



Functional Programming in Haskell

[The University of Glasgow](#)

[Get Unlimited learning](#)

[€23.99/month](#)

[TP](#)

- [Your learning](#)
- [Wishlist](#)
- [Profile](#)
- [Settings](#)
- [Account](#)
- [Sign out](#)

[Get Unlimited learning](#)

[€23.99/month](#)

- [To do](#)
- [Conversations](#)
- [Progress](#)



1.1

[Welcome to the Course](#)

[Video](#)

Haskell Basics: Expressions and Equations

1.2

[Basic Elements By Example](#)[Video](#)

1.3

[Introduction to Expressions and Equations](#)[Article](#)

1.4

[Do it Yourself: Expressions, Functions and Equations](#)[Exercise](#)

1.5

[Test Your Understanding](#)[Quiz](#)

1.6

[Summary](#)[Article](#)

Haskell Basics: Reduction, Functions and Lists

1.7

[More Basic Elements by Example](#)[Video](#)

6.5

You've completed 4 steps in Week 6

POSTULATES FOR THE FOUNDATION OF LOGIC. 355

$\lambda x[M]$ represents a function, whose value for a value L of the independent variable is equal to the result $S[M]$ of substituting L for x throughout M , whenever $S[M]$ turns out to have a meaning, and whose value is in any other case undefined.

The symbol H stands for a certain propositional function of two independent variables, such that $H(F, G)$ denotes, " $G(x)$ is a true proposition for all values of x for which $F(x)$ is a true proposition." It is necessary to distinguish between the proposition $H(F, G)$ and the proposition $x \cdot F(x) \supset G(x)$ (read, "For every x , $F(x)$ implies $G(x)$ "). The latter proposition justifies, for any value M of x , the inference $F(M) \supset G(M)$, and hence can be used only in the case that the functions F and G are defined for all values of their respective independent variables. The proposition $H(F, G)$ does not, on the other hand, justify this inference, although, when $F[M]$ is known to be true, it does justify the inference $G[M]$. And the proposition $H(F, G)$ is, therefore, suitable for use in the case that the ranges of definition of the functions F and G are limited.

[and Lists](#)

Excerpt from "Postulates for the foundation of Logic" by A. Church, 1932

[Do it Yourself: Functions and Lists](#)

© ISTEOR

[Exercise](#)

Introduction to the Lambda calculus

[Test Your Understanding](#)

[0 comments](#)

[Quiz](#)

Introduction to the Lambda Calculus

1.11

- The lambda calculus was developed in the 1930s by Alonzo Church (1903–1995), one of the leading developers of mathematical logic.

[Summary](#)

- The lambda calculus was an attempt to formalise functions as a means of computing.

[Article](#)

Significance to computability theory

[Finding Out More](#)

- A major (really *the* major) breakthrough in computability theory was the proof that the lambda calculus and the Turing machine have exactly the same computational power.

1.12

[Recommended Reading](#)

[Video](#)

- This led to *Church's thesis* — that the set of functions that are effectively computable are exactly the set computable by the Turing machine or the lambda calculus.

- The thesis was strengthened when several other mathematical computing systems (Post Correspondence Problem, and others) were also proved equivalent to lambda calculus.

[Spot the Difference](#)

- The point is that the set of effectively computable functions seems to be a fundamental reality, not just a quirk of how the {Turing machine, lambda calculus} was defined.

[Discussion](#)

1.14

Significance to programming languages

[End of Week 1](#)

[Video](#)

- The lambda calculus has turned out to capture two aspects of a function:
 - A mathematical object (set of ordered pairs from domain and range), and
 - An abstract black box machine that takes an input and produces an output.

[Week 2](#)

[Haskell Building Blocks](#)

- The lambda calculus is fundamental to denotational semantics, the mathematical theory of what computer programs mean.

[More Haskell](#)

- Functional programming languages were developed with the explicit goal of turning the lambda calculus into a practical programming language.

- The ghc Haskell compiler operates by (1) desugaring the source program, (2) transforming the program into a version of lambda calculus called *System F*, and (3) translating the System F to machine language using *graph reduction*.

2.1

[Welcome to week 2](#)

Abstract syntax of lambda calculus

[Video](#)

- We will work with the basic lambda calculus “enriched” with some constants and primitive functions (strictly speaking, that is not necessary).
- The language has constants, variables, applications, and functions.

[Do it Yourself: Boolean Values and Expressions](#)

[Exercise](#)

2.3

```

= const
| var
| exp exp
| \ var -> exp
  
```

[Zip that List](#)

Variables

[Video](#) Each occurrence of a variable in an expression is either *bound* or *free*

- In $\lambda x \rightarrow x + 1$, the occurrence of x in $x + 1$ is *bound* by the λx .
- 2.4 ◦ In $y * 3$, the occurrence of y is *free*. It must be defined somewhere else, perhaps as a global definition.

[Do it Yourself: Logical Thinking](#)

- In general, an occurrence of a variable is bound if there is some enclosing lambda expression that binds it; if there is no lambda binding, then the occurrence is free.

[Exercise](#)

2.5

We need to be careful: the first occurrence of a is free but the second occurrence is bound.

[Nothing but the Truth](#)

[Quiz](#)

```
a + (\ a -> 2^a) 3 --> a + 2^3
```

Being free or bound is a property of an *occurrence* of a variable, not of the variable itself.

[Input and Output](#)

Conversion rules

2.6

- Computing in the lambda calculus is performed using three *conversion rules*.
- The conversion rules allow you to replace an expression by another (“equal”) one.

[Video](#)

- Some conversions simplify an expression; these are called *reductions*.

2.7

Alpha conversion

[Do it Yourself: Input/Output](#)

[Exercise](#)

- Alpha conversion lets you change the name of a function parameter consistently.
- But you can’t change free variables with alpha conversion!
- 2.8 • The detailed definition of alpha conversion is a bit tricky, because you have to be careful to be consistent and avoid “name capture”. We won’t worry about the details right now.

[I/O and a First Encounter with Monads](#)

[Article](#)

```
(\x -> x+1) 3
```

```
(\y -> y+1) 3
```

[Installing GHC](#)

Beta conversion

- 2.9 • Beta conversion is the “workhorse” of lambda calculus: it defines how functions work.

[Installing Haskell for Yourself](#)

- To apply a lambda expression an argument, you take the body of the function, and replace each bound occurrence of the variable with the argument.

[Article](#)

```
(\x -> exp1) exp2
```

2.10

is evaluated as $exp1[exp2/x]$

[How to Run GHCi](#)

Example:

[Video](#)

```
(\x -> 2*x + g x) 42
```

2.11

is evaluated as $2 * 42 + g\ 42$

[Guessing Game](#)

Eta conversion

[Article](#)

- Eta conversion says that a function is equivalent to a lambda expression that

2.12 takes an argument and applies the function to the argument.

[What do you know about Haskell?](#)

is equivalent to f

Example (recall that $(*3)$ is a function that multiplies its argument by 3)

[\(\x -> \(*3\) x\) 2](#)

is equivalent to $(*3)$

Try applying both of these to 50:

Week 3
`(\x -> (*3) x) 50`
 Data Structures and Types
 is the same as $(*3) 50$

Functions on Lists

Removing a common trailing argument

There is a common usage of Eta conversion. Suppose we have a definition like this:

[f x, y = g y y](#)

This can be rewritten as follows:

$f = \lambda x \rightarrow (\lambda y \rightarrow g y)$
 $f' = \lambda x \rightarrow g = f x = g$

[Recursive Functions on Lists](#)

Thus the following two definitions are equivalent:

[f x y = g y](#)
 $f x = g$

3.3

In effect, since the last argument on both sides of the equation is the same (y), it

[Can be "factored out" Folds versus Imperative Loops](#)

[University of Glasgow](#)

Share this article:

[Do it Yourself: Lists and Recursion](#)

[Exercise](#)

3.5

[Do it Yourself: Function Composition](#)

[Exercise](#)

Mark as complete

[Next](#)

Mark as complete

[What Have We Learned About Lists?](#)

Comments

How:

All comments ▾

[Book Generator](#)

Sort by:

Newest ▾

Filter

3.9

[Summary](#)

- [Help Centre](#)
- [Child safety](#)

[Article](#)

- [Privacy](#)
- [T&Cs](#)

Custom Data Types

[Contact FutureLearn for Support](#)

Our website is updated regularly so this content may now be out of date, please go to <https://www.futurelearn.com> for the most up to date information.

3.9

[Define Your Own Data Types](#)

[Video](#)

3.10

[Grow a Tree](#)

[Article](#)

3.11

[Type Classes](#)

[Video](#)

Haskell History

3.12

[Interview with Simon Peyton Jones](#)

[Video](#)

3.13

[Brief History of Haskell](#)

[Article](#)

3.14

[Course Feedback](#)

[Article](#)

3.15

[End of Week 3](#)

[Video](#)

Week 4

When Programs Get Bigger

Program Structure

4.1

[Welcome to Week 4](#)

[Video](#)

4.2

[Keep Your Programs Tidy](#)

[Article](#)

4.3

[Guards, Guards!](#)

[Article](#)

4.4

[Dealing with Uncertainty](#)

[Video](#)

4.5

[Idiomatic Haskell](#)

[Quiz](#)

Parsing Text

4.6

[Parsing Text Using Higher-Order Functions](#)

[Article](#)

4.7

[Parsing using Parsec: a practical example](#)

[Video](#)

4.8

[Parser Puzzles](#)

[Quiz](#)

4.9

[Summary](#)

[Article](#)

Am I Right?

4.10

[Check my Program is Correct](#)

[Video](#)

4.11

[Using QuickCheck](#)

[Article](#)

4.12

[Talk with a Haskell Teacher](#)

[Video](#)

Week 5

Hardcore Haskell

Laziness and Infinite Data structures

5.1

[Welcome to Week 5](#)

[Video](#)

5.2

[Lazy is Good](#)

[Video](#)

5.3

[Infinite Data Structures](#)

[Article](#)

5.4

[To Infinity \(but not beyond\)](#)

[Quiz](#)

More about Types

5.5

[Type Horror Stories](#)

[Discussion](#)

5.6

[Types, lambda functions and type classes](#)

[Article](#)

5.7

[Curry is on the menu](#)

[Video](#)

5.8

[Type Inference by Example](#)

[Video](#)

5.9

[You are the type checker](#)

[Quiz](#)

5.10

[Summary](#)

[Article](#)

Haskell in the Real World

5.11

[Haskell at Facebook](#)

[Video](#)

5.12

[Haskell in the Wild](#)[Article](#)

5.13

[Course Feedback](#)[Article](#)

Week 6

Think like a Functional Programmer

Type Classes

6.1

[Welcome to Week 6](#)[Video](#)

6.2

[Types with Class](#)[Video](#)

6.3

[Type Classes in more Detail](#)[Article](#)

6.4

[Summary](#)[Article](#)

Geek Greek

6.5

[Introduction to the Lambda calculus](#)[Article](#)

6.6

[There are Only Functions! \(Optional\)](#)[Video](#)

6.7

[We Love Lambda!](#)

[Quiz](#)

6.8

[Summary](#)

[Article](#)

The M-word

6.9

[We Already Know About Monads](#)

[Video](#)

6.10

[Introduction to monad theory](#)

[Article](#)

6.11

[Example: the Maybe monad](#)

[Article](#)

6.12

[Monad metaphors](#)

[Discussion](#)

6.13

[Summary](#)

[Article](#)

So long and thanks for all the fun(ctions)!

6.14

[Functional Programming in Other Languages](#)

[Video](#)

6.15

[Will You Use Haskell in the Future?](#)

[Discussion](#)

6.16

[The End of the Affair](#)

[Video](#)