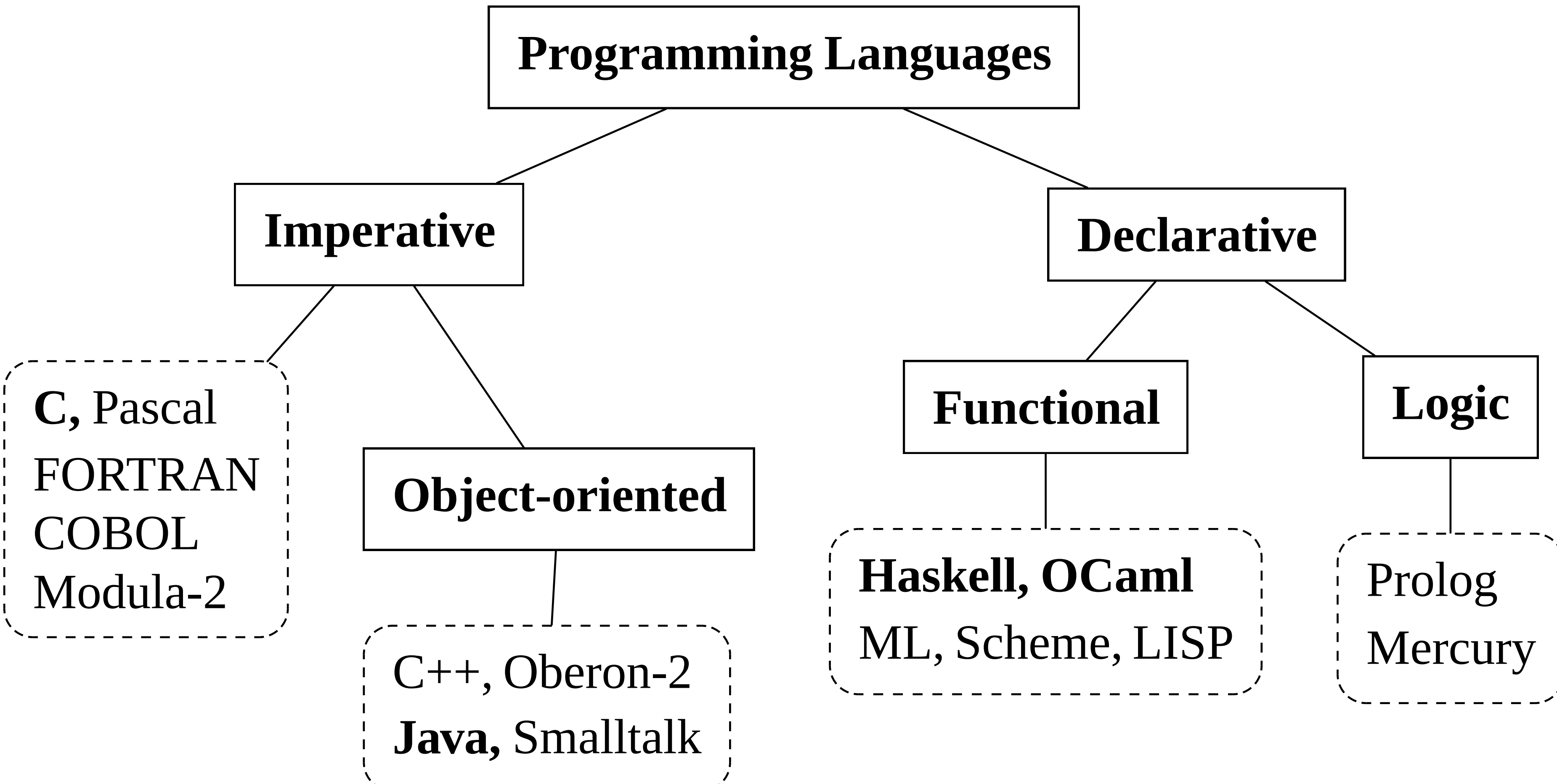
Declarative — “telling the machine what to **achieve**”



What Kinds of Programming Languages are There?

Imperative — “telling the machine what to **do**”

Declarative — “telling the machine what to **achieve**”



Programming Language Paradigms

Imperative Programming Languages

Statement oriented languages

Every statement changes the machine state

Object-oriented languages

Organising the state into *objects* with individual state and behaviour

Message passing paradigm (instead of subprogram call)

Rule-Based (Logical) Programming Languages

Specify rule that specifies problem solution (Prolog, BNF Parsing)

Other examples: Decision procedures, Grammar rules (BNF)

Programming consists of specifying the attributes of the answer

Functional (Applicative) Programming Languages

Goal is to understand the function that produces the answer

Function composition is major operation

Programming consists of building the function that computes the answer

Historical Development of Programming Languages

Historical Development of Programming Languages

Emphasis has changed:

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

– to making it easier for the programmer.

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

– to making it easier for the programmer.

Easier for the programmer means:

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

– to making it easier for the programmer.

Easier for the programmer means:

– Use languages that facilitate writing **error-free programs**

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

– to making it easier for the programmer.

Easier for the programmer means:

– Use languages that facilitate writing **error-free programs**

– Use languages that facilitate writing programs that are **easy to maintain**

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

– to making it easier for the programmer.

Easier for the programmer means:

– Use languages that facilitate writing **error-free programs**

– Use languages that facilitate writing programs that are **easy to maintain**

Goal of language development:

Historical Development of Programming Languages

Emphasis has changed:

– from making life easier for the computer

– to making it easier for the programmer.

Easier for the programmer means:

– Use languages that facilitate writing **error-free programs**

– Use languages that facilitate writing programs that are **easy to maintain**

Goal of language development:

– Developers concentrate on design (or even just specification)

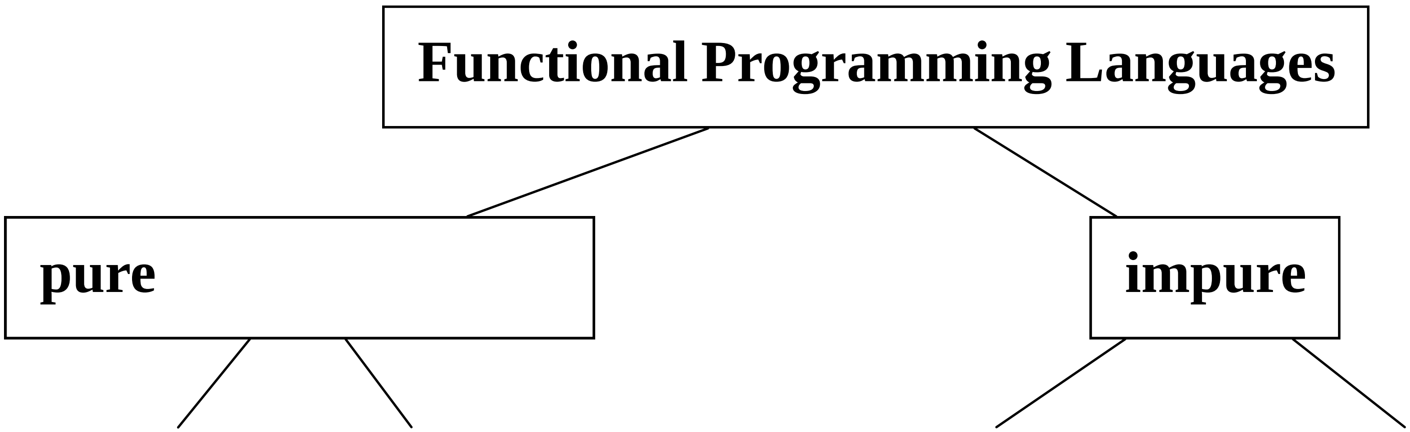
– Programming is trivial or handled by computer

(*executable specification languages, rapid prototyping*)

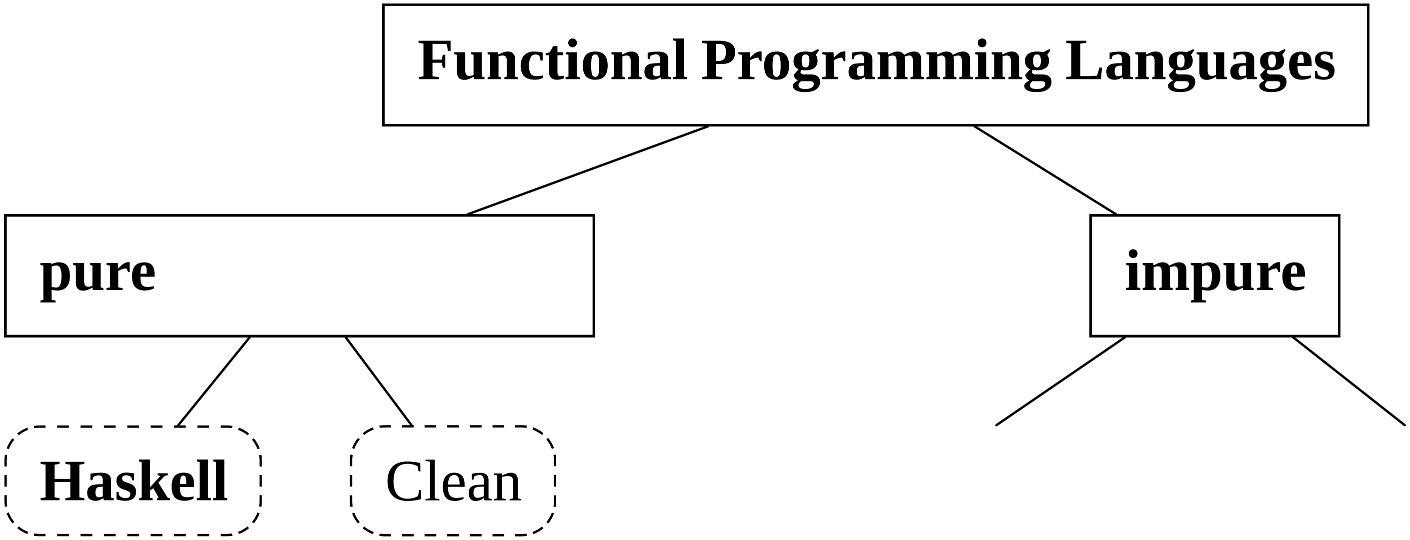
Important Functional Programming Languages

Functional Programming Languages

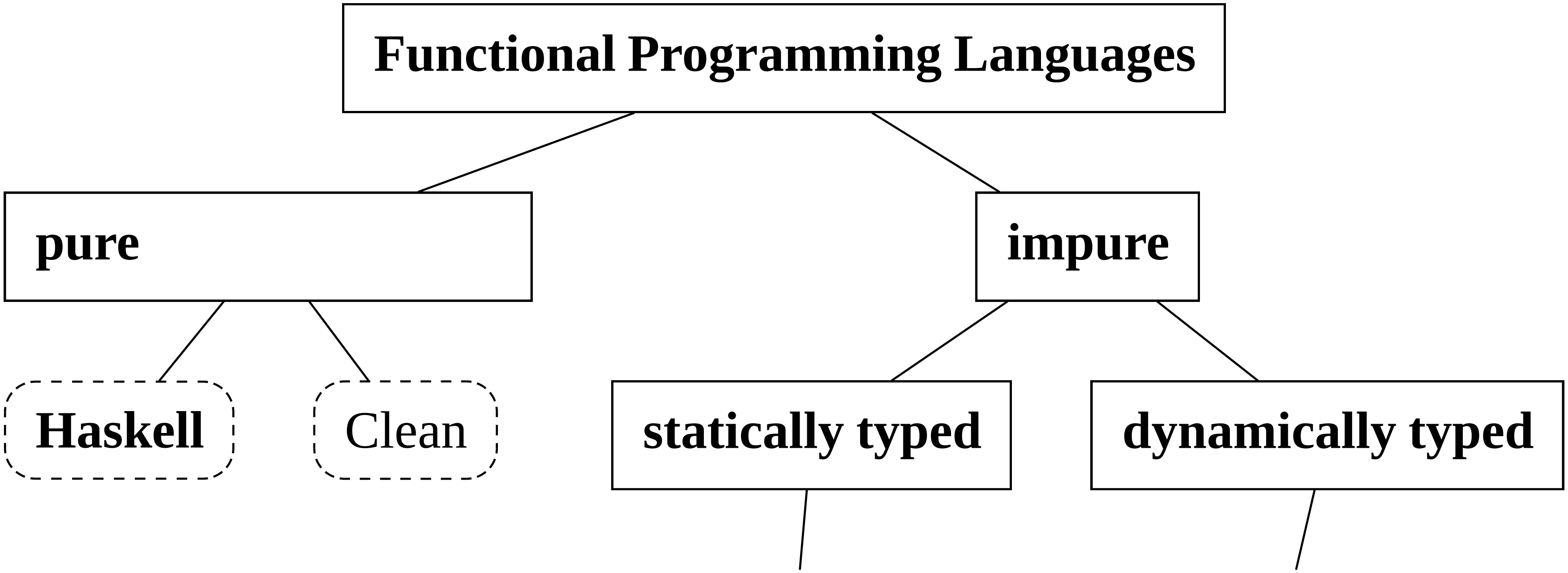
Important Functional Programming Languages



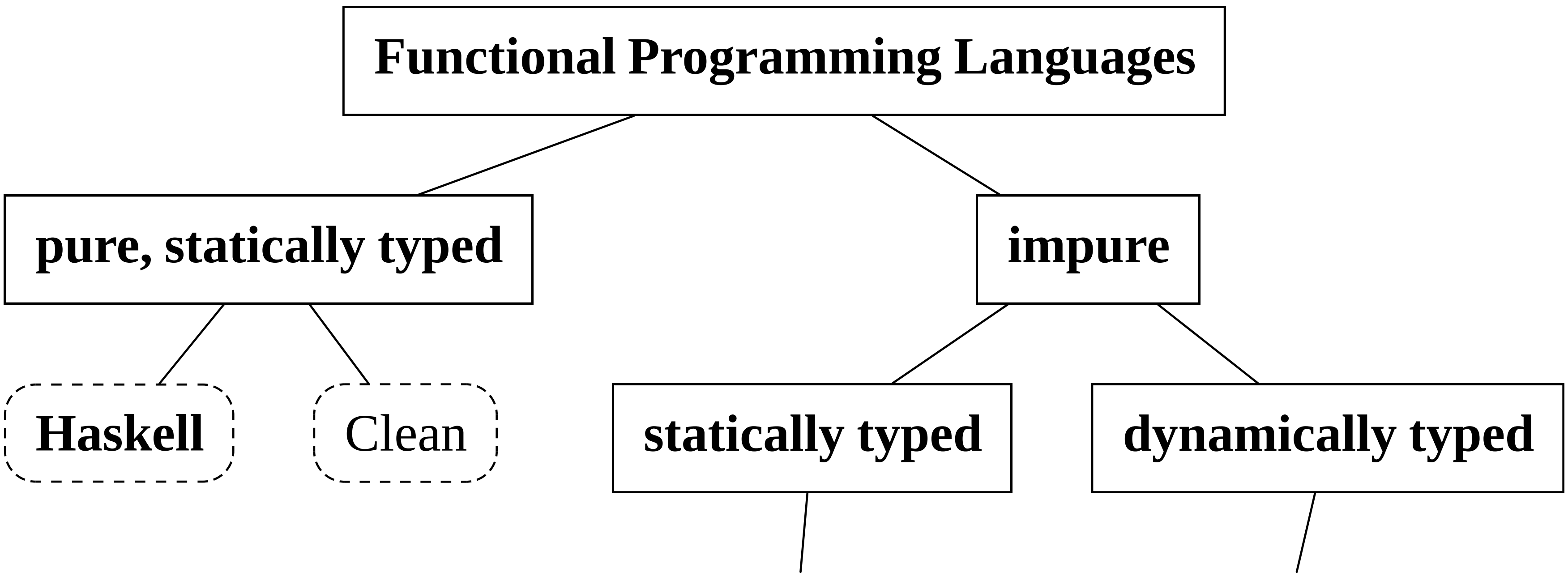
Important Functional Programming Languages



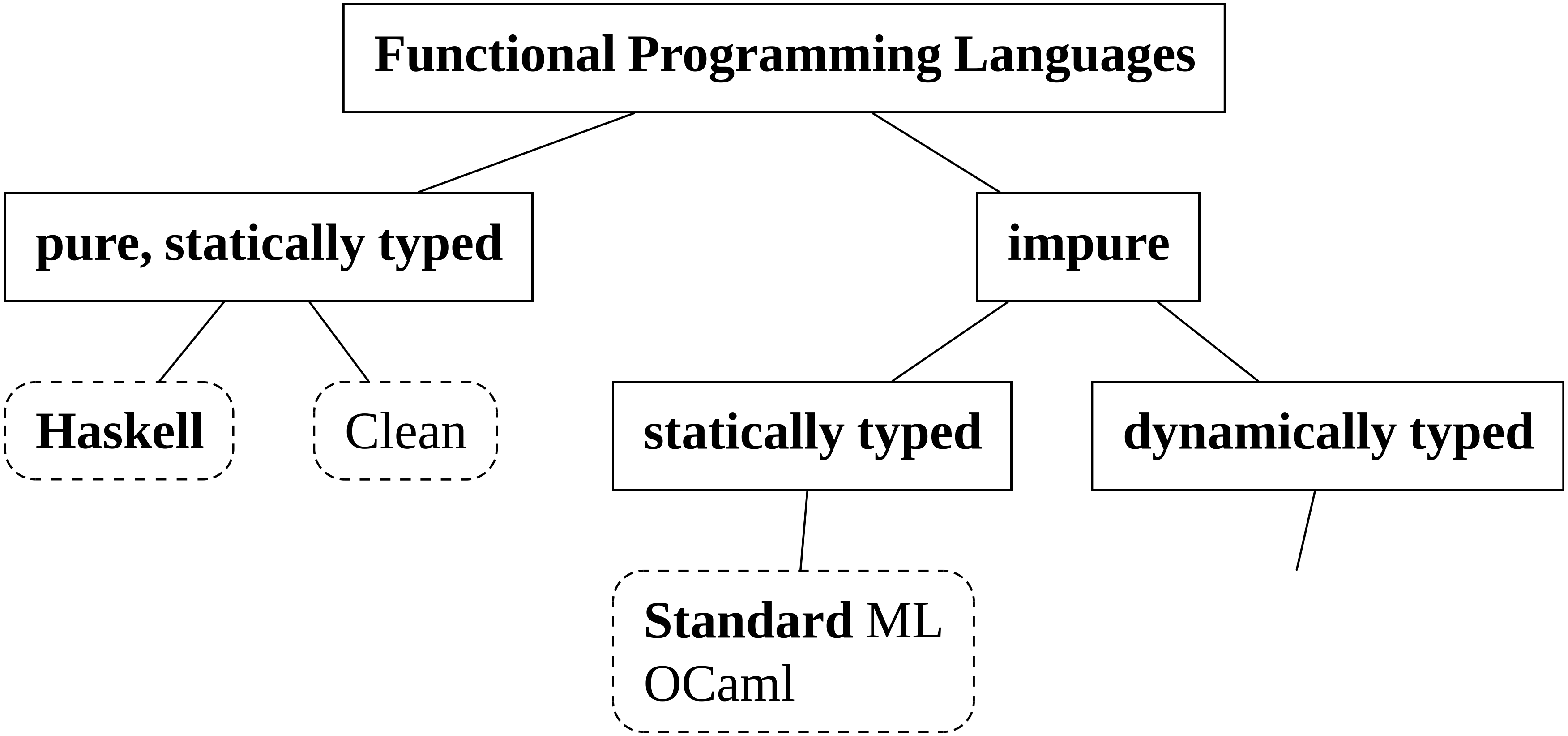
Important Functional Programming Languages



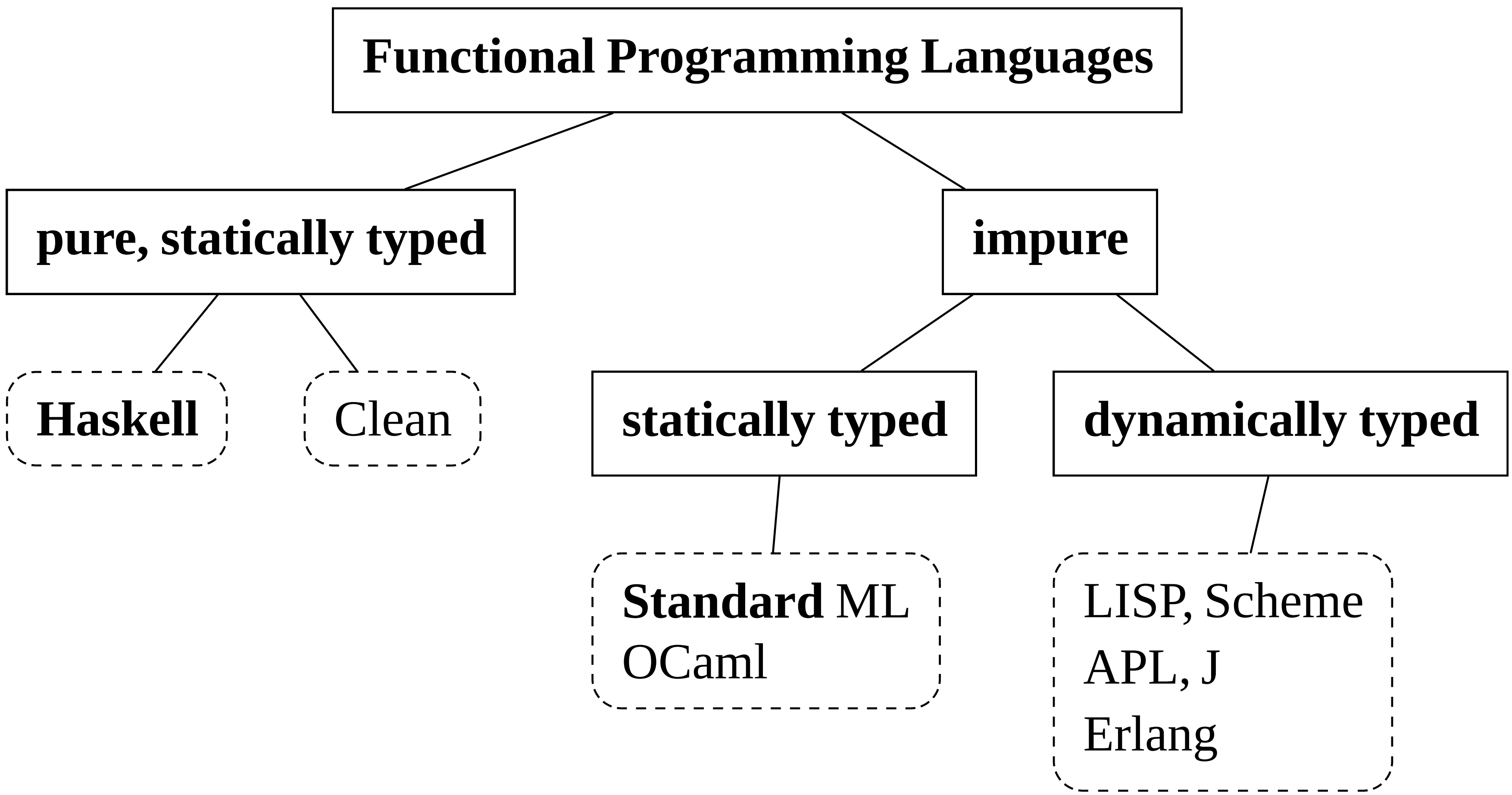
Important Functional Programming Languages



Important Functional Programming Languages



Important Functional Programming Languages



Haskell

Haskell

- **functional**

Haskell

- **functional** — programs are function definitions

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent)

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy)

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy) — arguments are evaluated only when needed

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed**

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading

Haskell

- **functional** — programs are function definitions; functions are **"first-class citizens"**
- **pure** (referentially transparent) — **"no side-effects"**
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading

- Standardised language version: **Haskell 98**

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading

- Standardised language version: **Haskell 98**
- Several compilers and interpreters available

Haskell

- **functional** — programs are function definitions; functions are **“first-class citizens”**
- **pure** (referentially transparent) — **“no side-effects”**
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading

- Standardised language version: **Haskell 98**
- Several compilers and interpreters available
- Comprehensive web site: http://haskell.org/

Important Points

Important Points

- Execution of Haskell programs

Important Points

- Execution of Haskell programs **is** **expression evaluation**

Important Points

- Execution of Haskell programs **<u>is</u> expression evaluation**

  — *(for the time being)*

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics**

Important Points

- Execution of Haskell programs **<u>is</u> expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics** than like defining procedures in C or classes and methods in Java

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics** than like defining procedures in C or classes and methods in Java

- One Haskell function may be defined by several “equations”

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics** than like defining procedures in C or classes and methods in Java

- One Haskell function may be defined by several "equations" — **the first that matches is used**

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics** than like defining procedures in C or classes and methods in Java

- One Haskell function may be defined by several "equations" — **the first that matches is used**

- **Lists** are an easy-to-use datastructure with lots of language and library support

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics** than like defining procedures in C or classes and methods in Java

- One Haskell function may be defined by several "equations" — **the first that matches is used**

- **Lists** are an easy-to-use datastructure with lots of language and library support — therefore, lists are heavily used in *beginners' material*.

Important Points

- Execution of Haskell programs <u>**is**</u> **expression evaluation**

  — *(for the time being)*

- Defining functions in Haskell is more like **defining functions in mathematics** than like defining procedures in C or classes and methods in Java

- One Haskell function may be defined by several "equations" — **the first that matches is used**

- **Lists** are an easy-to-use datastructure with lots of language and library support — therefore, lists are heavily used in *beginners' material*.

  In many cases, advanced Haskell programmers will use other datastructures, for example *Set*s, or *FiniteMap*s instead of association lists.

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

```
    4*(5+6)-2
```

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

`4*(5+6)-2` [subtraction & mult. impossible]

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

```
    4*(5+6)-2
=
    4*11-2
```

[subtraction & mult. impossible]

(addition)

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

	`4*(5+6)-2`	[subtraction & mult. impossible]
=	(addition)	
	`4*11-2`	[subtraction impossible]

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

	`4*(5+6)-2`		[subtraction & mult. impossible]
=		(addition)	
	`4*11-2`		[subtraction impossible]
=		(multiplication)	
	`44-2`		

Simple Expression Evaluation

The Haskell interpreters `hugs`, `ghci`, and `hi` accept any expression at their prompt and print (after the first ENTER) the value resulting from *evaluation* of that expression.

```
Prelude> 4*(5+6)-2
42
```

Expression evaluation proceeds by applying rules to subexpressions:

	`4*(5+6)-2`	[subtraction & mult. impossible]
=	(addition)	
	`4*11-2`	[subtraction impossible]
=	(multiplication)	
	`44-2`	
=	(subtraction)	
	`42`	

Simple Expression Evaluation — Explanation

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function *f* to a number of arguments,

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function *f* to a number of arguments, the value of the argument at position *i* is always needed

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position *i* is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its *i*-th argument, then the *i*-th argument has to be evaluated whenever a result is needed from f.

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument, then the i-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ undefined = undefined$.

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument, then the i-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**:

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument, then the i-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument, then the i-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$
  - Checking a list for emptyness is **strict:**

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument, then the i-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$
  - Checking a list for emptyness is **strict:** $\mathit{null}\ \mathit{undefined} = \mathit{undefined}$

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position i is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its i-th argument, then the i-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ undefined = undefined$.
  - **Constant functions** are **non-strict**: $(const\ 5)\ undefined = 5$
  - Checking a list for emptyness is **strict:** $null\ undefined = undefined$
  - **List construction** is **non-strict**:

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position *i* is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its *i*-th argument, then the *i*-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$
  - Checking a list for emptyness is **strict:** $\mathit{null}\ \mathit{undefined} = \mathit{undefined}$
  - **List construction** is **non-strict**: $\mathit{null}\ (\mathit{undefined} : \mathit{undefined}) = \mathbf{False}$

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position *i* is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its *i*-th argument, then the *i*-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$
  - Checking a list for emptyness is **strict:** $\mathit{null}\ \mathit{undefined} = \mathit{undefined}$
  - **List construction** is **non-strict**: $\mathit{null}\ (\mathit{undefined} : \mathit{undefined}) = \mathbf{False}$
  - **Standard arithmetic operators** are **strict in both arguments**:

Simple Expression Evaluation — Explanation

- Arguments to a fuction or operation are **evaluated only when needed.**
- If for obtaining a result from an application of a function f to a number of arguments, the value of the argument at position *i* is always needed. then f is called **strict in its *i*-th argument**
- Therefore: If f is strict in its *i*-th argument, then the *i*-th argument has to be evaluated whenever a result is needed from f.
- Simpler: A one-argument function f is strict iff $f\ \mathit{undefined} = \mathit{undefined}$.
  - **Constant functions** are **non-strict**: $(\mathit{const}\ 5)\ \mathit{undefined} = 5$
  - Checking a list for emptyness is **strict:** $\mathit{null}\ \mathit{undefined} = \mathit{undefined}$
  - **List construction** is **non-strict**: $\mathit{null}\ (\mathit{undefined} : \mathit{undefined}) = \mathbf{False}$
  - **Standard arithmetic operators** are **strict in both arguments**: $0 * \mathit{undefined} = \mathit{undefined}$

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)          (answer)
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)              (answer)
 = 41 * (magic * 42 - 23)                    (subtraction)
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)            (answer)
 = 41 * (magic * 42 - 23)                  (subtraction)
 = 41 * (7 * 42 - 23)                      (magic)
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)              (answer)
 = 41 * (magic * 42 - 23)                    (subtraction)
 = 41 * (7 * 42 - 23)                        (magic)
 = 41 * (294 - 23)                           (multiplication)
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)              (answer)
 = 41 * (magic * 42 - 23)                    (subtraction)
 = 41 * (7 * 42 - 23)                        (magic)
 = 41 * (294 - 23)                           (multiplication)
 = 41 * 271                                  (subtraction)
```

Unfolding Definitions

Assume the following definitions to be in scope:

```
answer = 42
magic = 7
```

Expression evaluation will **unfold** (or **expand**) definitions:

```
Prelude> (answer - 1) * (magic * answer - 23)
11111
```

```
   (answer - 1) * (magic * answer - 23)
 = (42 - 1) * (magic * 42 - 23)              (answer)
 = 41 * (magic * 42 - 23)                    (subtraction)
 = 41 * (7 * 42 - 23)                        (magic)
 = 41 * (294 - 23)                           (multiplication)
 = 41 * 271                                  (subtraction)
 = 11111                                     (multiplication)
```

How did I find those numbers?

Easy!

```
Prelude> [ n | n <- [1 .. 400] , 11111 ‘mod‘ n == 0 ]
[1,41,271]
```

This is a **list comprehension**:

- return all `n`
- where `n` is taken from then list `[1 .. 400]`
- and a result is returned only if `n` divides `11111`.

Conditional Expressions

Prelude> **if** 11111 ‘*mod*‘ 41 == 0 **then** 11111 ‘*div*‘ 41 **else** 5
271

The pattern is:

if *condition* **then** *expression1* **else** *expression2*

– If the condition evaluates to **True**, the conditional expression evaluates to the value of *expression1*.

– If the condition evaluates to **False**, the conditional expression evaluates to the value of *expression2*.

Conditional Expressions

Prelude> **if** 11111 ‘*mod*‘ 41 == 0 **then** 11111 ‘*div*‘ 41 **else** 5
271

The pattern is:

if *condition* **then** *expression1* **else** *expression2*

– If the condition evaluates to **True**, the conditional expression evaluates to the value of *expression1*.

– If the condition evaluates to **False**, the conditional expression evaluates to the value of *expression2*.

Therefore: **“if _ then _ else”** is **strict in the condition**.

Conditional Expressions

Prelude> **if** 11111 ‘*mod*‘ 41 == 0 **then** 11111 ‘*div*‘ 41 **else** 5
271

The pattern is:

if *condition* **then** *expression1* **else** *expression2*

– If the condition evaluates to **True**, the conditional expression evaluates to the value of *expression1*.

– If the condition evaluates to **False**, the conditional expression evaluates to the value of *expression2*.

Therefore: **“if _ then _ else”** is **strict in the condition**.

In C: (*condition* ? *expression1* : *expression2*)

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * 1
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * 1
 = 3 * 2 * 1
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * 1
 = 3 * 2 * 1
 = 3 * 2
```

Expanding Function Definitions

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact (n-1)
```

```
   fact 3
 = if 3 == 0 then 1 else 3 * fact (3-1)
 = if False then 1 else 3 * fact (3-1)
 = 3 * fact (3-1)
 = 3 * if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)
 = 3 * if 2 == 0 then 1 else 2 * fact (2-1)
 = 3 * if False then 1 else 2 * fact (2-1)
 = 3 * 2 * fact (2-1)
 = 3 * 2 * if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)
 = 3 * 2 * if 1 == 0 then 1 else 1 * fact (1-1)
 = 3 * 2 * if False then 1 else 1 * fact (1-1)
 = 3 * 2 * 1 * fact (1-1)
 = 3 * 2 * 1 * if (1-1) == 0 then 1 else (1-1) * fact ((1-1)-1)
 = 3 * 2 * 1 * if 0 == 0 then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * if True then 1 else 0 * fact (0-1)
 = 3 * 2 * 1 * 1
 = 3 * 2 * 1
 = 3 * 2
 = 6
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
fact 3
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
   fact 3
 = 3 * fact (3-1)                (fact n)
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
   fact 3
 = 3 * fact (3-1)
 = 3 * fact 2
```

`fact 3`	
`= 3 * fact (3-1)`	(`fact n`)
`= 3 * fact 2`	(determining which `fact` rule matches)

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
   fact 3
 = 3 * fact (3-1)              (fact n)
 = 3 * fact 2                  (determining which fact rule matches)
 = 3 * (2 * fact (2-1))        (fact n)
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

`  fact 3`	
`= 3 * fact (3-1)`	(`fact n`)
`= 3 * fact 2`	(determining which `fact` rule matches)
`= 3 * (2 * fact (2-1))`	(`fact n`)
`= 3 * (2 * fact 1)`	(determining which `fact` rule matches)

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

`fact 3`	
`= 3 * fact (3-1)`	(`fact n`)
`= 3 * fact 2`	(determining which `fact` rule matches)
`= 3 * (2 * fact (2-1))`	(`fact n`)
`= 3 * (2 * fact 1)`	(determining which `fact` rule matches)
`= 3 * (2 * (1 * fact (1-1)))`	(`fact n`)

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

`  fact 3`	
`= 3 * fact (3-1)`	(`fact n`)
`= 3 * fact 2`	(determining which `fact` rule matches)
`= 3 * (2 * fact (2-1))`	(`fact n`)
`= 3 * (2 * fact 1)`	(determining which `fact` rule matches)
`= 3 * (2 * (1 * fact (1-1)))`	(`fact n`)
`= 3 * (2 * (1 * fact 0))`	(determining which `fact` rule matches)

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

`  fact 3`	
`= 3 * fact (3-1)`	(`fact n`)
`= 3 * fact 2`	(determining which `fact` rule matches)
`= 3 * (2 * fact (2-1))`	(`fact n`)
`= 3 * (2 * fact 1)`	(determining which `fact` rule matches)
`= 3 * (2 * (1 * fact (1-1)))`	(`fact n`)
`= 3 * (2 * (1 * fact 0))`	(determining which `fact` rule matches)
`= 3 * (2 * (1 * 1))`	(`fact 0`)

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

`fact 3`	
`= 3 * fact (3-1)`	(`fact n`)
`= 3 * fact 2`	(determining which `fact` rule matches)
`= 3 * (2 * fact (2-1))`	(`fact n`)
`= 3 * (2 * fact 1)`	(determining which `fact` rule matches)
`= 3 * (2 * (1 * fact (1-1)))`	(`fact n`)
`= 3 * (2 * (1 * fact 0))`	(determining which `fact` rule matches)
`= 3 * (2 * (1 * 1))`	(`fact 0`)
`= 3 * (2 * 1)`	(multiplication)

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
   fact 3
 = 3 * fact (3-1)                    (fact n)
 = 3 * fact 2                        (determining which fact rule matches)
 = 3 * (2 * fact (2-1))              (fact n)
 = 3 * (2 * fact 1)                  (determining which fact rule matches)
 = 3 * (2 * (1 * fact (1-1)))        (fact n)
 = 3 * (2 * (1 * fact 0))            (determining which fact rule matches)
 = 3 * (2 * (1 * 1))                 (fact 0)
 = 3 * (2 * 1)                       (multiplication)
 = 3 * 2                             (multiplication)
```

Matching Function Definitions

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
  fact 3
= 3 * fact (3-1)                   (fact n)
= 3 * fact 2                       (determining which fact rule matches)
= 3 * (2 * fact (2-1))             (fact n)
= 3 * (2 * fact 1)                 (determining which fact rule matches)
= 3 * (2 * (1 * fact (1-1)))       (fact n)
= 3 * (2 * (1 * fact 0))           (determining which fact rule matches)
= 3 * (2 * (1 * 1))                (fact 0)
= 3 * (2 * 1)                      (multiplication)
= 3 * 2                            (multiplication)
= 6                                (multiplication)
```

Lists

- **List display:** between square brackets explicitly listing all elements, separated by commas:

```
[1,4,9,16,25]
```

Lists

- **List display:** between square brackets explicitly listing all elements, separated by commas:

```
[1,4,9,16,25]
```

- **Enumeration lists:** denoted by ellipsis "`..`" inside square brackets; defined by beginning (and end, if applicable):

```
[1 .. 10]    = [1,2,3,4,5,6,7,8,9,10]
[1,3 .. 10] = [1,3,5,7,9]
[1,3 .. 11] = [1,3,5,7,9,11]
[11,9 .. 1] = [11,9,7,5,3,1]
[11 .. 1]    = []
[1 .. ]      = [1,2,3,4,5,6,7,8,9,10, …]  -- infinite list
[1,3 .. ]    = [1,3,5,7,9,11, …]           -- infinite list
```

List Construction

List Construction

Display and enumeration lists are *syntactic sugar*

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs`

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

- either the **empty list**: `[]`,

- or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

“:” is used as *infix list constructor*:

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

`3 : []`

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

“:” is used as *infix list constructor*:

`3 : []` = `[3]`

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

```
3 : []        =        [3]
2 : [3]
```

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

```
3 : []     =     [3]
2 : [3]    =  [2, 3]
```

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

“:” is used as *infix list constructor*:

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

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

“:” is used as *infix list constructor*:

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

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

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

As an infix operator, ":" *associates to the right*

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

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

As an infix operator, ":" *associates to the right*:

$$x : y : ys \;=\; x : (y : ys)$$

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

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

As an infix operator, ":" *associates to the right*:

$$x : y : ys = x : (y : ys)$$

Example:

1 : 2 : [3,4]

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: “`x`es”)

`x : xs` — read: “`x` *cons* `x`es”.

“:” is used as *infix list constructor*:

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

As an infix operator, “:” *associates to the right*:

$$x : y : ys \;=\; x : (y : ys)$$

Example:

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

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

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

As an infix operator, ":" *associates to the right*:

$$x : y : ys = x : (y : ys)$$

Example:

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

List Construction

Display and enumeration lists are *syntactic sugar*: A list is

– either the **empty list**: `[]`,

– or **non-empty**, and **cons**tructed from a **head** `x` and a **tail** `xs` (read: "`x`es")

`x : xs` — read: "`x` *cons* `x`es".

":" is used as *infix list constructor*:

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

As an infix operator, ":" *associates to the right*:

$$x : y : ys \;=\; x : (y : ys)$$

Example:

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

Cons is Not Associative

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4])`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain circtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]`   is **nonsense**, since `2` is not a list!

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4] = [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!
- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]]`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!
- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]`

Cons is Not Associative

The convention that ":" ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, ":" is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!

- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]`
- Another *list of lists of integers*:
  `(1 : [2]) : [[3,4,5], [6,7]]`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!
- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]`
- Another *list of lists of integers*:
  `(1 : [2]) : [[3,4,5], [6,7]] = [[1,2],[3,4,5],[6,7]]`

Cons is Not Associative

The convention that “:” ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, “:” is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!
- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]`
- Another *list of lists of integers*:
  `(1 : [2]) : [[3,4,5], [6,7]] = [[1,2],[3,4,5],[6,7]]`
- `1 : ([2] : [[3,4,5], [6,7]])`

Cons is Not Associative

The convention that ":" ***associates to the right*** allows to save parentheses in certain cirtcumstances.

However, ":" is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`
- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!

- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]`
- Another *list of lists of integers*:
  `(1 : [2]) : [[3,4,5], [6,7]] = [[1,2],[3,4,5],[6,7]]`
- `1 : ([2] : [[3,4,5], [6,7]])` is **nonsense** again!

Cons is Not Associative

The convention that ":" ***associates to the right*** allows to save parentheses in certain circtumstances.

However, ":" is **not** associative:

- A *list of integers*:
  `1 : (2 : [3,4]) = 1 : 2 : [3,4]  =  [1, 2, 3, 4]`

- `(1 : 2) : [3,4]` is **nonsense**, since `2` is not a list!

- A *list of lists of integers*:
  `[2] : [[3,4,5], [6,7]] = [[2],[3,4,5],[6,7]]`

- Another *list of lists of integers*:
  `(1 : [2]) : [[3,4,5], [6,7]] = [[1,2],[3,4,5],[6,7]]`

- `1 : ([2] : [[3,4,5], [6,7]])` is **nonsense** again!
  Reason: `1` and `[2]` cannot be members of the same list (*type error*).

List Comprehensions

General shape:

$$[\ \mathit{term} \mid \mathit{generator}\ \ \{\,,\ \ \mathit{generator\_or\_constraint}\ \}^{*}\ ]$$

List Comprehensions

General shape:

$$[\ \textit{term} \mid \textit{generator} \ \ \{ , \textit{generator\_or\_constraint} \}^{*} \ ]$$

Examples:

$[\ n * n \mid n \leftarrow [1 \,..\, 5]\ ]$

List Comprehensions

General shape:

$$[\ \textit{term}\ \mathbf{|}\ \textit{generator}\ \ \{\ \mathbf{,}\ \textit{generator\_or\_constraint}\ \}^*\ ]$$

Examples:

$$[\ n*n\ |\ n \leftarrow [1\,..\,5]\ ] = [1,4,9,16,25]$$

List Comprehensions

General shape:

$$[\ \textit{term} \mid \textit{generator} \quad \{\, ,\ \textit{generator\_or\_constraint}\ \}^*\ ]$$

Examples:

$$[\ n*n \mid n \leftarrow [1\,..\,5]\ ] = [1,4,9,16,25]$$

$$[\ n*n \mid n \leftarrow [1\,..\,10],\ \textit{even}\ n\ ]$$

List Comprehensions

General shape:

$$[\ \mathit{term} \mid \mathit{generator}\ \ \{\, ,\ \mathit{generator\_or\_constraint}\ \}^*\ ]$$

Examples:

$[\ n*n \mid n \leftarrow [1\,..\,5]\ ] = [1,4,9,16,25]$

$[\ n*n \mid n \leftarrow [1\,..\,10],\ \mathit{even}\ n\ ] = [4,16,36,64,100]$

List Comprehensions

General shape:

$$[\ \textit{term} \mid \textit{generator}\ \ \{\, ,\ \textit{generator\_or\_constraint}\ \}^{*}\ ]$$

Examples:

$[\ n*n \mid n \leftarrow [1 \mathinner{..} 5]\ ] = [1,4,9,16,25]$

$[\ n*n \mid n \leftarrow [1 \mathinner{..} 10],\ \textit{even}\ n\ ] = [4,16,36,64,100]$

$[\ m*n \mid m \leftarrow [1,3,5],\ n \leftarrow [2,4,6]\ ]$

List Comprehensions

General shape:

$$[\ \textit{term}\ |\ \textit{generator}\quad \{\ ,\ \textit{generator\_or\_constraint}\ \}^*\ ]$$

Examples:

$[\ n * n \mid n \leftarrow [1 \,..\, 5]\ ] = [1,4,9,16,25]$

$[\ n * n \mid n \leftarrow [1 \,..\, 10],\ \textit{even}\ n\ ] = [4,16,36,64,100]$

$[\ m * n \mid m \leftarrow [1,3,5],\ n \leftarrow [2,4,6]\ ] = [2,4,6,6,12,18,10,20,30]$

List Comprehensions

General shape:

$$[\ term\ |\ generator\ \ \{,\ generator\_or\_constraint\ \}^{*}\ ]$$

Examples:

$[\ n * n \mid n \leftarrow [1 \,..\, 5]\ ] = [1,4,9,16,25]$

$[\ n * n \mid n \leftarrow [1 \,..\, 10],\ even\ n\ ] = [4,16,36,64,100]$

$[\ m * n \mid m \leftarrow [1,3,5],\ n \leftarrow [2,4,6]\ ] = [2,4,6,6,12,18,10,20,30]$

Note:

– The left generator “generates slower”.

List Comprehensions

General shape:

$$[\ \mathit{term}\ |\ \mathit{generator}\ \ \{\, ,\ \mathit{generator\_or\_constraint}\ \}^{*}\ ]$$

Examples:

$[\ n * n \mid n \leftarrow [1 \,..\, 5]\ ] = [1,4,9,16,25]$

$[\ n * n \mid n \leftarrow [1 \,..\, 10],\ \mathit{even}\ n\ ] = [4,16,36,64,100]$

$[\ m * n \mid m \leftarrow [1,3,5],\ n \leftarrow [2,4,6]\ ] = [2,4,6,6,12,18,10,20,30]$

Note:

– The left generator "generates slower".

– Haskell code fragments will frequently be presented like above in a form that is more readable than plain typewriter text — in that case, the "comes from" arrow "`<-`" in generators turns into "$\leftarrow$"

The Type Language

Haskell has a full-fledged **type language**, with

- Simple predefined datatypes: `Bool`, `Char`, `Integer`, …
- Predefined **type constructors**: lists, tuples, functions, …
- Type synonyms
- User-defined datatypes and type constructors
- Type variables — to express **parametric polymorphism**
- …

Simple Predefined Datatypes

`Bool`	truth values	False, True
`Char`	"Unicode" characters	(in GHC: ISO-10646)
`Integer`	integers	**arbitrary precision**
`Int`	"machine integers"	$\geq$ 32 bits
`Float`	real floating point	single precision
`Double`	real floating point	double precision
`Complex Float`	complex floating point	single precision
`Complex Double`	complex floating point	double precision

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: ???`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: ???`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: ???`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] ::` **???**

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`

List Types

If t is a type, then the **list type** $[t]$ is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`
- `"hello" :: ???`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`
- `"hello" :: [Char]`

List Types

If t is a type, then the **list type** $[t]$ is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`
- `"hello" :: [Char]`
- `[ "hello", "world" ] :: ???`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`
- `"hello" :: [Char]`
- `[ "hello", "world" ] :: [[Char]]`

List Types

If t is a type, then the **list type** `[`t`]` is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`
- `"hello" :: [Char]`
- `[ "hello", "world" ] :: [[Char]]`
- `[["first", "line"], ["second", "line"]] :: ???`

List Types

If t is a type, then the **list type** $[t]$ is the type of **lists** with elements of type t.

```
answer :: Integer
answer = 42
```

```
limit :: Int
limit = 100
```

Then:

- `[ 1, 2, 3, answer] :: [Integer]`
- `[ 1 .. limit ] :: [Int]`
- `[ [ 1 .. limit ] , [ 2 .. limit ] ] :: [[Int]]`
- `[ ’h’, ’e’, ’l’, ’l’, ’o’ ] :: [Char]`
- `"hello" :: [Char]`
- `[ "hello", "world" ] :: [[Char]]`
- `[["first", "line"], ["second", "line"]] :: [[[Char]]]`

Product Types (Pairs)

If t and u are types, then the **product type** `(t,u)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** (t, u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** (t, u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `("answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", 'X') :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** `(`t`,`u`)` is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `("answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", ’X’) :: ([Char], Char)`

Product Types (Pairs)

If t and u are types, then the **product type** (t,u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", 'X') :: ([Char], Char)`
- `(limit, ("???", 'X')) :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** (t, u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", 'X') :: ([Char], Char)`
- `(limit, ("???", 'X')) :: (Int, ([Char], Char))`

Product Types (Pairs)

If t and u are types, then the **product type** (t,u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", 'X') :: ([Char], Char)`
- `(limit, ("???", 'X')) :: (Int, ([Char], Char))`
- `(True, [("X",limit),("Y",5)]) :: ???`

Product Types (Pairs)

If t and u are types, then the **product type** (t, u) is the type of **pairs** with first component of type t and second component of type u (mathematically: $t \times u$).

Examples:

- `(answer, limit) :: (Integer, Int)`
- `(limit, answer) :: (Int, Integer)`
- `("???", answer) :: ([Char], Integer)`
- `("???", (limit, answer)) :: ([Char], (Int, Integer))`
- `("???", 'X') :: ([Char], Char)`
- `(limit, ("???", 'X')) :: (Int, ([Char], Char))`
- `(True, [("X",limit),("Y",5)]) :: (Bool, [([Char], Int)])`

Tuple Types

If $n \neq 1$ is a natural number and $t_1, \ldots, t_n$ are types, then the **tuple type** `(` t_1 `,` $\ldots$ `,` t_n `)` is the type of n-**tuples** with the ith component of type t_i.

Examples:

- `(answer, ’c’, limit) :: (Integer, Char, Int)`
- `(answer, ’c’, limit, "all") :: (Integer, Char, Int, [Char])`
- `() :: ()`

  — there is exactly one **zero-tuple**.

  The type `()` of zero-tuples is also called the **unit type**.

Simple Type Synonyms

If t is a type not containing any type variables, and *Name* is an identifier with a capital first letter, then

`type` *Name* `=` t

defines *Name* as a **type synonym** for t, i.e., *Name* can now be used interchangeably with t.

Examples:

```
type String = [Char]                        -- predefined

type Point = (Double, Double)               -- (1.5, 2.7)

type Triangle = (Point, Point, Point)

type CharEntity = (Char, String)            -- ('Ã¼', "&uuml;")

type Dictionary = [(String,String)] -- [("day","jour")]
```

Type Variables and Polymorphic Types

- Identifiers with lower-case first letter can be used as type variables.
- Type variables can be used like other types in the construction of types, e.g.:

```
[(a,b)]
(Bool, (a, Int))
[ ( String, [(key, val)] ) ]
```

- A type containing at least one type variable is called **polymorphic**
- Polymorphic types can be instantiated by instantiating type variables with types, e.g.:

```
[(a,b)]    ⇒    [(Char,b)]
[(a,b)]    ⇒    [(Char,Int)]
[(a,b)]    ⇒    [(a,[(String,Int)])]
[(a,b)]    ⇒    [(a,[(String,c)])]
```

Typing of List Construction

- The empty list can be used at any list type: `[] :: [a]`
- If an element `x :: a` and a list `xs :: [a]` are given, then

```
(x : xs) :: [a]
```

Examples:

```
2                                   :: Int
[]                                  :: [Int]
[2] = 2 : []                        :: [Int]
[[3,4,5], [6,7]]                    :: [[Int]]
[2] : [[3,4,5], [6,7]]              :: [[Int]]
1 : ([2] : [[3,4,5], [6,7]])
```
– – ***cannot be typed!***

Function Types and Function Application

If t and u are types, then the **function type** t`->`u is the type of all **functions** accepting arguments of type t and producing results of type u (mathematically: $t \rightarrow u$).

Then:

- If a function `f :: a -> b` and an argument `x :: a` are given, then we have `(f x) :: b`.
- If a function `f :: a -> b` is given and we know that `(f x) :: b`, then the argument `x` is used at type `a`.
- If an argument `x :: a` is given and we know that `(f x) :: b`, then the function `f` is used at type `a -> b`.

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                :: Char
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                :: Char

["hello", fst (x, 17)]
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x
```

```
fst ('c', False)                :: Char
```

```
["hello", fst (x, 17)]     ⇒     x :: String
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                :: Char

["hello", fst (x, 17)]      ⇒     x :: String

f p = limit + fst p
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                  :: Char

["hello", fst (x, 17)]       ⇒    x :: String

f p = limit + fst p          ⇒    p :: (Int,a)
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                          :: Char

["hello", fst (x, 17)]          ⇒          x :: String

f p = limit + fst p             ⇒          p :: (Int,a)
                                           f :: (Int,a) -> Int
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x


fst ('c', False)                    :: Char


["hello", fst (x, 17)]      ⇒     x :: String


f p = limit + fst p         ⇒     p :: (Int,a)
                                  f :: (Int,a) -> Int


g h = fst (h "") : [limit]
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                    :: Char

["hello", fst (x, 17)]       ⇒     x :: String

f p = limit + fst p          ⇒     p :: (Int,a)
                                   f :: (Int,a) -> Int

g h = fst (h "") : [limit]
    ⇒      h ::  String -> (Int,a)
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
= fst (length "", ’ ’ : "") : [limit]
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
= fst (length "", ’ ’ : "") : [limit]
= length "" : [limit]
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
= fst (length "", ’ ’ : "") : [limit]
= length "" : [limit]
= 0 : [limit]
```

Let’s Play the Evaluation Game Again — 1

```
h1 :: String -> (Int, String)
h1 str = (length str, ’ ’ : str)

g h = fst (h "") : [limit]
```

Then:

```
g h1
= fst (h1 "") : [limit]
= fst (length "", ’ ’ : "") : [limit]
= length "" : [limit]
= 0 : [limit]
= [0, 100]
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
= sum (map ord (notOccCaps "")) : [limit]
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
= sum (map ord (notOccCaps "")) : [limit]
= …
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
= sum (map ord (notOccCaps "")) : [limit]
= …
= 2015 : [limit]
```

Let’s Play the Evaluation Game Again — 2

```
h2 :: String -> (Int, Char)
h2 str = (sum (map ord (notOccCaps str)), head str)

notOccCaps :: String -> String
notOccCaps str = filter (‘notElem‘ str) [’A’ .. ’Z’]

g h = fst (h "") : [limit]
```

Then:

```
g h2
= fst (h2 "") : [limit]
= fst (sum (map ord (notOccCaps "")), head "") : [limit]
= sum (map ord (notOccCaps "")) : [limit]
= …
= 2015 : [limit]
= [2015, 100]
```

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: **Functions are first-class citizens**

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: Functions are first-class citizens

- Functions can be **arguments of other functions**: `g h2`

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: Functions are first-class citizens

- Functions can be **arguments of other functions**: `g h2`
- Functions can be **components of data structures**: `(7,h1)`, `[h1, h2]`

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: Functions are first-class citizens

- Functions can be **arguments of other functions**: `g h2`
- Functions can be **components of data structures**: `(7,h1)`, `[h1, h2]`
- Functions can be **results of function application**: `succ . succ`

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: Functions are first-class citizens

- Functions can be **arguments of other functions**: `g h2`
- Functions can be **components of data structures**: `(7,h1)`, `[h1, h2]`
- Functions can be **results of function application**: `succ . succ`

A **first-order function** accepts only non-functional values as arguments.

A **higher-order function** expects functions as arguments.

Higher-Order Functions

```
g h = fst (h "") : [limit]
```

Functional Programming: **Functions are first-class citizens**

- Functions can be **arguments of other functions**: `g h2`
- Functions can be **components of data structures**: `(7,h1)`, `[h1, h2]`
- Functions can be **results of function application**: `succ . succ`

A **first-order function** accepts only non-functional values as arguments.

A **higher-order function** expects functions as arguments.

`g` is a second-order function: it expects first-order functions like `h1`, `h2` as arguments.

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst ('c', False)                    :: Char

["hello", fst (x, 17)]      ⇒    x :: String

f p = limit + fst p         ⇒    p :: (Int,a)
                                 f :: (Int,a) -> Int

g h = fst (h "") : [limit]
     ⇒      h ::  String -> (Int,a)
```

Type Inference Examples

```
fst :: (a,b) -> a
fst (x,y) = x

fst (’c’, False)                      :: Char

["hello", fst (x, 17)]      ⇒     x :: String

f p = limit + fst p         ⇒     p :: (Int,a)
                                  f :: (Int,a) -> Int

g h = fst (h "") : [limit]
    ⇒      h ::  String -> (Int,a)
           g :: (String -> (Int,a)) -> [Int]
```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., "they accept their arguments one at a time":

```
cylVol r h = (pi :: Double) * r * r * h
```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., "they accept their arguments one at a time":

```
cylVol r h = (pi :: Double) * r * r * h
```

  Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

```
(cylVol r) :: ???
```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., "they accept their arguments one at a time":

  ```
  cylVol r h = (pi :: Double) * r * r * h
  ```

  Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

  ```
  (cylVol r) :: Double -> Double
  ```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., "they accept their arguments one at a time":

```
cylVol r h = (pi :: Double) * r * r * h
```

Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

```
(cylVol r) :: Double -> Double
cylVol     :: ???
```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., "they accept their arguments one at a time":

```
cylVol r h = (pi :: Double) * r * r * h
```

Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

```
(cylVol r) :: Double -> Double
cylVol     :: Double -> (Double -> Double)
```

Curried Functions

- **Function application associates to the left**, i.e.,

```
f x y  =  (f x) y
```

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., “they accept their arguments one at a time”:

```
cylVol r h = (pi :: Double) * r * r * h
```

  Since the right-hand side, `r`, and `h` obviously all have type `Double`, we have;

```
(cylVol r) :: Double -> Double
cylVol     :: Double -> (Double -> Double)
```

- **Function type construction associates to the right**, i.e.,

```
a -> b -> c  =  a -> (b -> c)
```

“Partial Application”

Let values with the following types be given:

$f \ :: \ a \rightarrow b \rightarrow c$

$x \ :: \ a$

$y \ :: \ b$

“Partial Application”

Let values with the following types be given:

$f :: a \to b \to c$

$x :: a$

$y :: b$

The type of f is the function type $a \to (b \to c)$

“Partial Application”

Let values with the following types be given:

$f :: a \to b \to c$
$x :: a$
$y :: b$

The type of f is the function type $a \to (b \to c)$, with

- argument type a,
- result type $b \to c$.

“Partial Application”

Let values with the following types be given:

$f :: a \to b \to c$
$x :: a$
$y :: b$

The type of f is the function type $a \to (b \to c)$, with

- argument type a,
- result type $b \to c$.

Therefore, we can apply f to x and obtain:

“Partial Application”

Let values with the following types be given:

$f :: a \to b \to c$
$x :: a$
$y :: b$

The type of f is the function type $a \to (b \to c)$, with

- argument type a,
- result type $b \to c$.

Therefore, we can apply f to x and obtain:

$(f\ x) :: b \to c$

“Partial Application”

Let values with the following types be given:

$f :: a \to b \to c$
$x :: a$
$y :: b$

The type of f is the function type $a \to (b \to c)$, with

- argument type a,
- result type $b \to c$.

Therefore, we can apply f to x and obtain:

$(f\ x) :: b \to c$

The application of a “two-argument function” to a single argument is a “one-argument function”

"Partial Application"

Let values with the following types be given:

$f :: a \to b \to c$

$x :: a$

$y :: b$

The type of f is the function type $a \to (b \to c)$, with

- argument type a,
- result type $b \to c$.

Therefore, we can apply f to x and obtain:

$(f\ x) :: b \to c$

The application of a "two-argument function" to a single argument is a "one-argument function", which can then be applied to a second argument:

$(f\ x)\ y :: c \qquad = \quad f\ x\ y$

Partial Application — Example

$$g :: (String \rightarrow (Int, a)) \rightarrow [Int]$$

$$g\ h = fst\ (h\ "")\ :\ [limit]$$

Partial Application — Example

$$g \;::\; (\mathit{String} \rightarrow (\mathit{Int}\,,\; a)) \rightarrow [\mathit{Int}\,]$$
$$g\;\; h \;=\; \mathit{fst}\;\; (h\;\text{""}) \;:\; [\mathit{limit}\,]$$

$$k\;\; n\;\; \mathit{str} \;=\; (n \;*\; (\mathit{length}\;\; \mathit{str} \;+\; 1)\,,\; \mathit{unwords}\;\; (\mathit{replicate}\;\; n\;\; \mathit{str}\,))$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "")\ :\ [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

Partial Application — Example

$$g :: (String \rightarrow (Int, a)) \rightarrow [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \rightarrow String \rightarrow (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$g\ (k\ 3)$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$g\ (k\ 3)$$
$$= fst\ (k\ 3\ "") : [limit]$$

Partial Application — Example

$$g :: (String \rightarrow (Int, a)) \rightarrow [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \rightarrow String \rightarrow (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$g\ (k\ 3)$$
$$= fst\ (k\ 3\ "") : [limit]$$
$$= fst\ (3 * (length\ "" + 1), unwords\ (replicate\ 3\ "")) : [limit]$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$g\ (k\ 3)$$
$$= fst\ (k\ 3\ "") : [limit]$$
$$= fst\ (3 * (length\ "" + 1), unwords\ (replicate\ 3\ "")) : [limit]$$
$$= (3 * (length\ "" + 1)) : [limit]$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "")\ :\ [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$g\ (k\ 3)$$
$$= fst\ (k\ 3\ "")\ :\ [limit]$$
$$= fst\ (3 * (length\ "" + 1), unwords\ (replicate\ 3\ ""))\ :\ [limit]$$
$$= (3 * (length\ "" + 1))\ :\ [limit]$$
$$= (3 * (0 + 1))\ :\ [limit]$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ 3\ str))$$

$$g\ (k\ 3)$$
$$= fst\ (k\ 3\ "") : [limit]$$
$$= fst\ (3 * (length\ "" + 1), unwords\ (replicate\ n\ "")) : [limit]$$
$$= (3 * (length\ "" + 1)) : [limit]$$
$$= (3 * (0 + 1)) : [limit]$$
$$= (3 * 1) : [limit]$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$g\ (k\ 3)$$
$$= fst\ (k\ 3\ "") : [limit]$$
$$= fst\ (3 * (length\ "" + 1), unwords\ (replicate\ 3\ "")) : [limit]$$
$$= (3 * (length\ "" + 1)) : [limit]$$
$$= (3 * (0 + 1)) : [limit]$$
$$= (3 * 1) : [limit]$$
$$= 3 : [limit]$$

Partial Application — Example

$$g :: (String \to (Int, a)) \to [Int]$$
$$g\ h = fst\ (h\ "") : [limit]$$

$$k :: Int \to String \to (Int, String)$$
$$k\ n\ str = (n * (length\ str + 1), unwords\ (replicate\ n\ str))$$

$$
\begin{aligned}
& g\ (k\ 3) \\
= {} & fst\ (k\ 3\ "") : [limit] \\
= {} & fst\ (3 * (length\ "" + 1), unwords\ (replicate\ 3\ "")) : [limit] \\
= {} & (3 * (length\ "" + 1)) : [limit] \\
= {} & (3 * (0 + 1)) : [limit] \\
= {} & (3 * 1) : [limit] \\
= {} & 3 : [limit] \\
= {} & [3, 100]
\end{aligned}
$$

Operations on Functions

```
id :: a -> a                                      -- identity function
id x = x

(.) :: (b -> c) -> (a -> b) -> (a -> c)           -- function composition
(f . g) x  =  f (g x)

flip :: (a -> b -> c) -> (b -> a -> c)            -- argument swapping
flip f x y  =  f y x

curry :: ((a,b) -> c) -> (a -> b -> c)            -- currying
curry g x y = g (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

Exercise (*necessary!*): Copy only the definitions to a sheet of paper, and then infer the types yourself!

Operator Sections

- Infix operators are turned into functions by surrounding them with parentheses:

```
(+) 2 3  =  2 + 3
```

- This is necessary in type declarations:

```
(+)  :: Int -> Int -> Int      -- not the "natural" type of (+)
(:)  ::  a  -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

- It is also possible to supply only one argument (which has to be an atomic expression):

```
    (2   +) 3    =    2   + 3       =  (+ 3  ) 2
    (8,3 /) 2.5  =    8.3 / 2.5     =  (/ 2.5) 8.3
    (7   :) []   =    7   : []      =  (: [] ) 7
((2^17) :) (16:[]) = (2^17) : 16 : [] = (: (16:[])) (2^17)
```

Turning Functions into Infix Operators

Surrounding a function name by **backquotes** turns it into an infix operator.

Frequently used examples (not the “natural” types throughout):

```
div, mod, max, min :: Int -> Int -> Int
elem :: Int -> [Int] -> Bool

12 ‘div‘  7             =    1
12 ‘mod‘  7             =    5
12 ‘max‘  7             =   12
12 ‘min‘  7             =    7
12 ‘elem‘ [1 .. 10]     =  False
```

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$\mathit{null} \quad :: \; [\,a\,] \rightarrow \mathit{Bool}$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$\begin{aligned} &\mathit{null} && :: [a] \to \mathit{Bool} \\ &\mathit{null}\ [\,] && = \end{aligned}$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$
\begin{array}{ll}
\mathit{null} & :: [a] \rightarrow \mathit{Bool} \\
\mathit{null}\ [\,] & = \\
\mathit{null}\ (x : xs) & =
\end{array}
$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$
\begin{aligned}
&\mathit{null} && :: [a] \rightarrow \mathit{Bool}\\
&\mathit{null}\ [\,] && = \textbf{True}\\
&\mathit{null}\ (x : xs) && =
\end{aligned}
$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$
\begin{aligned}
&null &&:: [a] \to Bool\\
&null\ [\,] &&= \mathbf{True}\\
&null\ (x : xs) &&= \mathbf{False}
\end{aligned}
$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$\begin{aligned} &\mathit{null} && :: [a] \to \mathit{Bool} \\ &\mathit{null}\ [\,] && = \textbf{True} \\ &\mathit{null}\ (x : xs) && = \textbf{False} \end{aligned}$$

$$\mathit{head} \quad :: [a] \to a$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$
\begin{aligned}
&\mathit{null} && :: [a] \to \mathit{Bool} \\
&\mathit{null}\ [\,] && = \mathbf{True} \\
&\mathit{null}\ (x : \mathit{xs}) && = \mathbf{False}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{head} && :: [a] \to a \\
&\mathit{head}\ (x : \mathit{xs}) && = x
\end{aligned}
$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$\begin{aligned}
&\mathit{null} && :: [a] \rightarrow \mathit{Bool} \\
&\mathit{null}\ [\,] && = \textbf{True} \\
&\mathit{null}\ (x : xs) && = \textbf{False}
\end{aligned}$$

$$\begin{aligned}
&\mathit{head} && :: [a] \rightarrow a \\
&\mathit{head}\ (x : xs) && = x
\end{aligned}$$

$$\mathit{tail} \quad :: [a] \rightarrow [a]$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$\begin{aligned}
&\mathit{null} && :: [a] \rightarrow \mathit{Bool} \\
&\mathit{null}\ [\,] && = \ \mathbf{True} \\
&\mathit{null}\ (x : xs) && = \ \mathbf{False}
\end{aligned}$$

$$\begin{aligned}
&\mathit{head} && :: [a] \rightarrow a \\
&\mathit{head}\ (x : xs) && = \ x
\end{aligned}$$

$$\begin{aligned}
&\mathit{tail} && :: [a] \rightarrow [a] \\
&\mathit{tail}\ (x : xs) && = \ xs
\end{aligned}$$

Defining Functions Over Lists by Pattern Matching

Some functions taking lists as arguments can be defined directly via **pattern matching**:

$$
\begin{aligned}
&\mathit{null} &&:: [a] \to \mathit{Bool}\\
&\mathit{null}\ [\,] &&= \textbf{True}\\
&\mathit{null}\ (x : xs) &&= \textbf{False}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{head} &&:: [a] \to a\\
&\mathit{head}\ (x : xs) &&= x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{tail} &&:: [a] \to [a]\\
&\mathit{tail}\ (x : xs) &&= xs
\end{aligned}
$$

(*head* and *tail* are **partial functions** — both are undefined on the empty list.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$length :: [a] \to Int$

$concat :: [[a]] \to [a]$

$product :: [Integer] \to Integer$

$(++) :: [a] \to [a] \to [a]$

$sum :: [Integer] \to Integer$

$(\text{`}elem\text{`}) :: Int \to [Int] \to Bool$

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{array}{lll} \mathit{length} & :: & [a] \to \mathit{Int} \\ \mathit{length}\ [\,] & = & \\ \mathit{length}\ (x : xs) & = & \end{array}$$

$$\begin{array}{lll} \mathit{concat} & :: & [[a]] \to [a] \\ \mathit{concat}\ [\,] & = & \\ \mathit{concat}\ (xs : xss) & = & \end{array}$$

$$\begin{array}{lll} (+\!\!+) & :: & [a] \to [a] \to [a] \\ [\,] & +\!\!+\ ys & = \\ (x : xs) & +\!\!+\ ys & = \end{array}$$

$$\begin{array}{ll} \mathit{sum}\ [\,] & = \\ \mathit{sum}\ (x : xs) & = \end{array}$$

$$\begin{array}{ll} \mathit{product}\ [\,] & = \\ \mathit{product}\ (x : xs) & = \end{array}$$

$$\begin{array}{ll} x\ \text{`}\mathit{elem}\text{`}\ [\,] & = \\ x\ \text{`}\mathit{elem}\text{`}\ (y : ys) & = \end{array}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$
\begin{array}{lll}
\mathit{length} & :: & [\,a\,] \to \mathit{Int} \\
\mathit{length}\ [\,] & = & 0 \\
\mathit{length}\ (x : xs) & = &
\end{array}
$$

$$
\begin{array}{lll}
\mathit{concat} & :: & [\,[\,a\,]\,] \to [\,a\,] \\
\mathit{concat}\ [\,] & = & \\
\mathit{concat}\ (xs : xss) & = &
\end{array}
$$

$$
\begin{array}{lll}
(+\!\!+) & :: & [\,a\,] \to [\,a\,] \to [\,a\,] \\
[\,] & +\!\!+\ ys = & \\
(x : xs) & +\!\!+\ ys = &
\end{array}
$$

$$
\begin{array}{ll}
\mathit{sum}\ [\,] & = \\
\mathit{sum}\ (x : xs) & =
\end{array}
$$

$$
\begin{array}{ll}
\mathit{product}\ [\,] & = \\
\mathit{product}\ (x : xs) & =
\end{array}
$$

$$
\begin{array}{ll}
x\ \text{`}\mathit{elem}\text{`}\ [\,] & = \\
x\ \text{`}\mathit{elem}\text{`}\ (y : ys) & =
\end{array}
$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{aligned} &\mathit{length} &&:: [a] \to \mathit{Int} \\ &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

$$\begin{aligned} &(+\!\!+) &&:: [a] \to [a] \to [a] \\ &[\,] +\!\!+ ys &&= \\ &(x : xs) +\!\!+ ys &&= \end{aligned}$$

$$\begin{aligned} &x\ `\mathit{elem}`\ [\,] &&= \\ &x\ `\mathit{elem}`\ (y : ys) &&= \end{aligned}$$

$$\begin{aligned} &\mathit{concat} &&:: [[a]] \to [a] \\ &\mathit{concat}\ [\,] &&= \\ &\mathit{concat}\ (xs : xss) &&= \end{aligned}$$

$$\begin{aligned} &\mathit{sum}\ [\,] &&= \\ &\mathit{sum}\ (x : xs) &&= \end{aligned}$$

$$\begin{aligned} &\mathit{product}\ [\,] &&= \\ &\mathit{product}\ (x : xs) &&= \end{aligned}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{array}{lcl} \mathit{length} & :: & [\,a\,] \to \mathit{Int} \\ \mathit{length}\ [\,] & = & 0 \\ \mathit{length}\ (x : \mathit{xs}) & = & 1 + \mathit{length}\ \mathit{xs} \end{array}$$

$$\begin{array}{lcl} \mathit{concat} & :: & [\,[\,a\,]\,] \to [\,a\,] \\ \mathit{concat}\ [\,] & = & \\ \mathit{concat}\ (\mathit{xs} : \mathit{xss}) & = & \end{array}$$

$$\begin{array}{lcl} (+\!\!+) & :: & [\,a\,] \to [\,a\,] \to [\,a\,] \\ [\,]\ +\!\!+\ \mathit{ys} & = & \mathit{ys} \\ (x : \mathit{xs})\ +\!\!+\ \mathit{ys} & = & \end{array}$$

$$\begin{array}{lcl} \mathit{sum}\ [\,] & = & \\ \mathit{sum}\ (x : \mathit{xs}) & = & \end{array}$$

$$\begin{array}{lcl} \mathit{product}\ [\,] & = & \\ \mathit{product}\ (x : \mathit{xs}) & = & \end{array}$$

$$\begin{array}{lcl} x\ \text{`}\mathit{elem}\text{`}\ [\,] & = & \\ x\ \text{`}\mathit{elem}\text{`}\ (y : \mathit{ys}) & = & \end{array}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$
\begin{array}{lcl}
\mathit{length} & :: & [\,a\,] \to \mathit{Int} \\
\mathit{length}\ [\,] & = & 0 \\
\mathit{length}\ (x : xs) & = & 1 + \mathit{length}\ xs
\end{array}
$$

$$
\begin{array}{lcl}
(+\!\!+) & :: & [\,a\,] \to [\,a\,] \to [\,a\,] \\
[\,]\ +\!\!+\ ys & = & ys \\
(x : xs)\ +\!\!+\ ys & = & x : (xs\ +\!\!+\ ys)
\end{array}
$$

$$
\begin{array}{lcl}
x\ \text{`}\mathit{elem}\text{`}\ [\,] & = & \\
x\ \text{`}\mathit{elem}\text{`}\ (y : ys) & = &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{concat} & :: & [\,[\,a\,]\,] \to [\,a\,] \\
\mathit{concat}\ [\,] & = & \\
\mathit{concat}\ (xs : xss) & = &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sum}\ [\,] & = & \\
\mathit{sum}\ (x : xs) & = &
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{product}\ [\,] & = & \\
\mathit{product}\ (x : xs) & = &
\end{array}
$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{array}{lll} \mathit{length} & :: & [a] \to \mathit{Int} \\ \mathit{length}\ [\,] & = & 0 \\ \mathit{length}\ (x : xs) & = & 1 + \mathit{length}\ xs \end{array}$$

$$\begin{array}{lll} (+\!\!+) & :: & [a] \to [a] \to [a] \\ [\,] & +\!\!+\ ys = & ys \\ (x : xs) & +\!\!+\ ys = & x : (xs +\!\!+ ys) \end{array}$$

$$\begin{array}{ll} x\ `\mathit{elem}`\ [\,] & = \\ x\ `\mathit{elem}`\ (y : ys) & = \end{array}$$

$$\begin{array}{lll} \mathit{concat} & :: & [[a]] \to [a] \\ \mathit{concat}\ [\,] & = & [\,] \\ \mathit{concat}\ (xs : xss) & = & \end{array}$$

$$\begin{array}{ll} \mathit{sum}\ [\,] & = \\ \mathit{sum}\ (x : xs) & = \end{array}$$

$$\begin{array}{ll} \mathit{product}\ [\,] & = \\ \mathit{product}\ (x : xs) & = \end{array}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$
\begin{aligned}
&\mathit{length} &&:: [a] \to \mathit{Int}\\
&\mathit{length}\ [\,] &&= 0\\
&\mathit{length}\ (x : \mathit{xs}) &&= 1 + \mathit{length}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{concat} &&:: [[a]] \to [a]\\
&\mathit{concat}\ [\,] &&= [\,]\\
&\mathit{concat}\ (\mathit{xs} : \mathit{xss}) &&= \mathit{xs} +\!\!+\ \mathit{concat}\ \mathit{xss}
\end{aligned}
$$

$$
\begin{aligned}
&(+\!\!+) &&:: [a] \to [a] \to [a]\\
&[\,] +\!\!+\ \mathit{ys} &&= \mathit{ys}\\
&(x : \mathit{xs}) +\!\!+\ \mathit{ys} &&= x : (\mathit{xs} +\!\!+\ \mathit{ys})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sum}\ [\,] &&=\\
&\mathit{sum}\ (x : \mathit{xs}) &&=
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{product}\ [\,] &&=\\
&\mathit{product}\ (x : \mathit{xs}) &&=
\end{aligned}
$$

$$
\begin{aligned}
&x\ \text{`}\mathit{elem}\text{`}\ [\,] &&=\\
&x\ \text{`}\mathit{elem}\text{`}\ (y : \mathit{ys}) &&=
\end{aligned}
$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{aligned} &\mathit{length} &&::\ [a] \to \mathit{Int} \\ &\mathit{length}\ [\,] &&=\ 0 \\ &\mathit{length}\ (x : \mathit{xs}) &&=\ 1 + \mathit{length}\ \mathit{xs} \end{aligned}$$

$$\begin{aligned} &\mathit{concat} &&::\ [[a]] \to [a] \\ &\mathit{concat}\ [\,] &&=\ [\,] \\ &\mathit{concat}\ (\mathit{xs} : \mathit{xss}) &&=\ \mathit{xs} +\!\!+\ \mathit{concat}\ \mathit{xss} \end{aligned}$$

$$\begin{aligned} &(+\!\!+) &&::\ [a] \to [a] \to [a] \\ &[\,] &&+\!\!+\ \mathit{ys} = \mathit{ys} \\ &(x : \mathit{xs}) &&+\!\!+\ \mathit{ys} = x : (\mathit{xs} +\!\!+ \mathit{ys}) \end{aligned}$$

$$\begin{aligned} &\mathit{sum}\ [\,] &&= 0 \\ &\mathit{sum}\ (x : \mathit{xs}) &&= \end{aligned}$$

$$\begin{aligned} &\mathit{product}\ [\,] &&= \\ &\mathit{product}\ (x : \mathit{xs}) &&= \end{aligned}$$

$$\begin{aligned} &x\ \text{`}\mathit{elem}\text{`}\ [\,] &&= \\ &x\ \text{`}\mathit{elem}\text{`}\ (y : \mathit{ys}) &&= \end{aligned}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{aligned} &\mathit{length} &&:: [a] \to \mathit{Int} \\ &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

$$\begin{aligned} &\mathit{concat} &&:: [[a]] \to [a] \\ &\mathit{concat}\ [\,] &&= [\,] \\ &\mathit{concat}\ (xs : xss) &&= xs +\!\!+\ \mathit{concat}\ xss \end{aligned}$$

$$\begin{aligned} &(+\!\!+) &&:: [a] \to [a] \to [a] \\ &[\,] &&+\!\!+\ ys = ys \\ &(x : xs) &&+\!\!+\ ys = x : (xs +\!\!+ ys) \end{aligned}$$

$$\begin{aligned} &\mathit{sum}\ [\,] &&= 0 \\ &\mathit{sum}\ (x : xs) &&= x + \mathit{sum}\ xs \end{aligned}$$

$$\begin{aligned} &\mathit{product}\ [\,] &&= \\ &\mathit{product}\ (x : xs) &&= \end{aligned}$$

$$\begin{aligned} &x\ \text{`}\mathit{elem}\text{`}\ [\,] &&= \\ &x\ \text{`}\mathit{elem}\text{`}\ (y : ys) &&= \end{aligned}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$
\begin{aligned}
&\mathit{length} &&:: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] &&= 0 \\
&\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{concat} &&:: [[a]] \to [a] \\
&\mathit{concat}\ [\,] &&= [\,] \\
&\mathit{concat}\ (xs : xss) &&= xs +\!\!+\ \mathit{concat}\ xss
\end{aligned}
$$

$$
\begin{aligned}
&(+\!\!+) &&:: [a] \to [a] \to [a] \\
&[\,] &&+\!\!+\ ys = ys \\
&(x : xs)\ +\!\!+\ ys &&= x : (xs +\!\!+ ys)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sum}\ [\,] &&= 0 \\
&\mathit{sum}\ (x : xs) &&= x + \mathit{sum}\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{product}\ [\,] &&= 0 \\
&\mathit{product}\ (x : xs) &&=
\end{aligned}
$$

$$
\begin{aligned}
&x\ \text{`}\mathit{elem}\text{`}\ [\,] &&= \\
&x\ \text{`}\mathit{elem}\text{`}\ (y : ys) &&=
\end{aligned}
$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{aligned} &\mathit{length} &&:: [a] \to \mathit{Int} \\ &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

$$\begin{aligned} &\mathit{concat} &&:: [[a]] \to [a] \\ &\mathit{concat}\ [\,] &&= [\,] \\ &\mathit{concat}\ (xs : xss) &&= xs +\!\!+\ \mathit{concat}\ xss \end{aligned}$$

$$\begin{aligned} &(+\!\!+) :: [a] \to [a] \to [a] \\ &[\,] +\!\!+ ys = ys \\ &(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys) \end{aligned}$$

$$\begin{aligned} &\mathit{sum}\ [\,] &&= 0 \\ &\mathit{sum}\ (x : xs) &&= x + \mathit{sum}\ xs \end{aligned}$$

$$\begin{aligned} &\mathit{product}\ [\,] &&= 0 \\ &\mathit{product}\ (x : xs) &&= x * \mathit{product}\ xs \end{aligned}$$

$$\begin{aligned} &x\ `\mathit{elem}`\ [\,] &&= \\ &x\ `\mathit{elem}`\ (y : ys) &&= \end{aligned}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$
\begin{array}{lcl}
\mathit{length} & :: & [\,a\,] \to \mathit{Int} \\
\mathit{length}\ [\,] & = & 0 \\
\mathit{length}\ (x : xs) & = & 1 + \mathit{length}\ xs
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{concat} & :: & [\,[\,a\,]\,] \to [\,a\,] \\
\mathit{concat}\ [\,] & = & [\,] \\
\mathit{concat}\ (xs : xss) & = & xs +\!\!+\ \mathit{concat}\ xss
\end{array}
$$

$$
\begin{array}{lcl}
(+\!\!+) & :: & [\,a\,] \to [\,a\,] \to [\,a\,] \\
[\,] & +\!\!+ & ys = ys \\
(x : xs)\ +\!\!+\ ys & = & x : (xs +\!\!+ ys)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sum}\ [\,] & = & 0 \\
\mathit{sum}\ (x : xs) & = & x + \mathit{sum}\ xs
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{product}\ [\,] & = & 0 \\
\mathit{product}\ (x : xs) & = & x * \mathit{product}\ xs
\end{array}
$$

$$
\begin{array}{lcl}
x\ \text{`}\mathit{elem}\text{`}\ [\,] & = & \mathbf{False} \\
x\ \text{`}\mathit{elem}\text{`}\ (y : ys) & = &
\end{array}
$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{aligned} &\mathit{length} &&:: [a] \to \mathit{Int} \\ &\mathit{length}\ [\,] &&= 0 \\ &\mathit{length}\ (x : xs) &&= 1 + \mathit{length}\ xs \end{aligned}$$

$$\begin{aligned} &\mathit{concat} &&:: [[a]] \to [a] \\ &\mathit{concat}\ [\,] &&= [\,] \\ &\mathit{concat}\ (xs : xss) &&= xs +\!\!+\ \mathit{concat}\ xss \end{aligned}$$

$$\begin{aligned} &(+\!\!+) &&:: [a] \to [a] \to [a] \\ &[\,] +\!\!+\ ys &&= ys \\ &(x : xs) +\!\!+\ ys &&= x : (xs +\!\!+\ ys) \end{aligned}$$

$$\begin{aligned} &\mathit{sum}\ [\,] &&= 0 \\ &\mathit{sum}\ (x : xs) &&= x + \mathit{sum}\ xs \end{aligned}$$

$$\begin{aligned} &\mathit{product}\ [\,] &&= 0 \\ &\mathit{product}\ (x : xs) &&= x * \mathit{product}\ xs \end{aligned}$$

$$\begin{aligned} &x\ \text{`}\mathit{elem}\text{`}\ [\,] &&= \mathbf{False} \\ &x\ \text{`}\mathit{elem}\text{`}\ (y : ys) &&= x \equiv y \ \|\ x\ \text{`}\mathit{elem}\text{`}\ ys \end{aligned}$$

(All these functions are in the standard prelude.)

Guarded Definitions

$$
\begin{array}{llll}
sign\ x & \mid x > 0 & = & 1 \\
 & \mid x == 0 & = & 0 \\
 & \mid x < 0 & = & -1
\end{array}
$$

Guarded Definitions

$$
\begin{array}{llcl}
\mathit{sign}\ x & \mid x > 0 & = & 1 \\
 & \mid x == 0 & = & 0 \\
 & \mid x < 0 & = & -1
\end{array}
$$

$$\mathit{choose} :: \mathit{Ord}\ a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{choose}\ (x, v)\ (y, w)} \\
\quad \mid x > y & = & v \\
\quad \mid x < y & = & w \\
\quad \mid \mathit{otherwise} & = & \mathit{error}\ \text{"I cannot decide!"}
\end{array}
$$

Guarded Definitions

$$
\begin{aligned}
&sign\ x \quad | \ x > 0 \ = \ 1 \\
&\qquad\quad\ \ | \ x == 0 \ = \ 0 \\
&\qquad\quad\ \ | \ x < 0 \ = \ \text{-}1
\end{aligned}
$$

$$
\begin{aligned}
&choose :: Ord\ a \Rightarrow (a,b) \rightarrow (a,b) \rightarrow b \\
&choose\ (x,v)\ (y,w) \\
&\qquad |\ x > y \quad = \ v \\
&\qquad |\ x < y \quad = \ w \\
&\qquad |\ otherwise \ = \ error\ \text{"I cannot decide!"}
\end{aligned}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
&take\_while\ p\ (x : xs)\ |\ p\ x \ = \ x : take\_while\ p\ xs \\
&take\_while\ p\ xs \qquad\qquad\quad\ = \ [\,]
\end{aligned}
$$

Guarded Definitions

$$
\begin{aligned}
sign\ x \quad & \mid x > 0 \;=\; 1 \\
& \mid x == 0 \;=\; 0 \\
& \mid x < 0 \;=\; -1
\end{aligned}
$$

$$
\begin{aligned}
& choose :: Ord\ a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b \\
& choose\ (x, v)\ (y, w) \\
& \quad \mid x > y \;=\; v \\
& \quad \mid x < y \;=\; w \\
& \quad \mid otherwise \;=\; error\ \text{"I cannot decide!"}
\end{aligned}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
& take\_while\ p\ (x : xs) \mid p\ x \;=\; x : take\_while\ p\ xs \\
& take\_while\ p\ xs \;=\; [\,]
\end{aligned}
$$

$$take\_while\ (< 5)\ [1, 2, 3]$$

Guarded Definitions

$$\begin{aligned} sign\ x\ \ & \mid x > 0 = 1 \\ & \mid x == 0 = 0 \\ & \mid x < 0 = -1 \end{aligned}$$

$$\begin{aligned} &choose :: Ord\ a \Rightarrow (a,b) \rightarrow (a,b) \rightarrow b \\ &choose\ (x,v)\ (y,w) \\ &\quad \mid x > y \quad = v \\ &\quad \mid x < y \quad = w \\ &\quad \mid otherwise = error \text{ "I cannot decide!"} \end{aligned}$$

If no guard succeeds, the next pattern is tried:

$$\begin{aligned} &take\_while\ p\ (x : xs) \mid p\ x = x : take\_while\ p\ xs \\ &take\_while\ p\ xs \qquad\qquad\quad = [\,] \end{aligned}$$

$$\begin{aligned} &take\_while\ (<5)\ [1, 2, 3] \\ &= take\_while\ (<5)\ (1 : 2 : 3 : [\,]) \end{aligned}$$

Guarded Definitions

$$
\begin{array}{llll}
sign\ x & \mid x > 0 & = & 1 \\
 & \mid x == 0 & = & 0 \\
 & \mid x < 0 & = & -1
\end{array}
$$

$$
\begin{array}{l}
choose :: Ord\ a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b \\
choose\ (x, v)\ (y, w) \\
\quad \mid x > y \quad = v \\
\quad \mid x < y \quad = w \\
\quad \mid otherwise = error\ \text{"I cannot decide!"}
\end{array}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{array}{ll}
take\_while\ p\ (x : xs) \mid p\ x & = x : take\_while\ p\ xs \\
take\_while\ p\ xs & = [\,]
\end{array}
$$

$$
\begin{array}{l}
take\_while\ (< 5)\ [1, 2, 3] \\
= take\_while\ (< 5)\ (1 : 2 : 3 : [\,]) \\
= 1 : take\_while\ (< 5)\ (2 : 3 : [\,])
\end{array}
$$

Guarded Definitions

$$
\begin{aligned}
\mathit{sign}\ x\ &|\ x > 0 = 1\\
&|\ x == 0 = 0\\
&|\ x < 0 = -1
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{choose} :: \mathit{Ord}\ a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b\\
&\mathit{choose}\ (x, v)\ (y, w)\\
&\quad |\ x > y = v\\
&\quad |\ x < y = w\\
&\quad |\ \mathit{otherwise} = \mathit{error}\ \text{"I cannot decide!"}
\end{aligned}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
&\mathit{take\_while}\ p\ (x : xs)\ |\ p\ x = x : \mathit{take\_while}\ p\ xs\\
&\mathit{take\_while}\ p\ xs = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{take\_while}\ (< 5)\ [1, 2, 3]\\
&= \mathit{take\_while}\ (< 5)\ (1 : 2 : 3 : [\,])\\
&= 1 : \mathit{take\_while}\ (< 5)\ (2 : 3 : [\,])\\
&= 1 : 2 : \mathit{take\_while}\ (< 5)\ (3 : [\,])
\end{aligned}
$$

Guarded Definitions

$$
\begin{aligned}
sign\ x\ \ & \mid x > 0 = 1 \\
& \mid x == 0 = 0 \\
& \mid x < 0 = -1
\end{aligned}
$$

$$
\begin{aligned}
& choose :: Ord\ a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b \\
& choose\ (x, v)\ (y, w) \\
& \quad \mid x > y \quad = v \\
& \quad \mid x < y \quad = w \\
& \quad \mid otherwise = error\ \text{"I cannot decide!"}
\end{aligned}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
& take\_while\ p\ (x : xs) \mid p\ x = x : take\_while\ p\ xs \\
& take\_while\ p\ xs \qquad\qquad\quad = [\,]
\end{aligned}
$$

$$
\begin{aligned}
& take\_while\ (<5)\ [1, 2, 3] \\
& = take\_while\ (<5)\ (1 : 2 : 3 : [\,]) \\
& = 1 : take\_while\ (<5)\ (2 : 3 : [\,]) \\
& = 1 : 2 : take\_while\ (<5)\ (3 : [\,]) \\
& = 1 : 2 : 3 : take\_while\ (<5)\ [\,]
\end{aligned}
$$

Guarded Definitions

$$
\begin{aligned}
\mathit{sign}\ x\ \ &|\ x > 0 \ = \ 1\\
&|\ x == 0 \ = \ 0\\
&|\ x < 0 \ = \ \text{-}1
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{choose} :: \mathit{Ord}\ a \Rightarrow (a, b) \to (a, b) \to b\\
&\mathit{choose}\ (x, v)\ (y, w)\\
&\quad |\ x > y \quad = \ v\\
&\quad |\ x < y \quad = \ w\\
&\quad |\ \mathit{otherwise} \ = \ \mathit{error}\ \text{"I cannot decide!"}
\end{aligned}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
&\mathit{take\_while}\ p\ (x : xs)\ |\ p\ x \ = \ x : \mathit{take\_while}\ p\ xs\\
&\mathit{take\_while}\ p\ xs \qquad\qquad\quad = \ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{take\_while}\ (<5)\ [1, 2, 3]\\
&= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : [\,])\\
&= 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : [\,])\\
&= 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : [\,])\\
&= 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ [\,]\\
&= 1 : 2 : 3 : [\,]
\end{aligned}
$$

Guarded Definitions

$$
\begin{aligned}
sign\ x\ &|\ x > 0 = 1 \\
&|\ x == 0 = 0 \\
&|\ x < 0 = -1
\end{aligned}
$$

$$
\begin{aligned}
&choose :: Ord\ a \Rightarrow (a,b) \rightarrow (a,b) \rightarrow b \\
&choose\ (x,v)\ (y,w) \\
&\quad |\ x > y \quad = v \\
&\quad |\ x < y \quad = w \\
&\quad |\ otherwise = error\ \text{"I cannot decide!"}
\end{aligned}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
&take\_while\ p\ (x : xs)\ |\ p\ x = x : take\_while\ p\ xs \\
&take\_while\ p\ xs \qquad\qquad\quad = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&take\_while\ (< 5)\ [1, 2, 3] \\
&= take\_while\ (< 5)\ (1 : 2 : 3 : [\,]) \\
&= 1 : take\_while\ (< 5)\ (2 : 3 : [\,]) \\
&= 1 : 2 : take\_while\ (< 5)\ (3 : [\,]) \\
&= 1 : 2 : 3 : take\_while\ (< 5)\ [\,] \\
&= 1 : 2 : 3 : [\,] \\
&= [1, 2, 3]
\end{aligned}
$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$
\begin{array}{lcl}
\mathit{take\_while}\ p\ (x : xs) \mid p\ x & = & x : \mathit{take\_while}\ p\ xs \\
\mathit{take\_while}\ p\ xs & = & [\,]
\end{array}
$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{array}{lcl} \mathit{take\_while}\ p\ (x : xs)\ \mid\ p\ x & = & x : \mathit{take\_while}\ p\ xs \\ \mathit{take\_while}\ p\ xs & = & [\,] \end{array}$$

$$\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{array}{lcl} \mathit{take\_while}\ p\ (x : xs) \mid p\ x & = & x : \mathit{take\_while}\ p\ xs \\ \mathit{take\_while}\ p\ xs & = & [\,] \end{array}$$

$$\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]$$
$$= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,])$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{array}{lcl} \mathit{take\_while}\ p\ (x : xs)\ \mid\ p\ x & = & x : \mathit{take\_while}\ p\ xs \\ \mathit{take\_while}\ p\ xs & = & [\,] \end{array}$$

$$\begin{array}{l} \mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\ = \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ = 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \end{array}$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{aligned} &\mathit{take\_while}\ p\ (x : xs) \mid p\ x && = x : \mathit{take\_while}\ p\ xs \\ &\mathit{take\_while}\ p\ xs && = [\,] \end{aligned}$$

$$\begin{aligned} &\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\ &= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ &= 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ &= 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \end{aligned}$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$
\begin{array}{lcl}
\mathit{take\_while}\ p\ (x : xs)\ |\ p\ x & = & x : \mathit{take\_while}\ p\ xs \\
\mathit{take\_while}\ p\ xs & = & [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,])
\end{array}
$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{array}{lcl} \mathit{take\_while}\ p\ (x : xs) \mid p\ x & = & x : \mathit{take\_while}\ p\ xs \\ \mathit{take\_while}\ p\ xs & = & [\,] \end{array}$$

$$\begin{array}{l} \mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\ = \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ = 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ = 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ = 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\ = 1 : 2 : 3 : 2 : \mathit{take\_while}\ (<5)\ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \end{array}$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{aligned} &\mathit{take\_while}\ p\ (x : xs) \mid p\ x &&= x : \mathit{take\_while}\ p\ xs \\ &\mathit{take\_while}\ p\ xs &&= [\,] \end{aligned}$$

$$\begin{aligned} &\mathit{take\_while}\ (<5)\ [1,2,3,2,3,4,3,4,5,4,3,4,5,6] \\ &= \mathit{take\_while}\ (<5)\ (1:2:3:2:3:4:3:4:5:4:3:4:5:6:[\,]) \\ &= 1 : \mathit{take\_while}\ (<5)\ (2:3:2:3:4:3:4:5:4:3:4:5:6:[\,]) \\ &= 1:2 : \mathit{take\_while}\ (<5)\ (3:2:3:4:3:4:5:4:3:4:5:6:[\,]) \\ &= 1:2:3 : \mathit{take\_while}\ (<5)\ (2:3:4:3:4:5:4:3:4:5:6:[\,]) \\ &= 1:2:3:2 : \mathit{take\_while}\ (<5)\ (3:4:3:4:5:4:3:4:5:6:[\,]) \\ &= 1:2:3:2:3 : \mathit{take\_while}\ (<5)\ (4:3:4:5:4:3:4:5:6:[\,]) \end{aligned}$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
&\mathit{take\_while}\ p\ (x : xs)\ |\ p\ x &&= x : \mathit{take\_while}\ p\ xs \\
&\mathit{take\_while}\ p\ xs &&= [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{take\_while}\ (<5)\ [1,2,3,2,3,4,3,4,5,4,3,4,5,6] \\
&= \mathit{take\_while}\ (<5)\ (1:2:3:2:3:4:3:4:5:4:3:4:5:6:[\,]) \\
&= 1 : \mathit{take\_while}\ (<5)\ (2:3:2:3:4:3:4:5:4:3:4:5:6:[\,]) \\
&= 1 : 2 : \mathit{take\_while}\ (<5)\ (3:2:3:4:3:4:5:4:3:4:5:6:[\,]) \\
&= 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ (2:3:4:3:4:5:4:3:4:5:6:[\,]) \\
&= 1 : 2 : 3 : 2 : \mathit{take\_while}\ (<5)\ (3:4:3:4:5:4:3:4:5:6:[\,]) \\
&= 1 : 2 : 3 : 2 : 3 : \mathit{take\_while}\ (<5)\ (4:3:4:5:4:3:4:5:6:[\,]) \\
&= 1 : 2 : 3 : 2 : 3 : 4 : \mathit{take\_while}\ (<5)\ (3:4:5:4:3:4:5:6:[\,])
\end{aligned}
$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$
\begin{aligned}
&\mathit{take\_while}\ p\ (x : xs) \mid p\ x &&= x : \mathit{take\_while}\ p\ xs \\
&\mathit{take\_while}\ p\ xs &&= [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
&= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : 2 : 3 : 2 : \mathit{take\_while}\ (<5)\ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : 2 : 3 : 2 : 3 : \mathit{take\_while}\ (<5)\ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : 2 : 3 : 2 : 3 : 4 : \mathit{take\_while}\ (<5)\ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
&= 1 : 2 : 3 : 2 : 3 : 4 : 3 : \mathit{take\_while}\ (<5)\ (4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,])
\end{aligned}
$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$\begin{array}{lcl} \mathit{take\_while}\ p\ (x : xs) \mid p\ x & = & x : \mathit{take\_while}\ p\ xs \\ \mathit{take\_while}\ p\ xs & = & [\,] \end{array}$$

$$\begin{array}{l}
\mathit{take_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \mathit{take_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : \mathit{take_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : \mathit{take_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : \mathit{take_while}\ (<5)\ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : \mathit{take_while}\ (<5)\ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : \mathit{take_while}\ (<5)\ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : \mathit{take_while}\ (<5)\ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : \mathit{take_while}\ (<5)\ (4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : \mathit{take_while}\ (<5)\ (5 : 4 : 3 : 4 : 5 : 6 : [\,])
\end{array}$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$
\begin{array}{lcl}
\mathit{take\_while}\ p\ (x : xs) \mid p\ x & = & x : \mathit{take\_while}\ p\ xs \\
\mathit{take\_while}\ p\ xs & = & [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : \mathit{take\_while}\ (<5)\ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : \mathit{take\_while}\ (<5)\ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : \mathit{take\_while}\ (<5)\ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : \mathit{take\_while}\ (<5)\ (4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : \mathit{take\_while}\ (<5)\ (5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : [\,]
\end{array}
$$

Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

$$
\begin{array}{lll}
\mathit{take\_while}\ p\ (x : xs) \mid p\ x & = & x : \mathit{take\_while}\ p\ xs \\
\mathit{take\_while}\ p\ xs & = & [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{take\_while}\ (<5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \mathit{take\_while}\ (<5)\ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : \mathit{take\_while}\ (<5)\ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : \mathit{take\_while}\ (<5)\ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : \mathit{take\_while}\ (<5)\ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : \mathit{take\_while}\ (<5)\ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : \mathit{take\_while}\ (<5)\ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : \mathit{take\_while}\ (<5)\ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : \mathit{take\_while}\ (<5)\ (4 : 5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : \mathit{take\_while}\ (<5)\ (5 : 4 : 3 : 4 : 5 : 6 : [\,]) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : [\,] \\
= [1, 2, 3, 2, 3, 4, 3, 4]
\end{array}
$$

`case` Expressions

```
sign x = case compare x 0 of
           GT ->  1
           EQ ->  0
           LT -> -1
```

`case` Expressions

```
sign x = case compare x 0 of
           GT ->  1
           EQ ->  0
           LT -> -1
```

The prelude datatype *Ordering* has three elements

data *Ordering* = *LT* | *EQ* | *GT*

`case` Expressions

```
sign x = case compare x 0 of
           GT ->  1
           EQ ->  0
           LT -> -1
```

The prelude datatype *Ordering* has three elements and is used mostly as result type of the prelude function *compare*:

data *Ordering* = *LT* | *EQ* | *GT*

$compare :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Ordering$

`case` Expressions

```
sign x = case compare x 0 of
           GT ->  1
           EQ ->  0
           LT -> -1
```

The prelude datatype *Ordering* has three elements and is used mostly as result type of the prelude function *compare*:

data *Ordering* = *LT* | *EQ* | *GT*

$compare :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Ordering$

Another example:

choose (*x*,*v*) (*y*,*w*) = **case** *compare x y* **of**
    GT $\rightarrow$ *v*
    LT $\rightarrow$ *w*
    EQ $\rightarrow$ *error* "I cannot decide!"

`if` … `then` … `else` … and `case` Expressions

The type *Bool* can be considered as a two-element enumeration type:

data *Bool* = **False** | **True**

`if` ... `then` ... `else` ... and `case` Expressions

The type *Bool* can be considered as a two-element enumeration type:

data *Bool* = **False** | **True**

Conditional expressions are "syntactic sugar" for **case** expressions over *Bool*:

if *condition* **then** *expr1* **else** *expr2*	$\equiv$	**case** *condition* **of** **True** $\rightarrow$ *expr1* **False** $\rightarrow$ *expr2*

`if` … `then` … `else` … **and** `case` **Expressions**

The type *Bool* can be considered as a two-element enumeration type:

data *Bool* = **False** | **True**

Conditional expressions are "syntactic sugar" for **case** expressions over *Bool*:

if *condition* **then** *expr1* **else** *expr2*	$\equiv$	**case** *condition* **of** **True** $\rightarrow$ *expr1* **False** $\rightarrow$ *expr2*

Two ways of defining functions:

Pattern Matching	`case`
not **True** = **False** *not* **False** = **True**	*not* *b* = **case** *b* **of** **True** $\rightarrow$ **False** **False** $\rightarrow$ **True**

`case` Expressions are "Anonymous" Pattern Matching

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \to \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : \mathit{xs}) = x +\!\!+\ \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \to [\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad \_ \ \to \texttt{","} : \mathit{commaWords}\ \mathit{xs}
\end{array}
$$

`case` Expressions are "Anonymous" Pattern Matching

$$
\begin{aligned}
&\mathit{commaWords} :: [\mathit{String}] \to \mathit{String} \\
&\mathit{commaWords}\ [\,] = [\,] \\
&\mathit{commaWords}\ (x : xs) = x +\!\!+\ \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad [\,] \to [\,] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \_ \ \to \text{","} : \mathit{commaWords}\ xs
\end{aligned}
$$

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching

`case` Expressions are "Anonymous" Pattern Matching

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \to \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : xs) = x +\!\!+ \ \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \to [\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad \_ \ \ \to "," : \mathit{commaWords}\ xs
\end{array}
$$

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \to \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : xs) = x +\!\!+ \ \mathit{commaWordsAux}\ xs
\end{array}
$$

`case` Expressions are "Anonymous" Pattern Matching

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \rightarrow \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : \mathit{xs}) = x \mathbin{+\!\!+} \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \rightarrow [\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad \_ \rightarrow \texttt{","} : \mathit{commaWords}\ \mathit{xs}
\end{array}
$$

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \rightarrow \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : \mathit{xs}) = x \mathbin{+\!\!+} \mathit{commaWordsAux}\ \mathit{xs}
\end{array}
$$

$$
\begin{array}{l}
\mathit{commaWordsAux}\ [\,] = [\,] \\
\mathit{commaWordsAux}\ \mathit{xs} = \texttt{","} : \mathit{commaWords}\ \mathit{xs}
\end{array}
$$

`where` Clauses

`where` Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**

`where` Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**, e.g.:

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \to \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : \mathit{xs}) = x +\!\!+\ \mathit{commaWordsAux}\ \mathit{xs} \\
\quad \textbf{where} \\
\qquad \mathit{commaWordsAux}\ [\,] = [\,] \\
\qquad \mathit{commaWordsAux}\ \mathit{xs} = \texttt{","} : \mathit{commaWords}\ \mathit{xs}
\end{array}
$$

`where` Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**, e.g.:

$$
\begin{aligned}
&\mathit{commaWords} :: [\mathit{String}] \to \mathit{String}\\
&\mathit{commaWords}\ [\,] = [\,]\\
&\mathit{commaWords}\ (x : xs) = x +\!\!+\ \mathit{commaWordsAux}\ xs\\
&\quad \textbf{where}\\
&\qquad \mathit{commaWordsAux}\ [\,] = [\,]\\
&\qquad \mathit{commaWordsAux}\ xs = \texttt{","} : \mathit{commaWords}\ xs
\end{aligned}
$$

`where` clauses are visible **only** within their enclosing clause, here "$\mathit{commaWords}\ (x : xs) = \ldots$"

`where` Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**, e.g.:

$$
\begin{array}{l}
\mathit{commaWords} :: [\mathit{String}] \to \mathit{String} \\
\mathit{commaWords}\ [\,] = [\,] \\
\mathit{commaWords}\ (x : \mathit{xs}) = x +\!\!+\ \mathit{commaWordsAux}\ \mathit{xs} \\
\quad \textbf{where} \\
\qquad \mathit{commaWordsAux}\ [\,] = [\,] \\
\qquad \mathit{commaWordsAux}\ \mathit{xs} = \texttt{","} : \mathit{commaWords}\ \mathit{xs}
\end{array}
$$

`where` clauses are visible **only** within their enclosing clause, here "$\mathit{commaWords}\ (x : \mathit{xs}) = \ldots$"

`where` clauses are visible within all guards:

```
f x y | y >  z  =  …
      | y == z  =  …
      | y <  z  =  …
  where z = x * x
```

`let` Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k ‘mod‘ n
        in if m == 0
           then n
           else f n m
```

`let` Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k ‘mod‘ n
        in if m == 0
           then n
           else f n m

h x y =  let x2 = x * x
             y2 = y * y
         in sqrt (x2 + y2)
```

`let` Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k ‘mod‘ n
        in if m == 0
           then n
           else f n m

h x y =  let x2 = x * x
             y2 = y * y
         in sqrt (x2 + y2)
```

Definitions can use **pattern bindings**:

```
g k n = let (d,m) = divMod k n
        in if d == 0
           then [m]
           else g d n ++ [m]
```

`let` Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k ‘mod‘ n
        in if m == 0
           then n
           else f n m

h x y =  let x2 = x * x
             y2 = y * y
         in sqrt (x2 + y2)
```

Definitions can use **pattern bindings**:

```
g k n = let (d,m) = divMod k n
        in if d == 0
           then [m]
           else g d n ++ [m]
```

Guards, `let` and `where` bindings, and `case` cases all are **layout sensitive**!

`let` or `where`?

`let` **or** `where`**?**

- `let` *bindings* `in` *expression*

  is an **expression**

`let` **or** `where`**?**

- `let` *bindings* `in` *expression*

  is an **expression**

- *fname patterns guardedRHSs* `where` *bindings*

  is a clause that is part of a **definition**

`let` **or** `where`**?**

- `let` *bindings* `in` *expression*

  is an **expression**

- *fname patterns guardedRHSs* `where` *bindings*

  is a clause that is part of a **definition**

- (`where` clauses can also modify `case` cases)

`let` **or** `where`?

- `let` *bindings* `in` *expression*

  is an **expression**

- *fname patterns guardedRHSs* `where` *bindings*

  is a clause that is part of a **definition**

- (`where` clauses can also modify `case` cases)

Frequently, the choice between `let` and `where` is a matter of *style*:

- `where` clauses result in a top-down presentation
- `let` expressions lend themselves also to bottom-up presentations

Some Prelude Functions — Elementary List Access

```
head                 :: [a] -> a
head (x:_)            = x

last                 :: [a] -> a
last [x]              = x
last (_:xs)           = last xs

tail                 :: [a] -> [a]
tail (_:xs)           = xs

init                 :: [a] -> [a]
init [x]              = []
init (x:xs)           = x : init xs

null                 :: [a] -> Bool
null []               = True
null (_:_)            = False
```

Some Prelude Functions — List Indexing

```
length                :: [a] -> Int
length                 = foldl' (\n _ -> n + 1) 0

(!!)                  :: [b] -> Int -> b
(x:_)  !! 0            = x
(_:xs) !! n | n>0      = xs !! (n-1)
(_:_)  !! _            = error "PreludeList.!!: negative index"
[]     !! _            = error "PreludeList.!!: index too large"
```

Some Prelude Functions — Positional List Splitting

```
take                    :: Int -> [a] -> [a]
take 0 _                 = []
take _ []                = []
take n (x:xs) | n>0      = x : take (n-1) xs
take _ _                 = error "take: negative argument"

drop                    :: Int -> [a] -> [a]
drop 0 xs                = xs
drop _ []                = []
drop n (_:xs) | n>0      = drop (n-1) xs
drop _ _                 = error "drop: negative argument"

splitAt                     :: Int -> [a] -> ([a], [a])
splitAt 0 xs                 = ([],xs)
splitAt _ []                 = ([],[])
splitAt n (x:xs) | n>0 = (x:xs’,xs”)
                 where (xs’,xs”) = splitAt (n-1) xs
splitAt _ _            = error "splitAt: negative argument"
```

Some Prelude Functions — Concatenation, Iteration

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

concat :: [[a]] -> [a]
concat = foldr (++) []

iterate                :: (a -> a) -> a -> [a]
iterate f x             = x : iterate f (f x)

repeat                 :: a -> [a]
repeat x                = xs  where  xs = x:xs
{– repeat x             = x : repeat x –}       – – for understanding

replicate              :: Int -> a -> [a]
replicate n x           = take n (repeat x)

cycle                  :: [a] -> [a]
cycle xs                = xs’  where  xs’ = xs ++ xs’
```

Separation of Concerns: Generation and Consumption

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                    -- replicate
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                        -- replicate
= take 3 ('!' : repeat '!')                  -- repeat
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                    -- replicate
= take 3 ('!' : repeat '!')              -- repeat
= '!' : take (3 - 1) (repeat '!')        -- take (iii)
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                      -- replicate
= take 3 ('!' : repeat '!')                -- repeat
= '!' : take (3 - 1) (repeat '!')          -- take (iii)
= '!' : take 2 (repeat '!')                -- subtraction
```

Separation of Concerns: Generation and Consumption

```
replicate 3 ’!’
= take 3 (repeat ’!’)                          –– replicate
= take 3 (’!’ : repeat ’!’)                    –– repeat
= ’!’ : take (3 - 1) (repeat ’!’)              –– take (iii)
= ’!’ : take 2 (repeat ’!’)                    –– subtraction
= ’!’ : take 2 (’!’ : repeat ’!’)              –– repeat
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                          -- replicate
= take 3 ('!' : repeat '!')                    -- repeat
= '!' : take (3 - 1) (repeat '!')              -- take (iii)
= '!' : take 2 (repeat '!')                    -- subtraction
= '!' : take 2 ('!' : repeat '!')              -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')        -- take (iii)
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                              -- replicate
= take 3 ('!' : repeat '!')                        -- repeat
= '!' : take (3 - 1) (repeat '!')                  -- take (iii)
= '!' : take 2 (repeat '!')                        -- subtraction
= '!' : take 2 ('!' : repeat '!')                  -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')            -- take (iii)
= '!' : '!' : take 1 (repeat '!')                  -- subtraction
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                              -- replicate
= take 3 ('!' : repeat '!')                        -- repeat
= '!' : take (3 - 1) (repeat '!')                  -- take (iii)
= '!' : take 2 (repeat '!')                        -- subtraction
= '!' : take 2 ('!' : repeat '!')                  -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')            -- take (iii)
= '!' : '!' : take 1 (repeat '!')                  -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')            -- repeat
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                                  -- replicate
= take 3 ('!' : repeat '!')                            -- repeat
= '!' : take (3 - 1) (repeat '!')                      -- take (iii)
= '!' : take 2 (repeat '!')                            -- subtraction
= '!' : take 2 ('!' : repeat '!')                      -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')                -- take (iii)
= '!' : '!' : take 1 (repeat '!')                      -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')                -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!')          -- take (iii)
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                                 -- replicate
= take 3 ('!' : repeat '!')                           -- repeat
= '!' : take (3 - 1) (repeat '!')                     -- take (iii)
= '!' : take 2 (repeat '!')                           -- subtraction
= '!' : take 2 ('!' : repeat '!')                     -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')               -- take (iii)
= '!' : '!' : take 1 (repeat '!')                     -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')               -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!')         -- take (iii)
= '!' : '!' : '!' : take 0 (repeat '!')               -- subtraction
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                                  -- replicate
= take 3 ('!' : repeat '!')                            -- repeat
= '!' : take (3 - 1) (repeat '!')                      -- take (iii)
= '!' : take 2 (repeat '!')                            -- subtraction
= '!' : take 2 ('!' : repeat '!')                      -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')                -- take (iii)
= '!' : '!' : take 1 (repeat '!')                      -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')                -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!')          -- take (iii)
= '!' : '!' : '!' : take 0 (repeat '!')                -- subtraction
= '!' : '!' : '!' : []                                 -- take (i)
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                               -- replicate
= take 3 ('!' : repeat '!')                         -- repeat
= '!' : take (3 - 1) (repeat '!')                   -- take (iii)
= '!' : take 2 (repeat '!')                         -- subtraction
= '!' : take 2 ('!' : repeat '!')                   -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')             -- take (iii)
= '!' : '!' : take 1 (repeat '!')                   -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')             -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!')       -- take (iii)
= '!' : '!' : '!' : take 0 (repeat '!')             -- subtraction
= '!' : '!' : '!' : []                              -- take (i)
= "!!!"
```

What We Have Seen So Far

What We Have Seen So Far

- **Functional programming:**

What We Have Seen So Far

- **Functional programming:** Higher-order functions

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:**

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables)

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:**

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:**

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:** not by value or reference

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:** not by value or reference, but by name

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:** not by value or reference, but by name
- **Powerful datatypes** with simple interface:

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:** not by value or reference, but by name
- **Powerful datatypes** with simple interface: *Integer*, lists, lists of lists of …

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, “associate to the left/right”
- **Argument passing:** not by value or reference, but by name
- **Powerful datatypes** with simple interface: *Integer*, lists, lists of lists of …
- **Non-local control** (evaluation on demand): modularity (e.g., generate / prune)

Some Prelude Functions — List Splitting with Predicates

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

dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile p []      = []
dropWhile p xs@(x:xs')
      | p x         = dropWhile p xs'
      | otherwise = xs

span, break        :: (a -> Bool) -> [a] -> ([a],[a])
span p []           = ([],[])
span p xs@(x:xs')
      | p x         = let (ys,zs) = span p xs' in (x:ys,zs)
      | otherwise = ([],xs)

break p             = span (not . p)
```

as-Patterns

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

as-Patterns

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

Consider matching of the third clause against *dropWhile* (< 5) $[1,2,3]$:

as-Patterns

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

Consider matching of the third clause against *dropWhile* (<5) $[1,2,3]$:

- $p =$
- $xs =$
- $x =$
- $xs' =$

as-Patterns

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

Consider matching of the third clause against $\mathit{dropWhile}\ (< 5)\ [1,2,3]$:

- $p = (< 5)$
- $xs =$
- $x =$
- $xs' =$

as-Patterns

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

Consider matching of the third clause against $\mathit{dropWhile}\ (< 5)\ [1,2,3]$:

- $p = (< 5)$
- $xs = [1,2,3]$
- $x =$
- $xs' =$

as-Patterns

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

Consider matching of the third clause against *dropWhile* (< 5) $[1,2,3]$:

- $p = (< 5)$
- $xs = [1,2,3]$
- $x = 1$
- $xs' =$

as-Patterns

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

Consider matching of the third clause against *dropWhile* (< 5) $[1,2,3]$:

- $p = (< 5)$
- $xs = [1,2,3]$
- $x = 1$
- $xs' = [2,3]$

as-Patterns

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

Consider matching of the third clause against $\mathit{dropWhile}\ (<5)\ [1,2,3]$:

- $p = (<5)$
- $xs = [1,2,3]$
- $x = 1$
- $xs' = [2,3]$
- $p\ x = (<5)\ 1 = 1 < 5 = \mathbf{True}$

as-Patterns

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

Consider matching of the third clause against *dropWhile* (< 5) $[1,2,3]$:

- $p = (< 5)$
- $xs = [1,2,3]$
- $x = 1$
- $xs' = [2,3]$
- $p\ x = (< 5)\ 1 = 1 < 5 =$ **True**

Therefore: *dropWhile* (< 5) $[1,2,3] =$

as-Patterns

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

Consider matching of the third clause against $\mathit{dropWhile}\ (<5)\ [1,2,3]$:

- $p = (<5)$
- $xs = [1,2,3]$
- $x = 1$
- $xs' = [2,3]$
- $p\ x = (<5)\ 1 = 1 < 5 = \mathbf{True}$

Therefore: $\mathit{dropWhile}\ (<5)\ [1,2,3] = \mathit{dropWhile}\ (<5)\ [2,3]$

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* `(< 5) [5,4,3]`:

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* `(< 5) [5,4,3]`:

- *p* =
- *xs* =
- *x* =
- *xs’* =

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* (< 5) [5,4,3]:

- $p = (< 5)$
- $xs =$
- $x =$
- $xs' =$

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* `(< 5) [5,4,3]`:

- p = `(< 5)`
- xs = `[5,4,3]`
- x =
- xs' =

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* (< 5) [5,4,3]:

- *p* = (< 5)
- *xs* = [5,4,3]
- *x* = 5
- *xs'* =

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* `(< 5) [5,4,3]`:

- *p* = `(< 5)`
- *xs* = `[5,4,3]`
- *x* = `5`
- *xs'* = `[4,3]`

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* (<5) $[5,4,3]$:

- $p = (<5)$
- $xs = [5,4,3]$
- $x = 5$
- $xs' = [4,3]$
- $p\ x = (<5)\ 5 = 5 < 5 =$ **False**

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* (< 5) $[5,4,3]$:

- $p = (< 5)$
- $xs = [5,4,3]$
- $x = 5$
- $xs' = [4,3]$
- $p\ x = (< 5)\ 5 = 5 < 5 =$ **False**

Therefore: *dropWhile* (< 5) $[5,4,3] =$

as-Patterns — 2

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

Consider matching of the third clause against *dropWhile* (< 5) [5,4,3]:

- *p* = (< 5)
- *xs* = [5,4,3]
- *x* = 5
- *xs*' = [4,3]
- *p x* = (< 5) 5 = 5 < 5 = **False**

Therefore: *dropWhile* (< 5) [5,4,3] = [5,4,3]

Some Prelude Functions — List Splitting with Predicates

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

dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile p []      = []
dropWhile p xs@(x:xs')
      | p x         = dropWhile p xs'
      | otherwise = xs

span, break        :: (a -> Bool) -> [a] -> ([a],[a])
span p []           = ([],[])
span p xs@(x:xs')
      | p x         = let (ys,zs) = span p xs' in (x:ys,zs)
      | otherwise = ([],xs)

break p             = span (not . p)
```

Some Prelude Functions — Text Processing

```
lines        :: String -> [String]
lines ""      = []
lines s       = let (l,s') = break ('\n'==) s
                in l : case s' of []      -> []
                                  (_:s'') -> lines s''

words        :: String -> [String]
words s       = case dropWhile isSpace s of
                     "" -> []
                     s' -> w : words s''
                           where (w,s'') = break isSpace s'

unlines      :: [String] -> String
unlines []        = []
unlines (l:ls)    = l ++ '\n' : unlines ls

unwords      :: [String] -> String
unwords []        =  ""
unwords [w]       = w
unwords (w:ws)    = w ++ ' ' : unwords ws
```

`map` and `filter`

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

`map` and `filter`

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

filter :: (a -> Bool) -> ([a] -> [a])
filter p [] = []
filter p (x : xs) = if p x then x : rest else rest
  where rest = filter p xs
```

`map` and `filter`

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

filter :: (a -> Bool) -> ([a] -> [a])
filter p [] = []
filter p (x : xs) = if p x then x : rest else rest
  where rest = filter p xs
```

These functions could also be defined via list comprehension:

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

`map` and `filter`

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

filter :: (a -> Bool) -> ([a] -> [a])
filter p [] = []
filter p (x : xs) = if p x then x : rest else rest
  where rest = filter p xs
```

These functions could also be defined via list comprehension:

```
map    f xs = [ f x | x <- xs ]
filter p xs = [   x | x <- xs, p x ]
```

`map` and `filter`

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

filter :: (a -> Bool) -> ([a] -> [a])
filter p [] = []
filter p (x : xs) = if p x then x : rest else rest
  where rest = filter p xs
```

These functions could also be defined via list comprehension:

```
map    f xs = [ f x | x <- xs ]
filter p xs = [   x | x <- xs, p x ]
```

Examples:

```
map   (7 *) [1 .. 6] = [7, 14, 21, 28, 35, 42]
```

`map` and `filter`

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

filter :: (a -> Bool) -> ([a] -> [a])
filter p [] = []
filter p (x : xs) = if p x then x : rest else rest
  where rest = filter p xs
```

These functions could also be defined via list comprehension:

```
map    f xs = [ f x | x <- xs ]
filter p xs = [   x | x <- xs, p x ]
```

Examples:

```
map   (7 *) [1 .. 6] = [7, 14, 21, 28, 35, 42]
filter even [1 .. 6] = [2, 4, 6]
```

foldr1

```
foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]      = x
foldr1 (⊗) (x:xs)   = x ⊗ (foldr1 (⊗) xs)
```

foldr1

```
foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]        = x
foldr1 (⊗) (x:xs)     = x ⊗ (foldr1 (⊗) xs)
```

foldr1 (⊗) [x_1, x_2, x_3, x_4, x_5]

foldr1

```
foldr1                :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]        = x
foldr1 (⊗) (x:xs)     = x ⊗ (foldr1 (⊗) xs)
```

foldr1 (⊗) $[x_1, x_2, x_3, x_4, x_5]$

= $x_1 \otimes$ (foldr1 (⊗) $[x_2, x_3, x_4, x_5]$)

foldr1

```
foldr1               :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]       = x
foldr1 (⊗) (x:xs)    = x ⊗ (foldr1 (⊗) xs)
```

foldr1 (⊗) [x_1, x_2, x_3, x_4, x_5]

= x_1 ⊗ (foldr1 (⊗) [x_2, x_3, x_4, x_5])

= x_1 ⊗ (x_2 ⊗ (foldr1 (⊗) [x_3, x_4, x_5]))

foldr1

```
foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]          = x
foldr1 (⊗) (x:xs)       = x ⊗ (foldr1 (⊗) xs)
```

foldr1 $(\otimes)$ $[x_1, x_2, x_3, x_4, x_5]$

$= x_1 \otimes (\text{foldr1 } (\otimes)\ [x_2, x_3, x_4, x_5])$

$= x_1 \otimes (x_2 \otimes (\text{foldr1 } (\otimes)\ [x_3, x_4, x_5]))$

$= x_1 \otimes (x_2 \otimes (x_3 \otimes (\text{foldr1 } (\otimes)\ [x_4, x_5])))$

foldr1

```
foldr1                  :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]          = x
foldr1 (⊗) (x:xs)       = x ⊗ (foldr1 (⊗) xs)
```

foldr1 ($\otimes$) [x_1, x_2, x_3, x_4, x_5]

= x_1 $\otimes$ (foldr1 ($\otimes$) [x_2, x_3, x_4, x_5])

= x_1 $\otimes$ (x_2 $\otimes$ (foldr1 ($\otimes$) [x_3, x_4, x_5]))

= x_1 $\otimes$ (x_2 $\otimes$ (x_3 $\otimes$ (foldr1 ($\otimes$) [x_4, x_5])))

= x_1 $\otimes$ (x_2 $\otimes$ (x_3 $\otimes$ (x_4 $\otimes$ (foldr1 ($\otimes$) [x_5]))))

foldr1

```
foldr1                 :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]         = x
foldr1 (⊗) (x:xs)      = x ⊗ (foldr1 (⊗) xs)
```

foldr1 ($\otimes$) [x_1, x_2, x_3, x_4, x_5]

= $x_1 \otimes$ (foldr1 ($\otimes$) [x_2, x_3, x_4, x_5])

= $x_1 \otimes$ ($x_2 \otimes$ (foldr1 ($\otimes$) [x_3, x_4, x_5]))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ (foldr1 ($\otimes$) [x_4, x_5])))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes$ (foldr1 ($\otimes$) [x_5]))))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes x_5$)))

foldr

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

foldrX

$$
\begin{aligned}
&\mathit{foldrX} && :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldrX}\ (***)\ z\ [\,] && = z \\
&\mathit{foldrX}\ (***)\ z\ (x : xs) && = x\ ***\ (\mathit{foldrX}\ (***)\ z\ xs)
\end{aligned}
$$

foldr

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

foldr ($\otimes$) z [x_1, x_2, x_3, x_4, x_5]

foldr

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

foldr ($\otimes$) z [x_1, x_2, x_3, x_4, x_5]

= x_1 $\otimes$ (foldr ($\otimes$) z [x_2, x_3, x_4, x_5])

foldr

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

foldr (⊗) z [x_1, x_2, x_3, x_4, x_5]

= x_1 ⊗ (foldr (⊗) z [x_2, x_3, x_4, x_5])

= x_1 ⊗ (x_2 ⊗ (foldr (⊗) z [x_3, x_4, x_5]))

foldr

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

foldr ($\otimes$) z [x_1, x_2, x_3, x_4, x_5]

= $x_1 \otimes$ (foldr ($\otimes$) z [x_2, x_3, x_4, x_5])

= $x_1 \otimes$ ($x_2 \otimes$ (foldr ($\otimes$) z [x_3, x_4, x_5]))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ (foldr ($\otimes$) z [x_4, x_5])))

foldr

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

foldr (⊗) z $[x_1, x_2, x_3, x_4, x_5]$

$= x_1 \otimes$ (foldr (⊗) z $[x_2, x_3, x_4, x_5]$)

$= x_1 \otimes (x_2 \otimes$ (foldr (⊗) z $[x_3, x_4, x_5]$))

$= x_1 \otimes (x_2 \otimes (x_3 \otimes$ (foldr (⊗) z $[x_4, x_5]$)))

$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes$ (foldr (⊗) z $[x_5]$))))

foldr

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

foldr (⊗) z [x_1, x_2, x_3, x_4, x_5]

= $x_1 \otimes$ (foldr (⊗) z [x_2, x_3, x_4, x_5])

= $x_1 \otimes$ ($x_2 \otimes$ (foldr (⊗) z [x_3, x_4, x_5]))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ (foldr (⊗) z [x_4, x_5])))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes$ (foldr (⊗) z [x_5]))))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes$ ($x_5 \otimes$ (foldr (⊗) z []))))))

foldr

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

foldr (⊗) z [x_1, x_2, x_3, x_4, x_5]

= $x_1 \otimes$ (foldr (⊗) z [x_2, x_3, x_4, x_5])

= $x_1 \otimes$ ($x_2 \otimes$ (foldr (⊗) z [x_3, x_4, x_5]))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ (foldr (⊗) z [x_4, x_5])))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes$ (foldr (⊗) z [x_5]))))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes$ ($x_5 \otimes$ (foldr (⊗) z [])))))

= $x_1 \otimes$ ($x_2 \otimes$ ($x_3 \otimes$ ($x_4 \otimes$ ($x_5 \otimes$ z))))

List Folding

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

foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 f [x]         = x
foldr1 f (x:xs)      = f x (foldr1 f xs)

foldl               :: (a -> b -> a) -> a -> [b] -> a
foldl f z []         = z
foldl f z (x:xs)     = foldl f (f z x) xs

foldl1              :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)      = foldl f x xs
```

Unfolding Definitions

A simple definition:

limit = 10 ^ 2

Unfolding Definitions

A simple definition:

$limit = 10 \wedge 2$

Expanding this definition:

$4 * (limit + 1)$

Unfolding Definitions

A simple definition:

$\mathit{limit} = 10 \wedge 2$

Expanding this definition:

$4 * (\mathit{limit} + 1)$
$= 4 * ((10 \wedge 2) + 1)$

Unfolding Definitions

A simple definition:

$limit = 10 \wedge 2$

Expanding this definition:

$4 * (limit + 1)$
$= 4 * ((10 \wedge 2) + 1)$
$= \ldots$

Unfolding Definitions

A simple definition:

$limit = 10 \wedge 2$

Expanding this definition:

$4 * (limit + 1)$
$= 4 * ((10 \wedge 2) + 1)$
$= \ldots$

Another definition:

$concat = foldr\ (+\!\!+)\ [\,]$

Unfolding Definitions

A simple definition:

$limit = 10 \wedge 2$

Expanding this definition:

$4 * (limit + 1)$
$= 4 * ((10 \wedge 2) + 1)$
$= \ldots$

Another definition:

$concat = foldr\ (++)\ [\,]$

Expanding this definition:

$concat\ [[1,2,3],[4,5]]$

Unfolding Definitions

A simple definition:

$limit = 10 \wedge 2$

Expanding this definition:

$4 * (limit + 1)$
$= 4 * ((10 \wedge 2) + 1)$
$= \ldots$

Another definition:

$concat = foldr\ (+\!\!+)\ [\,]$

Expanding this definition:

$concat\ [[1,2,3],[4,5]]$
$= (foldr\ (+\!\!+)\ [\,])\ [[1,2,3],[4,5]]$
$= \ldots$

Enumeration Type Definitions

$\textbf{data}\ \mathit{Bool} = \textbf{False} \mid \textbf{True} \qquad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Read}, \mathit{Show})$

$\textbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT} \qquad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Read}, \mathit{Show})$

$\textbf{data}\ \mathit{Suit} = \mathit{Diamonds} \mid \mathit{Hearts} \mid \mathit{Spades} \mid \mathit{Clubs} \quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})$

Pattern matching:

$\mathit{not}\ \textbf{False} = \textbf{True}$
$\mathit{not}\ \textbf{True} = \textbf{False}$

$\mathit{lexicalCombineOrdering} :: \mathit{Ordering} \to \mathit{Ordering} \to \mathit{Ordering}$
$\mathit{lexicalCombineOrdering}\ \mathit{LT}\ \_ = \mathit{LT}$
$\mathit{lexicalCombineOrdering}\ \mathit{EQ}\ x = x$
$\mathit{lexicalCombineOrdering}\ \mathit{GT}\ \_ = \mathit{GT}$

Simple data Type Definitions

$\textbf{data}\ \mathit{Point} = \mathit{Pt}\ \mathit{Int}\ \mathit{Int}\ \ \textbf{deriving}\ (\mathit{Eq})$ —— screen coordinates

$\textbf{data}\ \mathit{Transport} = \mathit{Feet}$
$\quad\quad |\ \mathit{Bike}$
$\quad\quad |\ \mathit{Train}\ \mathit{Int}$ —— price in cent

This defines at the same time **data constructors**:

$\mathit{Pt} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Point}$

$\mathit{Feet} :: \mathit{Transport}$
$\mathit{Bike} :: \mathit{Transport}$
$\mathit{Train} :: \mathit{Int} \rightarrow \mathit{Transport}$

Pattern matching:

$\mathit{addPt}\ (\mathit{Pt}\ x1\ y1)\ (\mathit{Pt}\ x2\ y2) = \mathit{Pt}\ (x1 + x2)\ (y1 + y2)$

$\mathit{cost}\ \mathit{Feet} = 0$
$\mathit{cost}\ \mathit{Bike} = 0$
$\mathit{cost}\ (\mathit{Train}\ \mathit{Int}) = \mathit{Int}$

Simple Polymorphic data Type Definitions

The prelude **type constructors** *Maybe*, *Either*, *Complex* are defined as follows:

$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a \qquad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Read}, \mathit{Show})$
$\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$

$\textbf{data}\ \mathit{Complex}\ r = r\ {:}{+}\ r\ \ \textbf{deriving}\ (\mathit{Eq}, \mathit{Read}, \mathit{Show})$

This defines at the same time **data constructors**:

$\mathit{Nothing} :: \mathit{Maybe}\ a$
$\mathit{Just} :: a \to \mathit{Maybe}\ a$

$\mathit{Left} :: a \to \mathit{Either}\ a\ b$
$\mathit{Right} :: b \to \mathit{Either}\ a\ b$

$({:}{+}) :: r \to r \to \mathit{Complex}\ r$