Functions

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

- Source files
- Functions:
  Declaration, Evaluation, Type Inference, ...
- Abstraction

Source Files

Source Files

So far, we have been using Haskell interactively: we have entered programs (expressions and declarations) at the GHCi prompt by hand. This quickly becomes inconvenient for larger programs.

Source Files

So far, we have been using Haskell interactively: we have entered programs (expressions and declarations) at the GHCi prompt by hand. This quickly becomes inconvenient for larger programs.

Programs are normally written with a text editor and saved in a file (i.e., a resource for persistent storage of information).

Source Files

So far, we have been using Haskell interactively: we have entered programs (expressions and declarations) at the GHCi prompt by hand. This quickly becomes inconvenient for larger programs.

Programs are normally written with a text editor and saved in a file (i.e., a resource for persistent storage of information).

By convention, we use the extension `.hs` for files that contain Haskell source code: e.g., `example.hs`.

Text Editors

To edit source files, you will need a **text editor**. We normally recommend Emacs, but there are several other options, like

- Atom
- Visual Studio Code
- Geany
- Vim

(and you can use any editor you like).

Text Editors

To edit source files, you will need a **text editor**. We normally recommend Emacs, but there are several other options, like

- Atom
- Visual Studio Code
- Geany
- Vim

(and you can use any editor you like).

See `https://www.haskell.org/haskellwiki/Editors` for a list of text editors with support for Haskell syntax highlighting and formatting.

Text Editors

To edit source files, you will need a **text editor**. We normally recommend Emacs, but there are several other options, like

- Atom
- Visual Studio Code
- Geany
- Vim

(and you can use any editor you like).

See `https://www.haskell.org/haskellwiki/Editors` for a list of text editors with support for Haskell syntax highlighting and formatting.

Install one today!

Haskell Sources: Declarations, Declarations, Declarations

A Haskell source file consists of declarations (and, optionally, their type signatures).

Haskell Sources: Declarations, Declarations, Declarations

A Haskell source file consists of declarations (and, optionally, their type signatures).

For instance, in a file `example.hs`:

Haskell Sources: Declarations, Declarations, Declarations

A Haskell source file consists of declarations (and, optionally, their type signatures).

For instance, in a file `example.hs`:

```
x = 42

greeting = "Hello!"

takeLast :: Int->String->String
takeLast n s = drop (length s - n) s
```

Loading Source Files in GHCi

In GHCi, you can use `:load` to load a Haskell source file:

Loading Source Files in GHCi

In GHCi, you can use `:load` to load a Haskell source file:

```
Prelude> :l example
[1 of 1] Compiling Main             ( example.hs, interpreted )
Ok, modules loaded: Main.
```

Loading Source Files in GHCi

In GHCi, you can use `:load` to load a Haskell source file:

```
Prelude> :l example
[1 of 1] Compiling Main             ( example.hs, interpreted )
Ok, modules loaded: Main.
```

After loading a file, you may refer to names that were bound in the file:

Loading Source Files in GHCi

In GHCi, you can use `:load` to load a Haskell source file:

```
Prelude> :l example
[1 of 1] Compiling Main             ( example.hs, interpreted )
Ok, modules loaded: Main.
```

After loading a file, you may refer to names that were bound in the file:

```
*Main> greeting
"Hello!"
```

Haskell Sources: Fine Points

- Top-level expressions are not allowed.

Haskell Sources: Fine Points

- Top-level expressions are not allowed.
- Usually, each declaration is given on a separate line.

Haskell Sources: Fine Points

- Top-level expressions are not allowed.
- Usually, each declaration is given on a separate line.
- The order of declarations doesn't matter.

Haskell Sources: Fine Points

- Top-level expressions are not allowed.
- Usually, each declaration is given on a separate line.
- The order of declarations doesn't matter.
- Multiple declarations of the same identifier are not allowed.

Haskell Sources: Fine Points

- Top-level expressions are not allowed.
- Usually, each declaration is given on a separate line.
- The order of declarations doesn't matter.
- Multiple declarations of the same identifier are not allowed.

From now on, I'll usually write code the way it is written in a source file, rather than how it may be entered at the GHCi prompt.

Indentation Matters

In Haskell (like in many other programming languages), it is important how you indent your code.

Indentation Matters

In Haskell (like in many other programming languages),
it is important how you indent your code.

The Golden Rule of Indentation

Code that is part of some expression or declaration should be indented further than the line where that expression (or declaration) starts.

Indentation Matters

In Haskell (like in many other programming languages),
it is important how you indent your code.

The Golden Rule of Indentation

Code that is part of some expression or declaration should be indented further than the line where that expression (or declaration) starts.

See `http://en.wikibooks.org/wiki/Haskell/Indentation` for examples and further explanations.

Functions

Declaration, Evaluation, Type Inference, ...

Motivation: Code Re-Use

This expression returns the last five characters from the string `"computer"`:

```
drop (length "computer" - 5) "computer"
```

Motivation: Code Re-Use

This expression returns the last five characters from the string `"computer"`:

```
drop (length "computer" - 5) "computer"
```

If we want to obtain a different number of characters from the end of a different string, we have to change the expression:

```
drop (length "programming" - 7) "programming"
```

Motivation: Code Re-Use

This expression returns the last five characters from the string `"computer"`:

```
drop (length "computer" - 5) "computer"
```

If we want to obtain a different number of characters from the end of a different string, we have to change the expression:

```
drop (length "programming" - 7) "programming"
```

If we write a program that needs to extract the last few characters from many different strings, we would end up copying (and changing) this expression many times.

Why Copy-and-Paste Programming is Bad

Copy-and-paste programming is considered an **anti-pattern** in software engineering, i.e., an ineffective (and even counterproductive) solution. Avoid it as much as you can!

Why Copy-and-Paste Programming is Bad

Copy-and-paste programming is considered an **anti-pattern** in software engineering, i.e., an ineffective (and even counterproductive) solution. Avoid it as much as you can!

Problems:

- Long, repeated sections of code can be difficult to understand.
  (Even realizing that the code is a copy takes time.)

Why Copy-and-Paste Programming is Bad

Copy-and-paste programming is considered an **anti-pattern** in software engineering, i.e., an ineffective (and even counterproductive) solution. Avoid it as much as you can!

Problems:

- Long, repeated sections of code can be difficult to understand.
  (Even realizing that the code is a copy takes time.)
- It can be difficult to spot the differences between copies,
  and to understand the specific purpose of each copy.

Why Copy-and-Paste Programming is Bad

Copy-and-paste programming is considered an **anti-pattern** in software engineering, i.e., an ineffective (and even counterproductive) solution. Avoid it as much as you can!

Problems:

- Long, repeated sections of code can be difficult to understand.
  (Even realizing that the code is a copy takes time.)
- It can be difficult to spot the differences between copies,
  and to understand the specific purpose of each copy.
- If it becomes necessary to make a change, every copy needs to be considered
  (and may need to be changed in a different way).

Why Copy-and-Paste Programming is Bad

Copy-and-paste programming is considered an **anti-pattern** in software engineering, i.e., an ineffective (and even counterproductive) solution. Avoid it as much as you can!

Problems:

- Long, repeated sections of code can be difficult to understand.
  (Even realizing that the code is a copy takes time.)
- It can be difficult to spot the differences between copies,
  and to understand the specific purpose of each copy.
- If it becomes necessary to make a change, every copy needs to be considered
  (and may need to be changed in a different way).

Why Copy-and-Paste Programming is Bad

Copy-and-paste programming is considered an **anti-pattern** in software engineering, i.e., an ineffective (and even counterproductive) solution. Avoid it as much as you can!

Problems:

- Long, repeated sections of code can be difficult to understand.
  (Even realizing that the code is a copy takes time.)
- It can be difficult to spot the differences between copies,
  and to understand the specific purpose of each copy.
- If it becomes necessary to make a change, every copy needs to be considered
  (and may need to be changed in a different way).

$\Longrightarrow$ Code becomes harder to maintain.

"Great! Now just make her blonde."

Functions to the Rescue

We want to write code that solves the problem of selecting the last few characters from a string once and for all, i.e., independently of the particular string and number of characters.

Functions to the Rescue

We want to write code that solves the problem of selecting the last few characters from a string once and for all, i.e., independently of the particular string and number of characters.

```
takeLast n str = drop (length str - n) str
```

Functions to the Rescue

We want to write code that solves the problem of selecting the last few characters from a string once and for all, i.e., independently of the particular string and number of characters.

```
takeLast n str = drop (length str - n) str
```

This is a **function declaration**. It defines a function `takeLast`. Here, `n` and `str` are names for the arguments of the function, i.e., for the data that is passed to the function.

Functions to the Rescue

We want to write code that solves the problem of selecting the last few characters from a string once and for all, i.e., independently of the particular string and number of characters.

```
takeLast n str = drop (length str - n) str
```

This is a **function declaration**. It defines a function `takeLast`. Here, `n` and `str` are names for the arguments of the function, i.e., for the data that is passed to the function.

Haskell programs typically declare many functions.

Function Declarations in Haskell

Here is another example:

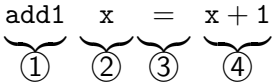
Function Declarations in Haskell

Here is another example:

$$\underbrace{\texttt{add1}}_{①}\ \underbrace{\texttt{x}}_{②}\ \underbrace{=}_{③}\ \underbrace{\texttt{x} + 1}_{④}$$

Function Declarations in Haskell

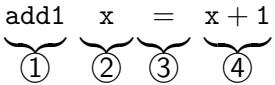
Here is another example:



1. the **name** of the function

Function Declarations in Haskell

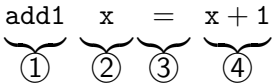
Here is another example:



1. the **name** of the function
2. a **formal argument** (there may be several)

Function Declarations in Haskell

Here is another example:



1. the **name** of the function
2. a **formal argument** (there may be several)
3. syntax for declarations in Haskell

Function Declarations in Haskell

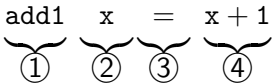
Here is another example:

$$\underbrace{\texttt{add1}}_{①}\ \underbrace{\texttt{x}}_{②}\ \underbrace{=}_{③}\ \underbrace{\texttt{x} + 1}_{④}$$

1. the **name** of the function
2. a **formal argument** (there may be several)
3. syntax for declarations in Haskell
4. the **body** of the function

Function Declarations in Haskell

Here is another example:



1. the **name** of the function
2. a **formal argument** (there may be several)
3. syntax for declarations in Haskell
4. the **body** of the function

A function declaration binds the name of the function (here: `add1`) to a value of function type.

Function Application

We have already seen the Haskell syntax to **apply** a function `f` to an argument `x`: it's just

```
f x
```

Function Application

We have already seen the Haskell syntax to **apply** a function `f` to an argument `x`: it's just

```
f x
```

Both built-in functions and your own functions are applied in this way.

Function Application

We have already seen the Haskell syntax to **apply** a function `f` to an argument `x`: it's just

```
f x
```

Both built-in functions and your own functions are applied in this way.

```
Prelude> length "computer"
8
Prelude> add1 42
43
Prelude> takeLast 5 "computer"
"puter"
```

Function Application

We have already seen the Haskell syntax to **apply** a function `f` to an argument `x`: it's just

```
f x
```

Both built-in functions and your own functions are applied in this way.

```
Prelude> length "computer"
8
Prelude> add1 42
43
Prelude> takeLast 5 "computer"
"puter"
```

Infix operators, such as +, are just functions of two arguments that are written between their arguments.

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3+4)
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3+4)
```

$\longrightarrow$ `x + 1` (where `x = 3+4`)

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3+4)
```

$\longrightarrow$ `x + 1` (where `x = 3+4`)

$\longrightarrow$ `x + 1` (where `x = 7`)

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3+4)
```
$\longrightarrow$ `x + 1` (where `x = 3+4`)
$\longrightarrow$ `x + 1` (where `x = 7`)
$\longrightarrow$ `7 + 1` (where `x = 7`)

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3+4)
  ⟶ x + 1 (where x = 3+4)
  ⟶ x + 1 (where x = 7)
  ⟶ 7 + 1 (where x = 7)
  ⟶ 8
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
```

$\longrightarrow$ `x + 1` (where `x = 3 * add1 4`)

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
  ⟶ x + 1 (where x = 3 * add1 4)
  ⟶ x + 1 (where x = 3 * [x+1 (where x=4)])
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
  ⟶ x + 1 (where x = 3 * add1 4)
  ⟶ x + 1 (where x = 3 * [x+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * [4+1 (where x=4)])
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
  ⟶ x + 1 (where x = 3 * add1 4)
  ⟶ x + 1 (where x = 3 * [x+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * [4+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * 5)
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
  ⟶ x + 1 (where x = 3 * add1 4)
  ⟶ x + 1 (where x = 3 * [x+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * [4+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * 5)
  ⟶ x + 1 (where x = 15)
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
  ⟶ x + 1 (where x = 3 * add1 4)
  ⟶ x + 1 (where x = 3 * [x+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * [4+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * 5)
  ⟶ x + 1 (where x = 15)
  ⟶ 15 + 1 (where x = 15)
```

Evaluation of Function Application

When a function is applied to (actual) arguments,
the function application is *replaced by the body of the function*.
This is then evaluated in an environment
where *formal arguments are bound to corresponding* **actual arguments**.

Example: `add1 x = x + 1`

```
add1 (3 * add1 4)
  ⟶ x + 1 (where x = 3 * add1 4)
  ⟶ x + 1 (where x = 3 * [x+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * [4+1 (where x=4)])
  ⟶ x + 1 (where x = 3 * 5)
  ⟶ x + 1 (where x = 15)
  ⟶ 15 + 1 (where x = 15)
  ⟶ 16
```

Evaluation of Function Application: Fine Points

Haskell uses lazy evaluation, so actual arguments are **not evaluated until (and unless) they are used** in the evaluation of the function body.

Evaluation of Function Application: Fine Points

Haskell uses lazy evaluation, so actual arguments are **not evaluated until (and unless) they are used** in the evaluation of the function body.

Example: `fortytwo x = 42`

Evaluation of Function Application: Fine Points

Haskell uses lazy evaluation, so actual arguments are **not evaluated until (and unless) they are used** in the evaluation of the function body.

Example: `fortytwo x = 42`

```
fortytwo (div 1 0)
```

Evaluation of Function Application: Fine Points

Haskell uses lazy evaluation, so actual arguments are **not evaluated until (and unless) they are used** in the evaluation of the function body.

Example: `fortytwo x = 42`

```
fortytwo (div 1 0)
```

$\longrightarrow$ `42` (where `x = div 1 0`)

Evaluation of Function Application: Fine Points

Haskell uses lazy evaluation, so actual arguments are **not evaluated until (and unless) they are used** in the evaluation of the function body.

Example: `fortytwo x = 42`

```
fortytwo (div 1 0)
  ⟶ 42 (where x = div 1 0)
  ⟶ 42
```

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

```
square (3+4)
```

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

```
square (3+4)
  ⟶ x * x (where x = 3+4)
```

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

```
square (3+4)
  ⟶ x * x (where x = 3+4)
  ⟶ x * x (where x = 7)
```

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

```
square (3+4)
  ⟶ x * x (where x = 3+4)
  ⟶ x * x (where x = 7)
  ⟶ 7 * x (where x = 7)
```

We do not need to evaluate 3+4 again.

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

```
square (3+4)
  ⟶ x * x (where x = 3+4)
  ⟶ x * x (where x = 7)
  ⟶ 7 * x (where x = 7)
  ⟶ 7 * 7 (where x = 7)
```

We do not need to evaluate 3+4 again.

Evaluation of Function Application: Fine Points (cont.)

Each actual argument is **evaluated at most once**, no matter how often the corresponding formal argument appears in the body of the function.

Example: `square x = x * x`

```
square (3+4)
  ⟶ x * x (where x = 3+4)
  ⟶ x * x (where x = 7)
  ⟶ 7 * x (where x = 7)
  ⟶ 7 * 7 (where x = 7)
  ⟶ 49
```

We do not need to evaluate 3+4 again.

Type Inference for Functions

In Haskell, the type of expressions (including function expressions) is **inferred** automatically.

Type Inference for Functions

In Haskell, the type of expressions (including function expressions) is **inferred** automatically.

```
Prelude> add1 x = x+1
Prelude> :t add1
add1 :: Num a => a -> a
```

Type Inference for Functions

In Haskell, the type of expressions (including function expressions) is **inferred** automatically.

```
Prelude> add1 x = x+1
Prelude> :t add1
add1 :: Num a => a -> a
```

In many other statically typed programming languages, the programmer needs to specify the argument types and result type of a function.

Type Inference for Functions

In Haskell, the type of expressions (including function expressions) is **inferred** automatically.

```
Prelude> add1 x = x+1
Prelude> :t add1
add1 :: Num a => a -> a
```

In many other statically typed programming languages, the programmer needs to specify the argument types and result type of a function.

However, it is possible to specify type signatures manually: e.g.,

```
Prelude> add1 :: Integer->Integer; add1 x = x+1
Prelude> :t add1
add1 :: Integer -> Integer
```

Type Inference for Functions

In Haskell, the type of expressions (including function expressions) is **inferred** automatically.

```
Prelude> add1 x = x+1
Prelude> :t add1
add1 :: Num a => a -> a
```

In many other statically typed programming languages, the programmer needs to specify the argument types and result type of a function.

However, it is possible to specify type signatures manually: e.g.,

```
Prelude> add1 :: Integer->Integer; add1 x = x+1
Prelude> :t add1
add1 :: Integer -> Integer
```

This is often a good idea—it makes code easier to understand.

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

1. `isAt` is a function that takes a single argument.
   Hence, the type of `isAt` is `... -> ...` (where we still need to fill in the dots).

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

1. `isAt` is a function that takes a single argument.
   Hence, the type of `isAt` is `... -> ...` (where we still need to fill in the dots).
2. In the body, `==` is applied to `x` and `'@'`. `'@'` is of type `Char`.
   Hence, `==` must be equality on `Char`, i.e., must have type `Char->Char->Bool`.

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

1. `isAt` is a function that takes a single argument.
   Hence, the type of `isAt` is `... -> ...` (where we still need to fill in the dots).
2. In the body, `==` is applied to `x` and `'@'`. `'@'` is of type `Char`.
   Hence, `==` must be equality on `Char`, i.e., must have type `Char->Char->Bool`.
3. Hence `x` must have type `Char`, so the argument type of `isAt` is `Char`.

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

1. `isAt` is a function that takes a single argument.
   Hence, the type of `isAt` is `... -> ...` (where we still need to fill in the dots).
2. In the body, `==` is applied to `x` and `'@'`. `'@'` is of type `Char`.
   Hence, `==` must be equality on `Char`, i.e., must have type `Char->Char->Bool`.
3. Hence `x` must have type `Char`, so the argument type of `isAt` is `Char`.
4. The return type of `==` is `Bool`. Hence, the return type of `isAt` is `Bool`.

Type Inference for Functions: Example

Haskell uses a sophisticated algorithm to infer the type of functions.
We'll only look at a simple example:

```
isAt x = x == '@'
```

1. `isAt` is a function that takes a single argument.
   Hence, the type of `isAt` is `... -> ...` (where we still need to fill in the dots).
2. In the body, `==` is applied to `x` and `'@'`. `'@'` is of type `Char`.
   Hence, `==` must be equality on `Char`, i.e., must have type `Char->Char->Bool`.
3. Hence `x` must have type `Char`, so the argument type of `isAt` is `Char`.
4. The return type of `==` is `Bool`. Hence, the return type of `isAt` is `Bool`.

Hence, the type of `isAt` is `Char -> Bool`

Ill-Typed Function Declarations

Function declarations may be rejected because of type errors:

Ill-Typed Function Declarations

Function declarations may be rejected because of type errors:

```
Prelude> oops x = x + length x

<interactive>:...:
    Couldn't match expected type `[a0]' with actual type `Int'
    In the first argument of `length', namely `x'
    In the second argument of `(+)', namely `length x'
    In the expression: x + length x
```

Ill-Typed Function Declarations

Function declarations may be rejected because of type errors:

```
Prelude> oops x = x + length x

<interactive>:...:
    Couldn't match expected type `[a0]' with actual type `Int'
    In the first argument of `length', namely `x'
    In the second argument of `(+)', namely `length x'
    In the expression: x + length x
```

Here, `length` requires an argument of type (e.g.) `String`.
But + requires arguments of type `Int`.

Ill-Typed Function Declarations

Function declarations may be rejected because of type errors:

```
Prelude> oops x = x + length x

<interactive>:...:
    Couldn't match expected type `[a0]' with actual type `Int'
    In the first argument of `length', namely `x'
    In the second argument of `(+)', namely `length x'
    In the expression: x + length x
```

Here, `length` requires an argument of type (e.g.) `String`.
But + requires arguments of type `Int`.

`x` cannot be both of type `String` and `Int` at the same time!

Functions Are First-Class

In functional programming, functions are **first-class citizens**, meaning they can be treated like other types of data.

Functions Are First-Class

In functional programming, functions are **first-class citizens**,
meaning they can be treated like other types of data.

Specifically, functions can be used as expressions, passed as arguments to other functions, returned from functions, etc.

Functions Are First-Class

In functional programming, functions are **first-class citizens**,
meaning they can be treated like other types of data.

Specifically, functions can be used as expressions, passed as arguments to other functions, returned from functions, etc.

For instance, the following if statement evaluates to a function:

```
if 2+2==5 then abs else add1
```

Functions Are First-Class

In functional programming, functions are **first-class citizens**,
meaning they can be treated like other types of data.

Specifically, functions can be used as expressions, passed as arguments to other functions, returned from functions, etc.

For instance, the following if statement evaluates to a function:

```
if 2+2==5 then abs else add1
```

```
Prelude> (if 2+2==5 then abs else add1) 42
43
```

Anonymous Functions

It is possible to write function expressions directly, without first declaring a name for the function. These are called **anonymous functions** or **lambda abstractions**.

Anonymous Functions

It is possible to write function expressions directly, without first declaring a name for the function. These are called **anonymous functions** or **lambda abstractions**.

```
Prelude> (\x -> x + 1) 42
43
```

Anonymous Functions

It is possible to write function expressions directly, without first declaring a name for the function. These are called **anonymous functions** or **lambda abstractions**.

```
Prelude> (\x -> x + 1) 42
43

Prelude> (\n s -> drop (length s - n) s) 5 "computer"
"puter"
```

Anonymous Functions

It is possible to write function expressions directly, without first declaring a name for the function. These are called **anonymous functions** or **lambda abstractions**.

```
Prelude> (\x -> x + 1) 42
43

Prelude> (\n s -> drop (length s - n) s) 5 "computer"
"puter"
```

(Often, there are more elegant ways in Haskell to express the same without using an anonymous function. We'll cover those later.)

Declarations: Functions vs. Values

It is possible to bind an identifier to a function expression.

Declarations: Functions vs. Values

It is possible to bind an identifier to a function expression.

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

Declarations: Functions vs. Values

It is possible to bind an identifier to a function expression.

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

The following are (mostly) equivalent.

Declarations: Functions vs. Values

It is possible to bind an identifier to a function expression.

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

The following are (mostly) equivalent.

```
Prelude> square x = x * x

Prelude> square = \x -> x * x
```

Functions Cannot Be Tested For Equality

In Haskell, it is not possible to compare two functions with == or /=.

Functions Cannot Be Tested For Equality

In Haskell, it is not possible to compare two functions with == or /=.

Attempting to do so will result in a type error:

```
Prelude> addition = (+)
Prelude> addition == (+)

<interactive>:...:
    No instance for (Eq (Integer -> Integer -> Integer))
      arising from a use of `=='
    ...
```

Functions Cannot Be Tested For Equality (cont.)

Reason: It can be difficult to figure out whether two functions are equal. For instance, consider

```
\x -> 2*x
\x -> x+x
```

Functions Cannot Be Tested For Equality (cont.)

Reason: It can be difficult to figure out whether two functions are equal. For instance, consider

```
\x -> 2*x
\x -> x+x
```

These are equal (i.e., they denote the same function), but how should Haskell know this?

Functions Cannot Be Tested For Equality (cont.)

Reason: It can be difficult to figure out whether two functions are equal. For instance, consider

```
\x -> 2*x
\x -> x+x
```

These are equal (i.e., they denote the same function), but how should Haskell know this?

In fact, in general it is impossible (i.e., there is no algorithm) to figure out whether two functions are equal—this is a fundamental result by Church.

Abstraction

Abstract and Concrete

Abstraction is a process by which (abstract) concepts are derived from literal (concrete) concepts, often by ignoring certain properties or details.

Abstract and Concrete

Abstraction is a process by which (abstract) concepts are derived from literal (concrete) concepts, often by ignoring certain properties or details.

Suppose you want to take the bus from Stenhagen to Sävja. Then you do not need to know the precise vehicle that you should go with—which will be different from time to time—but it is enough to know that you should take "bus route 5".

Abstract and Concrete

Abstraction is a process by which (abstract) concepts are derived from literal (concrete) concepts, often by ignoring certain properties or details.

Suppose you want to take the bus from Stenhagen to Sävja. Then you do not need to know the precise vehicle that you should go with—which will be different from time to time—but it is enough to know that you should take "bus route 5".

"Bus route 5" is an abstraction of the buses that actually run between Stenhagen and Sävja.

Abstraction in Programming

Abstraction is an extremely powerful concept in programming (and more generally in computer science). It allows us to write programs without having to worry about **irrelevant details**.

Abstraction in Programming

Abstraction is an extremely powerful concept in programming (and more generally in computer science). It allows us to write programs without having to worry about **irrelevant details**.

We've already seen one example of this: high-level programming languages abstract from hardware. We are happily writing Haskell programs without knowing the machine code that our computers will execute for them.

Abstraction in Programming

Abstraction is an extremely powerful concept in programming (and more generally in computer science). It allows us to write programs without having to worry about **irrelevant details**.

We've already seen one example of this: high-level programming languages abstract from hardware. We are happily writing Haskell programs without knowing the machine code that our computers will execute for them.

And we don't need to know—because there is a **contract** (namely the semantics of Haskell) that this machine code must fulfill.

Data Abstraction

Data abstraction separates the operations that are available for data from the concrete implementation of data, i.e., from the way data is represented in the computer.

Data Abstraction

Data abstraction separates the operations that are available for data from the concrete implementation of data, i.e., from the way data is represented in the computer.

For instance, how are integers (type `Integer`) represented in memory?

Data Abstraction

Data abstraction separates the operations that are available for data from the concrete implementation of data, i.e., from the way data is represented in the computer.

For instance, how are integers (type `Integer`) represented in memory?

We don't know in detail—and we don't need to know! What matters is that we know how to work with them in Haskell—e.g., how to add and multiply them.

Function Abstraction

A function declaration, like

```
takeLast n str = drop (length str - n) str
```

is also called a **function abstraction**.

Function Abstraction

A function declaration, like

```
takeLast n str = drop (length str - n) str
```

is also called a **function abstraction**.

We have abstracted from the concrete data (e.g., `"computer"` and `5`) and arrived at a more general—abstract—expression.

Function Abstraction

A function declaration, like

```
takeLast n str = drop (length str - n) str
```

is also called a **function abstraction**.

We have abstracted from the concrete data (e.g., `"computer"` and `5`) and arrived at a more general—abstract—expression.

Computations with concrete data have also become more abstract. Instead of `drop (length "computer"- 5)"computer"` we simply use

```
takeLast 5 "computer"
```

Thus, we only need to know the *name* of the function (i.e., `takeLast`), while the exact *algorithm* to compute the last few characters of a string has been abstracted away.

Function Abstraction in Real Life

Suppose you want to send a document to the CSN office in Sundsvall.

Function Abstraction in Real Life

Suppose you want to send a document to the CSN office in Sundsvall.

You could take the document, go to Sundsvall and deliver it in person.

Function Abstraction in Real Life

Suppose you want to send a document to the CSN office in Sundsvall.

You could take the document, go to Sundsvall and deliver it in person.

Or you could put your document into an envelope and mail it.

Function Abstraction in Real Life

Suppose you want to send a document to the CSN office in Sundsvall.

You could take the document, go to Sundsvall and deliver it in person.

Or you could put your document into an envelope and mail it.

Mailing is

- more abstract: you don't need to know the exact process for getting your letter from the mailbox to CSN.

Function Abstraction in Real Life

Suppose you want to send a document to the CSN office in Sundsvall.

You could take the document, go to Sundsvall and deliver it in person.

Or you could put your document into an envelope and mail it.

Mailing is

- more abstract: you don't need to know the exact process for getting your letter from the mailbox to CSN.
- more general: you can use the same method to send your document elsewhere.