Haskell Basics

Johannes Borgström
johannes.borgstrom@it.uu.se

Program Design and Data Structures

Based on notes by Tjark Weber, Lars-Henrik Eriksson, Pierre Flener, Sven-Olof Nyström

Haskell Basics

Type `Integer`

Value syntax: one or more digits. Negative numbers are preceded by `-`

Type `Integer`

Value syntax: one or more digits. Negative numbers are preceded by `-`

Examples: `1`, `-25`

Type `Integer`

Value syntax: one or more digits. Negative numbers are preceded by `-`

Examples: `1`, `-25`

Functions	Type[1]	Semantics

Type `Integer`

Value syntax: one or more digits. Negative numbers are preceded by `-`

Examples: `1`, `-25`

Functions	Type[1]	Semantics
`+`, `-`, `*`, `div`	`Integer->Integer->Integer`	the four basic arithmetic operations

Type `Integer`

Value syntax: one or more digits. Negative numbers are preceded by `-`

Examples: `1`, `-25`

Functions	Type[1]	Semantics
`+`, `-`, `*`, `div`	`Integer->Integer->Integer`	the four basic arithmetic operations
`mod`	`Integer->Integer->Integer`	modulo (e.g., `mod 7 3` $\longrightarrow$ `1`)

Type Integer

Value syntax: one or more digits. Negative numbers are preceded by -

Examples: 1, -25

Functions	Type[1]	Semantics
+, -, *, div	Integer->Integer->Integer	the four basic arithmetic operations
mod	Integer->Integer->Integer	modulo (e.g., mod 7 3 $\longrightarrow$ 1)
==, /=	Integer->Integer->Bool	equality, inequality

Type Integer

Value syntax: one or more digits. Negative numbers are preceded by `-`

Examples: `1`, `-25`

Functions	Type[1]	Semantics
`+`, `-`, `*`, `div`	`Integer->Integer->Integer`	the four basic arithmetic operations
`mod`	`Integer->Integer->Integer`	modulo (e.g., `mod 7 3` $\longrightarrow$ `1`)
`==`, `/=`	`Integer->Integer->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Integer->Integer->Bool`	comparison

Type Integer

Value syntax: one or more digits. Negative numbers are preceded by -

Examples: 1, -25

Functions	Type[1]	Semantics
+, -, *, div	Integer->Integer->Integer	the four basic arithmetic operations
mod	Integer->Integer->Integer	modulo (e.g., mod 7 3 ⟶ 1)
==, /=	Integer->Integer->Bool	equality, inequality
<, <=, >, >=	Integer->Integer->Bool	comparison
-, negate	Integer->Integer	integer negation (e.g., -(2-3) ⟶ 1)

Type Integer

Value syntax: one or more digits. Negative numbers are preceded by -

Examples: 1, -25

Functions	Type[1]	Semantics
+, -, *, div	Integer->Integer->Integer	the four basic arithmetic operations
mod	Integer->Integer->Integer	modulo (e.g., mod 7 3 $\longrightarrow$ 1)
==, /=	Integer->Integer->Bool	equality, inequality
<, <=, >, >=	Integer->Integer->Bool	comparison
-, negate	Integer->Integer	integer negation (e.g., -(2-3) $\longrightarrow$ 1)
abs	Integer->Integer	absolute value (e.g., abs (-1) $\longrightarrow$ 1)

Type Integer

Value syntax: one or more digits. Negative numbers are preceded by -

Examples: 1, -25

Functions	Type[1]	Semantics
+, -, *, div	Integer->Integer->Integer	the four basic arithmetic operations
mod	Integer->Integer->Integer	modulo (e.g., mod 7 3 $\longrightarrow$ 1)
==, /=	Integer->Integer->Bool	equality, inequality
<, <=, >, >=	Integer->Integer->Bool	comparison
-, negate	Integer->Integer	integer negation (e.g., -(2-3) $\longrightarrow$ 1)
abs	Integer->Integer	absolute value (e.g., abs (-1) $\longrightarrow$ 1)

[1] We'll talk more about the type of these functions later.

Type `Int`

Haskell also provides a type `Int`, with similar value syntax and operations as `Integer`.
What's the difference?

Type `Int`

Haskell also provides a type `Int`, with similar value syntax and operations as `Integer`.
What's the difference?

Computers have (large, but) finite amounts of memory.
Therefore, they can only represent a finite subset of all integers.

Type `Int`

Haskell also provides a type `Int`, with similar value syntax and operations as `Integer`.
What's the difference?

Computers have (large, but) finite amounts of memory.
Therefore, they can only represent a finite subset of all integers.

`Int` uses a **fixed** number of $N \geq 30$ bits (often $N = 32$ or $N = 64$) to represent each number.
It can only represent numbers from -2^{N-1} to $2^{N-1} - 1$.

Type `Int`

Haskell also provides a type `Int`, with similar value syntax and operations as `Integer`.
What's the difference?

Computers have (large, but) finite amounts of memory.
Therefore, they can only represent a finite subset of all integers.

`Int` uses a **fixed** number of $N \geq 30$ bits (often $N = 32$ or $N = 64$) to represent each number.
It can only represent numbers from -2^{N-1} to $2^{N-1} - 1$.

`Integer` is an **arbitrary precision** type: for each number, it uses as many bits as needed.
Thus, it will hold any number no matter how big, up to the limit of your machine's memory.

Integer Overflow

When an arithmetic operation attempts to create a numeric value that is too large to be represented, an **integer overflow** occurs.

Integer Overflow

When an arithmetic operation attempts to create a numeric value that is too large to be represented, an **integer overflow** occurs.

```
Prelude> 2 ^ 63 :: Int
-9223372036854775808
```

Integer Overflow

When an arithmetic operation attempts to create a numeric value that is too large to be represented, an **integer overflow** occurs.

```
Prelude> 2 ^ 63 :: Int
-9223372036854775808
```

Note the incorrect result!
Integer overflows are a frequent cause of programming errors.

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter e followed by an integer). Negative numbers are preceded by -

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter `e` followed by an integer). Negative numbers are preceded by `-`

Examples: `1.0`, `3e4` (= `30000.0`), `3.4e-2` (= `0.034`)

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter `e` followed by an integer). Negative numbers are preceded by `-`

Examples: `1.0`, `3e4` (= `30000.0`), `3.4e-2` (= `0.034`)

Functions	Type[1]	Semantics

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter `e` followed by an integer). Negative numbers are preceded by `-`

Examples: `1.0`, `3e4` (= `30000.0`), `3.4e-2` (= `0.034`)

Functions	Type[1]	Semantics
`+`, `-`, `*`, `/`	`Double->Double->Double`	the four basic arith. operations

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter `e` followed by an integer). Negative numbers are preceded by `-`

Examples: `1.0`, `3e4` (= `30000.0`), `3.4e-2` (= `0.034`)

Functions	Type[1]	Semantics
`+`, `-`, `*`, `/`	`Double->Double->Double`	the four basic arith. operations
`==`, `/=`	`Double->Double->Bool`	equality, inequality

Type Double

Value syntax: one or more digits with decimal point and/or in scientific notation (letter e followed by an integer). Negative numbers are preceded by -

Examples: 1.0, 3e4 (= 30000.0), 3.4e-2 (= 0.034)

Functions	Type[1]	Semantics
+, -, *, /	Double->Double->Double	the four basic arith. operations
==, /=	Double->Double->Bool	equality, inequality
<, <=, >, >=	Double->Double->Bool	comparison

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter `e` followed by an integer). Negative numbers are preceded by `-`

Examples: `1.0`, `3e4` ($=$ `30000.0`), `3.4e-2` ($=$ `0.034`)

Functions	Type[1]	Semantics
`+`, `-`, `*`, `/`	`Double->Double->Double`	the four basic arith. operations
`==`, `/=`	`Double->Double->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Double->Double->Bool`	comparison
`-`, `negate`	`Double->Double`	floating-point negation

Type `Double`

Value syntax: one or more digits with decimal point and/or in scientific notation (letter `e` followed by an integer). Negative numbers are preceded by `-`

Examples: `1.0`, `3e4` (= `30000.0`), `3.4e-2` (= `0.034`)

Functions	Type[1]	Semantics
`+`, `-`, `*`, `/`	`Double->Double->Double`	the four basic arith. operations
`==`, `/=`	`Double->Double->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Double->Double->Bool`	comparison
`-`, `negate`	`Double->Double`	floating-point negation
`abs`	`Double->Double`	absolute value

Type Double

Value syntax: one or more digits with decimal point and/or in scientific notation (letter e followed by an integer). Negative numbers are preceded by -

Examples: `1.0`, `3e4` (= `30000.0`), `3.4e-2` (= `0.034`)

Functions	Type[1]	Semantics
`+`, `-`, `*`, `/`	`Double->Double->Double`	the four basic arith. operations
`==`, `/=`	`Double->Double->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Double->Double->Bool`	comparison
`-`, `negate`	`Double->Double`	floating-point negation
`abs`	`Double->Double`	absolute value

[1] We'll talk more about the type of these functions later.

Floating-Point Numbers

Computers have (large, but) finite amounts of memory.
Therefore, only some real numbers can be represented exactly (e.g., 0.0),
while others (such as $\pi = 3.14159\ldots$) can only be represented **approximately**,
i.e., with a certain (finite) precision.

Floating-Point Numbers

Computers have (large, but) finite amounts of memory.
Therefore, only some real numbers can be represented exactly (e.g., 0.0),
while others (such as $\pi = 3.14159\ldots$) can only be represented **approximately**,
i.e., with a certain (finite) precision.

Watch out! This has strange consequences:

```
Prelude> 1.0 + 1e16 == 1e16
True
```

Floating-Point Numbers

Computers have (large, but) finite amounts of memory.
Therefore, only some real numbers can be represented exactly (e.g., 0.0), while others (such as $\pi = 3.14159\ldots$) can only be represented **approximately**, i.e., with a certain (finite) precision.

Watch out! This has strange consequences:

```
Prelude> 1.0 + 1e16 == 1e16
True
```

Floating-Point Numbers

Computers have (large, but) finite amounts of memory.
Therefore, only some real numbers can be represented exactly (e.g., 0.0),
while others (such as $\pi = 3.14159\ldots$) can only be represented **approximately**,
i.e., with a certain (finite) precision.

Watch out! This has strange consequences:

```
Prelude> 1.0 + 1e16 == 1e16
True
```

Haskell also offers a type `Float` that has even less precision than `Double` (but uses less memory).
Don't use `Float` unless you Really Know What You Are Doing!

Equality of Floating-Point Numbers

Because of potential **rounding errors**, testing floating-point numbers for equality is usually not the right thing to do!

Equality of Floating-Point Numbers

Because of potential **rounding errors**, testing floating-point numbers for equality is usually not the right thing to do!

Is `1.0 + 1e16 - 1e16` *equal to* `1.0`? *Should it be?*

Equality of Floating-Point Numbers

Because of potential **rounding errors**, testing floating-point numbers for equality is usually not the right thing to do!

Is `1.0 + 1e16 - 1e16` *equal to* `1.0`? *Should it be?*

Say you have a problem, so you use floating-point. Now you have 2.000001 problems.

Stephan T. Lavavej, Software Development Engineer

Equality of Floating-Point Numbers

Because of potential **rounding errors**, testing floating-point numbers for equality is usually not the right thing to do!

Is `1.0 + 1e16 - 1e16` *equal to* `1.0`? *Should it be?*

Say you have a problem, so you use floating-point. Now you have 2.000001 problems.

Stephan T. Lavavej, Software Development Engineer

Often, a better approach is to determine the magnitude of potential rounding errors, and test whether two floating-point numbers differ by less than that:

Equality of Floating-Point Numbers

Because of potential **rounding errors**, testing floating-point numbers for equality is usually not the right thing to do!

Is `1.0 + 1e16 - 1e16` *equal to* `1.0`? *Should it be?*

Say you have a problem, so you use floating-point. Now you have 2.000001 problems.

Stephan T. Lavavej, Software Development Engineer

Often, a better approach is to determine the magnitude of potential rounding errors, and test whether two floating-point numbers differ by less than that:

```
Prelude> abs ((1.0 + 1e16 - 1e16) - 1.0) < 10.0
True
```

Type `String`

Value syntax: any sequence of characters enclosed in `"`

Within a string, `"` needs to be written `\"`, and `\` needs to be written `\\`.
(We say that `\` is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ", b, \ and c.

Type `String`

Value syntax: any sequence of characters enclosed in `"`

Within a string, `"` needs to be written `\"`, and \ needs to be written `\\`.
(We say that \ is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ”, b, \ and c.

Functions	Type[1]	Semantics

Type String

Value syntax: any sequence of characters enclosed in "

Within a string, " needs to be written \", and \ needs to be written \\.
(We say that \ is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ", b, \ and c.

Functions	Type[1]	Semantics
`==`, `/=`	`String->String->Bool`	equality, inequality

Type `String`

Value syntax: any sequence of characters enclosed in `"`

Within a string, `"` needs to be written `\"`, and `\` needs to be written `\\`.
(We say that `\` is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ", b, \ and c.

Functions	Type[1]	Semantics
`==`, `/=`	`String->String->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`String->String->Bool`	comparison (lexicographic order)

Type `String`

Value syntax: any sequence of characters enclosed in `"`

Within a string, `"` needs to be written `\"`, and \ needs to be written `\\`.
(We say that \ is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ", b, \ and c.

Functions	Type[1]	Semantics
`==`, `/=`	`String->String->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`String->String->Bool`	comparison (lexicographic order)
`++`	`String->String->String`	concatenation

Type `String`

Value syntax: any sequence of characters enclosed in `"`

Within a string, `"` needs to be written `\"`, and `\` needs to be written `\\`.
(We say that `\` is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ", b, \ and c.

Functions	Type[1]	Semantics
`==`, `/=`	`String->String->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`String->String->Bool`	comparison (lexicographic order)
`++`	`String->String->String`	concatenation
`length`	`String->Int`	length

Type String

Value syntax: any sequence of characters enclosed in `"`

Within a string, `"` needs to be written `\"`, and \ needs to be written `\\`.
(We say that \ is an **escape character**.)

Example: `"a\"b\\c"` is a string of 5 characters: a, ", b, \ and c.

Functions	Type[1]	Semantics
`==`, `/=`	`String->String->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`String->String->Bool`	comparison (lexicographic order)
`++`	`String->String->String`	concatenation
`length`	`String->Int`	length

[1] We'll talk more about the type of these functions later.

Type `String` (cont.)

Further functions for strings:

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character

Type `String` (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char

Type `String` (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char
`take`	`Int->String->String`	the string's first *n* characters

Type `String` (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char
`take`	`Int->String->String`	the string's first *n* characters
`drop`	`Int->String->String`	the string without its first *n* chars

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char
`take`	`Int->String->String`	the string's first n characters
`drop`	`Int->String->String`	the string without its first n chars
`!!`	`String->Int->Char`	the string's n^{th} character

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char
`take`	`Int->String->String`	the string's first n characters
`drop`	`Int->String->String`	the string without its first n chars
`!!`	`String->Int->Char`	the string's n^{th} character

[1] We'll talk more about the type of these functions later.

Type String (cont.)

Further functions for strings:

Functions	Type[1]	Semantics
`head`	`String->Char`	first character
`last`	`String->Char`	last character
`tail`	`String->String`	the string without its first char
`init`	`String->String`	the string without its last char
`take`	`Int->String->String`	the string's first *n* characters
`drop`	`Int->String->String`	the string without its first *n* chars
`!!`	`String->Int->Char`	the string's n^{th} character

[1] We'll talk more about the type of these functions later.

`String` is actually just an abbreviation for `[Char]`, i.e., lists of characters.

Type `Char`

Value syntax: a single character enclosed in single quotes

Type `Char`

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Type `Char`

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Type Char

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics

Type `Char`

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
`==`, `/=`	`Char->Char->Bool`	equality, inequality

Type `Char`

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
==, /=	`Char->Char->Bool`	equality, inequality
<, <=, >, >=	`Char->Char->Bool`	comparison (lexicographic order)

Type Char

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
`==`, `/=`	`Char->Char->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Char->Char->Bool`	comparison (lexicographic order)
`Data.Char.isLower`	`Char->Bool`	selects lower-case chars

Type Char

Value syntax: a single character enclosed in single quotes

`'` and \ need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
`==`, `/=`	`Char->Char->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Char->Char->Bool`	comparison (lexicographic order)
`Data.Char.isLower`	`Char->Bool`	selects lower-case chars
`Data.Char.isUpper`	`Char->Bool`	selects upper-case chars

Type Char

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
==, /=	`Char->Char->Bool`	equality, inequality
<, <=, >, >=	`Char->Char->Bool`	comparison (lexicographic order)
`Data.Char.isLower`	`Char->Bool`	selects lower-case chars
`Data.Char.isUpper`	`Char->Bool`	selects upper-case chars
`Data.Char.toLower`	`Char->Char`	converts to lower case

Type Char

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
`==`, `/=`	`Char->Char->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Char->Char->Bool`	comparison (lexicographic order)
`Data.Char.isLower`	`Char->Bool`	selects lower-case chars
`Data.Char.isUpper`	`Char->Bool`	selects upper-case chars
`Data.Char.toLower`	`Char->Char`	converts to lower case
`Data.Char.toUpper`	`Char->Char`	converts to upper case

Type Char

Value syntax: a single character enclosed in single quotes

`'` and `\` need to be escaped (as `\'` and `\\`, respectively).

Examples: `'a'`, `'0'`, `'\''`, `'\\'`

Functions	Type[1]	Semantics
`==`, `/=`	`Char->Char->Bool`	equality, inequality
`<`, `<=`, `>`, `>=`	`Char->Char->Bool`	comparison (lexicographic order)
`Data.Char.isLower`	`Char->Bool`	selects lower-case chars
`Data.Char.isUpper`	`Char->Bool`	selects upper-case chars
`Data.Char.toLower`	`Char->Char`	converts to lower case
`Data.Char.toUpper`	`Char->Char`	converts to upper case

[1] We'll talk more about the type of these functions later.

Character Encodings

A **character encoding** maps (a fixed set of) characters to numbers (or bit patterns, . . .) in order to facilitate data storage and transmission.

Character Encodings

A **character encoding** maps (a fixed set of) characters to numbers (or bit patterns, . . .) in order to facilitate data storage and transmission.

Haskell's characters are based on the **Unicode** encoding. Unicode is a standard that defines over 110,000 characters in most of the world's writing systems.

Character Encodings

A **character encoding** maps (a fixed set of) characters to numbers (or bit patterns, . . .) in order to facilitate data storage and transmission.

Haskell's characters are based on the **Unicode** encoding. Unicode is a standard that defines over 110,000 characters in most of the world's writing systems.

See `http://unicode-table.com/en/` for the Unicode table.

Type Conversions

There are various functions to **convert** between data of different type.

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`
`round`	`Double->Integer`
`truncate`	`Double->Integer`
`floor`	`Double->Integer`
`ceiling`	`Double->Integer`

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`
`round`	`Double->Integer`
`truncate`	`Double->Integer`
`floor`	`Double->Integer`
`ceiling`	`Double->Integer`

Function	Type[1]
`[ ]`[2]	`Char->String`

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`
`round`	`Double->Integer`
`truncate`	`Double->Integer`
`floor`	`Double->Integer`
`ceiling`	`Double->Integer`

Function	Type[1]
`[ ]`[2]	`Char->String`
`toEnum`	`Int->Char`
`fromEnum`	`Char->Int`

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`
`round`	`Double->Integer`
`truncate`	`Double->Integer`
`floor`	`Double->Integer`
`ceiling`	`Double->Integer`

Function	Type[1]
`[ ]`[2]	`Char->String`
`toEnum`	`Int->Char`
`fromEnum`	`Char->Int`
`show`	`Integer->String`
`show`	`Double->String`

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`
`round`	`Double->Integer`
`truncate`	`Double->Integer`
`floor`	`Double->Integer`
`ceiling`	`Double->Integer`

Function	Type[1]
`[ ]`[2]	`Char->String`
`toEnum`	`Int->Char`
`fromEnum`	`Char->Int`
`show`	`Integer->String`
`show`	`Double->String`
`read`	`String->Integer`
`read`	`String->Double`

Type Conversions

There are various functions to **convert** between data of different type.

Examples:

Function	Type[1]
`toInteger`	`Int->Integer`
`fromInteger`	`Integer->Int`
`fromInteger`	`Integer->Double`
`round`	`Double->Integer`
`truncate`	`Double->Integer`
`floor`	`Double->Integer`
`ceiling`	`Double->Integer`

Function	Type[1]
`[ ]`[2]	`Char->String`
`toEnum`	`Int->Char`
`fromEnum`	`Char->Int`
`show`	`Integer->String`
`show`	`Double->String`
`read`	`String->Integer`
`read`	`String->Double`

[1] We'll talk more about the type of these functions later.

[2] E.g., `['a']`. This isn't actually a function, but special syntax.

Type Signatures

Because functions like `read` can return values of different types, it is sometimes necessary to use a **type signature** to indicate the desired type.

Type Signatures

Because functions like `read` can return values of different types, it is sometimes necessary to use a **type signature** to indicate the desired type.

```
Prelude> read "42"

<interactive>:21:1:
    No instance for (Read a0) arising from a use of `read'
    The type variable `a0' is ambiguous
    Possible fix: add a type signature that fixes these type
      variable(s)
```

Type Signatures

Because functions like `read` can return values of different types, it is sometimes necessary to use a **type signature** to indicate the desired type.

```
Prelude> read "42"

<interactive>:21:1:
    No instance for (Read a0) arising from a use of `read'
    The type variable `a0' is ambiguous
    Possible fix: add a type signature that fixes these type
      variable(s)
```

```
 Prelude> read "42" :: Integer
42
```

Type Signatures and Type Inference

Remember that Haskell performs type inference, i.e., it tries to work out the type of expressions automatically?

Type Signatures and Type Inference

Remember that Haskell performs type inference, i.e., it tries to work out the type of expressions automatically?

Even for functions like `read`, it is *not* necessary to indicate the desired type when this can be determined from the program context:

```
Prelude> read "42" + 1
43

Prelude> read "42" + 1.0
43.0
```

Type `Bool`

Value syntax: `True`, `False`

Type `Bool`

Value syntax: `True`, `False`

Explanation: `True` and `False` are (the only) values of type `Bool`, just like `1` and `42` are values of type `Integer`.

Type `Bool`

Value syntax: `True`, `False`

Explanation: `True` and `False` are (the only) values of type `Bool`, just like `1` and `42` are values of type `Integer`.

Functions	Type	Semantics
`==`, `/=`	`Bool->Bool->Bool`	equality, inequality

Type `Bool`

Value syntax: `True`, `False`

Explanation: `True` and `False` are (the only) values of type `Bool`, just like `1` and `42` are values of type `Integer`.

Functions	Type	Semantics
`==`, `/=`	`Bool->Bool->Bool`	equality, inequality
`not`	`Bool->Bool`	logical negation

Type Bool

Value syntax: `True`, `False`

Explanation: `True` and `False` are (the only) values of type `Bool`, just like `1` and `42` are values of type `Integer`.

Functions	Type	Semantics
`==`, `/=`	`Bool->Bool->Bool`	equality, inequality
`not`	`Bool->Bool`	logical negation
`&&`	`Bool->Bool->Bool`	logical and
`||`	`Bool->Bool->Bool`	logical or

Type `Bool`

Value syntax: `True`, `False`

Explanation: `True` and `False` are (the only) values of type `Bool`, just like `1` and `42` are values of type `Integer`.

Functions	Type	Semantics
`==`, `/=`	`Bool->Bool->Bool`	equality, inequality
`not`	`Bool->Bool`	logical negation
`&&`	`Bool->Bool->Bool`	logical and
`||`	`Bool->Bool->Bool`	logical or

Functions with result type `Bool` are sometimes called **predicates**.

Evaluation of && and ||

Remember that Haskell is non-strict, i.e., arguments to functions are evaluated only when they are needed to evaluate the function body?

Evaluation of `&&` and `||`

Remember that Haskell is non-strict, i.e., arguments to functions are evaluated only when they are needed to evaluate the function body?

The second argument to `&&` and `||` is only evaluated if the value of the first argument doesn't suffice to determine the value of the entire expression.

Evaluation of && and ||

Remember that Haskell is non-strict, i.e., arguments to functions are evaluated only when they are needed to evaluate the function body?

The second argument to `&&` and `||` is only evaluated if the value of the first argument doesn't suffice to determine the value of the entire expression.

Example:

Evaluation of && and ||

Remember that Haskell is non-strict, i.e., arguments to functions are evaluated only when they are needed to evaluate the function body?

The second argument to && and || is only evaluated if the value of the first argument doesn't suffice to determine the value of the entire expression.

Example:

```
3 > 2  ||  3 < div 1 0
⟶ True  ||  3 < div 1 0
⟶ True
```

Evaluation of && and ||

Remember that Haskell is non-strict, i.e., arguments to functions are evaluated only when they are needed to evaluate the function body?

The second argument to && and || is only evaluated if the value of the first argument doesn't suffice to determine the value of the entire expression.

Example:

```
 3 > 2  ||  3 < div 1 0
⟶ True  ||  3 < div 1 0
⟶ True
```

(With strict evaluation, this expression would throw an exception.)

The Haskell Prelude

Many other useful functions are provided by the **Haskell Prelude**: see

`http://hackage.haskell.org/package/base/docs/Prelude.html`

These are available by default in all Haskell programs.

Infix Operators

In Haskell, function application is usually written in **prefix** notation: e.g.,

```
length "Hello"
round 3.14159
```

Infix Operators

In Haskell, function application is usually written in **prefix** notation: e.g.,

```
length "Hello"
round 3.14159
```

However, we have already seen many examples of **infix operators**, i.e., functions that take two (or more) arguments and are written *between* their arguments. For instance,

```
1 + 2
2.72 == 3.14
"foo" ++ "bar"
```

Infix vs. Prefix Notation

In Haskell, names that consist of special symbols denote infix operators by default. However, it is easy to switch between infix and prefix notation.

Infix vs. Prefix Notation

In Haskell, names that consist of special symbols denote infix operators by default. However, it is easy to switch between infix and prefix notation.

To use an infix operator in prefix position, simply enclose the operator in parentheses. For instance,

```
Prelude> (+) 1 2
3
```

Infix vs. Prefix Notation

In Haskell, names that consist of special symbols denote infix operators by default. However, it is easy to switch between infix and prefix notation.

To use an infix operator in prefix position, simply enclose the operator in parentheses. For instance,

```
Prelude> (+) 1 2
3
```

To use a prefix function in infix position, simply enclose the function in backticks. For instance,

```
Prelude> 7 `mod` 3
1
```

Infix Operators: Precedence

How does Haskell know that `1 + 2 * 3` should be evaluated as `1 + (2 * 3)` rather than `(1 + 2) * 3`?

Infix Operators: Precedence

How does Haskell know that `1 + 2 * 3` should be evaluated as `1 + (2 * 3)` rather than `(1 + 2) * 3`?

Every infix operator has a **precedence**. Higher precedences bind more tightly.

Infix Operators: Precedence

How does Haskell know that `1 + 2 * 3` should be evaluated as `1 + (2 * 3)` rather than `(1 + 2) * 3`?

Every infix operator has a **precedence**. Higher precedences bind more tightly.

Operators	Precedence
`*`, `/`, `` `div` ``, `` `mod` ``	7
`+`, `-`	6
`++`	5
`==`, `/=`, `<`, `<=`, `>`, `>=`	4
`&&`	3
`||`	2

Source: `http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2`

Infix Operators: Precedence

How does Haskell know that `1 + 2 * 3` should be evaluated as `1 + (2 * 3)` rather than `(1 + 2) * 3`?

Every infix operator has a **precedence**. Higher precedences bind more tightly.

Operators	Precedence
`*`, `/`, `` `div` ``, `` `mod` ``	7
`+`, `-`	6
`++`	5
`==`, `/=`, `<`, `<=`, `>`, `>=`	4
`&&`	3
`||`	2

Source: http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2

(It is possible to change these precedence values.)

Infix Operators: Associativity

How does Haskell know that `3 - 2 - 1` should be evaluated as `(3 - 2) - 1` rather than `3 - (2 - 1)`?

Infix Operators: Associativity

How does Haskell know that `3 - 2 - 1` should be evaluated as `(3 - 2) - 1` rather than `3 - (2 - 1)`?

Infix operators may be **left**- or **right-associative**,
meaning operations are grouped from the left (or right, respectively).

Infix Operators: Associativity

How does Haskell know that `3 - 2 - 1` should be evaluated as `(3 - 2) - 1` rather than `3 - (2 - 1)`?

Infix operators may be **left**- or **right-associative**,
meaning operations are grouped from the left (or right, respectively).

Left associative	Non-associative	Right associative
`*`, `/`, `` `div` ``, `` `mod` ``		
`+`, `-`		
		`++`
	`==`, `/=`, `<`, `<=`, `>`, `>=`	
		`&&`
		`||`

Source: `http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2`

Infix Operators: Associativity

How does Haskell know that `3 - 2 - 1` should be evaluated as `(3 - 2) - 1` rather than `3 - (2 - 1)`?

Infix operators may be **left**- or **right-associative**,
meaning operations are grouped from the left (or right, respectively).

Left associative	Non-associative	Right associative
*, /, \`div\`, \`mod\`		
+, -		
		++
	==, /=, <, <=, >, >=	
		&&
		||

Source: http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2

(It is possible to change the associativity.)

GHCi's `:info`

In GHCi, you can use `:info` to show information about infix operators:

GHCi's `:info`

In GHCi, you can use `:info` to show information about infix operators:

```
Prelude> :i -
class Num a where
  ...
  (-) :: a -> a -> a
  ...
      -- Defined in `GHC.Num'
infixl 6 -
```

Tuples

Two or more expressions (of possibly different types) can be grouped together to form a **tuple**.

Tuples

Two or more expressions (of possibly different types) can be grouped together to form a **tuple**.

Examples:

```
(1, 2, 3)
("PKD", 2019)
(3, "three", 3.0)
```

Tuples

Two or more expressions (of possibly different types) can be grouped together to form a **tuple**.

Examples:

```
(1, 2, 3)
("PKD", 2019)
(3, "three", 3.0)
```

Fine points:

- There are no 1-tuples in Haskell: `(42)` is just 42 in parentheses.
- It is possible to have tuples of tuples: e.g., `((1,2),(3,4))`.
- The value `()` is the only "0-tuple". Its type is `()`.

Product Types

The type of a tuple is given by a (Cartesian) **product type**.

Product Types

The type of a tuple is given by a (Cartesian) **product type**.

Examples:

Product Types

The type of a tuple is given by a (Cartesian) **product type**.

Examples:

```
(1, 2, 3) :: (Integer, Integer, Integer)

("PKD", 2019) :: (String, Integer)

(3, "three", 3.0) :: (Integer, String, Double)
```

Pairs: `(,)`

The function `(,)` takes two arguments and returns the pair (2-tuple) that consists of these arguments (in the given order).

Pairs: (,)

The function `(,)` takes two arguments and returns the pair (2-tuple) that consists of these arguments (in the given order).

```
Prelude> (,) "PKD" 2019
("PKD",2019)

Prelude> (,) 1 2
(1,2)

Prelude> (,) 2 1
(2,1)
```

Pairs: `fst` and `snd`

Functions `fst` and `snd` extract the first and second, respectively, component of a pair.

Pairs: `fst` and `snd`

Functions `fst` and `snd` extract the first and second, respectively, component of a pair.

```
Prelude> fst (1,2)
1

Prelude> snd (1,2)
2
```

Pairs: `fst` and `snd`

Functions `fst` and `snd` extract the first and second, respectively, component of a pair.

```
Prelude> fst (1,2)
1

Prelude> snd (1,2)
2
```

Later, we will see how to extract components from n-tuples for $n > 2$.

Conditional Expressions

Often, we want our programs to perform different computations, depending on the value of some condition.

Conditional Expressions

Often, we want our programs to perform different computations, depending on the value of some condition.

If `condition` is an expression of type `Bool`, and `trueValue` and `falseValue` are two expressions that have *the same type*, then

Conditional Expressions

Often, we want our programs to perform different computations, depending on the value of some condition.

If `condition` is an expression of type `Bool`, and `trueValue` and `falseValue` are two expressions that have *the same type*, then

```
if condition then trueValue else falseValue
```

Conditional Expressions

Often, we want our programs to perform different computations, depending on the value of some condition.

If `condition` is an expression of type `Bool`, and `trueValue` and `falseValue` are two expressions that have *the same type*, then

```
if condition then trueValue else falseValue
```

is an expression, called a **conditional expression**.

Conditional Expressions

Often, we want our programs to perform different computations, depending on the value of some condition.

If `condition` is an expression of type `Bool`, and `trueValue` and `falseValue` are two expressions that have *the same type*, then

```
if condition then trueValue else falseValue
```

is an expression, called a **conditional expression**.

Example:

```
if 2+2==5 then "hel"++"lo" else "good"++"bye"
```

Conditional Expressions: Type

Remember that every expression has a type?

Conditional Expressions: Type

Remember that every expression has a type?

The type of a conditional expression

```
if condition then trueValue else falseValue
```

is simply the type of `trueValue`. (This is the same as the type of `falseValue`.)

Conditional Expressions: Type

Remember that every expression has a type?

The type of a conditional expression

```
if condition then trueValue else falseValue
```

is simply the type of `trueValue`. (This is the same as the type of `falseValue`.)

Example:

```
if 2+2==5 then "hel"++"lo" else "good"++"bye" :: String
```

Evaluation of Conditional Expressions

To evaluate a conditional expression

```
if condition then trueValue else falseValue
```

Haskell first evaluates `condition`. If `condition` evaluates to `True`, Haskell evaluates `trueValue`. If `condition` evaluates to `False`, Haskell evaluates `falseValue`.

Evaluation of Conditional Expressions

To evaluate a conditional expression

```
if condition then trueValue else falseValue
```

Haskell first evaluates `condition`. If `condition` evaluates to `True`, Haskell evaluates `trueValue`. If `condition` evaluates to `False`, Haskell evaluates `falseValue`.

Example:

```
if 2+2==5 then "hel"++"lo" else "good"++"bye"
```

Evaluation of Conditional Expressions

To evaluate a conditional expression

```
if condition then trueValue else falseValue
```

Haskell first evaluates `condition`. If `condition` evaluates to `True`, Haskell evaluates `trueValue`. If `condition` evaluates to `False`, Haskell evaluates `falseValue`.

Example:

```
   if 2+2==5 then "hel"++"lo" else "good"++"bye"
⟶ if  4 ==5 then "hel"++"lo" else "good"++"bye"
```

Evaluation of Conditional Expressions

To evaluate a conditional expression

```
if condition then trueValue else falseValue
```

Haskell first evaluates `condition`. If `condition` evaluates to `True`, Haskell evaluates `trueValue`. If `condition` evaluates to `False`, Haskell evaluates `falseValue`.

Example:

```
    if 2+2==5 then "hel"++"lo" else "good"++"bye"
--> if  4 ==5 then "hel"++"lo" else "good"++"bye"
--> if  False then "hel"++"lo" else "good"++"bye"
```

Evaluation of Conditional Expressions

To evaluate a conditional expression

```
if condition then trueValue else falseValue
```

Haskell first evaluates `condition`. If `condition` evaluates to `True`, Haskell evaluates `trueValue`. If `condition` evaluates to `False`, Haskell evaluates `falseValue`.

Example:

```
   if 2+2==5 then "hel"++"lo" else "good"++"bye"
⟶ if  4 ==5 then "hel"++"lo" else "good"++"bye"
⟶ if  False then "hel"++"lo" else "good"++"bye"
⟶                                 "good"++"bye"
```

Evaluation of Conditional Expressions

To evaluate a conditional expression

```
if condition then trueValue else falseValue
```

Haskell first evaluates `condition`. If `condition` evaluates to `True`, Haskell evaluates `trueValue`. If `condition` evaluates to `False`, Haskell evaluates `falseValue`.

Example:

```
   if 2+2==5 then "hel"++"lo" else "good"++"bye"
⟶ if  4 ==5 then "hel"++"lo" else "good"++"bye"
⟶ if  False then "hel"++"lo" else "good"++"bye"
⟶                                 "good"++"bye"
⟶                                   "goodbye"
```

Conditional Expressions: Fine Points

`if condition then trueValue else falseValue` is an expression.

Conditional Expressions: Fine Points

`if condition then trueValue else falseValue` is an expression.

(In case you are familiar with C: Haskell's `if-then-else` is more similar to C's `(?:)` than to C's `if-else`.)

Conditional Expressions: Fine Points

`if condition then trueValue else falseValue` is an expression.

(In case you are familiar with C: Haskell's `if-then-else` is more similar to C's `(?:)` than to C's `if-else`.)

Conditional Expressions: Fine Points

`if condition then trueValue else falseValue` is an expression.

(In case you are familiar with C: Haskell's `if-then-else` is more similar to C's `(?:)` than to C's `if-else`.)

All parts are mandatory. `if condition then trueValue` is *not* a valid expression.
(What should be its value when `condition` is false?!)

Exercises

1. Express each of the following expressions as `if-then-else` expressions. In other words, find `condition`, `trueValue` and `falseValue` such that the given expression is equivalent to

```
if condition then trueValue else falseValue
```

   1. `E || F`
   2. `E && F`

Exercises

1. Express each of the following expressions as `if-then-else` expressions. In other words, find `condition`, `trueValue` and `falseValue` such that the given expression is equivalent to

   ```
   if condition then trueValue else falseValue
   ```

   1. `E || F`
   2. `E && F`

2. Evaluate (step-by-step) the following expression:

   ```
   if 1 + 2 < 4 then length ("hel"++"lo!") else 4 `div` 2
   ```

Value Declarations

Value declarations associate a value to an identifier.

Value Declarations

Value declarations associate a value to an identifier.

```
Prelude> x = 1
Prelude> myPi = 3.14159
Prelude> twoPi = 2.0 * myPi
Prelude> (@@) = "use descriptive identifiers!"

Prelude> x + x
2
Prelude> twoPi
6.28318
```

Value Declarations

Value declarations associate a value to an identifier.

```
Prelude> x = 1
Prelude> myPi = 3.14159
Prelude> twoPi = 2.0 * myPi
Prelude> (@@) = "use descriptive identifiers!"

Prelude> x + x
2
Prelude> twoPi
6.28318
```

Note: value declarations are *not* expressions!

Identifiers: Syntax

Note that `3+(-2)` is different from `3+-2`! Haskell thinks that `+-` is an (undeclared) identifier.

Identifiers: Syntax

Note that `3+(-2)` is different from `3+-2`! Haskell thinks that `+-` is an (undeclared) identifier.

```
Prelude> 3+(-2)
1
```

```
Prelude> 3+-2

<interactive>:3:2:
    Not in scope: `+-'
    ...
```

Bindings and Environments

The execution of a declaration, say `x = expr`, creates a **binding**: the identifier `x` is *bound* to the value of the expression `expr`.

Bindings and Environments

The execution of a declaration, say `x = expr`, creates a **binding**: the identifier `x` is *bound* to the value of the expression `expr`.

A collection of bindings is called an **environment**.

Bindings and Environments

The execution of a declaration, say `x = expr`, creates a **binding**: the identifier `x` is *bound* to the value of the expression `expr`.

A collection of bindings is called an **environment**.

Bindings and Environments

The execution of a declaration, say `x = expr`, creates a **binding**: the identifier `x` is *bound* to the value of the expression `expr`.

A collection of bindings is called an **environment**.

In GHCi, you can use `:show bindings` to show the current bindings and their type:

Bindings and Environments

The execution of a declaration, say `x = expr`, creates a **binding**: the identifier `x` is *bound* to the value of the expression `expr`.

A collection of bindings is called an **environment**.

In GHCi, you can use `:show bindings` to show the current bindings and their type:

```
Prelude> :show bindings
x :: Integer = 1
myPi :: Double = 3.14159
twoPi :: Double = _
(@@) :: [Char] = _
```

Changing Environments

Later declarations of the same identifier change the environment.

Changing Environments

Later declarations of the same identifier change the environment.

```
Prelude> x = 1
Prelude> :show bindings
x :: Integer = 1

Prelude> x = 2
Prelude> :show bindings
x :: Integer = 2

Prelude> x = "x"
Prelude> :show bindings
x :: [Char] = "x"
```

Changing Environments

Later declarations of the same identifier change the environment.

```
Prelude> x = 1
Prelude> :show bindings
x :: Integer = 1

Prelude> x = 2
Prelude> :show bindings
x :: Integer = 2

Prelude> x = "x"
Prelude> :show bindings
x :: [Char] = "x"
```

Declarations in Haskell are similar to defining equations in mathematics.

GHCi's `it`

The identifier `it` is always bound to the value of the last expression that was evaluated.

GHCi's `it`

The identifier `it` is always bound to the value of the last expression that was evaluated.

```
Prelude> 1 + 2
3
Prelude> :show bindings
it :: Integer = 3
Prelude> it
3

Prelude> "another " ++ "expression"
"another expression"
Prelude> it
"another expression"
```

Execution of Declarations

Remember that Haskell is non-strict? When a declaration, say `x = expr`, is executed, the value of `expr` is not computed right away. Instead, it is computed later, when (and if) `x` is evaluated.

Execution of Declarations

Remember that Haskell is non-strict? When a declaration, say `x = expr`, is executed, the value of `expr` is not computed right away. Instead, it is computed later, when (and if) `x` is evaluated.

```
Prelude> x = 1 + 1 :: Int
Prelude> :show bindings
x :: Int = _

Prelude> x  -- force evaluation of x
2
Prelude> :show bindings
x :: Int = 2
it :: Int = 2
```

Execution of Declarations (cont.)

Therefore, it is possible to bind `x` to an expression whose evaluation would result in a runtime error. The runtime error will only be generated when (and if) `x` is evaluated.

Execution of Declarations (cont.)

Therefore, it is possible to bind `x` to an expression whose evaluation would result in a runtime error. The runtime error will only be generated when (and if) `x` is evaluated.

```
Prelude> x = 1 `div` 0
Prelude> x
*** Exception: divide by zero
```

Identifiers Are *Not* Variables

... and declarations are not variable updates.

Identifiers Are *Not* Variables

... and declarations are not variable updates.

```
Prelude> x = 10
Prelude> addX y = x+y
Prelude> addX 42
52

Prelude> x = 20
Prelude> addX 42
52
```

Practical Matters

Lab 1

Our first lab is scheduled for tomorrow (Thursday, 2018-11-07) 8:15-12:00.

Lab 1

Our first lab is scheduled for tomorrow (Thursday, 2018-11-07) 8:15-12:00.

It will take place in the lab rooms at Polacksbacken (1515, 1549, 2510).

Lab 1

Our first lab is scheduled for tomorrow (Thursday, 2018-11-07) 8:15-12:00.

It will take place in the lab rooms at Polacksbacken (1515, 1549, 2510).

Please see **Studium** for your specific room and a brief summary of the lab exercises. Do **not** go to the wrong room. (Note that rooms will change, so check again before the next lab!)

Lab 1

Our first lab is scheduled for tomorrow (Thursday, 2018-11-07) 8:15-12:00.

It will take place in the lab rooms at Polacksbacken (1515, 1549, 2510).

Please see **Studium** for your specific room and a brief summary of the lab exercises. Do **not** go to the wrong room. (Note that rooms will change, so check again before the next lab!)

Remember: attendance (at 7 out of 9 labs this Fall) is mandatory!