

Extracted from:

# Effective Haskell

## Solving Real-World Problems with Strongly Typed Functional Programming

This PDF file contains pages extracted from *Effective Haskell*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The  
Pragmatic  
Programmers

# Effective Haskell

Solving Real-World Problems with  
Strongly Typed Functional Programming



Rebecca Skinner  
*Edited by Michael Swaine*



# Effective Haskell

Solving Real-World Problems  
with Strongly Typed Functional Programming

Rebecca Skinner

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

The team that produced this book includes:

CEO: Dave Rankin

COO: Janet Furlow

Managing Editor: Tammy Coron

Development Editor: Michael Swaine

Copy Editor: Karen Galle

Indexing: Potomac Indexing, LLC

Layout: Gilson Graphics

Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact [support@pragprog.com](mailto:support@pragprog.com).

For international rights, please contact [rights@pragprog.com](mailto:rights@pragprog.com).

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-934-2

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—July 2023

Linked lists are a first class data structure that you will use frequently throughout this book and as you are writing programs in Haskell. In the last chapter, you were introduced to the basics of Haskell's syntax and learned how to create some simple lists. In this chapter, you'll learn more about how to work with lists using *higher-order functions* like `map` and `foldr`, how to write recursive functions over lists effectively using *pattern matching*, and finally, you'll learn about how to deal with streaming data, generators, and infinitely long lists by exploiting Haskell's laziness.

## Writing Code Using Lists

As you saw in the last chapter, lists of values are enclosed in square brackets, and separated by commas. We can make lists of any type we want, but a list can only hold a single type. Here are some examples of different types of lists. Try creating them in `ghci`:

```
λ listOfNums = [1, 2, 3]
λ listOfFloats = [1.1, 2.2, 3.3]
λ listOfStrings = ["hello", "world"]
```

There's also a special type of list that you've already used extensively: strings! In Haskell, regular strings are simply lists of characters. When you create a string using double quotes, it's really just a nice way of writing a list of individual characters. We can see this for ourselves in `ghci`:

```
λ ['h','e','l','l','o'] == "hello"
True
```

You'll rarely, if ever, write strings using the normal list syntax. However, you will frequently use other list functions when working with strings. This also means that some of the functions you've already used, like the `(< >)` operator, are actually more general than you might have realized. For example, we can combine lists of other values just as well as strings:

```
λ [1,2,3] <> [4,5,6]
[1,2,3,4,5,6]
λ ['h', 'e'] <> "llo"
"hello"
λ [[1,2,3],[4,5,6]] <> [[7,8,9]]
[[1,2,3],[4,5,6],[7,8,9]]
```

You can get the `nth` element of a list using the `(!!)` operator. List indices start at 0:

```
λ words = ["foo", "bar", "baz", "fizz", "buzz"]
words !! 0
```

```
"foo"
λ words !! 4
"buzz"
```

You need to be careful to not accidentally try to take an index that's larger than the length of a list:

```
λ words !! 5
*** Exception: Prelude.!!: index too large
```

We're not limited to just single values; you can also create lists of lists:

```
λ nums = [[1,3,5],[2,4],[0]]
λ strings = [["hello", "good morning"], ["so long", "farewell"]]
```

We can create empty lists with just an opening and closing bracket, like this: [].

A list can only hold a single type of value. Because of this, you can't have a list with lists of different types. This would be an error, for example:

```
λ badList = [[1,2,3],["one","two","three"]]
```

Lists, like all other Haskell values, are immutable. Although you can't change the value of a list, you can efficiently construct lists by prepending a new element to the start of an existing list. The process of adding a new element to the beginning of a list is called “cons-ing,” and the operator we use to add an element to the front of a list, (:), is generally pronounced “cons” as in “construct.” This style of prepending an element to the beginning of a list is so ubiquitous that when someone says that they are “adding an element to a list” you should assume they are prepending it unless they say otherwise.

Let's look at a couple of examples of constructing lists using (:). Try typing out these examples yourself to see what the generated lists look like, and to get more used to creating lists.

```
λ 1 : [2,3]
λ 1 : 2 : [3]
λ 1 : 2 : 3 : []
λ 'h' : "ello"
λ 'h' : 'e' : ['l','l','o']
λ [1,2,3] : []
λ [1] : [2] : [3] : []
```

Although you've seen that you can append elements to a list with (<>), it's far more common in Haskell to build our lists by prepending elements with (:). In fact, prepending elements is so common that it's not unusual to see functions where an entire list is built backwards only to be reversed at the end before being returned. The reason for this is that in an immutable language,

prepending an element to a list is much more efficient than appending an element to the end.

Later, [on page ?](#), you'll learn how to create list-like data structures yourself, which will help you better understand the details of why prepending is more efficient than appending to lists.

However you approach creating a list, in the end it comes down to putting one element in front of another. When you add an element to the front of a list, we call the part that you are adding the *head* of the list. In fact, a common alternative phrase *cons-ing* is to *push an element onto the head of a list*. The list that you are adding the element onto becomes the *tail*.

```
head : tail
```

The tail of a list is itself either any empty list, or a list with its own head and tail, so you can also look at a list as a series of heads preceding a final empty list:

```
head : tail = head : (head : (head : ... : []))
```

Head and tail aren't just the terms we use to talk about parts of a list. The head and tail functions let you deconstruct a list and get the first element and the rest of the elements back out of a list you've constructed. Let's look at some examples of using head and tail so you can get a feel for how they work:

```
λ head [1,2,3]
1
λ tail [1,2,3]
[2,3]
λ head (tail [1,2,3])
2
λ tail (tail [1,2,3])
[3]
λ tail [1]
[]
```

The head and tail functions are quite useful when you need to get at different parts of a list that you've built, but some caution is necessary with these functions. Both head and tail are *partial functions*. A *partial function* is a function that doesn't work for all of its possible inputs, and might raise a runtime exception or cause the program to crash. In the case of head and tail, these functions will cause a runtime exception if you use them on an empty list:

```
λ head []
*** Exception: Prelude.head: empty list
λ tail []
*** Exception: Prelude.tail: empty list
```



You'll learn another way to get the head and tail of a list [on page ?](#) that doesn't risk raising these exceptions, and later in this book you'll learn how to handle runtime exceptions. In the meantime, take care to make sure you check first to see if a list is empty before using these functions. You can check for an empty list with equality:

```
listIsEmpty list =
  if list == []
  then putStrLn "this list is empty"
  else putStrLn ("the first element of this list is: " <> show (head list))
```

Alternatively, you can use the null function, which will return True if a list is empty, and False otherwise:

```
listIsEmpty' list =
  if null list
  then putStrLn "this list is empty"
  else putStrLn ("the first element of this list is: " <> show (head list))
```