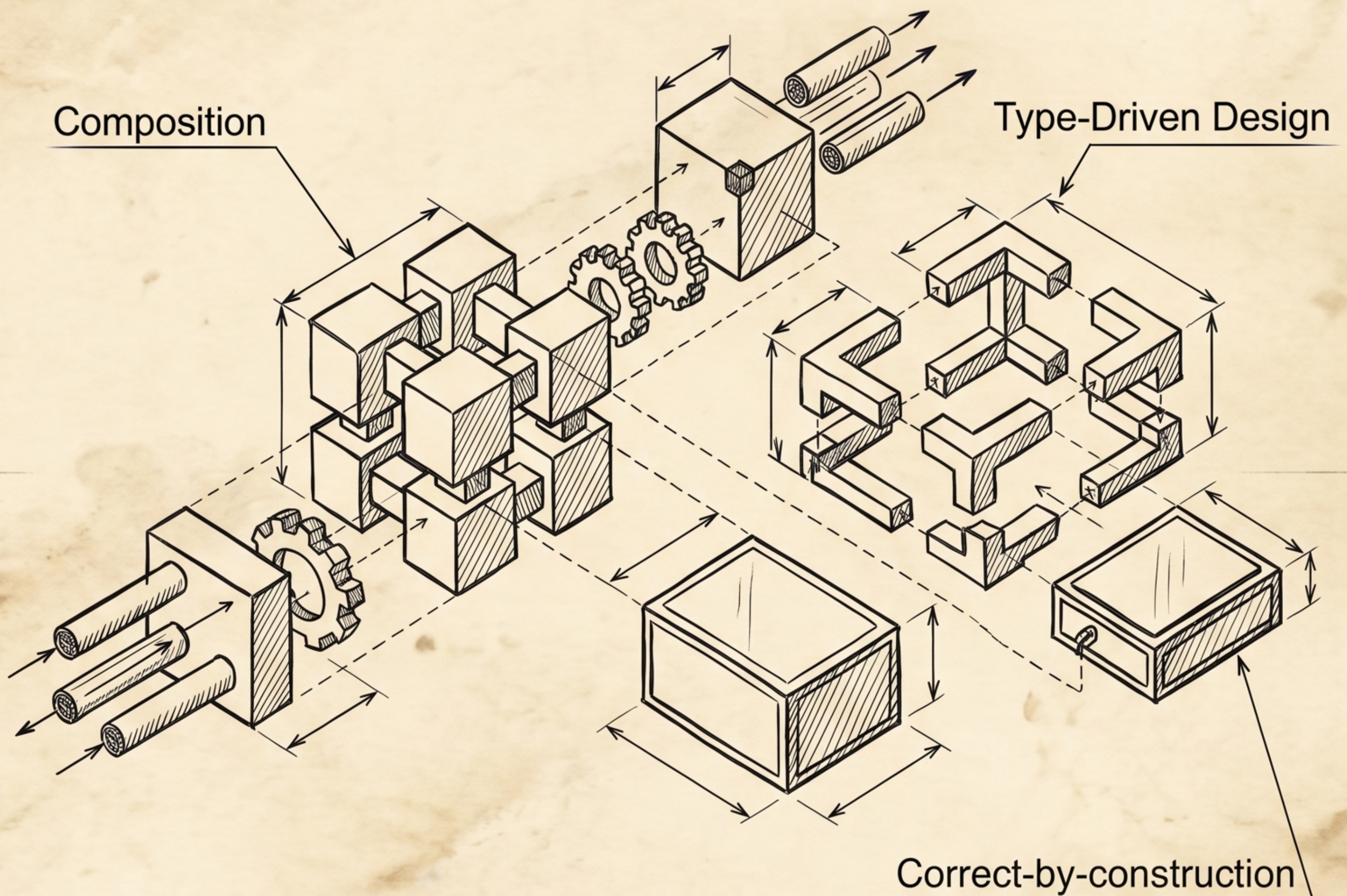
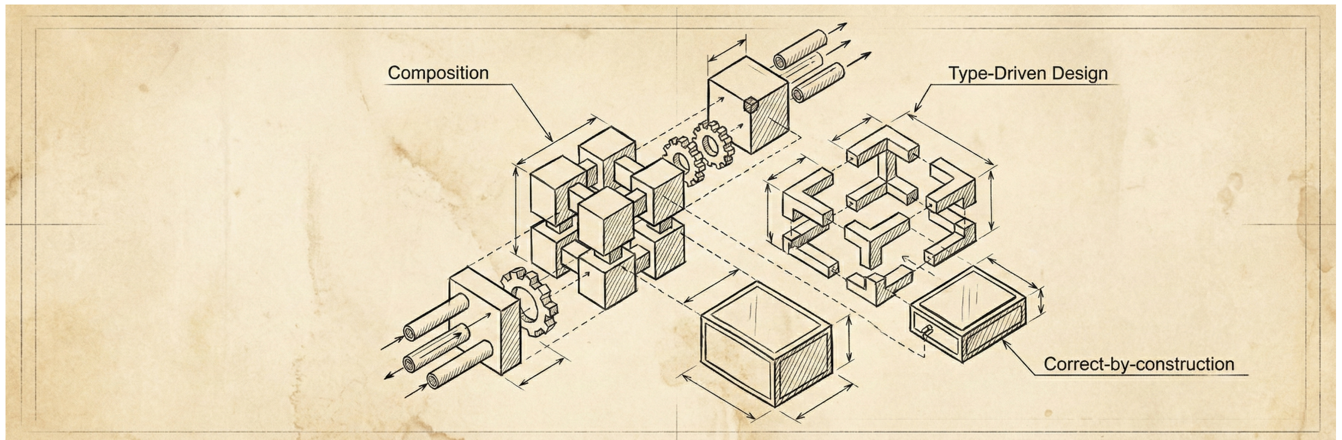
The Haskell Blueprint

A Direct Guide to Writing Maintainable, Real-World Software



Gautier DI FOLCO



The Haskell Blueprint

A Direct Guide to Writing Maintainable, Real-World Software

Gautier DI FOLCO

Version v1.0.0, 2026-06-14

Table of Contents

Copyright . . . 1
Dedication . . . 2
Book Description . . . 3
Introduction . . . 4
Getting started . . . 5
1. Introduction to Haskell and GHC . . . 6
1.1. Introduction . . . 6
1.2. Structure . . . 6
1.3. Objectives . . . 6
1.4. Understanding Functional Programming . . . 6
1.5. Origins and Evolution of Haskell . . . 9
1.6. Hands-on GHCi . . . 9
1.7. Hands-on: displaying time . . . 13
1.8. Hands-on: formatting a file into a tree . . . 17
1.9. Setting Up a Haskell Development Environment . . . 23
1.10. Conclusion . . . 28
1.11. Questions . . . 28
1.12. Exercises . . . 28
1.13. Answers to questions . . . 28
1.14. Answers to exercises . . . 28
2. Mastering Functions and Types . . . 30
2.1. Introduction . . . 30
2.2. Structure . . . 30
2.3. Objectives . . . 30
2.4. Anatomy of functions . . . 30
2.5. Immutability and referential transparency . . . 37
2.6. Creating functions . . . 38
2.7. Types of errors . . . 41
2.8. Navigating types . . . 43
2.9. Project: Playing with diagrams . . . 49
2.10. Conclusion . . . 56
2.11. Questions . . . 56
2.12. Exercises . . . 56
2.13. Answers to questions . . . 57
2.14. Answers to exercises . . . 58
3. Higher-Order Functions and Abstraction . . . 59
3.1. Introduction . . . 59
3.2. Structure . . . 59

3.3. Objectives . . . 59
3.4. Conditionals . . . 59
3.5. Recursion . . . 64
3.6. Higher-order functions . . . 66
3.7. Conclusion . . . 80
3.8. Questions . . . 81
3.9. Exercises . . . 81
3.10. Answers to questions . . . 83
3.11. Answers to exercises . . . 83
4. Data-types and Type Classes . . . 85
4.1. Introduction . . . 85
4.2. Structure . . . 85
4.3. Objectives . . . 85
4.4. Types aliases . . . 85
4.5. Newtypes . . . 88
4.6. Sum types . . . 92
4.7. Product types . . . 95
4.8. Space algebra . . . 98
4.9. Pattern-matching . . . 103
4.10. Strictness . . . 106
4.11. Recursive data types and folds . . . 107
4.12. Type class . . . 111
4.13. Constraints, laws, and properties . . . 114
4.14. Type class derivations . . . 116
4.15. Hands-on with aeson and yaml . . . 118
4.16. Conclusion . . . 125
4.17. Questions . . . 125
4.18. Exercises . . . 125
4.19. Answers to questions . . . 126
4.20. Answers to exercises . . . 126
Haskell into the Wild . . . 129
5. Essential type classes: Functor, Applicative, Monad . . . 130
5.1. Introduction . . . 130
5.2. Structure . . . 130
5.3. Objectives . . . 130
5.4. Study plan . . . 130
5.5. Plain values . . . 131
5.6. Containers . . . 139
5.7. Computations . . . 144
5.8. Conclusion . . . 158
5.9. Questions . . . 159

5.10. Exercises 159
5.11. Answers to questions 159
5.12. Answers to exercises 159
6. Running code in IO 165
6.1. Introduction 165
6.2. Structure 165
6.3. Objectives 165
6.4. Introducing IO 165
6.5. Purity vs controlled impurity vs imperative IO 174
6.6. IO and laziness 177
6.7. Exceptions 181
6.8. Sharing states 186
6.9. Playing with an HTTP server 189
6.10. Conclusion 197
6.11. Questions 197
6.12. Exercises 198
6.13. Answers to questions 198
6.14. Answers to exercises 198
7. Project Architecture and Cabal 201
7.1. Introduction 201
7.2. Structure 201
7.3. Objectives 201
7.4. Structuring Haskell applications 201
7.5. Modules and encapsulation 203
7.6. Project organization 210
7.7. Managing dependencies 215
7.8. Documentation 233
7.9. Libraries and large-scale projects 235
7.10. Conclusion 236
7.11. Questions 236
7.12. Exercises 236
7.13. Answers to questions 237
7.14. Answers to exercises 237
Advanced Design 238
8. Designing Type-Safe eDSLs 239
8.1. Introduction 239
8.2. Structure 239
8.3. Objectives 239
8.4. Introduction to embedded domain-specific languages (eDSLs) 239
8.5. Smart constructors vs correct-by-construction 240
8.6. Refactoring techniques and design patterns 246

8.7. SQL queries with Esqueleto 257
8.8. HTML generation with lucid2 265
8.9. Conclusion 270
8.10. Questions 270
8.11. Exercises 270
8.12. Answers to questions 271
8.13. Answers to exercises 271
9. Advanced data structures 275
9.1. Introduction 275
9.2. Structure 275
9.3. Objectives 275
9.4. Immutable and persistent data structures 275
9.5. Working with Maps, and Sets 277
9.6. Finger Trees and Performance Considerations 282
9.7. Efficient vector operations and mutable state 285
9.8. Implementing a Trie 289
9.9. Conclusion 292
9.10. Questions 292
9.11. Exercises 292
9.12. Answers to questions 293
9.13. Answers to exercises 293
Engineering Practices 297
10. Testing Haskell Applications 298
10.1. Introduction 298
10.2. Structure 298
10.3. Objectives 298
10.4. Testing 298
10.5. Conclusion 314
10.6. Questions 314
10.7. Exercises 315
10.8. Answers to questions 315
10.9. Answers to exercises 316
11. Packaging, Deployment, and Continuous Integration 318
11.1. Introduction 318
11.2. Structure 318
11.3. Objectives 318
11.4. Packaging and deployment 318
11.5. Continuous Integration (CI/CD) for Haskell applications 330
11.6. Conclusion 334
11.7. Questions 334
11.8. Exercises 334

11.9. Answers to questions 335
11.10. Answers to exercises 335
12. Final project: a blog engine 340
12.1. Introduction 340
12.2. Structure 340
12.3. Objectives 340
12.4. Project overview and structure 340
12.5. Building core types and logic 345
12.6. Adding CI/CD 359
12.7. Introducing repositories 360
12.8. Creating web handlers 367
12.9. Making an in-memory browsable website 378
12.10. Introducing persistence with sqlite-simple 382
12.11. Conclusion 391
12.12. Questions 391
12.13. Exercises 392
12.14. Answers to questions 392
12.15. Answers to exercises 392
Conclusion 393
Appendix A: Afterword: Where to Go from Here 394
A.1. Books 394
A.2. Community 394
A.3. Open-Source Projects 394
A.4. Topics to Explore Next 395
Appendix B: Reading GHC Error Messages 396
B.1. How to Read a GHC Error Message 396
B.2. Strategies for Reading Errors 396
B.3. Error Code Catalog 397
B.4. Warning Flags and Error Severity 407
B.5. The Haskell Error Index 408
Appendix C: Origins of Functional Programming 410
C.1. History of Functional Programming 410
C.2. History of Haskell 411
Appendix D: Glossary 413
D.1. Core Concepts 413
D.2. Functional Programming Principles 413
D.3. Type System 415
D.4. Abstractions and Effects 416
D.5. Architecture and Patterns 416
D.6. Tooling 417
Colophon 419

Bibliography . . . 421
Index . . . 423

Copyright

© 2026 Gautier DI FOLCO. All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

The source code for the examples and the companion project is available at https://codeberg.org/gdifolco/thb.

For any questions, error reports, or feedback, the author can be reached at thb.book@difolco.dev.

Dedication

I dedicate this book to my parents, who have supported me all these years.

And to my cat, who accompanied me over more than two decades.

Book Description

Haskell is an academically studied and peer-reviewed programming language. It is also production-grade, robust, and industry-oriented. Thus, it is the perfect foundation for building a reasoning framework for any domain in the problem or solution space.

This book aims to find the right balance between introducing the principles underlying functional programming and software engineering and providing practical examples of designing famous Haskell libraries. We will see how everything works, how to read errors, and what actions to take to solve them. From the ground up, we will explore Haskell and its ecosystem, and the main concepts of types, functions, and data types. We will then focus on the most common abstractions and how to build upon them. Finally, we will see how all of these come together while designing small programs using famous Haskell libraries.

The main idea behind this book is to give design tools usable in any programming language and paradigms to gain clarity during systems building.

Introduction

In 2012, I had been coding for a decade, and I had already experimented with many programming languages (PHP, C, Java, Prolog, Erlang, JavaScript, etc.). I had read "Learn You a Haskell for Great Good!" in a week, and I thought I was done.

I could not have been more wrong. Two years later, I took an advanced training course, and I had learned that I had not understood it. This was the beginning of a ten-year learning journey.

In the professional world, Haskell has the bad reputation of being complex, weird, and a toy language with no real-world usage. It is true that Haskell is the culmination of nearly a century of research in programming language design, type theory, and formal methods. Yet, they are only tools used to solve industry grade problems. Many systems are based on Haskell running in production at financial institutions, technology companies, and open-source projects worldwide.

That being said, discovering Haskell after years of writing software in mainstream languages can be disorienting. No mutable state, no null references, no uncontrolled side effects, at first glance, it seems impossible to build programs without them. In their place, Haskell offers a different contract: express what you mean, and let the compiler verify that it makes sense.

That contract is what this book is about.

The existing Haskell literature serves different audiences well. "Learn You a Haskell for Great Good!" provides a gentle, humorous introduction. "Real World Haskell" demonstrates practical applications. "Haskell Programming from First Principles" builds understanding from the ground up with remarkable thoroughness. Each of these books has its place on a programmer’s shelf.

This book occupies a different niche. The goal is to take the reader from understanding the basics to designing and building a real application as directly as possible. Every chapter introduces concepts with a clear purpose: to solve problems that arise when building software that others depend on.

The structure reflects this goal. We begin with functions, types, and type classes — the vocabulary of Haskell. We then move into I/O, project architecture, and embedded domain-specific languages. The final chapters cover testing, deployment, and a complete blog engine that ties everything together.

We do not assume prior experience with functional programming. We do assume familiarity with writing programs and a willingness to engage with a compiler that is more demanding, and more helpful, than most.

The ideas in this book are not specific to Haskell. Thinking in types, designing with composition, and isolating side effects are principles that improve software in any language. Haskell simply enforces them, which makes it an ideal vehicle for learning.

Let us get started.

Chapter 2. Mastering Functions and Types

2.1. Introduction

Functions are the cornerstones of functional programming; they shape our systems more than data types. This chapter exposes how functions are defined, explicitly or implicitly, and the different types of functions (higher-order, recursive, partial), and enabling us to classify them, which is the basis of using and composing them.

This chapter provides insights into designing functions such that they are easy to write, easy to reuse, and efficiently composable. At the end of this chapter, the reader will be able to read and interpret them from their type only.

More importantly, it equips us with the knowledge to read, understand, and debug type errors, a crucial skill for collaborating with GHC.

2.2. Structure

- Anatomy of functions
- Immutability and referential transparency
- Creating functions
- Types of errors
- Navigating types
- Project: Playing with diagrams

2.3. Objectives

Discover functions, how to define them, and how to read them. Learn to interpret type signatures and debug type errors using GHC's feedback. Understand immutability, referential transparency, and how to create functions through lambdas, currying, closures, and composition. Apply these concepts in a practical diagrams project.

2.4. Anatomy of functions

In the previous chapter, we have encountered a few functions, we even have defined a few of them.

To follow along, open a new file Chap02.hs with this content:

Module declaration and top-level binding

```
module Chap02 where

answer :: Int
answer = 42
```

This will enable us to experiment and write multi-line functions.

The reader will then be able to load, reload, and interact with the file:

Loading and reloading a module in GHCi

```
> :l Chap02.hs ①
[1 of 1] Compiling Chap02           ( Chap02.hs, interpreted )
Ok, one module loaded.
> :r ②
Ok, one module loaded.
> answer
42
it :: Int
```

① load a file

② reload everything (after a change)

2.4.1. Expressions

Everything defined at file-level (i.e. everything starting at the beginning of a line, not preceded by spaces) is called top-level binding.

It has the following syntax:

Top-level binding syntax

```
<name> :: <Type> ①
<name> = <expr> ②
```

① Type signature

② Binding definition

We could omit the type definition, but, not only is it essential for reasoning about the code, providing some documentation, it would also make the whole program brittle. Haskell has an advanced static type system, with a very clever type inference, the strategy is to come up with a type as generic as possible, and add constraints along the way, which does not produce readable type errors, moreover, it may move type errors away from the expression definition.

NOTE Most functional programming languages are statically typed (the type is known at compile-time). However, they do not all emphasize putting types first. Some dynamically typed languages also have type annotations (types being enforced at runtime), and gradual typing (types can be specified arbitrarily, not systematically). Nonetheless, only a handful of them are weakly typed (types change implicitly).

2.4.2. Functions

Functions are expressions of the type function (represented by (→)):

Defining a function with type signature

```
twice :: Int -> Int
twice x = x * 2

-- > twice 5
-- 10
-- it :: Int
```

NOTE We may sometimes encounter parameters named "`_", or prefixed with "_", it is used to silence compiler warnings when a binding is declared but not used. "`_"-prefixed bindings can be referenced, but it may be an issue since we cannot track anymore if some bindings are not used legitimately.

While **answer** (defined earlier), is of type Int, with all the operations associated (e.g. addition, display, etc.), the main operation we can perform with functions is application, which will give a final type here.

At the type level, function application consumes the left side of the function's type going from **Int → Int** to Int. It is a mechanical substitution which is also performed at the expression level, so the following are equivalent:

Beta reduction step by step

```
twice 5
5 * 2
10
```

This process is called *beta reduction*.

Haskell, and lambda calculus in general, only have one-parameter functions, which implies that functions with multiple parameters are in fact, functions producing functions, at the type level, these two definitions are equivalent:

Equivalence of curried and parenthesized function types

```
rectangleArea :: Int -> Int -> Int
rectangleArea :: Int -> (Int -> Int)
```

2.4.3. Long functions

While we should aim to keep functions short, top-level bindings have the major drawback of being accessible by everyone. It might look like a considerable benefit, but, not only does it create a lot of noise, but it might lead to unwanted behaviors. For example, consider we have a function fetching the nth month number of days:

Multi-parameter function with helper definitions

```
monthDays :: Int -> Int -> Int
monthDays year nthMonth =
  if isLeapYear year
    then monthDaysOf leapYearMonthsDays nthMonth
    else monthDaysOf regularYearMonthsDays nthMonth

isLeapYear :: Int -> Bool
isLeapYear year =
  (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)

monthDaysOf :: [Int] -> Int -> Int
monthDaysOf monthsDays nthMonth = monthsDays !! (nthMonth - 1)

regularYearMonthsDays :: [Int]
regularYearMonthsDays = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

leapYearMonthsDays :: [Int]
leapYearMonthsDays = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

It will work as expected:

Using monthDays in GHCi

```
> monthDays 2024 2
29
it :: Int
(0.01 secs, 74,288 bytes)
> monthDays 2025 2
28
it :: Int
(0.00 secs, 69,832 bytes)
```

However, if we aim to use **monthDaysOf** directly, with an empty list, it will explode:

Runtime exception from an out-of-bounds index

```
> monthDaysOf [] 2
*** Exception: Prelude.!!: index too large
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1366:14 in base:GHC.List
  tooLarge, called at libraries/base/GHC/List.hs:1376:50 in base:GHC.List
  !!, called at Chap02.hs:23:46 in main:Chap02
```

This is due to the assumption made by monthDaysOf that length monthsDays ⇐ nthMonth. We could enforce it, but it would add a lot of friction, while we could protect them.

There are two ways to encapsulate or group them, the first one is using where

Syntax template for a where clause

```
<name> :: <Type>
<name> = <expr>
  where <sub-bindings> ①
```

① can be anything a top-level binding can be (having even a `where`)

NOTE	Like Python, Haskell uses indentation to delimit expressions. As a general guideline, every expression continued on another should be indented more than its start.

Let us consider the following examples:

Valid and invalid indentation patterns

```
valid :: Type
valid = definition

valid :: Type
valid =
  definition

invalid :: Type
invalid =
definition

valid :: Type
valid = definition
  where sub = subdefinition

valid :: Type
valid =
  definition
  where sub = subdefinition

invalid :: Type
invalid =
  definition
where sub = subdefinition
```

We can regroup our functions like so:

Encapsulating helpers with where

```
monthDays' :: Int -> Int -> Int
monthDays' year nthMonth =
    if isLeapYear' year
      then monthDaysOf' leapYearMonthsDays' nthMonth
      else monthDaysOf' regularYearMonthsDays' nthMonth
  where
    isLeapYear' :: Int -> Bool
    isLeapYear' year' =
      (mod year' 4 == 0 && mod year' 100 /= 0) || (mod year' 400 == 0)

    monthDaysOf' :: [Int] -> Int -> Int
    monthDaysOf' monthsDays nthMonth' = monthsDays !! (nthMonth' - 1)

    regularYearMonthsDays' :: [Int]
    regularYearMonthsDays' = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

    leapYearMonthsDays' :: [Int]
    leapYearMonthsDays' = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

NOTE All the bindings and parameters are suffixed by an apostrophe (') to avoid name-shadowing (the declaration of a binding accessible in the scope). There is no special behavior associated with apostrophes, they can be part of identifiers.

Unlike top-level bindings, type definitions can be omitted, as the top-level binding already constrains the types.

Moreover, we can use the top-level parameters, given that, we can simplify our function:

Simplified monthDays using closure over top-level parameters

```
monthDays'' :: Int -> Int -> Int
monthDays'' year nthMonth =
    monthsDays !! (nthMonth - 1)
  where
    isLeapYear' =
      (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)

    monthsDays =
      if isLeapYear'
      then leapYearMonthsDays'
      else regularYearMonthsDays'

    regularYearMonthsDays' = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    leapYearMonthsDays' = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

There is also another syntax to define local bindings within a function:

Local bindings with let expression syntax

```
let <sub-expression>
in <expression>
```

Like the where syntax, they can be nested:

Nested let expressions

```
seven :: Int
seven =
  let four =
        let two = 2
          in two + two
      three = 3
    in four + three
```

We can rewrite our **monthDays** like so:

Rewriting monthDays with let bindings

```
monthDays''' :: Int -> Int -> Int
monthDays''' year nthMonth =
  let
    isLeapYear' =
      (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)

    monthsDays =
      if isLeapYear'
      then leapYearMonthsDays'
      else regularYearMonthsDays'

    regularYearMonthsDays' = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    leapYearMonthsDays' = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    in monthsDays !! (nthMonth - 1)
```

There is no difference between where and let in pure functions/expressions. The only impact is the way we read it, with where clauses, the most important part is put first, and the details later, while it is the other way around with let.

2.5. Immutability and referential transparency

It is important to emphasize the concept of immutability.

Nearly everything in Haskell is immutable, which means that, if we want to double an Int (as in twice), the argument is not changed, but a new one is produced. This is why *beta reduction* works. Also, having immutability enables *referential transparency*, the property of a

function to be replaceable by its output, there is no difference between twice 5 and 10.

> **NOTE** If coming from low-level programming languages or performance computing, it may seem to be wasteful or resource-intensive. In practice, not only GHC optimizes aggressively, but it also comes with a runtime system intelligent enough to mitigate these costs.

A practical consequence of *referential transparency* is that a pure expression can be evaluated once and shared wherever its result is needed. This lets GHC apply *common subexpression elimination* rather than recomputing the same value. Let us take the Fibonacci sequence:

Recursive function definition

```
fibonacci :: Integer -> Integer
fibonacci n =
  if n < 2
    then 1
    else fibonacci (n - 1) + fibonacci (n - 2) ①
```

① it is a recursive function, we will see later what it involves

Because the expression is pure, GHC can share its evaluation. When we bind the result to a name, it is computed only once, even if the name is used several times:

Sharing a pure result through a let-binding

```
> let x = fibonacci 20 in (x, x)
(10946,10946)
```

Both elements of the pair refer to the same evaluation of `fibonacci 20`.

> **NOTE** This sharing happens because the result is bound to a name (`x`). GHC does *not* memoize function calls by their arguments: evaluating `fibonacci 20` as two separate expressions recomputes it each time. Referential transparency is what makes sharing *safe*; a named binding is what makes it happen.

2.6. Creating functions

2.6.1. Lambda functions

Haskell is, like other functional programming languages, based on functions.

Unlike the ones we have seen earlier, there are many ways to define them.

The first approach we have seen involves explicit bindings:

Explicit binding with parameters

```
add :: Int -> Int -> Int
add x y = x + y
```

This is syntactic sugar for the following expression:

Lambda function desugaring

```
add :: Int -> Int -> Int
add = \x -> \y -> x + y
```

We call it *lambda functions* or *lambda* (the backslash / represents a lambda λ, the Greek letter).

We can observe the similarity between the type level and the expression level.

Alternatively, we can group parameters in the lambda abstraction:

Grouped lambda parameters

```
add :: Int -> Int -> Int
add = \x y -> x + y
```

Beta reduction becomes clearer when demonstrated:

Beta reduction of add applied to 2 and 3

```
add 2 3
(\x -> \y -> x + y) 2 3
(\y -> 2 + y) 3
2 + 3
5
```

2.6.2. Currying

If we look closely at the previous *beta reduction*, we should have noticed that, functions taking multiple parameters, are functions, which produce functions (which produce functions, etc.) which produce plain expressions. This property is known as currying.

It allows us to create functions incrementally.

For instance, if we want to increment each element of a list, we can write:

Partial application through currying

```
> map (add 1) [1 .. 5]
[2,3,4,5,6]
it :: [Int]
```

2.6.3. Closure

Sometimes, functions might reference (or capture) a value outside the lambda scope (it is the case when we have multiple parameters/we nest them).

Let us define a new one:

Closure capturing a free variable

```
addEach :: Int -> [Int] -> [Int]
addEach n elements = map (\x -> n + x) elements
```

The lambda captures **n**, from the top-level binding, it is called a free variable.

A free variable is a binding used in a *lambda function*, not declared in its parameters.

This can be used like so:

Using addEach in GHCi

```
> addEach 1 [1 .. 5]
[2,3,4,5,6]
it :: [Int]
```

2.6.4. Composition

Finally, in functional programming, we focus on creating small functions and composing them; however, this approach can sometimes involve some necessary setup code.

For example, suppose, we want to add 1 and double each element, we could run map twice:

Nested map calls (traverses the list twice)

```
> map (mul 2) (map (add 1) [1 .. 5])
[4,6,8,10,12]
```

But this would be inefficient because it traverses the list twice.

However, if we want to create a function performing both operations, we have to introduce a lambda parameter:

Combining operations with an explicit lambda

```
> map (\x -> mul 2 (add 1 x)) [1 .. 5]
[4,6,8,10,12]
```

Haskell comes with the composition operator (.), which removes the last lambda parameter:

Function composition with the dot operator

```
> map (mul 2 . add 1) [1 .. 5]
[4,6,8,10,12]
```

We call this *point-free functions* because the "subject" (the *point*, the last parameter), is not visible in the definition.

We can chain operations, they will be applied from right to left, as shown below:

Chaining multiple function compositions

```
> map (mul 3 . add 5 . mul 2 . add 1) [1 .. 5]
[27,33,39,45,51]
```

2.7. Types of errors

In programming, we often start by handling the nominal case (i.e., when everything goes right), then we enumerate edge cases.

For a compiler, the nominal case is when it does not compile.

A significant aspect of working in Haskell involves reading and fixing compiling errors.

Haskell errors can be less straightforward to interpret than those in many other programming languages. If we compare them to mainstream programming languages such as Java, they often contain a wealth of detailed information. Haskell has a Hindley–Milner-based type system, we will not go in-depth as it involves solid type theory knowledge, but, along the way, we will give examples of errors we will encounter.

Let us begin with a simple example: declaring a binding with a type, incompatible with its definition:

Type error: mismatched type and expression

```
anInt :: Int
anInt = True
```

Which gives the following errors:

GHC error: type mismatch between Int and Bool

```
Chap02.hs:98:9: error: [GHC-83865] ① ②
    • Couldn't match expected type ‘Int’ with actual type ‘Bool’ ③ ④
    • In the expression: True
      In an equation for ‘anInt’: anInt = True
   |
98 | anInt = True
   |         ^^^^
Failed, no modules loaded.
```

① expression location

② since GHC 9.6.1, a documentation effort has been done with The Haskell Error Index[1] referencing errors and warnings including examples and solving tactics

③ the expected expression type

④ the actual inferred expression type

This error is straightforward because the type and the expression are co-located.

Many errors arise from applying an incompatible expression, as in:

Type error: applying Bool where Int is expected

```
anInt' :: Int

anInt' = add 5 True
```

Which yields:

GHC error: mismatched argument type in function application

```
Chap02.hs:101:16: error: [GHC-83865]
    • Couldn't match expected type ‘Int’ with actual type ‘Bool’
    • In the second argument of ‘add’, namely ‘True’
      In the expression: add 5 True
      In an equation for ‘anInt'’: anInt' = add 5 True
    |
101 | anInt' = add 5 True
    |                ^^^^
```

The situation becomes more complex when an expression has no type annotation, the compilation error is moved to the place where it is used:

Type error propagated from unannotated binding

```
anInt'' :: Int
anInt'' = 5 + right
  where right = True
```

The error has the similar structure, but the original expression location is

GHC error: type mismatch propagated from a where clause

```
Chap02.hs:107:15: error: [GHC-83865]
    • Couldn't match expected type ‘Int’ with actual type ‘Bool’
    • In the second argument of ‘(+)’, namely ‘right’
      In the expression: 5 + right
      In an equation for ‘anInt''’:
          anInt''
            = 5 + right
            where
                right = True
    |
107 | anInt'' = 5 + right
    |               ^^^^^
```

It is, usually, acceptable when it concerns function bindings, but it is not manageable at code-base level.

2.8. Navigating types

Developing in Haskell is an ongoing conversation between the programmer and the compiler.

The idea is similar to Test-Driven Development[2]:

- Write the type with a mock expression
- Write the minimal expression to make it compile[3]
- Refactor

It is often referred to as `Type-Driven Development (TyDD)` or `Structure-Based Development (SBD)`.

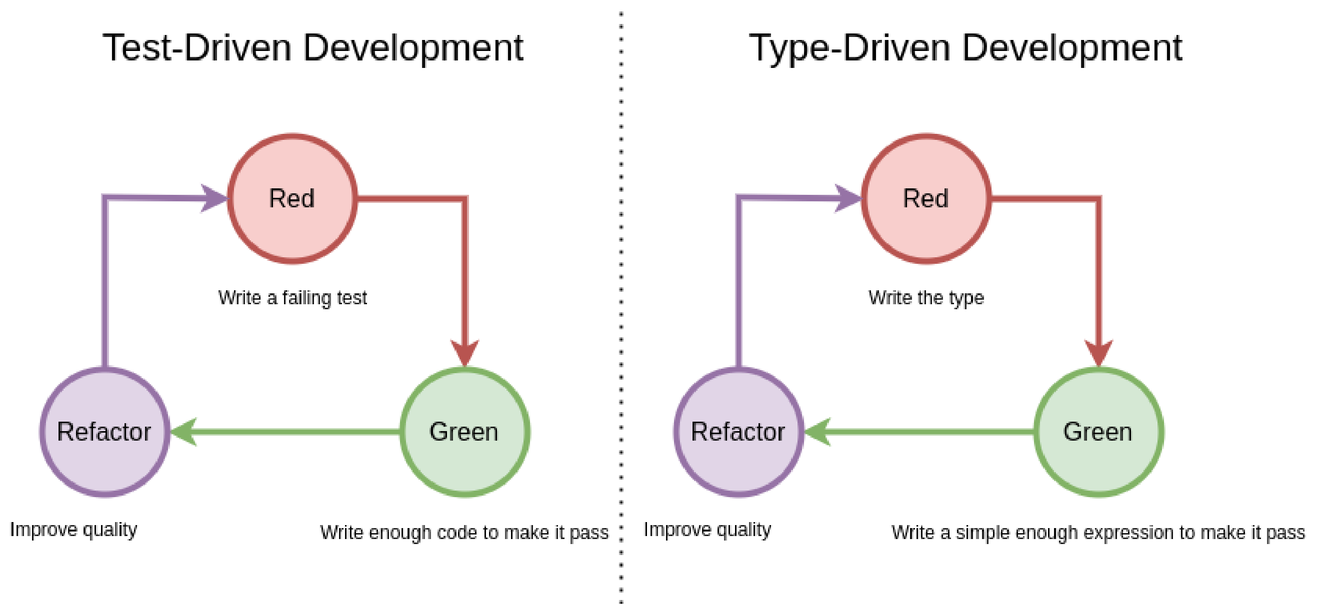


Figure 5. Test-Driven Development vs Type-Driven Development

Fortunately, Haskell comes with two tools.

The first one is `hole`, whenever a `_`, or a non-existing binding starting with `_` [4], the compiler will help us figure-out what is the expected type, and which bindings are available. It is called Hole-Driven Development.

Consider we wanted to use this in `monthDay`:

Using a typed hole to discover the expected type

```
monthDays'' :: Int -> Int -> Int
monthDays'' year nthMonth =
    monthsDays !! (nthMonth - 1)
  where
    isLeapYear' =
      (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)

    monthsDays =
      if isLeapYear'
      then _
      else regularYearMonthsDays'

    regularYearMonthsDays' = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    leapYearMonthsDays' = [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

Which gives a very detailed compilation error:

GHC error: typed hole report with relevant bindings

```
src/Chap02.hs:62:12: error: [GHC-88464] ①
    • Found hole: _ :: [Int] ②
```

```
    • In the expression: ③
        if isLeapYear' then _ else regularYearMonthsDays'
      In an equation for ‘monthsDays’:
          monthsDays = if isLeapYear' then _ else regularYearMonthsDays'
      In an equation for ‘monthDays''’:
          monthDays'' year nthMonth
            = monthsDays !! (nthMonth - 1)
            where
                isLeapYear'
                  = (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)
                monthsDays = if isLeapYear' then _ else regularYearMonthsDays'
                regularYearMonthsDays' = [31, ....]
                leapYearMonthsDays' = [31, ....]
    • Relevant bindings include ④
        monthsDays :: [Int] (bound at src/Chap02.hs:60:5)
        isLeapYear' :: Bool (bound at src/Chap02.hs:57:5)
        regularYearMonthsDays' :: [Int] (bound at src/Chap02.hs:65:5)
        leapYearMonthsDays' :: [Integer] (bound at src/Chap02.hs:66:5)
        nthMonth :: Int (bound at src/Chap02.hs:54:18)
        year :: Int (bound at src/Chap02.hs:54:13)
        (Some bindings suppressed; use -fmax-relevant-binds=N or -fno-max-relevant
-binds)
      Valid hole fits include ⑤
        regularYearMonthsDays' :: [Int] (bound at src/Chap02.hs:65:5)
        regularYearMonthsDays :: [Int] (defined at src/Chap02.hs:29:1)
        leapYearMonthsDays :: [Int] (defined at src/Chap02.hs:32:1)
        [] :: forall a. [a]
          with [] @Int
          (bound at <wired into compiler>)
        mempty :: forall a. Monoid a => a
          with mempty @[Int]
          (imported from ‘Prelude’ at src/Chap02.hs:1:8-13
           (and originally defined in ‘GHC.Base’))
        with :: forall d. Default d => d
          with with @[Int]
          (imported from ‘Diagrams.Prelude’ at src/Chap02.hs:4:1-23
           (and originally defined in ‘Diagrams.Util’))
        (Some hole fits suppressed; use -fmax-valid-hole-fits=N or -fno-max-valid-hole
-fits)
   |
62 |        then _
   |             ^
Failed, no modules loaded.
```

① hole location

② the deduced type

③ context of the hole

④ available bindings

⑤ compatible bindings

It works with functions too

Named hole for a missing function

```
monthDays'' :: Int -> Int -> Int
monthDays'' year nthMonth =
    _getAt monthsDays (nthMonth - 1) ①
```

① this hole has a proper name

Which yields a detailed compilation error:

GHC error: named hole with function type

```
src/Chap02.hs:55:5: error: [GHC-88464] ①
    • Found hole: _getAt :: [Int] -> Int -> Int ②
      Or perhaps ‘_getAt’ is mis-spelled, or not in scope ③
    • In the expression: _getAt monthsDays (nthMonth - 1)
      In an equation for ‘monthDays''’:
          monthDays'' year nthMonth
            = _getAt monthsDays (nthMonth - 1)
            where
                isLeapYear'
                  = (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)
                monthsDays
                  = if isLeapYear' then
                        leapYearMonthsDays'
                    else
                        regularYearMonthsDays'
                regularYearMonthsDays' = [31, ....]
                leapYearMonthsDays' = [31, ....]
    • Relevant bindings include ④
        monthsDays :: [Int] (bound at src/Chap02.hs:60:5)
        isLeapYear' :: Bool (bound at src/Chap02.hs:57:5)
        regularYearMonthsDays' :: [Integer] (bound at src/Chap02.hs:65:5)
        leapYearMonthsDays' :: [Integer] (bound at src/Chap02.hs:66:5)
        nthMonth :: Int (bound at src/Chap02.hs:54:18)
        year :: Int (bound at src/Chap02.hs:54:13)
        (Some bindings suppressed; use -fmax-relevant-binds=N or -fno-max-relevant
-binds)
      Valid hole fits include ⑤
        seq :: forall a b. a -> b -> b
          with seq @[Integer] @Int
          (imported from ‘Prelude’ at src/Chap02.hs:1:8-13
           (and originally defined in ‘GHC.Prim’))
        with :: forall d. Default d => d
          with with @([Integer] -> Int -> Int)
          (imported from ‘Diagrams.Prelude’ at src/Chap02.hs:4:1-23
           (and originally defined in ‘Diagrams.Util’))
```

```
          def :: forall a. Default a => a
            with def @([Integer] -> Int -> Int)
            (imported from Diagrams.Prelude at src/Chap02.hs:4:1-23
             (and originally defined in Data.Default.Class))
   |
55 |       _getAt monthsDays (nthMonth - 1)
   |       ^^^^^^
```

① hole location

② the deduced type

③ this is due to the ambiguity of named holes

④ context of the hole

⑤ available bindings

Holes also work for types:

Type wildcard hole in a signature

```
monthDays'' :: Int -> Int -> Int
monthDays'' year nthMonth =
    monthsDays !! (nthMonth - 1)
  where
    isLeapYear' :: _
    isLeapYear' =
      (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)
```

It will display the following error:

GHC error: type wildcard standing for Bool

```
src/Chap02.hs:57:20: error: [GHC-88464] ①
    • Found type wildcard ‘_’ standing for ‘Bool’ ②
      To use the inferred type, enable PartialTypeSignatures
    • In the type signature: isLeapYear' :: _
      In an equation for ‘monthDays''’:
          monthDays'' year nthMonth
            = monthsDays !! (nthMonth - 1)
            where
                isLeapYear' :: _
                isLeapYear'
                  = (mod year 4 == 0 && mod year 100 /= 0) || (mod year 400 == 0)
                monthsDays
                  = if isLeapYear' then
                        leapYearMonthsDays'
                    else
                        regularYearMonthsDays'
                regularYearMonthsDays' = [31, ....]
                ....
    • Relevant bindings include
        regularYearMonthsDays' :: [Int] (bound at src/Chap02.hs:66:5)
        leapYearMonthsDays' :: [Int] (bound at src/Chap02.hs:67:5)
        nthMonth :: Int (bound at src/Chap02.hs:54:18)
        year :: Int (bound at src/Chap02.hs:54:13)
        monthDays'' :: Int -> Int -> Int (bound at src/Chap02.hs:54:1)
   |
57 |     isLeapYear' :: _
   |                    ^
Failed, no modules loaded.
```

① hole location

② the deduced type

While it is usually quite efficient, recommendations are limited to bindings in scope, not to other modules or libraries. The second tool fills this gap: hoogle[5]

Consider we want to fill `_getAt`:

Hole type signature to search in Hoogle

```
src/Chap02.hs:55:5: error: [GHC-88464]
    • Found hole: _getAt :: [Int] -> Int -> Int
```

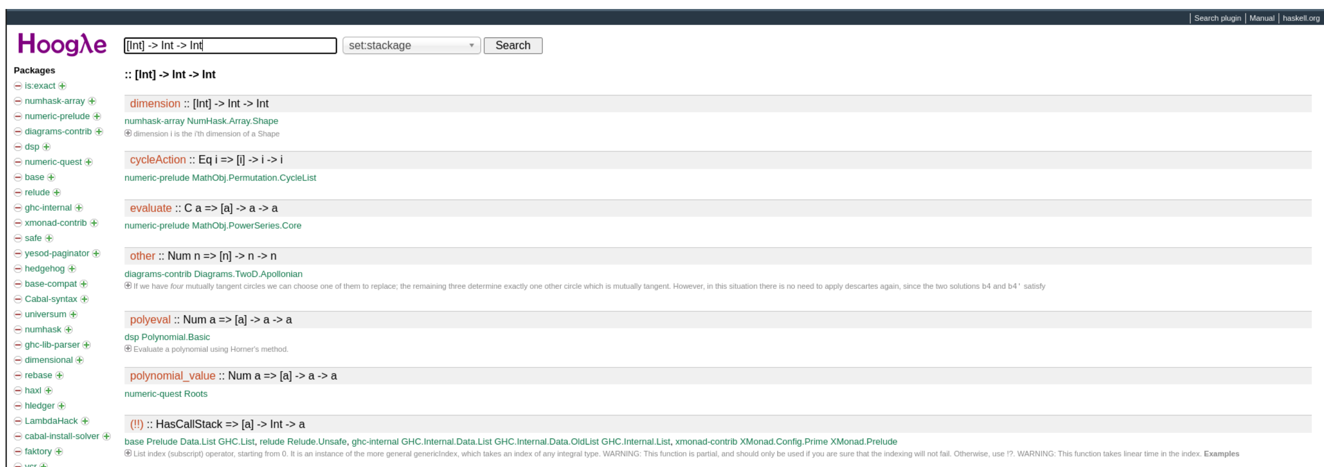


Figure 6. A hoogle search

The interesting result is the last of the screenshot, hoogle has been able to generalize our function to find a usable function. We will go more in-depth about that later, it will also help us to have more relevant results.

2.9. Project: Playing with diagrams

To do something more visual, let us play with diagrams[6], a library to build vector graphics.

First, install the library using `cabal`:

Installing the diagrams library

```
cabal install diagrams
```

2.9.1. A tiny house

The first step is to set up a minimal setup to create an SVG file:

First diagram: a filled triangle

```
house :: Diagram B ①
house = fc brown (triangle 2)

saveHouse :: IO () ②
saveHouse = renderSVG "house.svg" (dims2D 800 800) house
```

① the image definition

② save the image as a 800x800 image

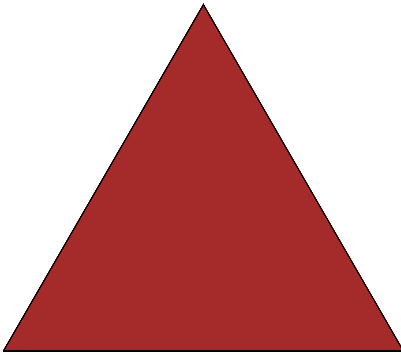


Figure 7. A simple triangle

This mechanism also works in two steps:

- Create a triangle of size 2
- Fill it in brown

It can be noticed that the type does not change, which means, the type contains the color.

The next step is to add a facade below the roof thanks to the dedicated operator `(===)`:

Vertical composition with the === operator

```
house :: Diagram B
house = roof === facade
  where
    roof = scaleY 0.5 (fc brown (triangle 2)) ①
    facade = fc grey (square 2)
```

① reduce the height of the triangle/roof

NOTE	Composability is one of the main goals of library building. Instead of having monolithic functions that create diagrams with many parameters (e.g. height, width, colors, etc.), the library provides a small set of basic shape, and a large set of "modifiers" which keep the type while changing the expression, making everything composable.

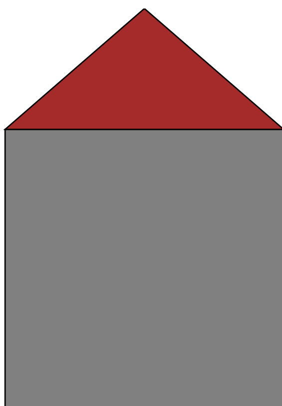


Figure 8. A simple house

It can be visualized as:

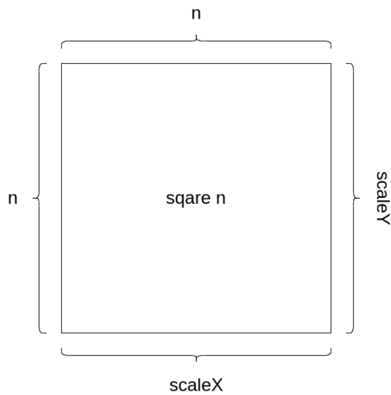


Figure 9. Scaling

If we aim to add a door and windows, we should adopt a more intricate approach.

There is a function `atop` which puts a diagram at the center of another one, we have a `translate` function to alleviate this issue.

Overlapping diagrams with atop and translate

```
house :: Diagram B
house = roof === facade
  where
    roof = scaleY 0.5 (fc brown (triangle 2))
    facade = (atop door . atop leftWindow . atop rightWindow) wall
    wall = fc grey (square 2)
    leftWindow = translate (r2 (-0.5, 0.5)) window
    rightWindow = translate (r2 (0.5, 0.5)) window
    door = translate (r2 (0, -0.6)) (scaleY 2 (fc brown (square 0.4)))
    window = fc white (square 0.5) ①
```

① a window is defined once, used twice

Which renders to the following picture:

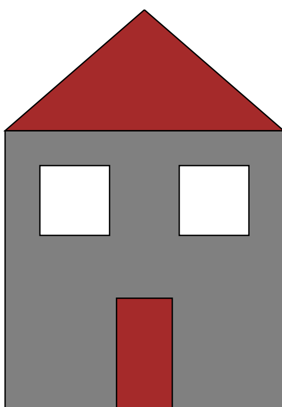


Figure 10. A house with a door and windows

Additionally, we can add a bush:

Adding a bush to the house

```
house :: Diagram B
house = roof === facade
  where
    roof = scaleY 0.5 (fc brown (triangle 2))
    facade = (atop bush . atop door . atop leftWindow . atop rightWindow) wall
    wall = fc grey (square 2)
    leftWindow = translate (r2 (-0.5, 0.5)) window
    rightWindow = translate (r2 (0.5, 0.5)) window
    door = translate (r2 (0, -0.6)) (scaleY 2 (fc brown (square 0.4)))
    window = fc white (square 0.5)
    bush = translate (r2 (0.55, -0.85)) (atop (translate (r2 (0.15, 0.15)) bushCircle)
(bushCircle ||| bushCircle))
    bushCircle = lw none (fc green (circle 0.15))
```

Which renders to the following picture:

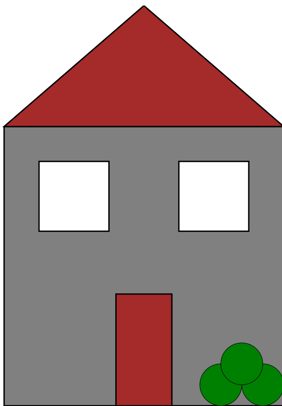


Figure 11. A house with a bush

2.9.2. A skyscraper

An individual house may be a dream, but it is not really sustainable, let us see if we can build a modest skyscraper.

As we did for the tiny house, we have a bit of boilerplate to save the SVG file:

A basic skyscraper diagram

```
skyscraper :: Diagram B
skyscraper = roof === facade
  where
    roof = scaleY 0.1 (fc black (square 2))
    facade = wall
    wall = scaleY 3 (fc grey (square 2))

saveSkyscraper :: IO ()
saveSkyscraper = renderSVG "skyscraper.svg" (dims2D 800 800) skyscraper
```

Which renders to the following picture:

Figure 12. A boring skyscraper

The next step is to add a door for entry into the building:

Adding a door to the skyscraper

```
skyscraper :: Diagram B
skyscraper = roof === facade
  where
    roof = scaleY 0.1 (fc black (square 2))
    facade = (atop door) wall
    wall = scaleY 3 (fc grey (square 2))
    door = translate (r2 (0, -2.8)) (scaleY 2 (fc brown (square 0.2)))
```

Which renders to the following picture:

Figure 13. A skyscraper with an entrance

The skyscraper still lacks windows, we can start by adding one, to see where it will land:

Adding a single window to the skyscraper

```
skyscraper :: Diagram B
skyscraper = roof === facade
  where
    roof = scaleY 0.1 (fc black (square 2))
    facade = (atop windows . atop door) wall
    wall = scaleY 3 (fc grey (square 2))
    door = translate (r2 (0, -2.8)) (scaleY 2 (fc brown (square 0.2)))
    windows = window
    window = fc white (scaleX 0.5 (square 0.5))
```

Which renders to the following picture:

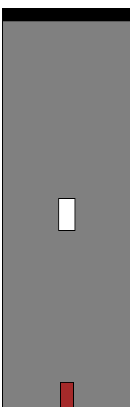


Figure 14. A skyscraper with a window

In the tiny house example, we have carefully placed each window, here, we have too many windows to do this. We can leverage two functions:

- **hsep** which, given a space-size and a list of diagrams, creates a diagram by concatenating each of them horizontally, adding some space between them
- **replicate** which creates a list of n elements given a number and an element

At this point, we may have realized that all values have the same weight, regardless of how they are constructed (hard-coded, or generated), they can be used transparently, this principle is the cornerstone of functional composition.

Repeating elements with replicate and hsep

```
skyscraper :: Diagram B
skyscraper = roof === facade
  where
    roof = scaleY 0.1 (fc black (square 2))
    facade = (atop windows . atop door) wall
    wall = scaleY 3 (fc grey (square 2))
    door = translate (r2 (0, -2.8)) (scaleY 2 (fc brown (square 0.2)))
    windows = windowsRow
    windowsRow = translate (r2 (-0.75, 0)) (hsep 0.12 (replicate 5 window))
    window = fc white (scaleX 0.5 (square 0.5))
```

Which renders to the following picture:

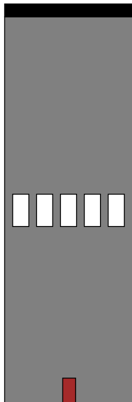


Figure 15. A skyscraper with a windows row

We also had to `translate` them to center the row.

We can use the same tactic to have many rows:

Stacking rows with vsep

```
skyscraper :: Diagram B
skyscraper = roof === facade
  where
    roof = scaleY 0.1 (fc black (square 2))
    facade = (atop windows . atop door) wall
    wall = scaleY 3 (fc grey (square 2))
    door = translate (r2 (0, -2.8)) (scaleY 2 (fc brown (square 0.2)))
    windows = translate (r2 (0, 2.5)) (vsep 0.40 (replicate 6 windowsRow))
    windowsRow = translate (r2 (-0.75, 0)) (hsep 0.12 (replicate 5 window))
    window = fc white (scaleX 0.5 (square 0.5))
```

Which renders to the following picture:

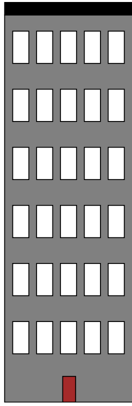


Figure 16. A skyscraper with a lot of windows

2.10. Conclusion

We have covered the essential building blocks of functions in Haskell. We started with the anatomy of function definitions, understanding how expressions and bindings work together. We explored immutability and referential transparency, which form the foundation of reasoning about Haskell programs. We learned to create functions using lambdas, currying, closures, and composition. We also developed the skill of reading type errors and navigating types using Hoogle and typed holes.

The diagrams project demonstrated how these concepts combine in practice: small, focused functions compose into complex visual outputs.

In the next chapter, we will build on this foundation to explore control flow, higher-order functions, and Haskell's evaluation model.

2.11. Questions

1. What are the two parts of a function?
2. What is the difference between a function and an expression?
3. What is referential transparency?
4. What is currying?
5. What is function composition and what operator is used for it?

2.12. Exercises

1. Write a function `addThree` that adds three to a number, using lambda syntax as follows:

Exercise template: addThree

```
addThree :: Int -> Int
-- > addThree 5
-- 8
```

2. Write a function `squareAll` that squares every element in a list using `map` as follows:

Exercise template: squareAll

```
squareAll :: [Int] -> [Int]
-- > squareAll [1, 2, 3]
-- [1,4,9]
```

3. Using the `diagrams` library, write a function that renders a horizontal row of five red circles as follows:

Exercise template: fiveRedCircles

```
fiveRedCircles :: Diagram B
-- (should render five red circles in a row)
```

2.13. Answers to questions

1. What are the two parts of a function?

The type signature and the definition (the expression or equations that make up its body).

2. What is the difference between a function and an expression?

An expression is any piece of Haskell code that has a value. A function is an expression that takes one or more parameters and produces a result.

3. What is referential transparency?

An expression is referentially transparent if it can be replaced with its value without changing the program's behavior.

4. What is currying?

Currying is the process by which a function with multiple parameters is represented as a sequence of functions, each taking a single parameter.

5. What is function composition and what operator is used for it?

Function composition combines two functions into one, feeding the output of the second function as input to the first. The operator is `(.)`.

2.14. Answers to exercises

1. Write a function `addThree` that adds three to a number:

Solution: addThree using lambda syntax

```
addThree :: Int -> Int
addThree = \x -> x + 3
```

2. Write a function `squareAll` that squares every element in a list:

Solution: squareAll using map

```
squareAll :: [Int] -> [Int]
squareAll = map (^ 2)
```

3. Using the `diagrams` library, render a horizontal row of five red circles:

Solution: fiveRedCircles using hsep and replicate

```
fiveRedCircles :: Diagram B
fiveRedCircles = hsep 1 (replicate 5 (fc red (circle 1)))
```

[1] https://errors.haskell.org/

[2] https://en.wikipedia.org/wiki/Test-driven_development

[3] sometimes, it could be as simple as introducing bindings not yet defined

[4] this is confusing to use a _-prefixed binding

[5] https://hoogle.haskell.org/

[6] https://diagrams.github.io