Functional Programming

Wolfram Kahl

kahl@mcmaster.ca

Department of Computing and Software
McMaster University
What Kinds of Programming Languages are There?

Programming Languages

- Imperative
  - C
  - Pascal
  - FORTRAN
  - COBOL
  - Modula-2

- Declarative
  - Functional
    - Haskell
    - OCaml
    - ML
    - Scheme
    - LISP
  - Logic
    - Prolog
    - Mercury

- Object-oriented
  - C++
  - Oberon-2
  - Java
  - Smalltalk

- Others
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
Important Functional Programming Languages

- Pure
  - Haskell
  - Clean
- Impure
  - Standard ML
  - OCaml
  - LISP, Scheme
  - APL, J
  - Erlang
Important Functional Programming Languages

- pure
  - Haskell
  - Clean

- impure
  - Standard ML
  - OCaml
  - LISP, Scheme
  - APL, J
  - Erlang
Important Functional Programming Languages

- **pure**
  - Haskell
  - Clean

- **impure**
  - statically typed
    - Standard ML
    - OCaml
  - dynamically typed
    - LISP, Scheme
    - APL, J
    - Erlang
Important Functional Programming Languages

- **Pure, statically typed**
  - Haskell
  - Clean

- **Impure**
  - Statically typed
  - Standard ML
  - OCaml

- **Dynamically typed**
  - LISP, Scheme
  - APL, J
  - Erlang
Important Functional Programming Languages

- pure, statically typed
  - Haskell
  - Clean
- impure
  - statically typed
    - Standard ML
    - OCaml
  - dynamically typed
    - Erlang
Important Functional Programming Languages

- pure, statically typed
  - Haskell
  - Clean
- impure
  - statically typed
    - Standard ML
    - OCaml
  - dynamically typed
    - LISP, Scheme
    - APL, J
    - Erlang
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 **is expression evaluation**
  
  — *(for the time being)*
Important Points

• Execution of Haskell programs is expression evaluation
  — (for the time being)

• Defining functions in Haskell
Important Points

• Execution of Haskell programs is expression evaluation
  — (for the time being)

• Defining functions in Haskell is more like defining functions in mathematics
Important Points

• Execution of Haskell programs is 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 is 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 is 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 is 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 is 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 is 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 Sets, or FiniteMaps 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:

\[4\times(5+6)−2\]
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.

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

Expression evaluation proceeds by applying rules to subexpressions:

\[ 4 \times (5+6) - 2 \quad [\text{subtraction & mult. impossible}] \]
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:

\[ 4 \times (5+6) - 2 \]

\[ = 4 \times 11 - 2 \] [subtraction & mult. impossible]

\[ = (addition) \]

4*11-2
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:

\[
4 \times (5 + 6) - 2 = 4 \times 11 - 2
\]

\[
\text{[subtraction & mult. impossible]} \quad \text{(addition)} \quad \text{[subtraction impossible]}
\]
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:

\[
4 \times (5+6) - 2 = 4 \times 11 - 2 = 44 - 2
\]

[+subtraction & mult. impossible] (addition) [subtraction impossible] (multiplication)
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:

\[
4 \times (5 + 6) - 2
\]

[subtraction & mult. impossible]

\[
= (addition)
\]

\[
4 \times 11 - 2
\]

[subtraction impossible]

\[
= (multiplication)
\]

\[
44 - 2
\]

= (subtraction)

\[
42
\]
Simple Expression Evaluation — Explanation
Simple Expression Evaluation — Explanation

• Arguments to a function or operation are evaluated only when needed.
Simple Expression Evaluation — Explanation

- Arguments to a function 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 function 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 function 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 function 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 function 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 function 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 $f \text{ undefined} = \text{undefined}$. 
Simple Expression Evaluation — Explanation

- Arguments to a function 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 $f \text{ undefined} = \text{undefined}$.
  - **Constant functions** are non-strict:
Simple Expression Evaluation — Explanation

- Arguments to a function 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 \( f \text{ undefined} = \text{undefined} \).
  - Constant functions are non-strict: \( (\text{const 5}) \text{ undefined} = 5 \)
Simple Expression Evaluation — Explanation

- Arguments to a function 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: $(\text{const } 5)$ undefined = 5
  - Checking a list for emptyness is **strict**:
Simple Expression Evaluation — Explanation

- Arguments to a function 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 $f \text{ undefined} = \text{ undefined}$.
  - **Constant functions** are non-strict: $(\text{const } 5) \text{ undefined} = 5$
  - Checking a list for emptiness is strict: $\text{null undefined} = \text{ undefined}$
Simple Expression Evaluation — Explanation

• Arguments to a function 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**: \((\text{const } 5) \) undefined = 5

  – Checking a list for emptiness is **strict**: \( \text{null undefined} = \text{undefined} \)

  – List construction is **non-strict**: 
Simple Expression Evaluation — Explanation

• Arguments to a function 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: $(\text{const 5}) \ undefined = 5$
  
  – Checking a list for emptiness is strict: $\text{null} \ undefined = \ undefined$
  
  – List construction is non-strict: $\text{null} (\ undefined : \ undefined) = \text{False}$
Simple Expression Evaluation — Explanation

- Arguments to a function 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: \((\text{const 5})\) undefined = 5
  - Checking a list for emptiness is strict: \( \text{null} \) undefined = undefined
  - List construction is non-strict: \( \text{null} (\text{undefined} : \text{undefined}) \) = False
  - Standard arithmetic operators are strict in both arguments:
Simple Expression Evaluation — Explanation

• Arguments to a function 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: \( (\text{const 5}) \ undefined = 5 \)

  – Checking a list for emptiness is strict: \( \text{null} \ undefined = \text{undefined} \)

  – List construction is non-strict: \( \text{null} (\text{undefined} : \text{undefined}) = \text{False} \)

  – Standard arithmetic operators are strict in both arguments:

    \[ 0 \times \text{undefined} = \text{undefined} \]
Unfolding Definitions

Assume the following definitions to be in scope:

\[ \text{answer} = 42 \]
\[ \text{magic} = 7 \]

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

\[
\text{Prelude> (answer - 1) \ast (magic \ast answer - 23)}
\]
\[ 11111 \]
Unfolding Definitions

Assume the following definitions to be in scope:

\[\text{answer} = 42\]
\[\text{magic} = 7\]

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

\[
\text{Prelude}> (\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23)
\]

11111

\[
(\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23)
\]
Unfolding Definitions

Assume the following definitions to be in scope:

\[
\begin{align*}
\text{answer} &= 42 \\
\text{magic} &= 7
\end{align*}
\]

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

\[
\text{Prelude}\> (\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23)
\]

\[
\begin{align*}
(\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23) \\
= (42 - 1) \times (\text{magic} \times 42 - 23) \\
\end{align*}
\]
Unfolding Definitions

Assume the following definitions to be in scope:

\[
\begin{align*}
\text{answer} &= 42 \\
\text{magic} &= 7
\end{align*}
\]

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

Prelude> (answer - 1) * (magic * answer - 23)
111111

\[
(\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23) = (42 - 1) \times (\text{magic} \times 42 - 23) \quad (\text{answer})
\]

\[
= 41 \times (\text{magic} \times 42 - 23) \quad (\text{subtraction})
\]
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)
111111

\[(answer - 1) \times (magic \times answer - 23)\]
\[= (42 - 1) \times (magic \times 42 - 23)\] (answer)
\[= 41 \times (magic \times 42 - 23)\] (subtraction)
\[= 41 \times (7 \times 42 - 23)\] (magic)
Unfolding Definitions

Assume the following definitions to be in scope:

\[
\text{answer} = 42 \\
\text{magic} = 7
\]

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

Prelude> (answer - 1) * (magic * answer - 23)
111111

\[
\begin{align*}
\text{(answer - 1) * (magic * answer - 23)} \\
= (42 - 1) * (7 * 42 - 23) & \quad \text{(answer)} \\
= 41 * (7 * 294 - 23) & \quad \text{(subtraction)} \\
= 41 * (294 - 23) & \quad \text{(magic)} \\
= 41 * (294 - 23) & \quad \text{(multiplication)}
\end{align*}
\]
Unfolding Definitions

Assume the following definitions to be in scope:

\[ \text{answer} = 42 \]
\[ \text{magic} = 7 \]

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

Prelude> \((\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23)\)

\[
(\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23) = (42 - 1) \times (\text{magic} \times 42 - 23) \\
= 41 \times (\text{magic} \times 42 - 23) \\
= 41 \times (7 \times 42 - 23) \\
= 41 \times (294 - 23) \\
= 41 \times 271
\]
Unfolding Definitions

Assume the following definitions to be in scope:

\[ \text{answer} = 42 \]
\[ \text{magic} = 7 \]

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

Prelude> \((\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23)\)

\[
(\text{answer} - 1) \times (\text{magic} \times \text{answer} - 23) \\
= (42 - 1) \times (\text{magic} \times 42 - 23) \\
= 41 \times (\text{magic} \times 42 - 23) \\
= 41 \times (7 \times 42 - 23) \\
= 41 \times (294 - 23) \\
= 41 \times 271 \\
= 11111
\]
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 the 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:

\[
\text{if } \text{condition} \text{ then expression1 else expression2}
\]

- If the condition evaluates to \text{True}, the conditional expression evaluates to the value of \text{expression1}.
- If the condition evaluates to \text{False}, the conditional expression evaluates to the value of \text{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:

\[
\text{if } \text{condition} \text{ then } \text{expression1} \text{ else } \text{expression2}
\]

- If the condition evaluates to \textbf{True}, the conditional expression evaluates to the value of \textit{expression1}.

- If the condition evaluates to \textbf{False}, the conditional expression evaluates to the value of \textit{expression2}.

\textit{Therefore:} “if _ then _ else” is \textbf{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

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\text{fact } 3
\]
Expanding Function Definitions

fact :: Integer -> Integer

\[
\text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\text{fact 3} \\
= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1)
\]
Expanding Function Definitions

```haskell
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)

\[
\begin{align*}
\text{fact 3} \\
= & \text{if 3 == 0 then 1 else } 3 \times \text{fact (3-1)} \\
= & \text{if False then 1 else } 3 \times \text{fact (3-1)} \\
= & 3 \times \text{fact (3-1)}
\end{align*}
\]
Expanding Function Definitions

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

\begin{align*}
\text{fact 3} \\
&= \text{if 3 == 0 then 1 else 3 * fact (3-1)} \\
&= \text{if False then 1 else 3 * fact (3-1)} \\
&= 3 * \text{fact (3-1)} \\
&= 3 * \text{if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)}
\end{align*}
Expanding Function Definitions

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\begin{align*}
\text{fact } 3 \\
= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
= \text{if False then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
= 3 \times \text{fact } (3-1) \\
= 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
= 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1)
\end{align*}
\]
Expanding Function Definitions

fact :: Integer -> Integer

\[
\text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\begin{align*}
\text{fact } 3 &= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= \text{if } \text{False} \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= 3 \times \text{fact } (3-1) \\
&= 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
&= 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times \text{if } \text{False} \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1)
\end{align*}
\]
Expanding Function Definitions

fact :: Integer -> Integer

fact \( n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \)

\[
\begin{align*}
\text{fact 3} &= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= \text{if False then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= 3 \times \text{fact } (3-1) \\
&= 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
&= 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times \text{if False then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times 2 \times \text{fact } (2-1)
\end{align*}
\]
Expanding Function Definitions

\[ \text{fact} :: \text{Integer} \rightarrow \text{Integer} \]
\[ \text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \]

\[
\begin{align*}
\text{fact } 3 \\
&= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= \text{if } \text{False} \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= 3 \times \text{fact } (3-1) \\
&= 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
&= 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times \text{if } \text{False} \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times 2 \times \text{fact } (2-1) \\
&= 3 \times 2 \times \text{if } (2-1) == 0 \text{ then } 1 \text{ else } (2-1) \times \text{fact } ((2-1)-1)
\end{align*}
\]
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

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\begin{align*}
\text{fact } 3 &= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
&= 3 \times \text{fact } (3-1) \\
&= 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
&= 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times \text{if } \text{False} \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
&= 3 \times 2 \times \text{fact } (2-1) \\
&= 3 \times 2 \times \text{if } (2-1) == 0 \text{ then } 1 \text{ else } (2-1) \times \text{fact } ((2-1)-1) \\
&= 3 \times 2 \times \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1) \\
&= 3 \times 2 \times \text{if } \text{False} \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1)
\end{align*}
\]
Expanding Function Definitions

\[ \text{fact} :: \text{Integer} \to \text{Integer} \]
\[ \text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \]

\[
\begin{align*}
\text{fact 3} &= \text{if 3 == 0 then 1 else 3 * fact (3-1)} \\
&= \text{if False then 1 else 3 * fact (3-1)} \\
&= 3 \times \text{fact (3-1)} \\
&= 3 \times \text{if (3-1) == 0 then 1 else (3-1) * fact ((3-1)-1)} \\
&= 3 \times \text{if 2 == 0 then 1 else 2 * fact (2-1)} \\
&= 3 \times \text{if False then 1 else 2 * fact (2-1)} \\
&= 3 \times 2 \times \text{fact (2-1)} \\
&= 3 \times 2 \times \text{if (2-1) == 0 then 1 else (2-1) * fact ((2-1)-1)} \\
&= 3 \times 2 \times \text{if 1 == 0 then 1 else 1 * fact (1-1)} \\
&= 3 \times 2 \times \text{if False then 1 else 1 * fact (1-1)} \\
&= 3 \times 2 \times 1 \times \text{fact (1-1)}
\end{align*}
\]
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

\[ \text{fact} :: \text{Integer} \rightarrow \text{Integer} \]
\[ \text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \]

\[
\text{fact } 3 \\
= \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
= \text{if False then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
= 3 \times \text{fact } (3-1) \\
= 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
= 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
= 3 \times \text{if False then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
= 3 \times 2 \times \text{fact } (2-1) \\
= 3 \times 2 \times \text{if } (2-1) == 0 \text{ then } 1 \text{ else } (2-1) \times \text{fact } ((2-1)-1) \\
= 3 \times 2 \times \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1) \\
= 3 \times 2 \times \text{if False then } 1 \text{ else } 1 \times \text{fact } (1-1) \\
= 3 \times 2 \times 1 \times \text{fact } (1-1) \\
= 3 \times 2 \times 1 \times \text{if } (1-1) == 0 \text{ then } 1 \text{ else } (1-1) \times \text{fact } ((1-1)-1) \\
= 3 \times 2 \times 1 \times \text{if } 0 == 0 \text{ then } 1 \text{ else } 0 \times \text{fact } (0-1) \\
= 3 \times 2 \times 1 \times \text{if True then } 1 \text{ else } 0 \times \text{fact } (0-1) \\
\]
Expanding Function Definitions

fact :: Integer -> Integer

\[
\text{fact } n = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times \text{fact}(n-1) & \text{else} 
\end{cases}
\]

\[
\text{fact } 3 = \begin{cases} 
1 & \text{if } 3 = 0 \\
3 \times \text{fact}(3-1) & \text{else} 
\end{cases} = 3 \times \text{fact}(2)
\]

\[
= \begin{cases} 
1 & \text{if } 2 = 0 \\
2 \times \text{fact}(2-1) & \text{else} 
\end{cases} = 2 \times \text{fact}(1) = 2 \times 1
\]

\[
= \begin{cases} 
1 & \text{if } 1 = 0 \\
1 \times \text{fact}(1-1) & \text{else} 
\end{cases} = 1 \times 1 = 1
\]

\[
= \begin{cases} 
1 & \text{if } 0 = 0 \\
0 \times \text{fact}(0-1) & \text{else} 
\end{cases} = 0 \times 1 = 0
\]

\[
= 3 \times 2 \times 1 \times 1
\]
Expanding Function Definitions

\textbf{fact} :: \texttt{Integer -> Integer}
\texttt{fact \ n = if \ n == 0 then 1 else \ n * fact \ (n-1)}

\begin{verbatim}
  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
\end{verbatim}
Expanding Function Definitions

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fact } n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)
\]

\[
\begin{align*}
\text{fact } 3 \\
= & \text{if } 3 == 0 \text{ then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
= & \text{if False then } 1 \text{ else } 3 \times \text{fact } (3-1) \\
= & 3 \times \text{fact } (3-1) \\
= & 3 \times \text{if } (3-1) == 0 \text{ then } 1 \text{ else } (3-1) \times \text{fact } ((3-1)-1) \\
= & 3 \times \text{if } 2 == 0 \text{ then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
= & 3 \times \text{if False then } 1 \text{ else } 2 \times \text{fact } (2-1) \\
= & 3 \times 2 \times \text{fact } (2-1) \\
= & 3 \times 2 \times \text{if } (2-1) == 0 \text{ then } 1 \text{ else } (2-1) \times \text{fact } ((2-1)-1) \\
= & 3 \times 2 \times \text{if } 1 == 0 \text{ then } 1 \text{ else } 1 \times \text{fact } (1-1) \\
= & 3 \times 2 \times \text{if False then } 1 \text{ else } 1 \times \text{fact } (1-1) \\
= & 3 \times 2 \times 1 \times \text{fact } (1-1) \\
= & 3 \times 2 \times 1 \times \text{if } (1-1) == 0 \text{ then } 1 \text{ else } (1-1) \times \text{fact } ((1-1)-1) \\
= & 3 \times 2 \times 1 \times \text{if } 0 == 0 \text{ then } 1 \text{ else } 0 \times \text{fact } (0-1) \\
= & 3 \times 2 \times 1 \times \text{if True then } 1 \text{ else } 0 \times \text{fact } (0-1) \\
= & 3 \times 2 \times 1 \times 1 \\
= & 3 \times 2 \times 1 \\
= & 3 \times 2
\end{align*}
\]
Expanding Function Definitions

fact :: Integer -> Integer

fact \ n = if \ n == 0 \ then \ 1 \ else \ n \ * \ fact \ (n-1)

---

\begin{align*}
\text{fact 3} &= \text{if } 3 = 0 \ \text{then } 1 \ \text{else } 3 \ * \ \text{fact } (3-1) \\
&= \text{if } \text{False} \ \text{then } 1 \ \text{else } 3 \ * \ \text{fact } (3-1) \\
&= 3 \ * \ \text{fact } (3-1) \\
&= 3 \ * \ \text{if } (3-1) = 0 \ \text{then } 1 \ \text{else } (3-1) \ * \ \text{fact } ((3-1)-1) \\
&= 3 \ * \ \text{if } 2 = 0 \ \text{then } 1 \ \text{else } 2 \ * \ \text{fact } (2-1) \\
&= 3 \ * \ \text{if } \text{False} \ \text{then } 1 \ \text{else } 2 \ * \ \text{fact } (2-1) \\
&= 3 \ * \ 2 \ * \ \text{fact } (2-1) \\
&= 3 \ * \ 2 \ * \ \text{if } (2-1) = 0 \ \text{then } 1 \ \text{else } (2-1) \ * \ \text{fact } ((2-1)-1) \\
&= 3 \ * \ 2 \ * \ \text{if } 1 = 0 \ \text{then } 1 \ \text{else } 1 \ * \ \text{fact } (1-1) \\
&= 3 \ * \ 2 \ * \ \text{if } \text{False} \ \text{then } 1 \ \text{else } 1 \ * \ \text{fact } (1-1) \\
&= 3 \ * \ 2 \ * \ 1 \ * \ \text{fact } (1-1) \\
&= 3 \ * \ 2 \ * \ 1 \ * \ \text{if } (1-1) = 0 \ \text{then } 1 \ \text{else } (1-1) \ * \ \text{fact } ((1-1)-1) \\
&= 3 \ * \ 2 \ * \ 1 \ * \ \text{if } 0 = 0 \ \text{then } 1 \ \text{else } 0 \ * \ \text{fact } (0-1) \\
&= 3 \ * \ 2 \ * \ 1 \ * \ \text{if } \text{True} \ \text{then } 1 \ \text{else } 0 \ * \ \text{fact } (0-1) \\
&= 3 \ * \ 2 \ * \ 1 \ * \ 1 \\
&= 3 \ * \ 2 \ * \ 1 \\
&= 3 \ * \ 2 \\
&= 6
\end{align*}
Matching Function Definitions

\[
\text{fact} :: \text{Integer} \to \text{Integer} \\
\text{fact} \ 0 = 1 \\
\text{fact} \ n = n \times \text{fact} \ (n-1)
\]
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)  \hspace{1cm} (fact n)
Matching Function Definitions

\[ \text{fact} :: \text{Integer} \to \text{Integer} \]
\[ \text{fact} \; 0 = 1 \]
\[ \text{fact} \; n = n \times \text{fact} \; (n-1) \]

\[
\begin{align*}
\text{fact} \; 3 &= 3 \times \text{fact} \; (3-1) \\
&= 3 \times \text{fact} \; 2 \\
\end{align*}
\]

(determining which \text{fact} rule matches)
Matching Function Definitions

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

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

(determining which fact rule matches)
Matching Function Definitions

\[\text{fact} :: \text{Integer} \rightarrow \text{Integer}\]
\[\text{fact} \ 0 = 1\]
\[\text{fact} \ n = n \times \text{fact} \ (n-1)\]

\[
\begin{align*}
\text{fact} \ 3 &= 3 \times \text{fact} \ (3-1) \\
&= 3 \times \text{fact} \ 2 \\
&= 3 \times (2 \times \text{fact} \ (2-1)) \\
&= 3 \times (2 \times \text{fact} \ 1)
\end{align*}
\]

(determining which \text{fact} rule matches)
Matching Function Definitions

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fact} \ 0 = 1 \\
\text{fact} \ n = n \ast \text{fact} \ (n-1)
\]

\[
\begin{align*}
\text{fact} \ 3 &= 3 \ast \text{fact} \ (3-1) \\
&= 3 \ast \text{fact} \ 2 \quad \text{(fact \ n)} \\
&= 3 \ast (2 \ast \text{fact} \ (2-1)) \quad \text{(determining which fact rule matches)} \\
&= 3 \ast (2 \ast \text{fact} \ 1) \quad \text{(fact \ n)} \\
&= 3 \ast (2 \ast (1 \ast \text{fact} \ (1-1))) \quad \text{(determining which fact rule matches)} \\
&= 3 \ast (2 \ast (1 \ast \text{fact} \ 0)) \quad \text{(fact \ n)}
\end{align*}
\]
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

```haskell
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

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer}
\]
\[
\text{fact} \ 0 = 1
\]
\[
\text{fact} \ n = n \times \text{fact} \ (n-1)
\]

\[
\begin{align*}
\text{fact} \ 3 &= 3 \times \text{fact} \ (3-1) \\
&= 3 \times \text{fact} \ 2 \\
&= 3 \times (2 \times \text{fact} \ (2-1)) \\
&= 3 \times (2 \times \text{fact} \ 1) \\
&= 3 \times (2 \times (1 \times \text{fact} \ (1-1))) \\
&= 3 \times (2 \times (1 \times \text{fact} \ 0)) \\
&= 3 \times (2 \times (1 \times 1)) \\
&= 3 \times (2 \times 1)
\end{align*}
\]

(determining which \text{fact} rule matches)

(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)
= 3 * fact 2
= 3 * (2 * fact (2-1))
= 3 * (2 * fact 1)
= 3 * (2 * (1 * fact (1-1)))
= 3 * (2 * (1 * fact 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
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):

  \[
  \begin{align*}
  [1 \, .\, . \, 10] & = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \\
  [1, 3 \, .\, . \, 10] & = [1, 3, 5, 7, 9] \\
  [1, 3 \, .\, . \, 11] & = [1, 3, 5, 7, 9, 11] \\
  [11 \, .\, . \, 1] & = [] \\
  [1 \, .\, . \, ] & = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots] \quad \text{-- infinite list} \\
  [1, 3 \, .\, . \, ] & = [1, 3, 5, 7, 9, 11, \ldots] \quad \text{-- infinite list}
  \end{align*}
  \]
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 **constructed** 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 **constructed** from a **head** x and a **tail** xs (read: “xes”)


List Construction

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

- either the **empty list**: \([\ ]\),

- or **non-empty**, and **constructed** from a **head** \(x\) and a **tail** \(xs\) (read: “xes”)

\[
x : \text{xs} \quad \text{— read: “} \times \text{cons xes”}.
\]
List Construction

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

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

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

  `x : xs` — read: “`x cons xes`”.

“:” 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 **constructed** from a **head** `x` and a **tail** `xs` (read: “xes”)

  \[ x : xs \rightarrow \text{read: “x cons xes”}. \]

“::” 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 **constructed** from a **head** x and a **tail** xs (read: “xes”)

\[
\text{x : xs} \quad \text{— read: “x cons xes”}.
\]

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

\[
3 : [ ] \quad = \quad [3]
\]
List Construction

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

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

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

  `x : xs` — read: “`x cons xes`”.

“`:`” 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 **constructed from a head** `x` and a **tail** `xs` (read: “xes”)

\[ x : xs \quad \text{— read: “}x \text{ cons } xes\text{”}. \]

“::” 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 **constructed** from a **head** `x` and a **tail** `xs` (read: “xes”)

  \[ x : \ x s \quad \text{— read: “} x \text{ cons xes”} \].

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

\[
\begin{align*}
3 & : [ ] = [3] \\
2 & : [3] = [2, 3] \\
1 & : [2, 3]
\end{align*}
\]
List Construction

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

- either the **empty list:** \([\ ]\),

- or **non-empty**, and **constructed** from a **head** \(x\) and a **tail** \(xs\) (read: “xes”)

\[ x : xs \quad \text{— read: “} x \text{ cons } x es\text{”} \]

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

\[
\begin{align*}
1 : [2, 3] & = [1, 2, 3]
\end{align*}
\]
List Construction

Display and enumeration lists are *syntactic sugar*: A list is
– either the **empty list**: `[ ]`,

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

\[ x : xs \quad \text{– read: “} x \text{ cons } xes \text{”} \]

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

\[
\begin{align*}
3 : [ ] & = [3] \\
1 : [2, 3] & = [1, 2, 3]
\end{align*}
\]

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 **constructed** from a **head** `x` and a **tail** `xs` (read: “xes”)
  \[ x : xs \rightarrow \text{read: "} x \text{ cons } xes\text{"} \].

“::” 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 **constructed** from a **head** `x` and a **tail** `xs` (read: "xes")

  `x : xs` — read: "`x cons xes`".

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

\[
\begin{align*}
3 & : [] & = & [3] \\
1 & : [2, 3] & = & [1, 2, 3]
\end{align*}
\]

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 **constructed** from a **head** \(x\) and a **tail** \(xs\) (read: “\(xes\)”)

\(x : xs\) — read: “\(x\ cons\ xes\)”.

“\(:\)” is used as **infix list constructor**:

\[
\begin{align*}
3 & : [\ ] \quad = \quad [3] \\
2 & : [3] \quad = \quad [2, 3] \\
1 & : [2, 3] \quad = \quad [1, 2, 3]
\end{align*}
\]

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 **constructed** from a **head** \(x\) and a **tail** \(xs\) (read: “\(xes\)”) \(\ x \ : \ xs \ — \ read: \ “x \ cons \ xes”.\)

“\(\cdot\)” is used as *infix list constructor*:

\[
\begin{align*}
3 \ : \ [\,] & = \ [3] \\
2 \ : \ [3] & = \ [2, \ 3] \\
1 \ : \ [2, \ 3] & = \ [1, \ 2, \ 3]
\end{align*}
\]

As an infix operator, “\(\cdot\)” **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 **constructed** from a **head** \( x \) and a **tail** \( x \cdot s \) (read: “\( x \cdot s \)”)

\[
x : x \cdot s \quad \text{— read: “} x \ \text{cons} \ x \cdot s \text{”}.
\]

“\( : \)” is used as **infix list constructor**:

\[
\begin{align*}
1 : [2, 3] & = [1, 2, 3]
\end{align*}
\]

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

\[
x : y : y \cdot s = x : (y : y \cdot s)
\]

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 circumstances.
Cons is Not Associative

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

However, “:” is **not** associative:
Cons is Not Associative

The convention that “:” \textit{associates to the right} allows to save parentheses in certain circumstances.

However, “:” is \textbf{not} associative:

- A \textit{list of integers}:
  
  \begin{verbatim}
  1 : (2 : [3, 4])
  \end{verbatim}
Cons is Not Associative

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

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 circumstances.

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 circumstances.

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 circumstances.

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 circumstances.

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 “:” \textit{associates to the right} allows to save parentheses in certain circumstances.

However, “:” is \textbf{not} associative:

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

- (1 : 2) : [3, 4] is \textbf{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 circumstances.

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 “:” \textit{associates to the right} allows to save parentheses in certain circumstances.

However, “:” is \textbf{not} associative:

- \textbf{A list of integers}:
  \[ 1 : (2 : [3, 4]) = 1 : 2 : [3, 4] = [1, 2, 3, 4] \]
- \( (1 : 2) : [3, 4] \) is \textbf{nonsense}, since 2 is not a list!

- \textbf{A list of lists of integers}:
  \[ [2] : [[3, 4, 5], [6, 7]] = [[2], [3, 4, 5], [6, 7]] \]
- \textbf{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 circumstances.

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 circumstances.

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 circumstances.

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 circumstances.

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:

\[
[ \text{term} \mid \text{generator} \{ , \text{generator_or_constraint} \}^* ]
\]
List Comprehensions

General shape:

\[
\left[ \text{term} \mid \text{generator} \ {\{, \text{generator}\_or\_constraint\}\}^* \right]
\]

Examples:

\[
\left[ n \cdot n \mid n \leftarrow [1..5] \right]
\]
List Comprehensions

General shape:

\[
\left[ \text{term} \mid \text{generator} \{ , \text{generator_or_constraint} \}^* \right]
\]

Examples:

\[
\left[ n \times n \mid n \leftarrow [1 .. 5] \right] = [1, 4, 9, 16, 25]
\]
List Comprehensions

General shape:

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

Examples:

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

\[
[ n* n \mid n \leftarrow [1..10], \text{even } n ]
\]
List Comprehensions

General shape:

\[
[ \text{term} \mid \text{generator} \{ , \text{generator_or_constraint} \}^* ]
\]

Examples:

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

\[
[ n \times n \mid n \leftarrow [1..10], \text{even } n ] = [4, 16, 36, 64, 100]
\]
List Comprehensions

General shape:

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

Examples:

\[
[ n \times n \mid n \leftarrow [1 \ldots 5] ] = [1,4,9,16,25]
\]

\[
[ n \times n \mid n \leftarrow [1 \ldots 10], \text{even n} ] = [4,16,36,64,100]
\]

\[
[ m \times n \mid m \leftarrow [1,3,5], n \leftarrow [2,4,6] ]
\]
List Comprehensions

General shape:

\[
[ \text{term} \mid \text{generator} \ {, \ generator\_or\_constraint \}* ]
\]

Examples:

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

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

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

General shape:

\[ [ \text{term} \mid \text{generator} \{ \text{, generator_or_constraint} \}^* ] \]

Examples:

\[ [ n \times n \mid n \leftarrow [1..5] ] = [1,4,9,16,25] \]
\[ [ n \times n \mid n \leftarrow [1..10], \text{even } n ] = [4,16,36,64,100] \]
\[ [ m \times n \mid m \leftarrow [1,3,5], \text{even } n \leftarrow [2,4,6] ] = [2,4,6,6,12,18,10,20,30] \]

Note:

- The left generator “generates slower”.
List Comprehensions

General shape:

\[
\begin{array}{l}
\left[ \text{term} \mid \text{generator} \ (\text{generator_or_constraint})^* \right]
\end{array}
\]

Examples:

\[
\begin{array}{l}
\left[ n \ast n \mid n \leftarrow [1 .. 5] \right] = [1, 4, 9, 16, 25] \\
\left[ n \ast n \mid n \leftarrow [1 .. 10], \text{even} \ n \right] = [4, 16, 36, 64, 100] \\
\left[ m \ast n \mid m \leftarrow [1, 3, 5], n \leftarrow [2, 4, 6] \right] = [2, 4, 6, 6, 12, 18, 10, 20, 30]
\end{array}
\]

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 “←”
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

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>truth values</td>
<td>False, True</td>
</tr>
<tr>
<td><strong>Char</strong></td>
<td>“Unicode” characters</td>
<td>(in GHC: ISO-10646)</td>
</tr>
<tr>
<td><strong>Integer</strong></td>
<td>integers</td>
<td>arbitrary precision</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>“machine integers”</td>
<td>≥ 32 bits</td>
</tr>
<tr>
<td><strong>Float</strong></td>
<td>real floating point</td>
<td>single precision</td>
</tr>
<tr>
<td><strong>Double</strong></td>
<td>real floating point</td>
<td>double precision</td>
</tr>
<tr>
<td><strong>Complex Float</strong></td>
<td>complex floating point</td>
<td>single precision</td>
</tr>
<tr>
<td><strong>Complex Double</strong></td>
<td>complex floating point</td>
<td>double precision</td>
</tr>
</tbody>
</table>
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 $t$.

```haskell
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 \( t \).

\[
\text{answer :: Integer} \\
\text{answer = 42} \\
\text{limit :: Int} \\
\text{limit = 100}
\]

Then:

* \([1, 2, 3, \text{answer}] :: ???*
List Types

If \( t \) is a type, then the list type \([t]\) is the type of lists with elements of type \( t \).

\[
\text{answer} :: \text{Integer} \\
\text{answer} = 42 \\
\text{limit} :: \text{Int} \\
\text{limit} = 100
\]

Then:

- \([1, 2, 3, \text{answer}] :: [\text{Integer}]\)
List Types

If \( t \) is a type, then the list type \([t]\) is the type of lists with elements of type \( t \).

```haskell
answer :: Integer
answer = 42

limit :: Int
limit = 100
```

Then:

- \([ 1, 2, 3, \text{answer} ] :: [\text{Integer}]\)
- \([ 1 .. \text{limit} ] :: ???\)
List Types

If \( t \) is a type, then the list type \([t]\) is the type of lists with elements of type \( t \).

```haskell
answer :: Integer
answer = 42

limit :: Int
limit = 100

Then:

- \([1, 2, 3, \text{answer}]\) :: [Integer]
- \([1..\text{limit}]\) :: [Int]
```
List Types

If \( t \) is a type, then the list type \([t]\) is the type of lists with elements of type \( t \).

```plaintext
answer :: Integer
answer = 42
limit :: Int
limit = 100
```

Then:

- \([1, 2, 3, \text{answer}]\) :: [Integer]
- \([1 .. \text{limit}]\) :: [Int]
- \([ [1 .. \text{limit}], [2 .. \text{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:

- \([1, 2, 3, \text{answer}] :: [\text{Integer}]\)
- \([1 .. \text{limit}] :: [\text{Int}]\)
- \([ [1 .. \text{limit}], [2 .. \text{limit}] ] :: [[[\text{Int}]]]\)
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:

• $[1, 2, 3, \text{answer}] :: [\text{Integer}]$
• $[1 .. \text{limit}] :: [\text{Int}]$
• $[[1 .. \text{limit}}, [2 .. \text{limit}]] :: [[[\text{Int}]]$
• $[\text{h'}, \text{e'}, \text{l'}, \text{l'}, \text{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:

- \([1, 2, 3, \text{answer}] :: [\text{Integer}]\)
- \([1 .. \text{limit}] :: [\text{Int}]\)
- \([ [1 .. \text{limit}], [2 .. \text{limit}] ] :: [[\text{Int}]]\)
- \(['h', 'e', 'l', 'l', 'o'] :: [\text{Char}]\)
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:

- \([1, 2, 3, \text{answer}]\) :: \([\text{Integer}]\)
- \([1 \ldots \text{limit}]\) :: \([\text{Int}]\)
- \([1 \ldots \text{limit}] , [2 \ldots \text{limit}]\) :: \([[[\text{Int}]]]\)
- \(['h', 'e', 'l', 'l', 'o']\) :: \([\text{Char}]\)
- "hello" :: ???
List Types

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

```haskell
answer :: Integer
answer = 42

limit :: Int
limit = 100

Then:

- $[1, 2, 3, \text{answer}] :: [\text{Integer}]$
- $[1 \ldots \text{limit}] :: [\text{Int}]$
- $[[1 \ldots \text{limit}}, [2 \ldots \text{limit}]] :: [[[\text{Int}]]]$
- $\text{['h', 'e', 'l', 'l', 'o']} :: [\text{Char}]$
- "$\text{hello}" :: [\text{Char}]$
```
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:

- $[1, 2, 3, \text{answer}] :: [\text{Integer}]$
- $[1..\text{limit}] :: [\text{Int}]$
- $[[1..\text{limit}], [2..\text{limit}]] :: [[\text{Int}]]$
- $[\text{h}', \text{e}', \text{l}', \text{l}', \text{o}'] :: [\text{Char}]$
- "hello" :: [\text{Char}]
- $[\text{"hello"}, \text{"world"}] :: ???$
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:

- $[1, 2, 3, \text{answer}] :: [\text{Integer}]$
- $[1 .. \text{limit}] :: [\text{Int}]$
- $[1 .. \text{limit}, 2 .. \text{limit}] :: [\text{Int}]$
- $['h', 'e', 'l', 'l', 'o'] :: [\text{Char}]$
- "hello" :: [\text{Char}]$
- $[\text{"hello", "world"}] :: [\text{[Char]}]$
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:

- \([1, 2, 3, \text{answer}] :: [\text{Integer}]\)
- \([1 \ldots \text{limit}] :: [\text{Int}]\)
- \([1 \ldots \text{limit}], [2 \ldots \text{limit}] :: [[\text{Int}]]\)
- \([\text{`h'`, `e'`, `l'`, `l'`, `o'}] :: [\text{Char}]\)
- \("\text{hello}\) :: [\text{Char}]\)
- \([\text{"hello"}, \text{"world"}] :: [[\text{Char}]]\)
- \([[[\text{"first"}, \text{"line"}], [\text{"second"}, \text{"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:

- $[1, 2, 3, \text{answer}] :: [\text{Integer}]$
- $[1 .. \text{limit}] :: [\text{Int}]$
- $[[1 .. \text{limit}], [2 .. \text{limit}]] :: [[\text{Int}]]$
- $[\text{'h'}, \text{'e'}, \text{'l'}, \text{'l'}, \text{'o'}] :: [\text{Char}]$
- "hello" :: [\text{Char}]
- $[\text{"hello"}, \text{"world"}] :: [[\text{Char}]]$
- $[[\text{"first"}, \text{"line"}], [\text{"second"}, \text{"line"}]] :: [[[\text{Char}]]]$
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:**

- \((\text{answer}, \text{limit}) : : ???\)
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:**

- $(\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})$
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:**

- $(\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})$
- $(\text{limit}, \text{answer}) :: ???$
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:**

- \( \text{(answer, limit)} :: \text{(Integer, Int)} \)
- \( \text{(limit, answer)} :: \text{(Int, Integer)} \)
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:**

- \((\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})\)
- \((\text{limit}, \text{answer}) :: (\text{Int}, \text{Integer})\)
- \("???", \text{answer}) :: ???\)
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:**

- \((\text{answer, limit}) :: (\text{Integer, Int})\)
- \((\text{limit, answer}) :: (\text{Int, Integer})\)
- \(("???", \text{answer}) :: ([\text{Char}], \text{Integer})\)
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:**

- \((\text{answer, limit}) :: (\text{Integer, Int})\)
- \((\text{limit, answer}) :: (\text{Int, Integer})\)
- \(("???", \text{answer}) :: ([Char], \text{Integer})\)
- \(("???", (\text{limit, answer})) :: ???\)
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:**

- \((\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})\)
- \((\text{limit}, \text{answer}) :: (\text{Int}, \text{Integer})\)
- \(\text{"???", answer} :: ([\text{Char}], \text{Integer})\)
- \(\text{"???", (limit, answer)} :: ([\text{Char}], (\text{Int}, \text{Integer}))\)
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:**

- $(\text{answer, limit}) :: (\text{Integer, Int})$
- $(\text{limit, answer}) :: (\text{Int, Integer})$
- $(\text{"???", answer}) :: ([\text{Char}], \text{Integer})$
- $(\text{"???", (limit, answer)}) :: ([\text{Char}], (\text{Int, Integer}))$
- $(\text{"???", 'X'}) :: ???$
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:**

- \((\text{answer}, \text{limit})::(\text{Integer}, \text{Int})\)
- \((\text{limit}, \text{answer})::(\text{Int}, \text{Integer})\)
- \("???", \text{answer})::([\text{Char}], \text{Integer})\)
- \("???", (\text{limit}, \text{answer}))::([\text{Char}], (\text{Int}, \text{Integer}))\)
- \("???", 'X')::([\text{Char}], \text{Char})\)
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:**

- $(\text{answer, limit}) :: (\text{Integer, Int})$
- $(\text{limit, answer}) :: (\text{Int, Integer})$
- $("???, \text{answer}) :: ([\text{Char}], \text{Integer})$
- $("???, (\text{limit, answer})) :: ([\text{Char}], (\text{Int, Integer}))$
- $("???, 'X') :: ([\text{Char}], \text{Char})$
- $(\text{limit, ("???, 'X')) :: ???}$
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:**

- \((\text{answer}, \text{limit})::(\text{Integer}, \text{Int})\)
- \((\text{limit}, \text{answer})::(\text{Int}, \text{Integer})\)
- \("???", \text{answer})::([\text{Char}], \text{Integer})\)
- \("???", (\text{limit}, \text{answer}))::([\text{Char}], (\text{Int}, \text{Integer}))\)
- \("???", 'X')::([\text{Char}], \text{Char})\)
- \((\text{limit}, ("???", 'X'))::(\text{Int}, ([\text{Char}], \text{Char}))\)
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:**

- \((\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})\)
- \((\text{limit}, \text{answer}) :: (\text{Int}, \text{Integer})\)
- \(\text{"???, answer} :: ([\text{Char}], \text{Integer})\)
- \(\text{"???, (limit, answer)} :: ([\text{Char}], (\text{Int}, \text{Integer}))\)
- \(\text{"???, 'X'} :: ([\text{Char}], \text{Char})\)
- \((\text{limit, ("???, 'X')} :: (\text{Int}, ([\text{Char}], \text{Char}))\)
- \((\text{True, [("X",limit),("Y",5)]}) :: ???\)
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:**

- $(\text{answer}, \text{limit}) :: (\text{Integer}, \text{Int})$
- $(\text{limit}, \text{answer}) :: (\text{Int}, \text{Integer})$
- $("???", \text{answer}) :: ([\text{Char}], \text{Integer})$
- $("???", (\text{limit}, \text{answer})) :: ([\text{Char}], (\text{Int}, \text{Integer}))$
- $("???", \text{`}X\text{'}") :: ([\text{Char}], \text{Char})$
- $(\text{limit}, ("???", \text{`}X\text{'}")) :: (\text{Int}, ([\text{Char}], \text{Char}))$
- $(\text{True}, [(\text{"X"},\text{limit}),(\text{"Y"},5)]) :: (\text{Bool}, [[[\text{Char}], \text{Int}]])$
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 $i$th component of type $t_i$.

Examples:

- $(\text{answer}, 'c', \text{limit}) :: (\text{Integer}, \text{Char}, \text{Int})$
- $(\text{answer}, 'c', \text{limit}, "all") :: (\text{Integer}, \text{Char}, \text{Int}, [\text{Char}])$
- $(()) :: ()$
  — 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

\[
\text{type } Name = t
\]

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

Examples:

\[
\begin{align*}
type \ String & = [\text{Char}] & \quad \text{-- predefined} \\
type \ Point & = (\text{Double, Double}) & \quad \text{-- (1.5, 2.7)} \\
type \ Triangle & = (\text{Point, Point, Point}) \\
type \ CharEntity & = (\text{Char, String}) & \quad \text{-- ('\AA\', '&uuml;\')}
\end{align*}
\]

\[
\text{type } \text{Dictionary} = [(\text{String, String})] & \quad \text{-- [("day","jour")]}
\]
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.:
  \[(a, b)\]  \(\Rightarrow\)  \[(Char, b)\]
  \[(a, b)\]  \(\Rightarrow\)  \[(Char, Int)\]
  \[(a, b)\]  \(\Rightarrow\)  \[(a, [(String, Int)])]\]
  \[(a, b)\]  \(\Rightarrow\)  \[(a, [(String, c)])]\]
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** \( t \rightarrow u \) 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 \( f :: a \rightarrow b \) and an argument \( x :: a \) are given, then we have \((f \ x) :: b\).

- If a function \( f :: a \rightarrow b \) 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 \( a \rightarrow b \).
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

\[
\text{fst} :: (a, b) \rightarrow a
\]
\[
\text{fst} (x, y) = x
\]

\[
\text{fst} (\text{\textquoteleft}c\textquoteleft, \text{False}) : : \text{Char}
\]

\[
[\text{"hello"}, \text{fst} (x, 17)] \Rightarrow x : : \text{String}
\]
Type Inference Examples

\[
\begin{align*}
\text{fst} &:: (a, b) \rightarrow a \\
\text{fst} (x, y) & = x \\
\text{fst} ('c', \text{False}) &:: \text{Char} \\
["hello", \text{fst} (x, 17)] &\Rightarrow x :: \text{String} \\
\text{f p} & = \text{limit} + \text{fst} p
\end{align*}
\]
Type Inference Examples

\[\text{fst :: (a,b) -> a}\]
\[\text{fst (x,y) = x}\]

\[\text{fst ('c', False) :: Char}\]

\[\text{["hello", fst (x, 17)] \Rightarrow x :: String}\]

\[\text{f p = limit + fst p \Rightarrow p :: (Int, a)}\]
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

\[
\begin{align*}
\text{fst} &:: (a,b) \rightarrow a \\
\text{fst} (x,y) &= x \\
\text{fst} ('c', \text{False}) &:: \text{Char} \\
["hello", \text{fst} (x, 17)] &\Rightarrow x :: \text{String} \\
\text{f p} &= \text{limit} + \text{fst} p \\
\Rightarrow p &:: (\text{Int},a) \\
\text{f} &:: (\text{Int},a) \rightarrow \text{Int} \\
\text{g h} &= \text{fst} (h "") : [\text{limit}] 
\end{align*}
\]
Type Inference Examples

\[ \text{fst :: (a\!,b) \to a} \]
\[ \text{fst (x, y) = x} \]
\[ \text{fst ('c', False)} :: \text{Char} \]
\[ \text{["hello", fst (x, 17)]} \Rightarrow \text{x :: String} \]
\[ \text{f p = limit + fst p} \Rightarrow \text{p :: (Int,a)} \]
\[ \text{f :: (Int,a) \to Int} \]
\[ \text{g h = fst (h "") : [limit]} \]
\[ \Rightarrow \text{h :: String \to (Int,a)} \]
Let’s Play the Evaluation Game Again — 1

\[
\text{h1} :: \text{String} \rightarrow (\text{Int}, \text{String})\\
\text{h1 str} = (\text{length str}, ' ' : \text{str})\\
\text{g h} = \text{fst (h "") : [limit]}
\]
Let’s Play the Evaluation Game Again — 1

h1 :: String → (Int, String)
h1 str = (length str, ‘ ’ : str)

\[ g \ h = \text{fst} \ (h \ "") : \text{[limit]} \]

Then:

\[ g \ h1 \]
Let’s Play the Evaluation Game Again — 1

\[
\text{h1 :: String} \rightarrow \text{(Int, String)}
\]

\[
h1 \text{ str} = (\text{length str, } ' ' : \text{str})
\]

\[
g \text{ h} = \text{fst (h "") : [limit]}
\]

Then:

\[
g \text{ h1} \\
= \text{fst (h1 "") : [limit]}
\]
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

\[
\begin{align*}
\text{h1} & : \text{String} \rightarrow (\text{Int}, \text{String}) \\
\text{h1} \; \text{str} & = (\text{length} \; \text{str}, \; '' : \; \text{str})
\end{align*}
\]

\[
\begin{align*}
\text{g} \; \text{h} & = \text{fst} \; (\text{h} \; "") : [\text{limit}]
\end{align*}
\]

Then:

\[
\begin{align*}
\text{g} \; \text{h1} & \\
& = \text{fst} \; (\text{h1} \; "") : [\text{limit}] \\
& = \text{fst} \; (\text{length} \; "", \; '' : \; "") : [\text{limit}] \\
& = \text{length} \; "" : [\text{limit}]
\end{align*}
\]
Let’s Play the Evaluation Game Again — 1

\[ h_1 :: \text{String} \rightarrow (\text{Int}, \text{String}) \]
\[ h_1 \text{ str} = (\text{length str}, '\ ' : \text{str}) \]

\[ g \ h = \text{fst} (h "") : [\text{limit}] \]

Then:

\[ g \ h_1 \]
\[ = \text{fst} (h_1 "") : [\text{limit}] \]
\[ = \text{fst} (\text{length } ",\ ' ' : ") : [\text{limit}] \]
\[ = \text{length } ": \text{limit} \]
\[ = 0 : [\text{limit}] \]
Let’s Play the Evaluation Game Again — 1

\[ \text{h1} :: \text{String} \rightarrow (\text{Int}, \text{String}) \]
\[ \text{h1 \ str} = (\text{length \ str}, ' ' : \text{str}) \]

\[ g \ h = \text{fst} (h "") : [\text{limit}] \]

\textbf{Then:}

\[ g \ h1 \]
\[ = \text{fst} (h1 "") : [\text{limit}] \]
\[ = \text{fst} (\text{length "", ' ' : ""}) : [\text{limit}] \]
\[ = \text{length ""} : [\text{limit}] \]
\[ = 0 : [\text{limit}] \]
\[ = [0, 100] \]
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']

\( g \ h = \text{fst} \ (h \ "") \ : \ [\text{limit}] \)

Then:

\( g \ h2 \)
\( = \text{fst} \ (h2 \ "") \ : \ [\text{limit}] \)
\( = \text{fst} \ (\text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} \ "")), \ \text{head} \ "") \ : \ [\text{limit}] \)
\( = \text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} \ "")) \ : \ [\text{limit}] \)
Let’s Play the Evaluation Game Again — 2

\( h_2 :: \text{String} \to (\text{Int}, \text{Char}) \)
\( h_2 \text{ str} = (\text{sum} (\text{map} \ \text{ord} \ (\text{notOccCaps} \text{ str})), \text{head} \text{ str}) \)

\( \text{notOccCaps} :: \text{String} \to \text{String} \)
\( \text{notOccCaps} \text{ str} = \text{filter} \ (\text{notElem} \text{ str}) [\text{'A'} .. \text{'Z'}] \)

\( g \ h = \text{fst} \ (h \ "") : [\text{limit}] \)

**Then:**

\( g \ h_2 \)
\( = \text{fst} \ (h_2 \ "") : [\text{limit}] \)
\( = \text{fst} \ (\text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} \ "")), \text{head} \ "") : [\text{limit}] \)
\( = \text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} \ "")) : [\text{limit}] \)
\( = \ldots \)
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

\[ h_2 :: \text{String} \rightarrow (\text{Int}, \text{Char}) \]
\[ h_2 \text{ str} = (\text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} \ \text{str})), \ \text{head} \ \text{str}) \]

\[ \text{notOccCaps} :: \text{String} \rightarrow \text{String} \]
\[ \text{notOccCaps} \ \text{str} = \text{filter} \ (\backslash \text{notElem} \ \text{str}) \ ['A' .. 'Z'] \]

\[ g \ h = \text{fst} \ (h ''): \text{[limit]} \]

\textbf{Then:}

\[ g \ h2 \]
\[ = \text{fst} \ (h2 ''): \text{[limit]} \]
\[ = \text{fst} \ (\text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} '')), \ \text{head} ''): \text{[limit]} \]
\[ = \text{sum} \ (\text{map} \ \text{ord} \ (\text{notOccCaps} '')): \text{[limit]} \]
\[ = ... \]
\[ = 2015: \text{[limit]} \]
\[ = [2015, 100] \]
Higher-Order Functions

g h = fst (h "") : [limit]
Higher-Order Functions

\[ g \ h = \text{fst} (h \ "") : \text{[limit]} \]

Functional Programming: \textbf{Functions are first-class citizens}
Higher-Order Functions

\[ \text{g h} = \text{fst (h "") : [limit]} \]

Functional Programming: Functions are first-class citizens

- Functions can be arguments of other functions: \( g \ h2 \)
Higher-Order Functions

\[ g \ h = \text{fst} (h \ "") : [\text{limit}] \]

Functional Programming: Functions are first-class citizens

- Functions can be arguments of other functions: \( g \ h_2 \)

- Functions can be components of data structures: \( (7, h_1), [h_1, h_2] \)
Higher-Order Functions

\[
g \ h = \text{fst} (h \ "") : \ [\text{limit}]\]

**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**: \( \text{succ} \ . \ \text{succ} \)
Higher-Order Functions

\[ g \ h = \text{fst} (h \ "") : [\text{limit}] \]

**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**: \( \text{succ} \ . \ \text{succ} \)

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

A **higher-order function** expects functions as arguments.
Higher-Order Functions

\[ g \ h = \text{fst} \ (h \; "") : [\text{limit}] \]

Functional Programming: Functions are first-class citizens

- Functions can be arguments of other functions: \( g \ h_2 \)
- Functions can be components of data structures: \( (7, h_1), [h_1, h_2] \)
- Functions can be results of function application: \( \text{succ} \; . \; \text{succ} \)

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 \( h_1, h_2 \) as arguments.
Type Inference Examples

\[
fst :: (a, b) \rightarrow a
\]
\[
fst (x, y) = x
\]
\[
fst ('c', False) :: Char
\]
\[
["hello", fst (x, 17)] \Rightarrow x :: String
\]
\[
f p = limit + fst p \Rightarrow p :: (Int, a)
f :: (Int, a) \rightarrow Int
\]
\[
g h = fst (h "") : [limit]
\Rightarrow h :: String \rightarrow (Int, a)
Type Inference Examples

\[ \text{fst} :: (a,b) \rightarrow a \]
\[ \text{fst} (x,y) = x \]

\[ \text{fst} ('c', \text{False}) \quad \text{:: Char} \]

\[ ["\text{hello}", \text{fst} (x, 17)] \quad \Rightarrow \quad x \quad \text{:: String} \]

\[ \text{f p} = \text{limit} + \text{fst} p \quad \Rightarrow \quad p \quad \text{:: (Int,a)} \]
\[ \quad \text{f} \quad \text{:: (Int,a) \rightarrow Int} \]

\[ \text{g h} = \text{fst} (h "") : [\text{limit}] \]
\[ \Rightarrow \quad h \quad \text{:: String \rightarrow (Int,a)} \]
\[ \quad \text{g} \quad \text{:: (String \rightarrow (Int,a)) \rightarrow [Int]} \]
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”:
  \[ \text{cylVol} \ r \ h = (\pi :: \text{Double}) \ast r \ast r \ast h \]
Curried Functions

- Function application associates to the left, i.e.,
  \[ f \times y = (f \times x) \times y \]

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., “they accept their arguments one at a time”:
  \[ \text{cylVol} \ r \ h = (\pi :: \text{Double}) \times r \times r \times h \]

Since the right-hand side, \( r \) and \( h \) obviously all have type \( \text{Double} \), we have;

\[ (\text{cylVol} \ r) :: ??? \]
Curried Functions

- **Function application associates to the left**, i.e.,
  \[ f \times y = (f \times x) \times y \]

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., “they accept their arguments one at a time”:
  \[ \text{cylVol } r \times h = (\pi :: \text{Double}) \times r \times r \times h \]
  Since the right-hand side, \( r \) and \( h \) obviously all have type \text{Double}, we have;
  \[ (\text{cylVol } r) :: \text{Double} \rightarrow \text{Double} \]
Curried Functions

- **Function application associates to the left**, i.e.,
  \[ f \times y = (f \times x) \times y \]

- Multi-argument functions in Haskell are typically defined as **curried** function, i.e., “they accept their arguments one at a time”:
  \[ \text{cylVol } r \ h = (\pi :: \text{Double}) \times r \times r \times h \]

Since the right-hand side, \( r \), and \( h \) obviously all have type \text{Double}, we have;

\[
(cylVol \ r) :: \text{Double} \rightarrow \text{Double}
\]

\[
cylVol :: ???
\]
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”:
  \[ \texttt{cylVol } r \ h = (\pi :: \text{Double}) \times r \times r \times h \]

Since the right-hand side, \( r \) and \( h \) obviously all have type \texttt{Double}, we have;

\[
\texttt{(cylVol } r) :: \text{Double} \to \text{Double} \\
\texttt{cylVol} :: \text{Double} \to (\text{Double} \to \text{Double})
\]
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”:

\[ \text{cylVol} \ r \ h = (\pi :: \text{Double}) * r * r * h \]

Since the right-hand side, \( r \) and \( h \) obviously all have type \( \text{Double} \), we have:

\[ (\text{cylVol} \ r) :: \text{Double} \rightarrow \text{Double} \]
\[ \text{cylVol} :: \text{Double} \rightarrow (\text{Double} \rightarrow \text{Double}) \]

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

\[ a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c) \]
“Partial Application”

Let values with the following types be given:

\[ f :: a \to b \to c \]
\[ x :: a \]
\[ y :: b \]
“Partial Application”

Let values with the following types be given:

\[ f :: a \rightarrow b \rightarrow c \]
\[ x :: a \]
\[ y :: b \]

The type of \( f \) is the function type \( a \rightarrow (b \rightarrow c) \)
“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:

\[ f :: a \rightarrow b \rightarrow c \]
\[ x :: a \]
\[ y :: b \]

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

- argument type \( a \),
- result type \( b \rightarrow c \).

Therefore, we can apply \( f \) to \( x \) and obtain:
“Partial Application”

Let values with the following types be given:

\[ f :: a \rightarrow b \rightarrow c \]
\[ x :: a \]
\[ y :: b \]

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

- argument type \( a \),
- result type \( b \rightarrow c \).

Therefore, we can apply \( f \) to \( x \) and obtain:

\[ (f \ x) :: b \rightarrow c \]
“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:

\[ f :: a \rightarrow b \rightarrow c \]
\[ x :: a \]
\[ y :: b \]

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

- argument type \( a \),
- result type \( b \rightarrow c \).

Therefore, we can apply \( f \) to \( x \) and obtain:

\[ (f \ x) :: b \rightarrow c \]

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:

\[ (f \ x) \ y :: c = f \ x \ y \]
Partial Application — Example

\[ g :: (\text{String} \rightarrow (\text{Int}, a)) \rightarrow [\text{Int}] \]
\[ g \ h = \text{fst} (h "") : [\text{limit}] \]
Partial Application — Example

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

\[ k n str = (n \ast (\text{length str} + 1), \text{unwords} (\text{replicate n str})) \]
Partial Application — Example

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

\[ k :: \text{Int} \rightarrow \text{String} \rightarrow (\text{Int}, \text{String}) \]
\[ k \ n \ \text{str} = (n \ast (\text{length str} + 1), \text{unwords}(\text{replicate} \ n \ \text{str})) \]
Partial Application — Example

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

\[
k :: \text{Int} \rightarrow \text{String} \rightarrow (\text{Int}, \text{String})
k n \text{ str} = (n * (\text{length str} + 1), \text{unwords} (\text{replicate n str}))
\]

\[
g (k 3)
\]
Partial Application — Example

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

\[ k :: \text{Int} \rightarrow \text{String} \rightarrow (\text{Int}, \text{String}) \]
\[ k \ n \ \text{str} = (n \ast (\text{length} \ \text{str} + 1), \text{unwords} (\text{replicate} \ n \ \text{str})) \]

\[ g \ (k \ 3) \]
\[ = \text{fst} (k \ 3 \ "") : \text{[limit]} \]
Partial Application — Example

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

\[ k :: \text{Int} \rightarrow \text{String} \rightarrow (\text{Int}, \ \text{String}) \]
\[ k \ n \ \text{str} = (n \ast (\text{length} \ \text{str} + 1), \ \text{unwords} (\ \text{replicate} \ n \ \text{str})) \]

\[ g \ (k \ 3) \]
\[ = \text{fst} (k \ 3 "") : [\ \text{limit}] \]
\[ = \text{fst} (3 \ast (\text{length} "" + 1), \ \text{unwords} (\ \text{replicate} 3 "")) : [\ \text{limit}] \]
Partial Application — Example

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

\[ k :: \text{Int} \to \text{String} \to (\text{Int}, \text{String}) \]
\[ k \ n \ \text{str} = (n \ast (\text{length str} + 1), \text{unwords} (\text{replicate n str})) \]

\[ g \ (k \ 3) \]
\[ = \text{fst}(k \ 3 \\text{""}) : \text{[limit]} \]
\[ = \text{fst}(3 \ast (\text{length \\text{""} + 1), \text{unwords} (\text{replicate 3 \\text{""})) : \text{[limit]} \]
\[ = (3 \ast (\text{length \\text{""} + 1)) : \text{[limit]} \]
Partial Application — Example

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

\[ k :: Int \rightarrow String \rightarrow (Int, String) \]
\[ k \ n \ str = (n \ast (\text{length} \ str + 1), \text{unwords} (\text{replicate} \ n \ str)) \]

\[ g \ (k \ 3) \]
\[ = \text{fst} (k \ 3 "") : [\text{limit}] \]
\[ = \text{fst} (3 \ast (\text{length} "" + 1), \text{unwords} (\text{replicate} \ 3 "")) : [\text{limit}] \]
\[ = (3 \ast (\text{length} "" + 1)) : [\text{limit}] \]
\[ = (3 \ast (0 + 1)) : [\text{limit}] \]
Partial Application — Example

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

\[ k :: \text{Int} \rightarrow \text{String} \rightarrow (\text{Int}, \text{String}) \]
\[ k \; n \; \text{str} = (n \ast (\text{length} \; \text{str} + 1), \text{unwords} (\text{replicate} \; 3 \; \text{str})) \]

\[ g \; (k \; 3) \]
\[ = \text{fst} \; (k \; 3 \\) : [\text{limit}] \]
\[ = \text{fst} \; (3 \ast (\text{length} \\ + 1), \text{unwords} (\text{replicate} \; n \\)) : [\text{limit}] \]
\[ = (3 \ast (\text{length} \\ + 1)) : [\text{limit}] \]
\[ = (3 \ast (0 + 1)) : [\text{limit}] \]
\[ = (3 \ast 1) : [\text{limit}] \]
Partial Application — Example

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

\[ k :: Int \to String \to (Int, String) \]
\[ k \ n \ str = (n \times (\text{length} \ str + 1), \text{unwords} (\text{replicate} \ n \ str)) \]

\[ g (k \ 3) \]
\[ = \text{fst} (k \ 3 "") : [\text{limit}] \]
\[ = \text{fst} (3 \times (\text{length} "" + 1), \text{unwords} (\text{replicate} \ 3 "")) : [\text{limit}] \]
\[ = (3 \times (\text{length} "" + 1)) : [\text{limit}] \]
\[ = (3 \times (0 + 1)) : [\text{limit}] \]
\[ = (3 \times 1) : [\text{limit}] \]
\[ = 3 : [\text{limit}] \]
Partial Application — Example

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

\[ k :: \text{Int} \to \text{String} \to (\text{Int}, \text{String}) \]
\[ k \ n \ \text{str} = (n \ast (\text{length} \ \text{str} + 1), \text{unwords} (\text{replicate} \ n \ \text{str})) \]

\[ g \ (k \ 3) \]
\[ = \text{fst} \ (k \ 3 \ "") : [\text{limit}] \]
\[ = \text{fst} \ (3 \ast (\text{length} \ "" + 1), \text{unwords} (\text{replicate} \ 3 \ "")) : [\text{limit}] \]
\[ = (3 \ast (\text{length} \ "" + 1)) : [\text{limit}] \]
\[ = (3 \ast (0 + 1)) : [\text{limit}] \]
\[ = (3 \ast 1) : [\text{limit}] \]
\[ = 3 : [\text{limit}] \]
\[ = [3, 100] \]
Operations on Functions

\[ \text{id} :: a \to a \quad \text{--- identity function} \]
\[ \text{id} \ x = x \]

\[ (.) :: (b \to c) \to (a \to b) \to (a \to c) \quad \text{--- function composition} \]
\[ (f \ . \ g) \ x = f (g \ x) \]

\[ \text{flip} :: (a \to b \to c) \to (b \to a \to c) \quad \text{--- argument swapping} \]
\[ \text{flip} \ f \ x \ y = f \ y \ x \]

\[ \text{curry} :: ((a,b) \to c) \to (a \to b \to c) \quad \text{--- currying} \]
\[ \text{curry} \ g \ x \ y = g \ (x,y) \]

\[ \text{uncurry} :: (a \to b \to c) \to ((a,b) \to c) \]
\[ \text{uncurry} \ f \ (x,y) = f \ x \ y \]

**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:

\[(+) \ :: \ \text{Int} \rightarrow \ \text{Int} \rightarrow \ \text{Int} \quad \quad \text{-- not the “natural” type of (+)}\]
\[(:) \ :: \ a \rightarrow [a] \rightarrow [a]\]
\[++] \ :: \ [a] \rightarrow [a] \rightarrow [a]\]

• 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):

```haskell
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:

\[ \text{null} :: [a] \to \text{Bool} \]
Defining Functions Over Lists by Pattern Matching

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

\[
\text{null} \quad :: \quad [a] \rightarrow \text{Bool}
\]

\[
\text{null} \quad [\ ] \quad =
\]
Defining Functions Over Lists by Pattern Matching

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

\[
\begin{align*}
null & : [a] \rightarrow \text{Bool} \\
null \; [\;] & = \\
null \; (x : xs) & = 
\end{align*}
\]
Defining Functions Over Lists by Pattern Matching

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

\[
\text{null} :: [a] \to \text{Bool}
\]
\[
\text{null} \ [ \ ] = \text{True}
\]
\[
\text{null} \ (x : xs) =
\]
Defining Functions Over Lists by Pattern Matching

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

\[
\text{null} :: [a] \rightarrow \text{Bool} \\
\text{null} \; [\;] = \text{True} \\
\text{null} \; (x : xs) = \text{False}
\]
Defining Functions Over Lists by Pattern Matching

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

\[
\begin{align*}
\text{null} &:: [a] \to \text{Bool} \\
\text{null} \; [ ] &\; = \; \text{True} \\
\text{null} \; (x : xs) &\; = \; \text{False}
\end{align*}
\]

\[
\begin{align*}
\text{head} &:: [a] \to a
\end{align*}
\]
Defining Functions Over Lists by Pattern Matching

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

\[
\text{null} :: [a] \rightarrow \text{Bool} \\
\text{null} \ [\ ] = \text{True} \\
\text{null} \ (x : xs) = \text{False}
\]

\[
\text{head} :: [a] \rightarrow a \\
\text{head} \ (x : xs) = x
\]
Defining Functions Over Lists by Pattern Matching

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

\[ \text{null} :: [a] \rightarrow \text{Bool} \]
\[ \text{null} \; \text{[]} = \text{True} \]
\[ \text{null} \; (x : xs) = \text{False} \]

\[ \text{head} :: [a] \rightarrow a \]
\[ \text{head} \; (x : xs) = x \]

\[ \text{tail} :: [a] \rightarrow [a] \]
Defining Functions Over Lists by Pattern Matching

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

\[\text{null} :: [a] \rightarrow \text{Bool}\]
\[\text{null} \ [\ ] = \text{True}\]
\[\text{null} \ (x : xs) = \text{False}\]

\[\text{head} :: [a] \rightarrow a\]
\[\text{head} \ (x : xs) = x\]

\[\text{tail} :: [a] \rightarrow [a]\]
\[\text{tail} \ (x : xs) = xs\]
Defining Functions Over Lists by Pattern Matching

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

\[
\textit{null} :: [a] \rightarrow \text{Bool} \\
\textit{null} [\ ] = \text{True} \\
\textit{null} (x : xs) = \text{False}
\]

\[
\textit{head} :: [a] \rightarrow a \\
\textit{head} (x : xs) = x \\
\]

\[
\textit{tail} :: [a] \rightarrow [a] \\
\textit{tail} (x : xs) = xs \\
\]

(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:

\begin{align*}
\text{length} & \quad :: \ [a] \rightarrow \text{Int} \\
\text{concat} & \quad :: \ [\ [a] ] \rightarrow \ [a] \\
( \text{++} ) & \quad :: \ [a] \rightarrow \ [a] \rightarrow \ [a] \\
\text{product} & \quad :: \ [\ \text{Integer} \ ] \rightarrow \text{Integer} \\
\text{sum} & \quad :: \ [\ \text{Integer} \ ] \rightarrow \text{Integer} \\
(\text{\textquote{elem}}) & \quad :: \ \text{Int} \rightarrow \ [\ \text{Int} \ ] \rightarrow \text{Bool}
\end{align*}
Defining Functions Over Lists by Structural Induction

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

\[
\text{length} :: [a] \rightarrow \text{Int} \\
\text{length} \ [ \ ] = \\
\text{length} \ (x : xs) =
\]

\[
( \text{++)} :: [a] \rightarrow [a] \rightarrow [a] \\
[ \ ] \text{++)} ys = \\
(x : xs) \text{++)} ys =
\]

\[
x \text{‘elem’} \ [ \ ] = \\
x \text{‘elem’} \ (y : ys) =
\]

(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:

\[
\begin{align*}
\text{length} & \quad :: \ [a] \rightarrow \text{Int} \\
\text{length} \ [\ ] & \quad = \ 0 \\
\text{length} \ (x : xs) & \quad = \\
\text{concat} & \quad :: \ [[a]] \rightarrow [a] \\
\text{concat} \ [\ ] & \quad = \\
\text{concat} \ (xs : xss) & \quad = \\
\text{sum} & \quad :: \ [a] \rightarrow [a] \rightarrow [a] \\
\text{sum} \ [\ ] & \quad = \\
\text{sum} \ (x : xs) & \quad = \\
\text{product} & \quad :: \ [a] \rightarrow [a] \rightarrow [a] \\
\text{product} \ [\ ] & \quad = \\
\text{product} \ (x : xs) & \quad = \\
x \ ‘\text{elem}‘ \ [\ ] & \quad = \\
x \ ‘\text{elem}‘ \ (y : ys) & \quad =
\end{align*}
\]

(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:

\[
\begin{align*}
\text{length} &: \mathbb{[a]} \to \mathbb{Int} \\
\text{length} [ ] &= 0 \\
\text{length} (x : xs) &= 1 + \text{length} \hspace{1pt} xs \\
\text{(++)} &: \mathbb{[a]} \to \mathbb{[a]} \to \mathbb{[a]} \\
[ ] + ys &= \\
(x : xs) + ys &= \\
x \text{\ 'elem'} [ ] &= \\
x \text{\ 'elem'} (y : ys) &= 
\end{align*}
\]

(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:

\[
\begin{align*}
\text{length} & : \ [\ a\ ] \rightarrow \text{Int} \\
\text{length} \ [\ ] & = 0 \\
\text{length} \ (x : xs) & = 1 + \text{length} \ xs \\
\text{concat} & : [[\ a\ ]] \rightarrow [\ a\ ] \\
\text{concat} \ [\ ] & = \ [\ ] \\
\text{concat} \ (xs : xss) & = \\
(\ +\ ) & : [\ a\ ] \rightarrow [\ a\ ] \rightarrow [\ a\ ] \\
[\ ] + ys & = ys \\
(x : xs) + ys & = \\
\text{sum} & : \ [\ a\ ] \rightarrow \text{Int} \\
\text{sum} \ [\ ] & = \ \\
\text{sum} \ (x : xs) & = \\
\text{product} & : \ [\ a\ ] \rightarrow \text{Int} \\
\text{product} \ [\ ] & = \ \\
\text{product} \ (x : xs) & = \\
\text{x `elem` [ ]} & = \ \\
\text{x `elem` (y : ys)} & = \\
\end{align*}
\]

(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:

\[
\begin{align*}
\text{length} & : [a] \to \text{Int} \\
\text{length} [ ] & = 0 \\
\text{length} (x : xs) & = 1 + \text{length} xs
\end{align*}
\]

\[
\begin{align*}
(+) & : [a] \to [a] \to [a] \\
[ ] + ys & = ys \\
(x : xs) + ys & = x : (xs + ys)
\end{align*}
\]

\[
\begin{align*}
x \text{ \texttt{\textquotesingle}elem\textquotesingle} [ ] & = \\
x \text{ \texttt{\textquotesingle}elem\textquotesingle} (y : ys) & =
\end{align*}
\]

(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 \textit{structural induction}:

\begin{align*}
\text{length} &: \mathbb{[a]} \rightarrow \mathbb{Int} \\
\text{length} [ &] = 0 \\
\text{length} (x : xs) &= 1 + \text{length} \; xs \\
\text{concat} &: \mathbb{[[a]]} \rightarrow \mathbb{[a]} \\
\text{concat} [ &] = [ ] \\
\text{concat} (xs : xss) &= \text{concat} xs \; \text{concat} xss \\
\text{(++)} &: \mathbb{[a]} \rightarrow \mathbb{[a]} \rightarrow \mathbb{[a]} \\
[ &] + ys &= ys \\
(x : xs) + ys &= x : (xs + ys) \\
\text{sum} &: \mathbb{[a]} \rightarrow \mathbb{a} \\
\text{sum} [ &] = \\
\text{sum} (x : xs) &= \\
\text{product} &: \mathbb{[a]} \rightarrow \mathbb{a} \\
\text{product} [ &] = \\
\text{product} (x : xs) &= \\
x \text{‘elem‘} [ &] = \\
x \text{‘elem‘} (y : ys) &= \\
\end{align*}

(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:

\[
\begin{align*}
\text{length} & \quad :: \ [a] \to \text{Int} \\
\text{length} \ [\] & = 0 \\
\text{length} \ (x : xs) & = 1 + \text{length} \ xs \\
\text{concat} & \quad :: \ [[a]] \to [a] \\
\text{concat} \ [\] & = [\] \\
\text{concat} \ (xs : xss) & = xs ++ \text{concat} \ xss \\
(\) & \quad :: \ [a] \to [a] \to [a] \\
[\] & + ys = ys \\
(x : xs) & + ys = x : (xs ++ ys) \\
\text{sum} & \quad :: \ [a] \to a \\
\text{sum} \ [\] & = \\
\text{sum} \ (x : xs) & = \\
\text{product} & \quad :: \ [a] \to a \\
\text{product} \ [\] & = \\
\text{product} \ (x : xs) & = \\
x \ 'elem' \ [\] & = \\
x \ 'elem' \ (y : ys) & = 
\end{align*}
\]

(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 \textit{structural induction}:

\begin{align*}
\text{\textit{length}} & \quad :: \quad [\ a\ ] \rightarrow \text{Int} \\
\text{\textit{length}} \ [\ ] & \quad = \quad 0 \\
\text{\textit{length}} \ (x : xs) & \quad = \quad 1 + \text{\textit{length}} \ xs \\
\text{\textit{concat}} & \quad :: \quad [[\ a\ ]] \rightarrow [\ a\ ] \\
\text{\textit{concat}} \ [\ ] & \quad = \quad [\ ] \\
\text{\textit{concat}} \ (xs : xss) & \quad = \quad xs \ + \ \text{\textit{concat}} \ xss \\
\text{\textit{(+ )}} & \quad :: \quad [\ a\ ] \rightarrow [\ a\ ] \rightarrow [\ a\ ] \\
[\ ] \ + \ ys & \quad = \quad ys \\
(x : xs) \ + \ ys & \quad = \quad x : (xs \ + \ ys) \\
\text{\textit{sum}} & \quad :: \quad [\ ] \rightarrow \text{Int} \\
\text{\textit{sum}} \ [\ ] & \quad = \quad 0 \\
\text{\textit{sum}} \ (x : xs) & \quad = \quad \text{\textit{sum}} \ xs \ + \ x \\
\text{\textit{product}} & \quad :: \quad [\ ] \rightarrow \text{Int} \\
\text{\textit{product}} \ [\ ] & \quad = \quad 1 \\
\text{\textit{product}} \ (x : xs) & \quad = \quad \text{\textit{product}} \ xs \ \times \ x \\
\text{\textit{x `elem`}} & \quad :: \quad [\ ] \rightarrow \text{Int} \\
\text{\textit{x `elem`}} \ [\ ] & \quad = \quad 0 \\
\text{\textit{x `elem`}} \ (y : ys) & \quad = \quad \text{\textit{elem}} \ ys \ y
\end{align*}

(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**:

- **length**: 
  \[ \text{length} :: [a] \rightarrow \text{Int} \]
  \[ \text{length} \ [\] = 0 \]
  \[ \text{length} \ (x : xs) = 1 + \text{length} \ xs \]

- **concat**: 
  \[ \text{concat} :: [[a]] \rightarrow [a] \]
  \[ \text{concat} \ [\] = [\] \]
  \[ \text{concat} \ (xs : xss) = xs ++ \text{concat} \ xss \]

- **(++)**: 
  \[ (\cdot\cdot\cdot) :: [a] \rightarrow [a] \rightarrow [a] \]
  \[ [\] ++ ys = ys \]
  \[ (x : xs) ++ ys = x : (xs ++ ys) \]

- **sum**: 
  \[ \text{sum} :: [a] \rightarrow \text{Int} \]
  \[ \text{sum} \ [\] = 0 \]
  \[ \text{sum} \ (x : xs) = x + \text{sum} \ xs \]

- **product**: 
  \[ \text{product} :: [a] \rightarrow \text{Int} \]
  \[ \text{product} \ [\] = \text{product} \ (x : xs) = \]

  (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:

\[
\begin{align*}
\text{length} &: \mathbb{[a]} \to \mathbb{Int} \\
\text{length} [ ] &= 0 \\
\text{length} (x : xs) &= 1 + \text{length} xs \\
\text{concat} &: \mathbb{[[a]]} \to \mathbb{[a]} \\
\text{concat} [ ] &= [ ] \\
\text{concat} (xs : xss) &= xs \mathbin{+} \text{concat} xss \\
(++) &: \mathbb{[a]} \to \mathbb{[a]} \to \mathbb{[a]} \\
[ ] + ys &= ys \\
(x : xs) \mathbin{+} ys &= x : (xs \mathbin{+} ys) \\
x \text{‘elem’} [ ] &= \\
x \text{‘elem’} (y : ys) &= \\
\end{align*}
\]

(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:

\[ length \quad :: \quad [a] \rightarrow \text{Int} \]
\[ length \; [\;] \; = \; 0 \]
\[ length \; (x : \; xs) \; = \; 1 + \; length \; xs \]

\( (++\) \quad :: \quad [a] \rightarrow [a] \rightarrow [a] \)
\[ [\;] \; + \; ys \; = \; ys \]
\[ (x : \; xs) \; + \; ys \; = \; x \; : \; (xs \; + \; ys) \]

\[ sum \quad :: \quad [\;] \rightarrow \text{Int} \]
\[ sum \; [\;] \; = \; 0 \]
\[ sum \; (x : \; xs) \; = \; x \; + \; sum \; xs \]

\[ product \quad :: \quad [\;] \rightarrow \text{Int} \]
\[ product \; [\;] \; = \; 0 \]
\[ product \; (x : \; xs) \; = \; x \; \ast \; product \; xs \]

\[ x \; \text{`elem`} \; [\;] \; = \; \text{False} \]
\[ x \; \text{`elem`} \; (y : \; ys) \; = \; \text{True} \]

(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:

\[\text{length} \quad :: \quad [\, a \,] \rightarrow \text{Int}\]
\[\text{length} \quad [\,] \quad = \quad 0\]
\[\text{length} \quad (x : xs) \quad = \quad 1 + \text{length} \quad xs\]

\[\text{concat} \quad :: \quad [[[\, a \,]]] \rightarrow [\, a \,]\]
\[\text{concat} \quad [\,] \quad = \quad [\,]\]
\[\text{concat} \quad (xs : xss) \quad = \quad xs + \text{concat} \quad xss\]

\[\text{sum} \quad :: \quad [\, a \,] \rightarrow \text{Int}\]
\[\text{sum} \quad [\,] \quad = \quad 0\]
\[\text{sum} \quad (x : xs) \quad = \quad x + \text{sum} \quad xs\]

\[\text{product} \quad :: \quad [\, a \,] \rightarrow \text{Int}\]
\[\text{product} \quad [\,] \quad = \quad 0\]
\[\text{product} \quad (x : xs) \quad = \quad x * \text{product} \quad xs\]

\[x \quad \text{\textquotesingle elem\textquoteright} \quad [\,] \quad = \quad \text{False}\]
\[x \quad \text{\textquotesingle elem\textquoteright} \quad (y : ys) \quad = \quad \]

(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:

\[
\begin{align*}
\text{length} & : \mathbb{[a]} \rightarrow \mathbb{\text{Int}} \\
\text{length} [ ] & = 0 \\
\text{length} (x : xs) & = 1 + \text{length} xs
\end{align*}
\]

\[
\begin{align*}
\text{concat} & : \mathbb{[[a]]} \rightarrow \mathbb{[a]}
\text{concat} [ ] & = [ ] \\
\text{concat} (xs : xss) & = xs ++ \text{concat} xss
\end{align*}
\]

\[
\begin{align*}
(+) & : \mathbb{[a]} \rightarrow \mathbb{[a]} \rightarrow \mathbb{[a]}
[ ] & + ys = ys \\
(x : xs) & + ys = x : (xs + ys)
\end{align*}
\]

\[
\begin{align*}
\text{sum} & : \mathbb{[a]} \rightarrow \mathbb{\text{Int}} \\
\text{sum} [ ] & = 0 \\
\text{sum} (x : xs) & = x + \text{sum} xs
\end{align*}
\]

\[
\begin{align*}
\text{product} & : \mathbb{[a]} \rightarrow \mathbb{\text{Int}} \\
\text{product} [ ] & = 0 \\
\text{product} (x : xs) & = x \ast \text{product} xs
\end{align*}
\]

\[
\begin{align*}
x \text{ `elem` } [ ] & = \text{False} \\
x \text{ `elem` } (y : ys) & = x \equiv y \parallel x \text{ `elem` } ys
\end{align*}
\]

(All these functions are in the standard prelude.)
Guarded Definitions

\[
\text{sign } x \quad | \quad x > 0 = 1 \\
| \quad x == 0 = 0 \\
| \quad x < 0 = -1
\]
Guarded Definitions

\[ \text{sign } x \quad | \quad x > 0 = 1 \]
\[ | \quad x == 0 = 0 \]
\[ | \quad x < 0 = -1 \]

\[ \text{choose} :: \text{Ord } a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b \]
\[ \text{choose } (x, v) (y, w) \quad | \quad x > y = v \]
\[ | \quad x < y = w \]
\[ | \quad \text{otherwise} = \text{error } "\text{I cannot decide!}" \]
Guarded Definitions

\[
\begin{align*}
\text{sign } x & \quad | \quad x > 0 \quad = \quad 1 \\
& \quad | \quad x == 0 \quad = \quad 0 \\
& \quad | \quad x < 0 \quad = \quad -1
\end{align*}
\]

\[
\text{choose} :: \text{Ord } a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b
\]

\[
\text{choose} (x, v) (y, w)
\]

| \quad x > y \quad = \quad v \\
| \quad x < y \quad = \quad w \\
| \quad \text{otherwise} \quad = \quad \text{error} \quad \text{"I cannot decide!"}

If no guard succeeds, the next pattern is tried:

\[
\text{take_while } p (x : xs) \mid p \ x \quad = \quad x : \text{take_while } p \ xs
\]

\[
\text{take_while } p \ xs \quad = \quad [ ]
\]
Guarded Definitions

\[
\text{sign } x \quad | \quad x > 0 = 1 \\
\quad | \quad x == 0 = 0 \\
\quad | \quad x < 0 = -1
\]

\[
\text{choose} :: \text{Ord } a \Rightarrow (a, b) \to (a, b) \to b
\]

\[
\text{choose} \ (x, v) \ (y, w) \\
\quad | \ x > y = v \\
\quad | \ x < y = w \\
\quad | \ otherwise = \text{error } "\text{I cannot decide!}" 
\]

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p \ (x : xs) \ | \ p \ x = x : \text{take\_while } p \ xs
\]

\[
\text{take\_while } p \ xs = []
\]

\[
\text{take\_while } (\ < 5) \ [1, 2, 3]
\]
Guarded Definitions

\[
\begin{align*}
    \text{sign } x &| \begin{array}{ll}
    x > 0 & = 1 \\
    x == 0 & = 0 \\
    x < 0 & = -1
    \end{array} \\
\end{align*}
\]

\[
\begin{align*}
    \text{choose :: Ord } a &\Rightarrow (a, b) \to (a, b) \to b \\
    \text{choose } (x, v) (y, w) &| x > y = v \\
    &| x < y = w \\
    &| \text{otherwise} = \text{error} "\text{I cannot decide!}" \\
\end{align*}
\]

If no guard succeeds, the next pattern is tried:

\[
\begin{align*}
    \text{take\_while } p (x : xs) &| p x = x : \text{take\_while } p xs \\
    \text{take\_while } p \, xs &\quad = [] \\
\end{align*}
\]

\[
\begin{align*}
    \text{take\_while } (\ < 5) \, [1, 2, 3] \\
    &\quad = \text{take\_while } (\ < 5) \, (1 : 2 : 3 : []) \\
\end{align*}
\]
Guarded Definitions

\[\text{sign } x \begin{cases} x > 0 & = 1 \\ x == 0 & = 0 \\ x < 0 & = -1 \end{cases}\]

\[\text{choose } :: \text{Ord } a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b\]

\[\text{choose } (x, v) (y, w) \begin{cases} x > y & = v \\ x < y & = w \\ \text{otherwise} & = \text{error } "I cannot decide!" \end{cases}\]

If no guard succeeds, the next pattern is tried:

\[\text{take\_while } p \ (x : xs) | p x = x : \text{take\_while } p \ xs\]

\[\text{take\_while } p \ xs = []\]

\[\text{take\_while } ( < 5) [1, 2, 3]\]

\[= \text{take\_while } ( < 5) (1 : 2 : 3 : [])\]

\[= 1 : \text{take\_while } ( < 5) (2 : 3 : [])\]
Guarded Definitions

\[
\begin{align*}
sign x & \mid x > 0 = 1 \\
& \mid x == 0 = 0 \\
& \mid x < 0 = -1
\end{align*}
\]

\[\text{choose :: Ord a} \Rightarrow (a,b) \rightarrow (a,b) \rightarrow b\]

\[\text{choose } (x,v)(y,w) \rightarrow \begin{cases} x > y & = v \\
& x < y & = w \\
& \text{otherwise} & = \text{error} "\text{I cannot decide!}" \end{cases}\]

If no guard succeeds, the next pattern is tried:

\[
\begin{align*}
take\_while \ p \ (x : xs) \mid p \ x & = x : take\_while \ p \ xs \\
take\_while \ p \ xs & = []
\end{align*}
\]

\[
\begin{align*}
take\_while \ (< 5) [1, 2, 3] & = take\_while \ (< 5) (1 : 2 : 3 : []) \\
& = 1 : take\_while \ (< 5) (2 : 3 : []) \\
& = 1 : 2 : take\_while \ (< 5) (3 : [])
\end{align*}
\]
Guar ded Definitions

\( \text{sign } x \mid x > 0 = 1 \\
\quad x == 0 = 0 \\
\quad x < 0 = -1 \)

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

\( \text{choose} (x, v) (y, w) \)
\( \quad | x > y = v \)
\( \quad | x < y = w \)
\( \quad | \text{otherwise} = \text{error} "I cannot decide!" \)

If no guard succeeds, the next pattern is tried:

\( \text{take\_while } p \ (x : xs) \mid p x = x : \text{take\_while } p \ xs \)

\( \text{take\_while } p \ xs \)
\( = [ ] \)

\( \text{take\_while } (< 5) [1, 2, 3] \)
\( = \text{take\_while } (< 5) (1 : 2 : 3 : [ ] ) \)
\( = 1 : \text{take\_while } (< 5) (2 : 3 : [ ] ) \)
\( = 1 : 2 : \text{take\_while } (< 5) (3 : [ ] ) \)
\( = 1 : 2 : 3 : \text{take\_while } (< 5) [ ] \)
Guarded Definitions

\[\text{sign } x \mid x > 0 = 1\]
\[x == 0 = 0\]
\[x < 0 = -1\]

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

\(\text{choose } (x, v) (y, w)\)
\[| x > y \quad = \quad v\]
\[| x < y \quad = \quad w\]
\[| \text{otherwise} \quad = \quad \text{error } "\text{I cannot decide!}"\]

If no guard succeeds, the next pattern is tried:

\(\text{take}_\text{while} \ p \ (x : xs) \mid p \ x = x : \text{take}_\text{while} \ p \ xs\)

\(\text{take}_\text{while} \ p \ xs\)
\[= [\ ]\]

\(\text{take}_\text{while} \ (\ < 5 \) [1, 2, 3]\)
\[= \text{take}_\text{while} \ (\ < 5 \) (1 : 2 : 3 : [ ]))\]
\[= 1 : \text{take}_\text{while} \ (\ < 5 \) (2 : 3 : [ ]))\]
\[= 1 : 2 : \text{take}_\text{while} \ (\ < 5 \) (3 : [ ]))\]
\[= 1 : 2 : 3 : \text{take}_\text{while} \ (\ < 5 \) [ ]\]
\[= 1 : 2 : 3 : [ ]\]
Guarded Definitions

\[
\begin{align*}
\text{sign } x & \mid x > 0 = 1 \\
& \mid x == 0 = 0 \\
& \mid x < 0 = -1
\end{align*}
\]

\[
\text{choose :: Ord } a \Rightarrow (a, b) \rightarrow (a, b) \rightarrow b
\]

\[
\text{choose } (x, v) (y, w) \\
\mid x > y = v \\
\mid x < y = w \\
\mid \text{otherwise} = \text{error } "\text{I cannot decide!"
}
\]

If no guard succeeds, the next pattern is tried:

\[
\text{take}_\text{while } p (x : xs) \mid p x = x : \text{take}_\text{while } p xs
\]

\[
\text{take}_\text{while } p xs = []
\]

\[
\text{take}_\text{while } (\lt 5) [1, 2, 3]
\]

\[
= \text{take}_\text{while } (\lt 5) (1 : 2 : 3 : [])
\]

\[
= 1 : \text{take}_\text{while } (\lt 5) (2 : 3 : [])
\]

\[
= 1 : 2 : \text{take}_\text{while } (\lt 5) (3 : [])
\]

\[
= 1 : 2 : 3 : \text{take}_\text{while } (\lt 5) []
\]

\[
= 1 : 2 : 3 : []
\]

\[
= [1, 2, 3]
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\begin{align*}
\text{take\_while } p \ (x : xs) \mid p \ x &= x : \text{take\_while } p \ xs \\
\text{take\_while } p \ xs &= []
\end{align*}
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\begin{align*}
take\_while\ p\ (x : xs)\ |\ p\ x &= x : take\_while\ p\ xs \\
take\_while\ p\ xs &= [] \\
take\_while\ ( < 5)\ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
\end{align*}
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
take\_while\ p\ (x:xs) | p\ x = x : take\_while\ p\ xs \\
take\_while\ p\ xs = []
\]

\[
take\_while\ (<5) [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] = take\_while\ (<5) (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p \ (x : xs) \mid p \ x = x : \text{take\_while } p \ xs
\]

\[
\text{take\_while } p \ xs = []
\]

\[
\text{take\_while } (\ < \ 5) \ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
\]

\[
= \text{take\_while } (\ < \ 5) \ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
\]

\[
= 1 : \text{take\_while } (\ < \ 5) \ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p \ (x : xs) \mid p \ x = x : \text{take\_while } p \ xs
\]
\[
\text{take\_while } p \ xs = [ ]
\]

\[
\text{take\_while } (\ < 5) [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
\]
\[
= \text{take\_while } (\ < 5) (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : \text{take\_while } (\ < 5) (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : \text{take\_while } (\ < 5) (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p (x : xs) | p x = x : \text{take\_while } p xs
\]

\[
\text{take\_while } p xs = [ ]
\]

\[
\text{take\_while } ( < 5 ) [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
\]

\[
= \text{take\_while } ( < 5 ) (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]

\[
\]

\[
\]

\[
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p \ (x : xs) \ | \ p \ x = x : \text{take\_while } p \ xs \\
\text{take\_while } p \ xs = []
\]

\[
\text{take\_while } (\ < 5) \ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \text{take\_while } (\ < 5) \ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : \text{take\_while } (\ < 5) \ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : \text{take\_while } (\ < 5) \ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : \text{take\_while } (\ < 5) \ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : \text{take\_while } (\ < 5) \ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [])
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\textit{take\_while} \; p \; (x : xs) \mid p \; x = \; x : \textit{take\_while} \; p \; xs
\]

\[
\textit{take\_while} \; p \; xs = \; [\;]
\]

\[
\textit{take\_while} \; (\; < \; 5) \; [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
\]


Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p (x : xs) \mid p x = x : \text{take\_while } p x s \\
\text{take\_while } p x s = [ ]
\]

\[
\text{take\_while } ( < 5) [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \text{take\_while } ( < 5) (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take\_while } p \ (x : xs) \ | \ p \ x = x : \text{take\_while } p \ xs
\]

\[
\text{take\_while } p \ xs = [ ]
\]

\[
\text{take\_while } (< 5) \ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6]
\]
\[
= \text{take\_while } (< 5) \ (1 : 2 : 3 : 2 : 3 : 4 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : \text{take\_while } (< 5) \ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : \text{take\_while } (< 5) \ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : 3 : \text{take\_while } (< 5) \ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : 3 : 2 : \text{take\_while } (< 5) \ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : 3 : 2 : 3 : \text{take\_while } (< 5) \ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : 3 : 2 : 3 : 4 : \text{take\_while } (< 5) \ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
\[
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : \text{take\_while } (< 5) \ (4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ])
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[ \text{take\_while } p \ (x : xs) \ | \ p \ x = x : \text{take\_while } p \ xs \]
\[ \text{take\_while } p \ xs \quad = \ [ ] \]

\[ \text{take\_while } ( < 5) \ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \]
\[ = \text{take\_while } ( < 5) \ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : \text{take\_while } ( < 5) \ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : \text{take\_while } ( < 5) \ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : 3 : \text{take\_while } ( < 5) \ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : 3 : 2 : \text{take\_while } ( < 5) \ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : 3 : 2 : 3 : \text{take\_while } ( < 5) \ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : 3 : 2 : 3 : 4 : \text{take\_while } ( < 5) \ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : 3 : 2 : 3 : 4 : 3 : \text{take\_while } ( < 5) \ (4 : 5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
\[ = 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : \text{take\_while } ( < 5) \ (5 : 4 : 3 : 4 : 5 : 6 : [ ] ) \]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
\text{take}_\text{while} \ p \ (x : xs) \ | \ p \ x = x : \text{take}_\text{while} \ p \ xs \\
\text{take}_\text{while} \ p \ xs = []
\]

\[
\text{take}_\text{while} \ (<5) \ [1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 3, 4, 5, 6] \\
= \text{take}_\text{while} \ (<5) \ (1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : \text{take}_\text{while} \ (<5) \ (2 : 3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : \text{take}_\text{while} \ (<5) \ (3 : 2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : \text{take}_\text{while} \ (<5) \ (2 : 3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : \text{take}_\text{while} \ (<5) \ (3 : 4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : 3 : \text{take}_\text{while} \ (<5) \ (4 : 3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : 3 : 4 : \text{take}_\text{while} \ (<5) \ (3 : 4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : \text{take}_\text{while} \ (<5) \ (4 : 5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : \text{take}_\text{while} \ (<5) \ (5 : 4 : 3 : 4 : 5 : 6 : []) \\
= 1 : 2 : 3 : 2 : 3 : 4 : 3 : 4 : []
\]
Guarded Definitions — Fall-Through

If no guard succeeds, the next pattern is tried:

\[
take\textunderscore while\ p\ (x:\ xs)\ |\ p\ x\ =\ x:\ take\textunderscore while\ p\ xs
\]
\[
take\textunderscore while\ p\ xs\ =\ []
\]

\[
take\textunderscore while\ (\ <\ 5)\ [1,\ 2,\ 3,\ 2,\ 3,\ 4,\ 3,\ 4,\ 5,\ 4,\ 3,\ 4,\ 5,\ 6]
\]
\[
=\ take\textunderscore while\ (\ <\ 5)\ (1:2:3:2:3:4:3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:\ take\textunderscore while\ (\ <\ 5)\ (2:3:2:3:4:3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:\ take\textunderscore while\ (\ <\ 5)\ (3:2:3:4:3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:\ take\textunderscore while\ (\ <\ 5)\ (2:3:4:3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:2:\ take\textunderscore while\ (\ <\ 5)\ (3:4:3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:2:3:\ take\textunderscore while\ (\ <\ 5)\ (4:3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:2:3:4:\ take\textunderscore while\ (\ <\ 5)\ (3:4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:2:3:4:3:\ take\textunderscore while\ (\ <\ 5)\ (4:5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:2:3:4:3:4:\ take\textunderscore while\ (\ <\ 5)\ (5:4:3:4:5:6:[])
\]
\[
=\ 1:2:3:2:3:4:3:4:[]
\]
\[
=\ [1,\ 2,\ 3,\ 2,\ 3,\ 4,\ 3,\ 4]
\]
case Expressions

\[
\text{sign } x = \text{case } \text{compare } x \ 0 \ \text{of} \\
\quad \text{GT} \to 1 \\
\quad \text{EQ} \to 0 \\
\quad \text{LT} \to -1
\]
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`:

```haskell
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> 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 => a -> a -> Ordering

Another example:

choose (x, v) (y, w) = case compare x y of
    GT -> v
    LT -> w
    EQ -> error "I cannot decide!"
if ... then ... else ... and case Expressions

The type \textit{Bool} can be considered as a two-element enumeration type:

\begin{verbatim}
data Bool = False | True
\end{verbatim}
if ... then ... else ... and case Expressions

The type $Bool$ can be considered as a two-element enumeration type:

\[
data Bool = \text{False} \mid \text{True}\]

Conditional expressions are “syntactic sugar” for case expressions over $Bool$:

\[
\begin{align*}
\text{if} & \quad \text{condition} \\
\text{then} & \quad \text{expr1} \\
\text{else} & \quad \text{expr2}
\end{align*}
\]
\[
\equiv
\begin{align*}
\text{case} & \quad \text{condition of} \\
\text{True} & \quad \rightarrow \quad \text{expr1} \\
\text{False} & \quad \rightarrow \quad \text{expr2}
\end{align*}
\]
if ... then ... else ...

and case Expressions

The type \textit{Bool} can be considered as a two-element enumeration type:

\texttt{data Bool = False | True}

Conditional expressions are “syntactic sugar” for case expressions over \textit{Bool}:

\[\begin{array}{l}
\text{if condition then expr1 else expr2} \\
\end{array} \quad \equiv \quad \begin{array}{l}
\text{case condition of} \\
\text{True } \rightarrow \text{expr1} \\
\text{False } \rightarrow \text{expr2}
\end{array}\]

Two ways of defining functions:

\textit{Pattern Matching}

\[\begin{array}{l}
\text{not True } = \text{False} \\
\text{not False } = \text{True}
\end{array}\]

\textit{case}

\[\begin{array}{l}
\text{not } b = \text{case } b \text{ of} \\
\text{True } \rightarrow \text{False} \\
\text{False } \rightarrow \text{True}
\end{array}\]
Case Expressions are “Anonymous” Pattern Matching

\[\text{commaWords} :: [\text{String}] \rightarrow \text{String}\]
\[\text{commaWords} \ [\ ] = [\ ]\]
\[\text{commaWords} \ (x : xs) = x \ + \ \text{case} \ xs \ \text{of}\]
\[\ [\ ] \rightarrow [\ ]\]
\[\_ \rightarrow "," : \text{commaWords} \ xs\]
case Expressions are “Anonymous” Pattern Matching

case Expr essions ar e “Anonymous” P attern Matching

case Expr essions ar e “Anonymous” P attern Matching

\[
\text{commaWords} :: [\text{String}] \rightarrow \text{String}
\]

\[
\text{commaWords} [ ] = [ ]
\]

\[
\text{commaWords} (x : xs) = x \uplus \text{case } xs \text{ of}
\]

\[
[] \rightarrow []
\]

\[
_ \rightarrow "", " : \text{commaWords } xs
\]

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

```haskell
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ case xs of
                         [ ] → []
                         _ → ""," : commaWords xs
```

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

```haskell
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
```
**case Expressions are “Anonymous” Pattern Matching**

```haskell
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ case xs of
  [ ] → []
  _ → "," : commaWords xs
```

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

```haskell
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs

commaWordsAux [] = []
commaWordsAux xs = "," : commaWords xs
```
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.:

```haskell
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
    where
        commaWordsAux [] = []
        commaWordsAux xs = "," : commaWords xs
```
where Clauses

If an auxiliary definition is used only locally, it should be inside a local definition, e.g.:

```
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
where
  commaWordsAux [] = []
  commaWordsAux xs = "," : commaWords xs
```

where clauses are visible only within their enclosing clause, here “commaWords (x : xs) = …”
where Clauses

If an auxiliary definition is used only locally, it should be inside a local definition, e.g.:

```haskell
commaWords :: [String] → String
commaWords [] = []
commaWords (x : xs) = x ++ commaWordsAux xs
where
    commaWordsAux [] = []
    commaWordsAux xs = "," : commaWords xs
```

where clauses are visible only within their enclosing clause, here “commaWords (x : xs) = …”

where clauses are visible within all guards:

```haskell
f x y | y > z = ...
     | y == z = ...
     | y < z = ...
where z = x * x
```
let Expressions

Local definitions can also be part of expressions:

\[
f k n = \text{let } m = k \mod n \\
\text{in if } m == 0 \\
\text{then } n \\
\text{else } f n m
\]
let Expressions

Local definitions can also be part of expressions:

```plaintext
f k n = let m = k \text{ `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:

```haskell
f k n = let m = k `mod` n
    in if m == 0
        then n
        else f n m
```

```haskell
h x y = let x2 = x * x
        y2 = y * y
    in sqrt (x2 + y2)
```

Definitions can use **pattern bindings**:

```haskell
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:

\[
\text{f k n} = \text{let } m = k \mod n \\
\text{in if } m == 0 \\
\text{then } n \\
\text{else } f n m
\]

\[
\text{h x y} = \text{let } x2 = x \times x \\
\text{y2} = y \times y \\
\text{in } \sqrt{(x2 + y2)}
\]

Definitions can use **pattern bindings:**

\[
\text{g k n} = \text{let (d,m) = divMod k n} \\
\text{in if } d == 0 \\
\text{then } [m] \\
\text{else } g d n ++ [m]
\]

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 \( \textit{bindings} \text{ in } \textit{expression} \)
  
is an \textit{expression}

- \textit{fname patterns guardedRHSs where bindings} 
  
is a clause that is part of a \textit{definition}
let or where?

• let bindings in expression
  is an expression

• fname patterns guarded RHSs 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 guarded RHSs 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

```haskell
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

\[
\text{take} \quad :: \quad \text{Int} \to \text{[a]} \to \text{[a]}
\]
\[
\text{take} 0 \_ \quad = \quad \text{[]} \\
\text{take} \_ \text{[]} \quad = \quad \text{[]} \\
\text{take} n \ (x:xs) \quad | \quad n>0 \quad = \quad x : \text{take} \ (n-1) \ xs \\
\text{take} \_ \_ \quad = \quad \text{error} \ "\text{take: negative argument}" \\
\]

\[
\text{drop} \quad :: \quad \text{Int} \to \text{[a]} \to \text{[a]}
\]
\[
\text{drop} 0 \ xs \quad = \quad xs \\
\text{drop} \_ \text{[]} \quad = \quad \text{[]} \\
\text{drop} n \ (_{:xs}) \quad | \quad n>0 \quad = \quad \text{drop} \ (n-1) \ xs \\
\text{drop} \_ \_ \quad = \quad \text{error} \ "\text{drop: negative argument}" \\
\]

\[
\text{splitAt} \quad :: \quad \text{Int} \to \text{[a]} \to (\text{[a]}, \text{[a]})
\]
\[
\text{splitAt} 0 \ xs \quad = \quad ([],xs) \\
\text{splitAt} \_ \text{[]} \quad = \quad ([],[[]]) \\
\text{splitAt} n \ (x:xs) \quad | \quad n>0 \quad = \quad (x:xs',xs'') \\
\quad \quad \text{where} \ (xs',xs'') \quad = \quad \text{splitAt} \ (n-1) \ xs \\
\text{splitAt} \_ \_ \quad = \quad \text{error} \ "\text{splitAt: negative argument}" \
\]
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 '!!')
= take 3 ('!!' : repeat '!!')
= '!!' : take (3 - 1) (repeat '!!')
Separation of Concerns: Generation and Consumption

replicate 3 '!'  
= take 3 (repeat '!')  
= take 3 ('!' : repeat '!')  
= '!' : take (3 - 1) (repeat '!')  
= '!' : take 2 (repeat '!')  

-- replicate  
-- repeat  
-- take (iii)  
-- subtraction
Separation of Concerns: Generation and Consumption

replicate 3 '!!'
= take 3 (repeat '!!')
= take 3 ('!!' : repeat '!!')
= '!!' : take (3 - 1) (repeat '!!')
= '!!' : take 2 (repeat '!!')
= '!!' : take 2 ('!!' : repeat '!!')

-- replicate
-- repeat
-- take (iii)
-- subtraction
-- repeat
Separation of Concerns: Generation and Consumption

replicate 3 '!'
= take 3 (repeat '!')
= take 3 ('!' : repeat '!')
= '!' : take (3 - 1) (repeat '!')
= '!' : take 2 (repeat '!')
= '!' : '!' : take (2 - 1) (repeat '!')
Separation of Concerns: Generation and Consumption

replicate 3 '!!'
= take 3 (repeat '!!')
= take 3 ('!!' : repeat '!!')
= '!!' : take (3 - 1) (repeat '!!')
= '!!' : take 2 (repeat '!!')
= '!!' : take 2 ('!!' : repeat '!!')
= '!!' : '!!' : take (2 - 1) (repeat '!!')
= '!!' : '!!' : take 1 (repeat '!!')
Separation of Concerns: Generation and Consumption

replicate 3 ‘!’
= take 3 (repeat ‘!’)
= take 3 (‘!’ : repeat ‘!’)
= ‘!’ : take (3 - 1) (repeat ‘!’)
= ‘!’ : take 2 (repeat ‘!’)
= ‘!’ : ‘!’ : take (2 - 1) (repeat ‘!’)
= ‘!’ : ‘!’ : take 1 (repeat ‘!’)
= ‘!’ : ‘!’ : take 1 (‘!’ : repeat ‘!’)

-- replicate
-- repeat
-- take (iii)
-- subtraction
-- repeat
-- take (iii)
-- subtraction
-- 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 '!'') 
= take 3 ('!' : repeat '!'') 
= '!' : take (3 - 1) (repeat '!'') 
= '!' : take 2 (repeat '!'') 
= '!' : take 2 ('!' : repeat '!'') 
= '!' : '!' : take (2 - 1) (repeat '!'') 
= '!' : '!' : take 1 (repeat '!'') 
= '!' : '!' : take 1 ('!' : repeat '!'') 
= '!' : '!' : '!' : take (1 - 1) (repeat '!'') 
= '!' : '!' : '!' : take 0 (repeat '!'') 
= '!' : '!' : '!' : [] 

-- replicate 
-- repeat 
-- take (iii) 
-- subtraction 
-- repeat 
-- take (iii) 
-- subtraction 
-- repeat 
-- take (iii) 
-- 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 \textit{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 \textit{dropWhile} \((< 5)\) \([1,2,3]\):

- \(p = (< 5)\)
- \(xs = \)
- \(x = \)
- \(xs' = \)
as-Patterns

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs') |
     p x = dropWhile p xs'
  otherwise = xs

Consider matching of the third clause against dropWhile (< 5) [1,2,3]:

- \( p = (< 5) \)
- \( xs = [1,2,3] \)
- \( x = \)
- \( xs' = \)
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 \( \text{dropWhile} \ (< 5) \ [1,2,3] \):

- \( p = (< 5) \)
- \( xs = [1,2,3] \)
- \( x = 1 \)
- \( xs' = \)
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 \(\text{dropWhile} (~< 5) ~[1,2,3]\):

- \(p = (~< 5)\)
- \(xs = [1,2,3]\)
- \(x = 1\)
- \(xs' = [2,3]\)
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 \textit{dropWhile} \((< 5)\) \([1,2,3]\):

\begin{itemize}
  \item \(p = (< 5)\)
  \item \(xs = [1,2,3]\)
  \item \(x = 1\)
  \item \(xs' = [2,3]\)
  \item \(p x = (< 5) 1 = 1 < 5 = \textbf{True}\)
\end{itemize}
as-Patterns

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs) |
  p x = dropWhile p xs'
  otherwise = x

Consider matching of the third clause against \textit{dropWhile} \( (< 5) \ [1,2,3] \): 

- \( p = ( < 5) \)
- \( x s = [1,2,3] \)
- \( x = 1 \)
- \( x s' = [2,3] \)
- \( p \ x = ( < 5) 1 = 1 < 5 = \text{True} \)

Therefore: \textit{dropWhile} \( (< 5) \ [1,2,3] = \)
as-Patterns

\[
\text{dropWhile} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
dropWhile \ p \ [] = [] \\
dropWhile \ p \ \text{xs@(x:xsp')} |
| \ p \ x \ = \ \text{dropWhile} \ p \ \text{xs'} \\
| \ \text{otherwise} \ = \ \text{xs}
\]

Consider matching of the third clause against \( \text{dropWhile} \ (< 5) \ [1,2,3] \):

- \( p = (\ < 5) \)
- \( xs = [1,2,3] \)
- \( x = 1 \)
- \( xs' = [2,3] \)
- \( p \ x = (< 5) \ 1 = 1 < 5 = \text{True} \)

Therefore: \( \text{dropWhile} \ (< 5) \ [1,2,3] = \text{dropWhile} \ (< 5) \ [2,3] \)
Consider matching of the third clause against \( \text{dropWhile} (< 5) [5, 4, 3] \):
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 \( \text{dropWhile} \ (< 5) \ [5,4,3] \):

- \( p = (\ < 5) \)
- \( xs = [5,4,3] \)
- \( x = \)
- \( xs' = \)
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 \(\text{dropWhile} \ (< 5) \ [5,4,3]\):

• \(p = (< 5)\)

• \(xs = [5,4,3]\)

• \(x = 5\)

• \(xs' = \)
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 \textit{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

\[
dropWhile :: (a -> \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
\[
dropWhile \ p \ [] \ = \ []
\]
\[
dropWhile \ p \ \text{x}\@\text{x}s@\text{x}s'\)
\quad | \ p \ x \ = \ \text{dropWhile} \ p \ \text{x}s'
\quad | \ \text{otherwise} \ = \ \text{x}s
\]

Consider matching of the third clause against \text{dropWhile} \ (\ < \ 5) \ [5,4,3]:

- \(p = (\ < \ 5)\)

- \(x\ s = [5,4,3]\)

- \(x = 5\)

- \(x\ s' = [4,3]\)

- \(p \ x = (\ < \ 5) \ 5 = 5 < 5 = \text{False}\)

Therefore: \text{dropWhile} \ (\ < \ 5) \ [5,4,3] =
as-Patterns — 2

\[
\text{dropWhile} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]
\[
\text{dropWhile } p \ [\] = []
\]
\[
\text{dropWhile } p \ (x : xs') =
\begin{align*}
& \quad \text{p } x \quad = \text{dropWhile } p \ xs' \\
& \quad \text{otherwise } = \ xs
\end{align*}
\]

Consider matching of the third clause against \text{dropWhile} \ (< 5) \ [5,4,3]:

- \( p = (< 5) \)
- \( xs = [5,4,3] \)
- \( x = 5 \)
- \( xs' = [4,3] \)
- \( p \ x = (< 5) \ 5 = 5 < 5 = \text{False} \)

Therefore: \text{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

```haskell
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 (⊗) [x₁, x₂, x₃, x₄, x₅]
foldr1

foldr1 :: (a -> a -> a) -> [a] -> a

foldr1 (⊗) [x] = x

foldr1 (⊗) (x:xs) = x ⊗ (foldr1 (⊗) xs)

foldr1 (⊗) [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr1 (⊗) [x₂, x₃, x₄, x₅])
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 (⊗) [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr1 (⊗) [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr1 (⊗) [x₃, x₄, x₅]))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (foldr1 (⊗) [x₄, x₅])))
foldr1

foldr1 :: (a -> a -> a) -> [a] -> a

foldr1 (⊗) [x] = x

foldr1 (⊗) (x:xs) = x ⊗ (foldr1 (⊗) xs)

foldr1 (⊗) [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr1 (⊗) [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr1 (⊗) [x₃, x₄, x₅]))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (foldr1 (⊗) [x₄, x₅])))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (foldr1 (⊗) [x₅])))))
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]))
= x_1 ⊗ (x_2 ⊗ (x_3 ⊗ (foldr1 (⊗) [x_4, x_5])))
= x_1 ⊗ (x_2 ⊗ (x_3 ⊗ (x_4 ⊗ (foldr1 (⊗) [x_5]))))
= 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)
foldrX

foldrX :: (a → b → b) → b → [a] → b
foldrX (*** ) z [] = z
foldrX (*** ) z (x:xs) = x *** (foldrX (*** ) z xs)
foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr (⊗) z [] = z

foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅]
foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr (⊗) z [] = z

foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅ ]
= x₁ ⊗ (foldr (⊗) z [x₂, x₃, x₄, x₅ ])

foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr (⊗) z [] = z

foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr (⊗) z [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr (⊗) z [x₃, x₄, x₅]))
foldr

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (⊗) z [] = z
foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr (⊗) z [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr (⊗) z [x₃, x₄, x₅]))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (foldr (⊗) z [x₄, x₅])))
foldr

foldr :: (a → b → b) → b → [a] → b

foldr (⊗) z [] = z

foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr (⊗) z [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr (⊗) z [x₃, x₄, x₅]))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (foldr (⊗) z [x₄, x₅])))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (foldr (⊗) z [x₅]))))
foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr (⊗) z [] = z

foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr (⊗) z [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr (⊗) z [x₃, x₄, x₅]))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (foldr (⊗) z [x₄, x₅])))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (foldr (⊗) z [x₅]))))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (x₅ ⊗ (foldr (⊗) z [])))))
foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr (⊗) z [] = z
foldr (⊗) z (x:xs) = x ⊗ (foldr (⊗) z xs)

foldr (⊗) z [x₁, x₂, x₃, x₄, x₅]
= x₁ ⊗ (foldr (⊗) z [x₂, x₃, x₄, x₅])
= x₁ ⊗ (x₂ ⊗ (foldr (⊗) z [x₃, x₄, x₅]))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (foldr (⊗) z [x₄, x₅])))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (foldr (⊗) z [x₅]))))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (x₅ ⊗ (foldr (⊗) z [])))))
= x₁ ⊗ (x₂ ⊗ (x₃ ⊗ (x₄ ⊗ (x₅ ⊗ 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:

\[ \text{limit} = 10 ^ 2 \]
Unfolding Definitions

A simple definition:

\[ \text{limit} = 10^2 \]

Expanding this definition:

\[ 4 \times (\text{limit} + 1) \]
Unfolding Definitions

A simple definition:

\[ \text{limit} = 10^2 \]

Expanding this definition:

\[ 4 \times (\text{limit} + 1) \]
\[ = 4 \times ((10^2) + 1) \]
Unfolding Definitions

A simple definition:

\( \text{limit} = 10^2 \)

Expanding this definition:

\[
4 \times (\text{limit} + 1) \\
= 4 \times ((10^2) + 1) \\
= \ldots
\]
Unfolding Definitions

A simple definition:

\( \text{limit} = 10^2 \)

Expanding this definition:

\[ 4 \times (\text{limit} + 1) \]
\[ = 4 \times ((10^2) + 1) \]
\[ = \ldots \]

Another definition:

\( \text{concat} = \text{foldr} \ (++) \ [ ] \)
Unfolding Definitions

A simple definition:

\( \text{limit} = 10^2 \)

Expanding this definition:

\[
4 \times (\text{limit} + 1)
= 4 \times ((10^2) + 1)
= \ldots
\]

Another definition:

\( \text{concat} = \text{foldr} \ (\ + \) \ [ \ ] \)

Expanding this definition:

\( \text{concat} \ [\ [1,2,3], [4,5]] \)
Unfolding Definitions

A simple definition:

\[ \text{limit} = 10^2 \]

Expanding this definition:

\[ 4 \cdot (\text{limit} + 1) = 4 \cdot ((10^2) + 1) = \ldots \]

Another definition:

\[ \text{concat} = \text{foldr} \, (\text{\texttt{++}}) \, [] \]

Expanding this definition:

\[ \text{concat} \, [[1,2,3], [4,5]] = (\text{foldr} \, (\text{\texttt{++}}) \, []) \, [[1,2,3], [4,5]] = \ldots \]
Enumeration Type Definitions

data Bool = False | True deriving (Eq, Ord, Read, Show)
data Ordering = LT | EQ | GT deriving (Eq, Ord, Read, Show)

data Suit = Diamonds | Hearts | Spades | Clubs deriving (Eq, Ord)

Pattern matching:

not False = True
not True = False

lexicalCombineOrdering :: Ordering → Ordering → Ordering
lexicalCombineOrdering LT _ = LT
lexicalCombineOrdering EQ x = x
lexicalCombineOrdering GT _ = GT
Simple data Type Definitions

```haskell
data Point = Pt Int Int  deriving ( Eq )  --- screen coordinates

data Transport = Feet
    | Bike
    | Train Int  --- price in cent

This defines at the same time data constructors:

Pt :: Int \rightarrow Int \rightarrow Point

Feet :: Transport
Bike :: Transport
Train :: Int \rightarrow Transport

Pattern matching:

addPt ( Pt x1 y1 ) ( Pt x2 y2 ) = Pt ( x1 + x2 ) ( y1 + y2 )

cost Feet = 0

cost Bike = 0

cost ( Train Int ) = Int
```
Simple Polymorphic data Type Definitions

The prelude type constructors \texttt{Maybe}, \texttt{Either}, \texttt{Complex} are defined as follows:

\begin{verbatim}
data Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show) 
data Either a b = Left a | Right b 

data Complex r = r :+ r deriving (Eq, Read, Show)
\end{verbatim}

This defines at the same time data constructors:

\begin{verbatim}
Nothing :: Maybe a 
Just :: a \rightarrow{} Maybe a 

Left :: a \rightarrow{} Either a b 
Right :: b \rightarrow{} Either a b 

( :+ ) :: r \rightarrow{} r \rightarrow{} Complex r
\end{verbatim}