Introduction to Functional Programming and Haskell

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

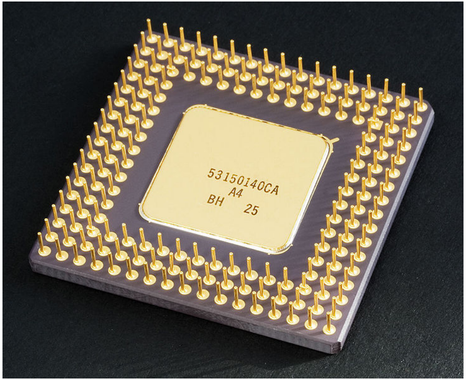
Today

- Introduction to functional programming
- Introduction to Haskell

Introduction to Functional Programming

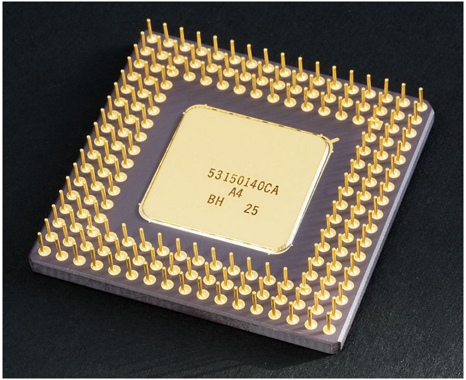
Abstraction from Hardware

Allmost all computer hardware is imperative (i.e., executes a sequence of instructions that change the state of the machine).



Abstraction from Hardware

Allmost all computer hardware is imperative (i.e., executes a sequence of instructions that change the state of the machine).



However, high-level programming languages provide a **higher level of abstraction**.

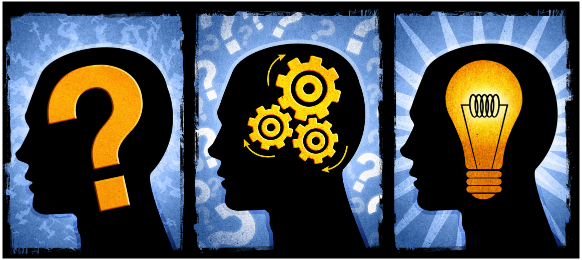
High-Level Languages

High-level languages are not executed directly, but need to be **translated** into executable machine code first (by a compiler or interpreter).

High-Level Languages

High-level languages are not executed directly, but need to be **translated** into executable machine code first (by a compiler or interpreter).

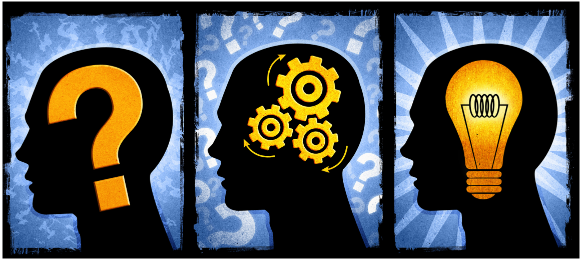
Therefore, they can support advanced concepts that allow the programmer to focus on **solving problems** (rather than instructing a machine).



High-Level Languages

High-level languages are not executed directly, but need to be **translated** into executable machine code first (by a compiler or interpreter).

Therefore, they can support advanced concepts that allow the programmer to focus on **solving problems** (rather than instructing a machine).



Programmers use many different high-level programming languages: e.g., C, C++, Java, PHP, Perl, Python, MATLAB, Lisp, Haskell.

Programming Paradigms

Programming Paradigms

imperative
(e.g., C)

Programming Paradigms

imperative	object-oriented
(e.g., C)	(e.g., C++, Java)

Programming Paradigms

imperative	object-oriented	logic-based
(e.g., C)	(e.g., C++, Java)	(e.g., Prolog)

Programming Paradigms

imperative
(e.g., C)

object-oriented
(e.g., C++, Java)

logic-based
(e.g., Prolog)

functional
(e.g., Lisp, **Haskell**)

General-Purpose Languages

All programming paradigms offer **general-purpose** languages—that is, languages which may be used to write software for a wide range of different application domains.

General-Purpose Languages

All programming paradigms offer **general-purpose** languages—that is, languages which may be used to write software for a wide range of different application domains.

All modern general-purpose languages can compute the same mathematical functions. (In technical terms, they are *Turing-complete*.) In other words, if a mathematical problem can be solved by a computer program at all, it can be solved in any of these languages.

General-Purpose Languages

All programming paradigms offer **general-purpose** languages—that is, languages which may be used to write software for a wide range of different application domains.

All modern general-purpose languages can compute the same mathematical functions. (In technical terms, they are *Turing-complete*.) In other words, if a mathematical problem can be solved by a computer program at all, it can be solved in any of these languages.

In practice many “problems” are engineering challenges (rather than mathematical functions), and these can be much **easier** to solve in one language or another.

Programming without mutable variables, loops or shared state.

Think different.

Why Functional Programming?

- Shorter, more readable code

Why Functional Programming?

- Shorter, more readable code
- Easy to execute concurrently

Why Functional Programming?

- Shorter, more readable code
- Easy to execute concurrently
- Unit testing becomes easy

Functional Programming Languages

- Lisp (McCarthy 1962): Scheme, Racket, Clojure

https://en.wikipedia.org/wiki/Category:Functional_languages

Functional Programming Languages

- Lisp (McCarthy 1962): Scheme, Racket, Clojure
- ML (Milner et al. 1977): OCaml, **Haskell** (Hudak 1990)

https://en.wikipedia.org/wiki/Category:Functional_languages

Functional Programming Languages

- Lisp (McCarthy 1962): Scheme, Racket, Clojure
- ML (Milner et al. 1977): OCaml, **Haskell** (Hudak 1990)
- Erlang (Armstrong et al. 1993)

https://en.wikipedia.org/wiki/Category:Functional_languages

Functional Programming Languages

- Lisp (McCarthy 1962): Scheme, Racket, Clojure
- ML (Milner et al. 1977): OCaml, **Haskell** (Hudak 1990)
- Erlang (Armstrong et al. 1993)
- Scala (Odersky 2003)

https://en.wikipedia.org/wiki/Category:Functional_languages

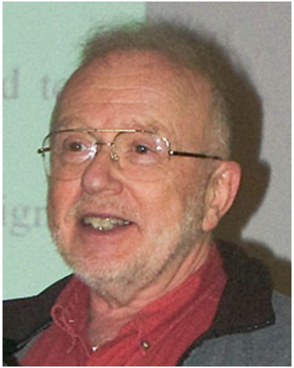
Functional Programming Languages

- Lisp (McCarthy 1962): Scheme, Racket, Clojure
- ML (Milner et al. 1977): OCaml, **Haskell** (Hudak 1990)
- Erlang (Armstrong et al. 1993)
- Scala (Odersky 2003)
- F# (Syme 2005)

https://en.wikipedia.org/wiki/Category:Functional_languages

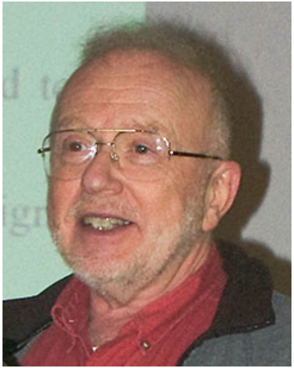
Fundamental Ideas

"I would advise students to pay more attention to the fundamental ideas rather than the latest technology. The technology will be out-of-date before they graduate. Fundamental ideas never get out of date."



Fundamental Ideas

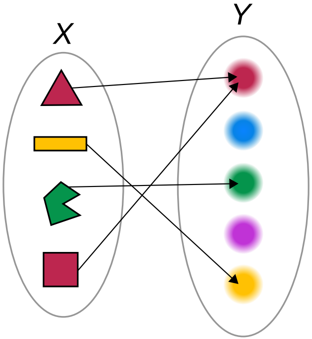
"I would advise students to pay more attention to the fundamental ideas rather than the latest technology. The technology will be out-of-date before they graduate. Fundamental ideas never get out of date."



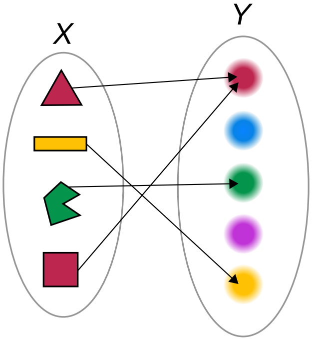
David Parnas (born 1941) is a Canadian early pioneer of software engineering. He developed the concept of information hiding in modular programming.

http://en.wikipedia.org/wiki/David_Parnas

Functions



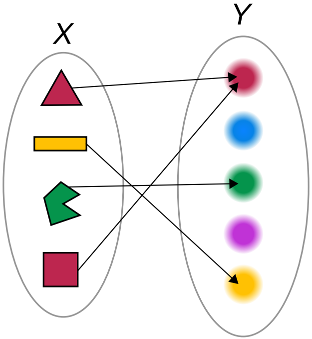
Functions



A (total) **function** is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to *exactly one* output.

`http://en.wikipedia.org/wiki/Function_%28mathematics%29`

Functions

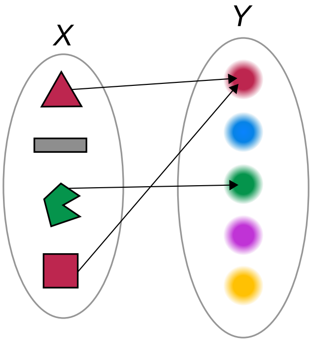


A (total) **function** is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to *exactly one* output.

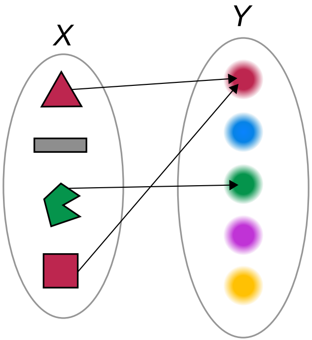
http://en.wikipedia.org/wiki/Function_%28mathematics%29

We write $f : X \to Y$ for a function f from X to Y.

Partial Functions

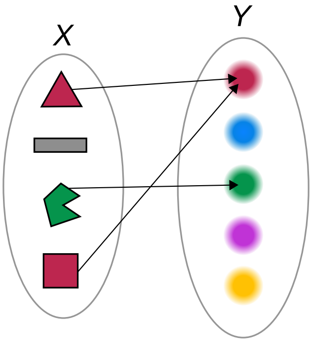


Partial Functions



A **partial function** is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to *at most one* output.

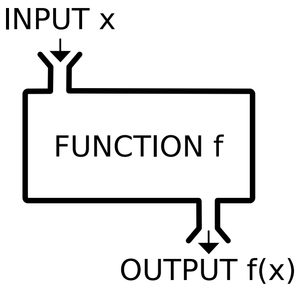
Partial Functions



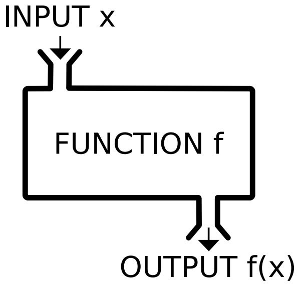
A **partial function** is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to *at most one* output.

We write $f\colon X \rightharpoonup Y$ for a partial function f from X to Y.

Specifying Functions



Specifying Functions



A function may be given by an **expression** that determines the output value. For example,

```
double n = 2 * n
```

Function Composition

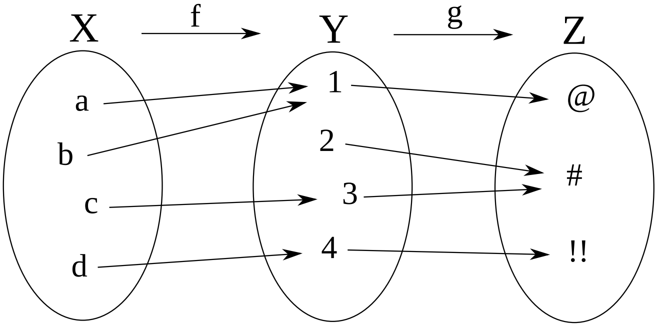
Function composition is the application of one function to the results of another, to produce a new function.

http://en.wikipedia.org/wiki/Function_composition

Function Composition

Function composition is the application of one function to the results of another, to produce a new function.

http://en.wikipedia.org/wiki/Function_composition



Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

Fundamental principles:

- Declaration of functions

Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

Fundamental principles:

- Declaration of functions
- Application of functions

Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

Fundamental principles:

- Declaration of functions
- Application of functions
- Execution by evaluation of expressions

Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

Fundamental principles:

- Declaration of functions
- Application of functions
- Execution by evaluation of expressions
- Inductive data types

Principles of Functional Programming

Functional programming treats computation as the **evaluation of mathematical functions**, while avoiding state and mutable data.

http://en.wikipedia.org/wiki/Functional_programming

Fundamental principles:

- Declaration of functions
- Application of functions
- Execution by evaluation of expressions
- Inductive data types
- Recursion

Introduction to Haskell

Haskell

Haskell is a general-purpose, modular, purely functional programming language. It is garbage collected, has compile-time type inference and type checking, and is type safe.

Haskell

Haskell is a general-purpose, modular, purely functional programming language. It is garbage collected, has compile-time type inference and type checking, and is type safe.

Haskell is a popular functional language. There is a very active community, with several Haskell implementations (compilers), many third-party tools and libraries, and good online resources.

Haskell

Haskell is a general-purpose, modular, purely functional programming language. It is garbage collected, has compile-time type inference and type checking, and is type safe.

Haskell is a popular functional language. There is a very active community, with several Haskell implementations (compilers), many third-party tools and libraries, and good online resources.

Some aspects of the language are slightly intricate. We'll try to introduce these gently, but they may surprise you from time to time.

Glasgow Haskell Compiler

Remember that programs written in a high-level language are not executed directly, but first translated into machine code by a compiler or interpreter?

Glasgow Haskell Compiler

Remember that programs written in a high-level language are not executed directly, but first translated into machine code by a compiler or interpreter?

In this course, we use an implementation (i.e., a compiler/interpreter) for Haskell that is called the **Glasgow Haskell Compiler**, or **GHC**.

Glasgow Haskell Compiler

Remember that programs written in a high-level language are not executed directly, but first translated into machine code by a compiler or interpreter?

In this course, we use an implementation (i.e., a compiler/interpreter) for Haskell that is called the **Glasgow Haskell Compiler**, or **GHC**.

GHC is already installed on the lab machines. To install it on your own computer, download the Haskell Platform from

`http://www.haskell.org/platform/`

Glasgow Haskell Compiler

Remember that programs written in a high-level language are not executed directly, but first translated into machine code by a compiler or interpreter?

In this course, we use an implementation (i.e., a compiler/interpreter) for Haskell that is called the **Glasgow Haskell Compiler**, or **GHC**.

GHC is already installed on the lab machines. To install it on your own computer, download the Haskell Platform from

`http://www.haskell.org/platform/`

(If you install from some other source, make sure that you get "Haskell Platform", and not one of the cut-down distributions.)

Using GHC Interactively

The Glasgow Haskell Compiler includes a tool called **GHCi** that allows you to evaluate Haskell expressions **interactively**.

Using GHC Interactively

The Glasgow Haskell Compiler includes a tool called **GHCi** that allows you to evaluate Haskell expressions **interactively**.

To start GHCi, open a terminal and run

```
ghci
```

Using GHC Interactively

The Glasgow Haskell Compiler includes a tool called **GHCi** that allows you to evaluate Haskell expressions **interactively**.

To start GHCi, open a terminal and run

```
ghci
```

You'll be greeted with something like

```
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Using GHC Interactively

The Glasgow Haskell Compiler includes a tool called **GHCi** that allows you to evaluate Haskell expressions **interactively**.

To start GHCi, open a terminal and run

```
ghci
```

You'll be greeted with something like

```
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude>
```

`Prelude>` is the GHCi prompt. You can now enter Haskell expressions, and GHCi will evaluate them for you!

Haskell Expressions: Arithmetic

```
2 + 15

49 * 100

1892 - 1472

5 / 2

50 * 100 - 4999

(50 * 100) - 4999

50 * (100 - 4999)

5 * (-3)
```

Haskell Expressions: Boolean Algebra

```haskell
True && False

True && True

False || True

not False

not (True && True)
```

GHCi: Interrupting and Leaving

To interrupt an ongoing computation, press **Ctrl+C**:

```
Prelude> [1..] -- how long are you willing to wait?
[1,2,3,4Interrupted.
```

GHCi: Interrupting and Leaving

To interrupt an ongoing computation, press **Ctrl+C**:

```
Prelude> [1..] -- how long are you willing to wait?
[1,2,3,4Interrupted.
```

To leave GHCi, type **:quit** at its prompt:

```
Prelude> :quit
Leaving GHCi.
```

Alternatively, press **Ctrl+D**.

Underwhelmed?

Did we just turn a 10,000 SEK laptop into a 50 SEK pocket calculator?

Underwhelmed?

Did we just turn a 10,000 SEK laptop into a 50 SEK pocket calculator?

Well, right now it may look like that. But that's just because we haven't seen much of Haskell yet.

Underwhelmed?

Did we just turn a 10,000 SEK laptop into a 50 SEK pocket calculator?

Well, right now it may look like that. But that's just because we haven't seen much of Haskell yet.

In fact, Haskell—like other modern languages—is Turing-complete. Any (mathematical) problem that can be solved by a computer program at all, can be solved by a Haskell program.

Syntax

The **syntax** of a programming language is a set of rules that defines what combinations of symbols constitute valid programs in that language.

Syntax

The **syntax** of a programming language is a set of rules that defines what combinations of symbols constitute valid programs in that language.

For instance, `2 + 15` is a valid Haskell program.

Syntax

The **syntax** of a programming language is a set of rules that defines what combinations of symbols constitute valid programs in that language.

For instance, `2 + 15` is a valid Haskell program.

On the other hand, `5 * -3` is not a valid Haskell program. (But it could be a valid program in some other language.)

Syntax

The **syntax** of a programming language is a set of rules that defines what combinations of symbols constitute valid programs in that language.

For instance, `2 + 15` is a valid Haskell program.

On the other hand, `5 * -3` is not a valid Haskell program. (But it could be a valid program in some other language.)

(These are just examples. You can't really tell valid from invalid programs yet, because we haven't discussed the syntax of Haskell. We will introduce it bit by bit in the course, mostly through examples of valid programs.)

Syntax Errors

English has syntax too, but if there is a mistak, you can (usually) still figure out the meaning.

Syntax Errors

English has syntax too, but if there is a mistak, you can (usually) still figure out the meaning.

Compilers/interpreters are less forgiving. If your program doesn't follow the Haskell syntax rules, GHC will print an **error message**.

Syntax Errors

English has syntax too, but if there is a mistak, you can (usually) still figure out the meaning.

Compilers/interpreters are less forgiving. If your program doesn't follow the Haskell syntax rules, GHC will print an **error message**. For instance,

```
Prelude> 5 * -3
<interactive>:1:0:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] ...
```

Compiler Error Messages

```
Prelude> 5 * -3
<interactive>:1:0:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] ...
```

Compiler Error Messages

```
Prelude> 5 * -3
<interactive>:1:0:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] ...
```

You will get many error messages when you write your own programs.

Compiler Error Messages

```
Prelude> 5 * -3
<interactive>:1:0:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] ...
```

You will get many error messages when you write your own programs.

Read and try to understand them! Sometimes GHC's error messages are daunting, but usually they give some indication what's wrong with your code.

Compiler Error Messages

```
Prelude> 5 * -3
<interactive>:1:0:
    Precedence parsing error
        cannot mix `*' [infixl 7] and prefix `-' [infixl 6] ...
```

You will get many error messages when you write your own programs.

Read and try to understand them! Sometimes GHC's error messages are daunting, but usually they give some indication what's wrong with your code.

However, GHC can't know what you had in mind ...

Semantics

The **semantics** of a programming language defines the meaning of programs.

Semantics

The **semantics** of a programming language defines the meaning of programs.

:	Syntax	Semantics
	dog	

Semantics

The **semantics** of a programming language defines the meaning of programs.

:

Syntax	Semantics
dog	

For instance, in Haskell . . .

- + means addition

Semantics

The **semantics** of a programming language defines the meaning of programs.

: Syntax	Semantics
dog	

For instance, in Haskell . . .

- + means addition
- 2+15 means "the sum of 2 and 15", i.e., 17

Semantics

The **semantics** of a programming language defines the meaning of programs.

 : Syntax	Semantics
dog	

For instance, in Haskell . . .

- `+` means addition
- `2+15` means "the sum of 2 and 15", i.e., 17
- `f x` means "the function `f` applied to the argument `x`"

Syntax and Semantics: Common Descriptions

The syntax of programming languages is often defined very precisely, e.g., by formal grammars or syntax diagrams. Thus, it is (usually) possible to determine precisely—and by computer—whether a program is valid.

Syntax and Semantics: Common Descriptions

The syntax of programming languages is often defined very precisely, e.g., by formal grammars or syntax diagrams. Thus, it is (usually) possible to determine precisely—and by computer—whether a program is valid.

A syntax definition for Haskell is available online at
`http://www.haskell.org/onlinereport/haskell2010/`

Syntax and Semantics: Common Descriptions

The syntax of programming languages is often defined very precisely, e.g., by formal grammars or syntax diagrams. Thus, it is (usually) possible to determine precisely—and by computer—whether a program is valid.

A syntax definition for Haskell is available online at
`http://www.haskell.org/onlinereport/haskell2010/`

It is possible—but a lot of work—to define the semantics of every program construct equally precisely. In practice, programmers mostly rely on informal descriptions (e.g., in English).

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`
`3.14 * 1.0`	

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`
`3.14 * 1.0`	`3.14`

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`
`3.14 * 1.0`	`3.14`
`"Hello "++"world"`	

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`
`3.14 * 1.0`	`3.14`
`"Hello "++"world"`	`"Hello world"`

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`
`3.14 * 1.0`	`3.14`
`"Hello "++"world"`	`"Hello world"`
`not False`	

Semantics: Evaluating Expressions to Values

Executing a functional program amounts to evaluating an expression. This yields a **value**. Values are expressions that cannot be evaluated further. (Values are also said to be in **normal form**.)

Expressions that are not values	Expressions that are values
`1 + 2`	`3`
`3.14 * 1.0`	`3.14`
`"Hello "++"world"`	`"Hello world"`
`not False`	`True`

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

```
3 + 4 * 2 < 5 * 2
```

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

```
   3 + 4 * 2 < 5 * 2
⟶ 3 +   8   < 5 * 2
```

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

```
   3 + 4 * 2 < 5 * 2
⟶ 3 +   8   < 5 * 2
⟶     11     < 5 * 2
```

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

```
    3 + 4 * 2 < 5 * 2
⟶  3 +   8   < 5 * 2
⟶      11     < 5 * 2
⟶      11     <  10
```

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

```
   3 + 4 * 2 < 5 * 2
⟶ 3 +   8  < 5 * 2
⟶     11    < 5 * 2
⟶     11    <  10
⟶       False
```

Evaluation of Expressions

For complex expressions, evaluation happens in several steps.

```
    3 + 4 * 2 < 5 * 2
--> 3 +   8  < 5 * 2
-->     11    < 5 * 2
-->     11    <  10
-->       False
```

To evaluate functions applied to arguments, Haskell uses an evaluation strategy known as **non-strict evaluation**. Arguments are evaluated only if their value is needed in the evaluation of the function body. More on this later.

Evaluation Order

An expression is called **pure** if it has no side effects (such as I/O).

Evaluation Order

An expression is called **pure** if it has no side effects (such as I/O).

For pure expressions, the order of evaluation does not matter—the result is always the same.

Evaluation Order

An expression is called **pure** if it has no side effects (such as I/O).

For pure expressions, the order of evaluation does not matter—the result is always the same.

```
2*3 + 4*5 ⟶ 6 + 4*5 ⟶ 6 + 20 ⟶ 26
```

Evaluation Order

An expression is called **pure** if it has no side effects (such as I/O).

For pure expressions, the order of evaluation does not matter—the result is always the same.

2*3 + 4*5 ⟶ 6 + 4*5 ⟶ 6 + 20 ⟶ 26
2*3 + 4*5 ⟶ 2*3 + 20 ⟶ 6 + 20 ⟶ 26

Evaluation Order

An expression is called **pure** if it has no side effects (such as I/O).

For pure expressions, the order of evaluation does not matter—the result is always the same.

$2*3 + 4*5 \longrightarrow 6 + 4*5 \longrightarrow 6 + 20 \longrightarrow 26$

$2*3 + 4*5 \longrightarrow 2*3 + 20 \longrightarrow 6 + 20 \longrightarrow 26$

Even simultaneous evaluation is possible:

$2*3 + 4*5 \longrightarrow 6 + 20 \longrightarrow 26$

Evaluation Order

An expression is called **pure** if it has no side effects (such as I/O).

For pure expressions, the order of evaluation does not matter—the result is always the same.

$2*3 + 4*5 \longrightarrow 6 + 4*5 \longrightarrow 6 + 20 \longrightarrow 26$

$2*3 + 4*5 \longrightarrow 2*3 + 20 \longrightarrow 6 + 20 \longrightarrow 26$

Even simultaneous evaluation is possible:

$2*3 + 4*5 \longrightarrow 6 + 20 \longrightarrow 26$

Therefore, functional programming is well-suited for parallel computing (e.g., multi-core machines).

Whitespace

You may freely insert spaces between expressions:

```
Prelude> 1+2
3
Prelude> 1  +    2
3
```

Whitespace

You may freely insert spaces between expressions:

```
Prelude> 1+2
3
Prelude> 1  +    2
3
```

Spaces are usually not required, except to separate different names: e.g., `length x` is not the same as `lengthx`

Whitespace

You may freely insert spaces between expressions:

```
Prelude> 1+2
3
Prelude> 1  +    2
3
```

Spaces are usually not required, except to separate different names: e.g., `length x` is not the same as `lengthx`

Later, we will see programs where spaces do matter.

Comments

Haskell programs may contain **comments**, i.e., text that is not evaluated.

Comments

Haskell programs may contain **comments**, i.e., text that is not evaluated.

Single line comments begin with `--`:

```
Prelude> 1+2 -- this is a comment
3
```

Comments

Haskell programs may contain **comments**, i.e., text that is not evaluated.

Single line comments begin with `--`:

```
Prelude> 1+2 -- this is a comment
3
```

Comments can also be enclosed in `{- ... -}`:

```
Prelude> 1 + {- another comment -} 2
3
```

Comments

Haskell programs may contain **comments**, i.e., text that is not evaluated.

Single line comments begin with `--`:

```
Prelude> 1+2 -- this is a comment
3
```

Comments can also be enclosed in `{- ... -}`:

```
Prelude> 1 + {- another comment -} 2
3
```

These comments may extend over several lines when used in Haskell programs.

Runtime Errors (aka Exceptions)

Not every expression can be evaluated to a (sensible) value: e.g.,

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

Runtime Errors (aka Exceptions)

Not every expression can be evaluated to a (sensible) value: e.g.,

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

Note that `div 1 0` is a valid expression. There is no error when it is compiled. However, when the expression is evaluated, there is a runtime error, known as an **exception**.

Runtime Errors (aka Exceptions)

Not every expression can be evaluated to a (sensible) value: e.g.,

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

Note that `div 1 0` is a valid expression. There is no error when it is compiled. However, when the expression is evaluated, there is a runtime error, known as an **exception**.

Later, we'll see how to recover from an exception, and how you can write your own functions that throw exceptions.

Types

Programs (usually) manipulate data. There are many different types of data: numbers, characters, strings, dates and times, Booleans, . . .

Types

Programs (usually) manipulate data. There are many different types of data: numbers, characters, strings, dates and times, Booleans, . . .

A **(data) type** is a set of data with similar properties: e.g., possible values, operations that can be applied, and meaning.

Types

Programs (usually) manipulate data. There are many different types of data: numbers, characters, strings, dates and times, Booleans, . . .

A **(data) type** is a set of data with similar properties: e.g., possible values, operations that can be applied, and meaning.

At the hardware level, data is (mostly) just bits and bytes. Consequently, machine code is (mostly) untyped. However, many high-level languages offer data types.

Basic Types in Haskell

- Integers (`Integer`): `0`, `1`, `2`, `-10`, `42`, ...

Basic Types in Haskell

- Integers (`Integer`): `0`, `1`, `2`, `-10`, `42`, ...
- Floating point numbers (`Double`): `0.0`, `-10.4`, `3.14159`, `1.234e6`, ...

Basic Types in Haskell

- Integers (`Integer`): `0`, `1`, `2`, `-10`, `42`, ...
- Floating point numbers (`Double`): `0.0`, `-10.4`, `3.14159`, `1.234e6`, ...
- Characters (`Char`): `'a'`, `'B'`, `'1'`, ...

Basic Types in Haskell

- Integers (`Integer`): `0`, `1`, `2`, `-10`, `42`, `...`
- Floating point numbers (`Double`): `0.0`, `-10.4`, `3.14159`, `1.234e6`, `...`
- Characters (`Char`): `'a'`, `'B'`, `'1'`, `...`
- Strings (`String`): `"Hello"`, `"PKD"`, `"1"`, `...`

Basic Types in Haskell

- Integers (`Integer`): `0, 1, 2, -10, 42, ...`
- Floating point numbers (`Double`): `0.0, -10.4, 3.14159, 1.234e6, ...`
- Characters (`Char`): `'a', 'B', '1', ...`
- Strings (`String`): `"Hello", "PKD", "1", ...`
- Booleans (`Bool`): `True, False`

Basic Types in Haskell

- Integers (`Integer`): `0, 1, 2, -10, 42, ...`
- Floating point numbers (`Double`): `0.0, -10.4, 3.14159, 1.234e6, ...`
- Characters (`Char`): `'a', 'B', '1', ...`
- Strings (`String`): `"Hello", "PKD", "1", ...`
- Booleans (`Bool`): `True, False`
- Unit (`()`): has only one value, written `()`

Basic Types in Haskell

- Integers (`Integer`): `0`, `1`, `2`, `-10`, `42`, ...
- Floating point numbers (`Double`): `0.0`, `-10.4`, `3.14159`, `1.234e6`, ...
- Characters (`Char`): `'a'`, `'B'`, `'1'`, ...
- Strings (`String`): `"Hello"`, `"PKD"`, `"1"`, ...
- Booleans (`Bool`): `True`, `False`
- Unit (`()`): has only one value, written `()`

Note that `1`, `1.0`, `'1'` and `"1"` are all different.

Seeing Types in GHCi

You can use `:type expr` to have GHCi show the type of `expr`.

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t not True
not True :: Bool
```

Seeing Types in GHCi

You can use `:type expr` to have GHCi show the type of `expr`.

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t not True
not True :: Bool
```

You can also `:set +t` to see the type of all expressions that you enter.

```
Prelude> :set +t
Prelude> 'a'
'a'
it :: Char
Prelude> not True
False
it :: Bool
```

Function Types

Also functions have types. For instance,

```
Prelude> :t not
not :: Bool -> Bool
```

Function Types

Also functions have types. For instance,

```
Prelude> :t not
not :: Bool -> Bool
```

Here, the first `Bool` is called the **argument type** and the second `Bool` is called the **result type** of the function.

Function Types

Also functions have types. For instance,

```
Prelude> :t not
not :: Bool -> Bool
```

Here, the first `Bool` is called the **argument type** and the second `Bool` is called the **result type** of the function.

-> is a **type constructor**: when applied to two existing types, it constructs a new type (namely, a function type).

Expression Language, Type Language

We have seen ways to write Haskell expressions, such as

```
2 + 15
not True
()
```

Expression Language, Type Language

We have seen ways to write Haskell expressions, such as

```
2 + 15
not True
()
```

We have seen ways to write Haskell types, such as

```
Integer
Bool -> Bool
()
```

Expression Language, Type Language

We have seen ways to write Haskell expressions, such as

```
2 + 15
not True
()
```

We have seen ways to write Haskell types, such as

```
Integer
Bool -> Bool
()
```

When we write a program, it is clear from the context whether we are writing an expression or a type. For instance,

```
expr :: Type
```

Type Checking

What should `32 + "1"` mean?

Type Checking

What should `32 + "1"` mean?

Adding values of different types is not obviously meaningful. Different programming languages offer different semantics.

Type Checking

What should `32 + "1"` mean?

Adding values of different types is not obviously meaningful. Different programming languages offer different semantics.

In a **strongly typed** language, this is never allowed.

Type Checking

What should `32 + "1"` mean?

Adding values of different types is not obviously meaningful. Different programming languages offer different semantics.

In a **strongly typed** language, this is never allowed.

In a **weakly typed** language, the result depends (on the language). It could be `33`, or `"321"`, or . . .

Static vs. Dynamic Type Checking

If `32 + "1"` is not allowed, when exactly do you get an error?

Static vs. Dynamic Type Checking

If `32 + "1"` is not allowed, when exactly do you get an error?

In a language with **static** type checking, the compiler inspects the program and generates an error. Translation to machine code fails.

Static vs. Dynamic Type Checking

If `32 + "1"` is not allowed, when exactly do you get an error?

In a language with **static** type checking, the compiler inspects the program and generates an error. Translation to machine code fails.

In a language with **dynamic** type checking, the compiler translates the program to machine code that performs type checking at run time, i.e., that generates an error when it is executed.

Type Checking in Haskell

Haskell is a strongly typed language with static type checking. This helps to prevent programming errors.

Type Checking in Haskell

Haskell is a strongly typed language with static type checking. This helps to prevent programming errors.

Every expression has a type.

Type Inference

In many languages, the programmer needs to declare the type of variables, functions, etc.

Type Inference

In many languages, the programmer needs to declare the type of variables, functions, etc.

In Haskell, this is (mostly) not necessary. Instead, the types of expressions are automatically **inferred** by (a type inference algorithm that is part of) the Haskell implementation.

Type Inference

In many languages, the programmer needs to declare the type of variables, functions, etc.

In Haskell, this is (mostly) not necessary. Instead, the types of expressions are automatically **inferred** by (a type inference algorithm that is part of) the Haskell implementation.

This happens statically, i.e., without running the program.

Type Errors

Ill-typed expressions, similar to syntax errors, cause **error messages**:

Type Errors

Ill-typed expressions, similar to syntax errors, cause **error messages**:

```
Prelude> 32 + "1"
```

Type Errors

Ill-typed expressions, similar to syntax errors, cause **error messages**:

```
Prelude> 32 + "1"

<interactive>:2:4:
    No instance for (Num [Char]) arising from a use of `+'
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 32 + "1"
    In an equation for `it': it = 32 + "1"
```

Type Errors

Ill-typed expressions, similar to syntax errors, cause **error messages**:

```
Prelude> 32 + "1"

<interactive>:2:4:
    No instance for (Num [Char]) arising from a use of `+'
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: 32 + "1"
    In an equation for `it': it = 32 + "1"
```

Again, **try to understand** these error messages! And be patient with GHC: it can't know what you really had in mind ...