Learn You Some Erlang for Great Good!

A Beginner's Guide

Fred Hébert

Foreword by Joe Armstrong

no starch press

LEARN YOU SOME ERLANG FOR GREAT GOOD!

Learn You Some Erlang for Great Good!

A Beginner's Guide

Fred Hébert

no starch press

San Francisco

Printed in USA

First printing

17 16 15 14 13   1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-435-1
ISBN-13: 978-1-59327-435-1

Publisher: William Pollock
Production Editor: Alison Law
Cover Design: Sonia Brown
Developmental Editor: Keith Fancher
Technical Reviewer: Geoff Cant
Copyeditor: Marilyn Smith
Compositor: Susan Glinert Stevens
Proofreader: Greg Teague

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
38 Ringold Street, San Francisco, CA 94103
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data
A catalog record of this book is available from the Library of Congress.

BRIEF CONTENTS

CONTENTS IN DETAIL

ABOUT THE AUTHOR

Fred Hébert is a self-taught programmer with experience in frontend web development, web services, and general backend programming in various languages. His online tutorial, *Learn You Some Erlang for Great Good!*, is widely regarded as the best way to learn Erlang. While at Erlang Solutions Ltd., he wrote training materials and taught Erlang all around the Western world. He currently works with Erlang on a real-time bidding platform (AdGear) and was named Erlang User of the Year 2012.

FOREWORD

Learning to program is fun, or at least it should be fun. If it's not fun, you won't enjoy doing it. During my career as a programmer, I have taught myself several different programming languages, and it hasn't always been fun. Whether or not learning a language is fun depends to a large extent on how the language is introduced.

When you start working with a new programming language, on the surface it seems that all you are doing is learning a new language. But at a deeper level, you are doing something much more profound—you are learning a new way of thinking. It's this new way of thinking that is exciting, not the minor details of the punctuation or how the language looks compared to your favorite programming language.

Functional programming is one of those areas of programming that has acquired a reputation for being "hard" (concurrent programming even more so), and so writing a book about Erlang that covers the ideas of functional programming *plus* concurrent programming is a daunting prospect.

Make no mistake about it: Introducing functional programming is not so easy, and introducing concurrent programming has its difficulties. Doing both with humor and ease requires a very particular kind of talent.

Fred Hebert has shown that he has this talent. He explains complex ideas in a way that makes them seem simple.

One of the biggest barriers to learning Erlang is not so much that the ideas involved are intrinsically difficult but that they are very different from the ideas in most of the other languages that you will have encountered. To learn Erlang, you have to temporarily unlearn what you have learned in other programming languages. Variables in Erlang do not vary. You're not supposed to program defensively. Processes are really, really cheap, and you can have thousands of them or even millions, if you feel like it. Oh, and then there is the strange syntax. Erlang doesn't look like Java; there are no methods or classes and no objects. And wait a moment . . . even the equals sign doesn't mean "equals"—it means "match this pattern."

Fred is completely undaunted by these issues; he treats the subject matter with a delicate dry humor and teaches complex subjects in such a way that we forget the complexity.

This is now the fourth major text on Erlang and is a great addition to the Erlang library. But it's not only about Erlang. Many of the ideas in Fred's book are equally applicable to Haskell or OCaml or F#.

I hope that many of you will enjoy reading Fred's book as much as I did and that you find learning Erlang to be an agreeable and thought-provoking process. If you type in the programs in this book and run them as you go along, you'll learn even more. Writing programs is much more difficult than reading them, and the first step is just letting your fingers get used to typing in the programs and getting rid of the small syntax errors that inevitably occur. As you get deeper into the book, you'll be writing programs that are pretty tricky to write in most other languages—but hopefully you won't realize this. Soon you'll be writing distributed programs. This is when the fun starts. . .

Thanks, Fred, for a great book.

Joe Armstrong
Stockholm, Sweden
November 6, 2012

PREFACE

This book initially started as a website, which is still available at *http://learnyousomeerlang.com/* (thanks to No Starch Press's open-mindedness regarding all things related to publishing and technical material). Since the first chapters were made public in 2009, *Learn You Some Erlang* has grown from a three-chapter micro-tutorial with a request for proofreading on the *erlang-questions* mailing list into one of the official documentation's suggestions for learning Erlang, a book, and a major accomplishment in my life. I'm baffled and thankful for all it has brought me, from friends to jobs to the title of Erlang User of the Year 2012.

To the Foreigner

When you're looking at Erlang programmers from afar, as an outsider, they may seem like a weird little community of people who believe in principles that nearly nobody else needs or wants to follow. Their principles look impractical, limited in how they can be applied. To make matters worse, Erlang citizens may appear similar to members of a religious sect, entirely sure that they know the one true way to the heart of software. This is the same kind of "one true way" previously preached by fanatics of languages like those of the Lisp family, Haskellers, proud members of the formal proof school of thought, Smalltalk programmers, stack aficionados from the world of Forth, and so on. Same old, same old; they all offer great promises of success, and deliver in various ways, but the programs we programmers write are still buggy, too expensive, or unmaintainable.

With Erlang, it's likely the promise of concurrency or parallelism that brings you here. Maybe it's the distributed computing aspect of the language, or possibly its unusual approach to fault tolerance. Of course, approaching Erlang with skepticism is a good thing. It won't solve all your problems—that's your job, after all. Erlang is merely a nifty toolbox to help you do so.

To the Erlang Regular

You already know Erlang, possibly very well. In that case, I hope this book becomes an interesting read or a possible reference, or that a few of its chapters help you learn more about bits of the language and its environment that you weren't too familiar with before.

It's also possible that you know Erlang better than I do in every respect. In that case, I hope this book makes an adequate paperweight or space-filler in your library.

To the Person Who Has Read This Online

Thanks for your support, and I hope you enjoy what professional editing has brought to the original text, along with a boost into R15B+ versions of Erlang.

ACKNOWLEDGMENTS

Thanks to Miran Lipovača for coming up with the *Learn You a Language* idea first, and for letting me borrow the concept for this book and its related website.

Thanks to Jenn (my girlfriend) for the original website design, the long yeoman's work required to redraw most of the images of this book so they would be suitable for print, her support, and her patience in letting me spend so many hours working on this project.

Thanks to all the people who gave their time away to help review the online copy of this book, find errors, and offer support (in no particular order): Michael Richter, OJ Reeves, Dave Pawson, Robert Virding, Richard O'Keefe, Ulf Wiger, Lukas Larsson, Dale Harvey, Richard Carlsson, Nick Fitzgerald, Brendon Hogger, Geoff Cant, Andrew Thompson, Bartosz Fabianowski, Richard Jones, Tuncer Ayaz, William King, Mahesh Paolini-Subramanya, and Malcolm Matalka. There were also many other people who provided minor reviews and spotted typos and other errors.

A second thanks to Geoff Cant, who was the official tech reviewer for this version of the book.

Thanks to the team at No Starch Press (Keith, Alison, Leigh, Riley, Jessica, Tyler, and Bill) for their professional work.

Finally, thanks to the countless readers of the online copy of this book: those who bought this version, and those who read it without buying it.

INTRODUCTION

This is the beginning of *Learn You Some Erlang for Great Good!* Reading this book should be one of your first steps in learning Erlang, so let's talk about it a bit.

I got the idea to write this book after reading Miran Lipovača's *Learn You a Haskell for Great Good!* (LYAH) tutorial. I thought he did a great job making the language attractive and the learning experience friendly. As I already knew him, I asked him how he felt about me writing an Erlang version of his book. He liked the idea, being somewhat interested in Erlang.

So I began writing this book.

Of course, there were other sources to my motivation. When I began, I found the entry to the language to be hard (the Web had sparse documentation, and books are expensive), and I thought the community would benefit from a LYAH-like guide. Also, I had seen people attributing Erlang too little—or sometimes too much—merit based on sweeping generalizations.

This book is a way to learn Erlang for people who have a basic knowledge of programming in imperative languages (such as C/C++, Java, Python, Ruby, and so on) but may or may not be familiar with functional programming languages (such as Haskell, Scala, Clojure, and OCaml, as well as Erlang). I also wanted to write this book in an honest manner, selling Erlang for what it is, acknowledging its weaknesses and strengths.

So What's Erlang?

Erlang is a functional programming language. If you have ever worked with imperative languages, statements such as `i++` may be normal to you, but in functional programming, they are not allowed. In fact, changing the value of any variable is strictly forbidden! This may sound weird at first, but if you remember your math classes, that's how you learned it:

```
y = 2
x = y + 3
x = 2 + 3
x = 5
```

If I added the following, you would have been very confused.

```
x = 5 + 1
x = x
∴ 5 = 6
```

Functional programming recognizes this. If I say `x` is `5`, then I can't logically claim it is also `6`! This would be dishonest. This is also why a function should return the same result every time it's called with the same parameter:

```
x = add_two_to(3) = 5
∴ x = 5
```

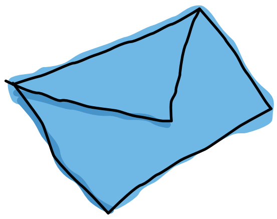
The concept of functions always returning the same result for the same parameter is called *referential transparency*. It's what lets us replace `add_two_to(3)` with `5`, as the result of `3+2` will always be `5`. That means we can glue dozens of functions together in order to resolve more complex problems while being sure nothing will break. Logical and clean, isn't it? There's a problem though:

```
x = today() = 2013/10/22
    -- wait a day --
x = today() = 2013/10/23
x = x
∴ 2013/10/22 = 2013/10/23
```

Oh no! My beautiful equations! They suddenly all turned wrong! How come my function returns a different result every day?

Obviously, there are some cases where it's useful to break referential transparency. Erlang has this very pragmatic approach with functional programming: Obey its purest principles (referential transparency, avoiding mutable data, and so on), but break away from them when real-world problems pop up.

Although Erlang is a functional programming language, there's also a large emphasis on concurrency and high reliability. To be able to have dozens of tasks being performed at the same time, Erlang uses the actor model, and each actor is a separate process in the virtual machine. In a nutshell, if you were an actor in Erlang's world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message. Once you got a message, you would react to it in a specific way: You pay the bills, you respond to birthday cards with a "thank you" letter, and you ignore the letters you can't understand.



Erlang's actor model can be imagined as a world where everyone is sitting alone in a room and can perform a few distinct tasks. Everyone communicates strictly by writing letters, and that's it. While it sounds like a boring life (and a new age for the postal service), it means you can ask many people to perform very specific tasks for you, and none of them will ever do something wrong or make mistakes that will have repercussions on the work of others. They may not even know of the existence of people other than you (and that's great).

In practice, Erlang forces you to write actors (processes) that will share no information with other bits of code unless they pass messages to each other. Every communication is explicit, traceable, and safe.

Erlang is not just a language but also a development environment as a whole. The code is compiled to bytecode and runs inside a virtual machine. So Erlang, much like Java and kids with ADD, can run anywhere. Here are just some of the components of the standard distribution:

- Development tools (compiler, debugger, profiler, and test frameworks, optional type analyzer)
- The Open Telecom Platform (OTP) framework
- A web server
- Advanced tracing tools
- The Mnesia database (a key/value storage system able to replicate itself on many servers, which supports nested transactions and lets you store any kind of Erlang data)

The virtual machine and libraries also allow you to update the code of a running system without interrupting any program, distribute your code with ease on many computers, and manage errors and faults in a simple but powerful manner.

We'll cover how to use most of these tools and achieve safety in this book.

Speaking of safety, you should be aware of a related general policy in Erlang: Let it crash—not like a plane with dozens of passengers dying, but more like a tightrope walker with a safety net below. While you should avoid making mistakes, you won't need to check for every type or error condition in most cases.

Erlang's ability to recover from errors, organize code with actors, and scale with distribution and concurrency all sound awesome, which brings us to the next section . . .

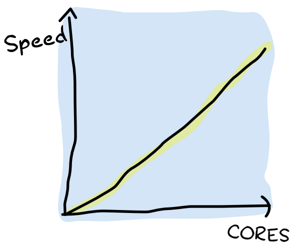
Don't Drink Too Much Kool-Aid

This book has many little boxed sections named like this one (you'll recognize them when you see them). Erlang is currently gaining a lot of popularity due to zealous talks, which may lead people to believe it's more than what it really is. The following are some reminders to help you keep your feet on the ground if you're one of these overenthusiastic learners.

First is the talk of Erlang's massive scaling abilities due to its lightweight processes. It is true that Erlang processes are very light; you can have hundreds of thousands of them existing at the same time. But this doesn't mean you *should* use Erlang that way just because you *can*. For example, creating a shooter game where everything including bullets is its own actor is madness. The only thing you'll shoot with a game like that is your own foot. There is still a small cost in sending a message from actor to actor, and if you divide tasks too much, you will make things slower!

I'll cover this in more depth when we're far enough into the tutorial to actually worry about it, but just keep in mind that randomly throwing parallelism at a problem is not enough to make it go fast. (Don't be sad; occasionally, using hundreds of processes is both possible and useful!)

Erlang is also said to be able to scale in a directly proportional manner to how many cores your computer has, but this is usually not true. It is possible, but in most cases, problems do not behave in a way that lets you just run everything at the same time.



Something else to keep in mind is that while Erlang does some things very well, it's technically still possible to get the same results from other languages. The opposite is also true. You should evaluate each problem that you need to solve and choose the best tool for that problem and its solution. Erlang is no silver bullet and will be particularly bad at things like image and signal processing, operating system device drivers, and other functions. It will shine at things like large software for server use (for example, queue middleware, web servers, real-time bidding and distributed database implementations), doing some lifting coupled with other languages, higher-level protocol implementation, and so on. Areas in the middle will depend on you.

You should not necessarily limit yourself to server software with Erlang. People have done unexpected and surprising things with it. One example is IANO, a robot created by the Unict team (the Eurobot team of the University of Catania), which uses Erlang for its artificial intelligence. IANO won the silver medal at the 2009 Eurobot competition. Another example is Wings 3D, an open source, cross-platform 3D modeler (but not a renderer) written in Erlang.

What You Need to Dive In

All you need to get started is a text editor and the Erlang environment. You can get the source code and the Windows binaries from the official Erlang website.

For Windows systems, just download and run the binary files. Don't forget to add your Erlang directory to your `PATH` system variable to be able to access it from the command line.

On Debian-based Linux distributions, you should be able to install the package with this command:

```
$ sudo apt-get install erlang
```

On Fedora (if you have `yum` installed), you can install Erlang by typing this:

```
# yum install erlang
```

However, these repositories often hold outdated versions of the Erlang packages. Using an outdated version could give you some different results from those shown for the examples in this book, as well as a hit in performance with certain applications. I therefore encourage you to compile from source. Consult the *README* file within the package and Google to get all the installation details you'll need.

On FreeBSD, many options are available. If you're using `portmaster`, you can use this command:

```
$ portmaster lang/erlang
```

For standard ports, enter the following:

```
$ cd /usr/ports/lang/erlang; make install clean
```

Finally, if you want to use packages, enter this:

```
$ run pkg_add -rv erlang
```

If you're on a Mac OS X system, you can install Erlang with Homebrew:

```
$ brew install erlang
```

Or, if you prefer, use MacPorts:

```
$ port install erlang
```

NOTE *At the time of this writing, I'm using Erlang version R15B+, so for the best results, you should use that version or a newer one. However, most of the content in this book is also valid for versions as old as R13B.*

Along with downloading and installing Erlang, you should also download the complete set of files available for this book. They contain tested copies of any program and module written within these pages, and they might prove useful for fixing your own programs. They also can provide a base for later chapters if you feel like skipping around. The files are all packaged in a zip file, available at *http://learnyousomeerlang.com/static/erlang/learn-you-some-erlang.zip*. Otherwise, the examples in *Learn You Some Erlang* depend on no other external dependency.

Where to Get Help

If you're using Linux, you can access the man pages for good technical documentation. For example, Erlang has a `lists` module (as you'll see in Chapter 1). To get the documentation on lists, just type in this command:

```
$ erl -man lists
```

On Windows, the installation should include HTML documentation. You can download it at any time from the official Erlang site, or consult one of the alternative sites.

Good coding practices can be found at *http://www.erlang.se/doc/programming_rules.shtml* when you feel you need to get your code clean. The code in this book will attempt to follow these guidelines, too.

Now, there are times when just getting the technical details isn't enough. When that happens, I tend to turn to two main sources: the official Erlang mailing list (you should follow it just to learn a bunch) and the *#erlang* channel on *irc.freenode.net*.

1

STARTING OUT

In Erlang, you can test most of your code in an emulator. It will run your scripts when they are compiled and deployed, but it will also let you edit stuff live.

In this chapter, you'll learn how to use the Erlang shell and be introduced to some basic Erlang data types.

Using the Erlang Shell

To start the Erlang shell in a Linux or Mac OS X system, open a terminal, and then type **erl**. If you've set up everything correctly, you should see something like this:

```
$ erl
Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:0] [hipe]
[kernel-poll:false]

Eshell V5.9  (abort with ^G)
```

Congratulations, you're running the Erlang shell!

If you're a Windows user, you can run the shell by executing *erl.exe* at the command prompt, but it's recommended that you use *werl.exe* instead, which can be found in your Start menu (choose **All Programs ▸ Erlang**). Werl is a Windows-only implementation of the Erlang shell that has its own window with scroll bars and supports line-editing shortcuts (which are not available with the standard *cmd.exe* shell in Windows). However, the *erl.exe* shell is still required if you want to redirect standard input or output, or use pipes.

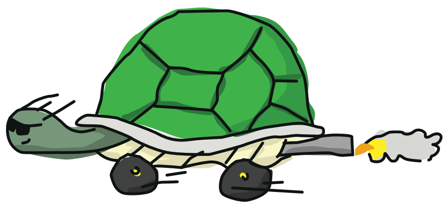
Now we can enter code into the shell and run it in the emulator. But first, let's see how we can get around in it.

Entering Shell Commands

The Erlang shell has a built-in line editor based on a subset of Emacs, a popular text editor that has been in use since the 1970s. If you know Emacs, you should be fine. And even if you don't know Emacs, you'll do fine anyway.

To begin, type some text in the Erlang shell, and then press CTRL-A (^A). Your cursor should move to the beginning of the line. Similarly, pressing CTRL-E (^E) moves the cursor to the end of the line. You can also use the left and right arrow keys to move the cursor forward and backward, and cycle through previously written lines of code by using the up and down arrow keys.

Let's try something else. Type **`li`**, and then press TAB. The shell will expand the term for you to `lists:`. Press TAB again, and the shell will suggest all the functions available in the `lists` module. You may find the notation weird, but don't worry, you'll become familiar with it soon enough. (We'll learn more about modules in Chapter 2.)



Exiting the Shell

At this point, you've seen most of the basic Erlang shell functionality, except for one very important thing: You don't know how to exit! Luckily, there's a fast way to find out: type **`help().`** into the shell and press ENTER. You'll see information about a bunch of commands, including functions to inspect processes, manage how the shell works, and so on. We'll use many of these in this book, but the only one of interest right now is the following expression:

```
q() -- quit - shorthand for init:stop()
```

So, this is one way to exit (two ways, in fact). But this won't help if the shell freezes!

If you were paying attention when you started the shell, you probably saw a comment about "aborting with ^G." So let's press CTRL-G, and then type **h** to get help.

```
User switch command
--> h
c [nn]            - connect to job
i [nn]            - interrupt job
k [nn]            - kill job
j                 - list all jobs
s [shell]         - start local shell
r [node [shell]]  - start remote shell
q                 - quit erlang
? | h             - this message
-->
```

If you are wearing a monocle, now would be the time to drop it. The Erlang shell isn't just a simple shell as with other languages. Instead, it is a bundle of shell instances, each running different *jobs*. Moreover, you can manage them like processes in an operating system. If you type `k N`, where `N` is a job number, you will terminate that shell and all the code it was running at the time. If you want to stop the code that is running without killing the shell, then `i N` is the command you need. You can also create new shell instances by typing in `s`, list them with `j`, and connect to them with `c N`.

At some point, you might see an asterisk (`*`) next to some shell jobs:

```
 --> j
   1* {shell,start,[init]}
```

The `*` means that this is the last shell instance you were using. If you use the command `c`, `i`, or `k` without any number following it, that command will operate on this last shell instance.

If your shell ever freezes, a quick sequence to help is to press CTRL-G, type **i**, press ENTER, type **c**, and press ENTER (^G **i** ENTER **c** ENTER). This will get you to the shell manager, interrupt the current shell job, and then connect back to it:

```
Eshell V5.9  (abort with ^G)
1> "OH NO THIS SHELL IS UNRESPONSIVE!!! *hits ctrl+G*"
User switch command
 --> i
 --> c
** exception exit: killed
1> "YESS!"
```

There's one important thing to know before you start typing "real" stuff into the shell: *A sequence of expressions must be terminated with a period followed by whitespace* (a line break, a space, and so on); otherwise, it won't be executed. You can separate expressions with commas, but only the result of the last one will be shown (although the others are still executed). This is certainly unusual syntax for most people, and it comes from the days Erlang was implemented directly in Prolog, a logic programming language.

Now let's get things started (for real) by learning about the basic Erlang data types and how to write your first bits of programs in the shell.

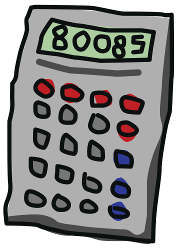
Some Erlang Basics

Although you've just seen a rather advanced mechanism to handle different jobs and shell sessions, Erlang is considered to be a relatively small and simple language (in the way that C is simpler than C++). The language has only a few basic built-in data types (and few syntactic elements around them). First, we'll take a look at numbers.

Numbers

Open the Erlang shell as described earlier, and let's type some things:

```
1> 2 + 15.
17
2> 49 * 100.
4900
3> 1892 - 1472.
420
4> 5 / 2.
2.5
```



As you can see, Erlang doesn't care if you enter floating-point numbers or integers. Both types are supported when dealing with arithmetic.

Note that if you want to perform integer-to-integer division, rather than floating-point division, you should use `div`. To get the remainder (modulo) of an integer division, use `rem` (remainder).

```
5> 5 div 2.
2
6> 5 rem 2.
1
```

You can use several operators in a single expression, and mathematical operations obey the normal precedence rules:

```
7> (50 * 100) - 4999.
1
8> -(50 * 100 - 4999).
-1
9> -50 * (100 - 4999).
244950
```

If you want to express integers in other bases than base 10, just enter the number in the form *Base#Value* (as long as *Base* is in the range of 2 through 36), like this:

```
10> 2#101010.
42
11> 8#0677.
447
12> 16#AE.
174
```

Here, we're converting binary, octal, and hexadecimal values to base 10. Awesome! Erlang has the power of the calculator you have on the corner of your desk, but with a weird syntax on top of it. Absolutely exciting!

Invariable Variables

Doing arithmetic is all right, but you won't get far without being able to store the results somewhere. For that, you use variables. If you read the Introduction to this book, you know that variables can't be variable in functional programming.

In Erlang, variables begin with an uppercase letter by definition. The basic behavior of variables can be demonstrated with these six expressions:

```
1> One.
* 1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
```

The first thing these commands tell us is that you can assign a value to a variable exactly once. Then you can "pretend" to assign a value to a variable if it's the same value the variable already has. If the value is different, Erlang will complain. It's a correct observation, but the explanation is a bit more complex and depends on the = operator. The = operator (not the variables) has the role of comparing values and complaining if they're different. If they're the same, Erlang returns the value:

```
7> 47 = 45 + 2.
47
8> 47 = 45 + 3.
** exception error: no match of right hand side value 48
```

When you use the = operator with variables on both sides of it, with the variable on the left side being unbound (without any value associated with it), Erlang will automatically bind the value on the right to the variable on the left. Both variables will then have the same value. The comparison will consequently succeed, and the variable on the left side will keep the value in memory.

Here's another example:

```
9> two = 2.
** exception error: no match of right hand side value 2
```

The command fails because the word `two` begins with a lowercase letter.

NOTE *Technically, variables can also start with an underscore (_), but by convention, their use is restricted to values you do not care about.*

This behavior of the = operator is the basis of something called *pattern matching*, which many functional programming languages have, although Erlang's way of doing things is usually regarded as more flexible and complete than the alternatives. You'll learn more about Erlang pattern matching when we visit other data types in this chapter, and also see how it works with functions in the following chapters.

Note that if you're testing in the shell and save the wrong value to a variable, it is possible to "erase" that variable by using the function `f(Variable).`. If you wish to clear all variable names, use `f().`. These functions are designed to help you when you're testing, and they only work in the shell. When you're writing real programs, you won't be able to destroy values this way. This restriction makes sense if you think about Erlang being usable in industrial scenarios. It's wholly possible that a shell will be active for years without interruption, and you can bet that a given variable will be used more than once in that time period.

Atoms

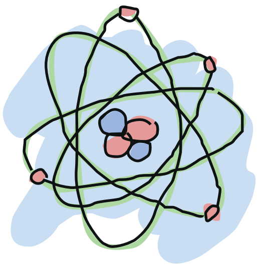
There is a reason why variables names can't begin with a lowercase character: *atoms*. Atoms are literals, which means that they're just constants whose only value is their own name. In other words, what you see is what you get—don't expect more. The atom `cat` means "cat," and that's it. You can't play with it. You can't change it. You can't smash it to pieces. It's `cat`. Deal with it.

While using single words starting with a lowercase letter is one way to write an atom, there are also other ways:

```
1> atom.
atom
2> atoms_rule.
atoms_rule
```

```
3> atoms_rule@erlang.
atoms_rule@erlang
4> 'Atoms can be cheated!'.
'Atoms can be cheated!'
5> atom = 'atom'.
atom
```

An atom should be enclosed in single quotes (`'`) if it does not begin with a lowercase letter or if it contains any characters other than alphanumeric characters, an underscore (`_`), or an at sign (`@`). Line 5 also shows that an atom with single quotes is exactly the same as a similar atom without them.



I compared atoms to constants that have their name as their values. You may have worked with code that used constants before. For example, let's say you have values for eye colors: 1 for blue, 2 for brown, 3 for green, and 4 for other. You need to match the name of the constant to some underlying value. Atoms let you forget about the underlying values. Your eye colors can simply be blue, brown, green, or other. These colors can be used anywhere in any piece of code. The underlying values will never clash, and it is impossible for such a constant to be undefined! (We'll see how to create constants with values associated with them in Chapter 2.)

Therefore, an atom is mainly useful to express or qualify data coupled with it, usually in a tuple (described in "Tuples" on page 16). Atoms are sometimes (but not often) useful when used alone. This is why we won't spend more time toying with them here. You'll see them coupled with other types of data in later examples.

DON'T DRINK TOO MUCH KOOL-AID

Atoms are really nice and a great way to send messages or represent constants. However, there are pitfalls to using atoms for too many things. An atom is referred to in an *atom table*, which consumes memory (4 bytes per atom in a 32-bit system and 8 bytes per atom in a 64-bit system). The atom table is not garbage collected, so atoms will accumulate until the system tips over, either from memory usage or because 1,048,577 atoms were declared.

This means atoms should not be generated dynamically. If your system needs to be reliable, and user input lets someone crash it at will by telling it to create atoms, you're in serious trouble.

Atoms should be seen as tools for the developer because, honestly, that's what they are. To reiterate: You should feel perfectly safe using atoms in your everyday code as long as you type them in yourself. It's only dynamic generation of atoms that is risky.

NOTE *Some atoms are reserved words and cannot be used except for what the language designers wanted them to be: function names, operators, expressions, and so on. These reserved words are as follows: `after`, `and`, `andalso`, `band`, `begin`, `bnot`, `bor`, `bsl`, `bsr`, `bxor`, `case`, `catch`, `cond`, `div`, `end`, `fun`, `if`, `let`, `not`, `of`, `or`, `orelse`, `query`, `receive`, `rem`, `try`, `when`, and `xor`.*

Boolean Algebra and Comparison Operators

We would be in pretty deep trouble if we couldn't tell the difference between what's small and big, or what's true and false. Like any other language, Erlang has ways to let you use Boolean operations and to compare items.

Boolean algebra is dirt simple:

```
1> true and false.
false
2> false or true.
true
3> true xor false.
true
4> not false.
true
5> not (true and true).
false
```

NOTE *The Boolean operators `and` and `or` will always evaluate arguments on both sides of the operator. If you want a short-circuit operator (which will evaluate the right-side argument only if necessary), use `andalso` or `orelse`.*

Testing for equality or inequality is also dirt simple, but involves slightly different symbols from those you see in many other languages:

```
6> 5 =:= 5.
true
7> 1 =:= 0.
false
8> 1 =/= 0.
true
9> 5 =:= 5.0.
false
10> 5 == 5.0.
true
11> 5 /= 5.0.
false
```

There's a good chance that your usual language uses `==` and `!=` to test for and against equality, but Erlang uses `=:=` and `=/=`. The three last expressions (lines 9 through 11) also introduce a pitfall: Erlang doesn't care about the difference between floats and integers in arithmetic, but does distinguish between the two when comparing them. No worry though, because the `==` and `/=` operators are there to help you in these cases. Thus, it is important to consider whether or not you want exact equality. As a general rule of thumb, you should always start by using `=:=` and `=/=`, and switch to `==` and `/=` only when you know you do not need exact equality. This could help you avoid some unfortunate comparisons when the types of numbers you expect are not what you get.

Other operators for comparisons are `<` (less than), `>` (greater than), `>=` (greater than or equal to), and `=<` (less than or equal to). That last one is backward (in my opinion) and is the source of many syntax errors in my code. Keep an eye on that `=<`.

```
12> 1 < 2.
true
13> 1 < 1.
false
14> 1 >= 1.
true
15> 1 =< 1.
true
```

What happens when you enter something like `5 + llama` or `5 =:= true`? There's no better way to know than trying it and subsequently getting scared by error messages!

```
12> 5 + llama.
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 5 + llama
```

Erlang doesn't really like you misusing some of its fundamental types. The emulator returns an error message here, indicating it doesn't like one of the two arguments used around the `+` operator.

Erlang doesn't always get mad at you for using the wrong types though:

```
13> 5 =:= true.
false
```

Why does it refuse different types in some operations but not others? While Erlang doesn't let you *add* two operands of different types, it will let you *compare* them. This is because the creators of Erlang thought pragmatism beats theory and decided it would be great to be able to simply write things like general sorting algorithms that could order any terms. It's there to make your life simpler and can do so the vast majority of the time.

There is one last thing to keep in mind when doing Boolean algebra and comparisons:

```
14> 0 == false.
false
15> 1 < false.
true
```

Chances are you're pulling out your hair if you come from procedural languages or most object-oriented languages. Line 14 should evaluate to `true` and line 15 to `false`! After all, `false` means `0` and `true` is anything else! Except in Erlang. Because I lied to you. Yes, I did that. Shame on me.

Erlang has no such things as Boolean `true` and `false`. The terms `true` and `false` are atoms, but they are integrated well enough into the language that you shouldn't have a problem with them, as long as you don't expect `false` and `true` to mean anything but false and true.

NOTE *The correct ordering of each element in a comparison is the following:* `number < atom < reference < fun < port < pid < tuple < list < bit string`*. Some of these types won't be familiar to you, but you will get to know them through this book. Just remember that this is why you can compare anything with anything. To quote Joe Armstrong, one of the creators of Erlang, "The actual order is not important—but that a total ordering is well defined is important."*

Tuples

A *tuple* is a way to group together a set number of terms. In Erlang, a tuple is written in the form `{Element1, Element2, ..., ElementN}`. For example, you would give me the coordinates (*x*,*y*) if you wanted to tell me the position of a point in a Cartesian graph. We can represent this point as a tuple of two terms:

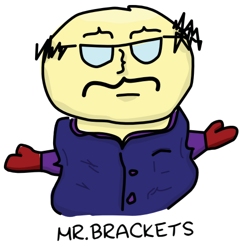
```
1> X = 10, Y = 4.
4
2> Point = {X,Y}.
{10,4}
```

In this case, a point will always be two terms. Instead of carrying the variables `X` and `Y` everywhere, you need to carry only one. However, what can you do if you receive a point and only want the *x*-coordinate? It's not hard to extract that information. Remember that when you assign values, Erlang will never complain if they are the same. Let's exploit that. (You may need to clear the variables we just set with `f()` before typing in the following example.)

```
3> Point = {4,5}.
{4,5}
4> {X,Y} = Point.
{4,5}
5> X.
4
6> {X,_} = Point.
{4,5}
```

From now on, we can use `X` to get the first value of the tuple. How did that happen? First, `X` and `Y` had no value and were thus considered unbound variables. When you set them in the tuple `{X,Y}` on the left side of the `=` operator, the `=` operator compares both values: `{X,Y}` versus `{4,5}`. Erlang is smart enough to unpack the values from the tuple and distribute them to the unbound variables on the left side. Then the comparison is only `{4,5} = {4,5}`, which obviously succeeds. That's one of the many forms of pattern matching.

Note that line 6 uses the don't care variable (`_`). This is exactly how it's meant to be used: to drop the value that would usually be placed there, since we won't use that value. The `_` variable is always seen as unbound and acts as a wildcard for pattern matching. Pattern matching to unpack tuples will work only if the number of elements (the tuple's length) is the same.



```
7> {_,_} = {4,5}.
{4,5}
8> {_,_} = {4,5,6}.
** exception error: no match of right hand side value {4,5,6}
```

Tuples can also be useful when working with single values. For example, suppose that we want to store the following temperature:

```
9> Temperature = 23.213.
23.213
```

Looks like a good day to go to the beach! But wait—is this temperature in Kelvin, Celsius, or Fahrenheit? We can use a tuple to store the temperature's units along with its value:

```
10> PreciseTemperature = {celsius, 23.213}.
{celsius,23.213}
11> {kelvin, T} = PreciseTemperature.
** exception error: no match of right hand side value {celsius,23.213}
```

This raises an exception, but that's exactly what we want. This is, again, pattern matching at work. The `=` operator compares `{kelvin, T}` and `{celsius, 23.213}`, and even if the variable `T` is unbound, Erlang can see that the `celsius` atom is different from the `kelvin` atom. An exception is raised, which stops the execution of code. So, the part of the program that expects a temperature in Kelvin won't be able to process temperatures sent in Celsius. This makes it easier for the programmer to know what kind of data is being sent, and it also works as a debugging aid.

A tuple that contains an atom with one element following it is called a *tagged tuple*. Any element of a tuple can be of any type, even another tuple:

```
12> {point, {X,Y}}.
{point,{4,5}}
```

But what if we want to carry around more than one point? For that, we have *lists*.

Lists

Lists are the bread and butter of many functional languages. They're used to solve all kinds of problems and are undoubtedly the most-used data structure in Erlang. Lists can contain anything—numbers, atoms, tuples, other lists—your wildest dreams in a single structure.

The basic notation of a list is [*Element1, Element2, ..., ElementN*], and you can mix more than one type of data in it:

```
1> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

Simple enough, right? Let's try another one:

```
2> [97, 98, 99].
"abc"
```

Uh-oh! This is one of the most disliked things in Erlang: strings. Strings are lists, and the notation is exactly the same. Why do people dislike it? Because of this:

```
3> [97,98,99,4,5,6].
[97,98,99,4,5,6]
4> [233].
"é"
```

Erlang will print lists of numbers as numbers only when at least one of them could not also represent a letter. There is no such thing as a real string in Erlang! This will no doubt come to haunt you in the future, and you'll hate the language for it. Don't despair, because there are other ways to write strings, as you'll see in "Binary Strings" on page 27.

DON'T DRINK TOO MUCH KOOL-AID

This is why some programmers say that Erlang sucks at string manipulation: It does not have a built-in string type as in most other languages. The lack is due to Erlang's origins as a language created and used by telecom companies. They never (or rarely) used strings, so they were not officially added to the language as their own data type. However, this problem is getting fixed with time. The virtual machine (VM) now partially supports Unicode and is getting faster at string manipulations all the time. There is also a way to store strings as a binary data structure, making them really light and faster to work with. We'll discuss this in "Binary Strings" on page 27.

All in all, some functions are still missing from the standard library when it comes to strings. While string processing is definitely doable in Erlang, other languages, such as Perl and Python, are better suited for tasks that need a lot of it.

To glue lists together, use the ++ operator. To remove elements from a list, use --.

```
5> [1,2,3] ++ [4,5].
[1,2,3,4,5]
6> [1,2,3,4,5] -- [1,2,3].
[4,5]
7> [2,4,2] -- [2,4].
[2]
8> [2,4,2] -- [2,4,2].
[]
```

Both ++ and -- are right-associative. This means the elements of many -- or ++ operations will be done from right to left, as in the following examples:

```
9> [1,2,3] -- [1,2] -- [3].
[3]
10> [1,2,3] -- [1,2] -- [2].
[2,3]
```

In the first example, proceeding from right to left, we first remove [3] from [1,2], leaving us with [1,2]. Then we remove [1,2] from [1,2,3], leaving us with only [3]. For the last one, we first remove [2] from [1,2], giving [1]. Then we take [1] out of [1,2,3], producing the final result [2,3].

Let's keep going. The first element of a list is named the *head*, and the rest of the list is named the *tail*. We will use two built-in functions (BIFs) to get them:

```
11> hd([1,2,3,4]).
1
12> tl([1,2,3,4]).
[2,3,4]
```

NOTE *BIFs are usually functions that could not be implemented in pure Erlang, and as such are defined in C, or whichever language Erlang happens to be implemented in (it was Prolog in the 1980s). There are still some BIFs that could be done in Erlang but were implemented in C in order to provide more speed to common operations. One example of this is the* `length(List)` *function, which will return the (you've guessed it) length of the list passed in as the argument.*

Accessing or adding the head is fast and efficient. Virtually all applications where you need to deal with lists will operate on the head first. As it's used so frequently, Erlang provides an easier way to separate the head from the tail of a list, with the help of pattern matching: `[Head|Tail]`. For example, here's how you would add a new head to a list:

```
13> List = [2,3,4].
[2,3,4]
14> NewList = [1|List].
[1,2,3,4]
```

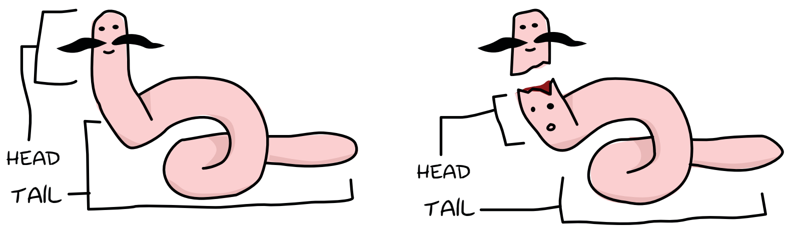
When processing lists, it's also helpful to have a quick way to store the tail, so you can operate on the tail later. If you remember the way tuples work and how we used pattern matching to unpack the values of a point (`{X,Y}`), you'll understand how we can get the first element (the head) sliced off a list in a similar manner:

```
15> [Head|Tail] = NewList.
[1,2,3,4]
16> Head.
1
17> Tail.
[2,3,4]
18> [NewHead|NewTail] = Tail.
[2,3,4]
19> NewHead.
2
```

The | we used is called the *cons* operator (constructor). In fact, any list can be built with only cons operators and values:

```
20> [1 | []].
[1]
21> [2 | [1 | []]].
[2,1]
22> [3 | [2 | [1 | []]]].
[3,2,1]
```

In other words, any list can be built with the following formula: `[Term1 | [Term2 | [... | [TermN]]]]`. Thus, you can define lists recursively as a head preceding a tail, which is itself a head followed by more heads. In this sense, you could imagine a list being a bit like an earthworm; you can slice it in half, and you'll then have two worms.



The ways Erlang lists can be built are sometimes confusing to people who are not used to similar constructors. To help you get familiar with the concept, read all of these examples (hint: they're all equivalent):

```
[a, b, c, d]
[a, b, c, d | []]
[a, b | [c, d]]
[a, b | [c | [d]]]
[a | [b | [c | [d]]]]
[a | [b | [c | [d | []]]]]
```

With this understood, you should be able to deal with list comprehensions, which are discussed in the next section.

NOTE *Using the form* `[1 | 2]` *gives what is called an* improper list. *Improper lists will work when you pattern match in the* `[Head|Tail]` *manner, but will fail when used with standard functions of Erlang (even* `length()`*). This is because Erlang expects* proper lists. *Proper lists end with an empty list as their last cell. When declaring an item like* `[2]`*, the list is automatically formed in a proper manner. As such,* `[1|[2]]` *would work. Improper lists, although syntactically valid, are of very limited use outside of user-defined data structures.*

List Comprehensions

List comprehensions are ways to build or modify lists. They also make programs short and easy to understand compared to other ways of manipulating lists. They may be hard to grasp at first, but they're worth the effort. Don't hesitate to try the examples in this section until you understand them!

List comprehensions are based on the mathematical idea of *set notation*, so if you've ever taken a math class that dealt with set theory, list comprehensions may look familiar to you. Set notation describes how to build a set by specifying properties its members must satisfy. For instance, here's a basic example: $\{x \in \mathbb{R} : x = x^2\}$. This describes the set of all real numbers that are equal to their own square. (The result of that set would be `{0,1}`.) A simpler example of set notation is $\{x : x > 0\}$. This describes the set of all numbers greater than zero.

Like set notation, list comprehensions are about building sets from other sets. For example, given the set $\{2n : n \in L\}$, where L is the list

[1,2,3,4], we could read this as "for all n values in [1,2,3,4], give me n*2." The set built from this would be [2,4,6,8]. The Erlang implementation of this same set is as follows:

```
1> [2*N || N <- [1,2,3,4]].
[2,4,6,8]
```

Compare the mathematical notation to the Erlang one, and you'll see that not a lot changes: brackets ({}) become square brackets ([]), the colon (:) becomes two pipes (||), and the operator ∈ becomes the arrow (<-). In other words, we change symbols but keep the same logic. In the example, each value of [1,2,3,4] is sequentially pattern matched to N. The arrow acts exactly like the = operator, with the exception that it doesn't throw exceptions.

You can also add constraints to a list comprehension by using operations that return Boolean values. So if you want all the even numbers from one to ten, you could write something like this:

```
2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].
[2,4,6,8,10]
```

Here, X rem 2 =:= 0 checks if a number is even.

The recipe for list comprehensions in Erlang is as follows:

```
NewList = [Expression || Pattern <- List, Condition1, Condition2, ... ConditionN]
```

The Pattern <- List part is called a *generator expression.*

List comprehensions are useful when you want to apply a function to each element of a list, forcing it to respect constraints. For example, say you own a restaurant. A customer enters, sees your menu, and asks if he could have the prices of all the items costing between $3 and $10, with taxes (say 7 percent) counted in afterward.

```
3> RestaurantMenu = [{steak, 5.99}, {beer, 3.99}, {poutine, 3.50}, {kitten, 20.99}, {water, 0.00}].
[{steak,5.99},
{beer,3.99},
{poutine,3.5},
{kitten,20.99},
{water,0.0}]
4> [{Item, Price*1.07} || {Item,Price} <- RestaurantMenu, Price >= 3, Price =< 10].
[{steak,6.409300000000001},{beer,4.2693},{poutine,3.745}]
```

The decimals are not rounded in a readable manner, but you get the point.

Another nice thing about list comprehensions is that you can have more than one generator expression, as in this example:

```
5> [X+Y || X <- [1,2], Y <- [3,4]].
[4,5,5,6]
```

This runs the operations 1+3, 1+4, 2+3, 2+4. So if you want to make the list comprehension recipe more generic, you get this:

```
NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN,
Condition1, Condition2, ... ConditionM]
```

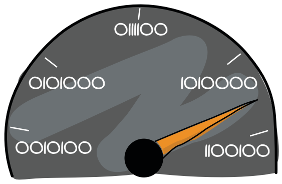
Note that the generator expressions coupled with pattern matching can also act as a filter:

```
6> Weather = [{toronto, rain}, {montreal, storms}, {london, fog},
6>            {paris, sun}, {boston, fog}, {vancouver, snow}].
[{toronto,rain},
 {montreal,storms},
 {london,fog},
 {paris,sun},
 {boston,fog},
 {vancouver,snow}]
7> FoggyPlaces = [X || {X, fog} <- Weather].
[london,boston]
```

If an element of the list Weather doesn't match the {X, fog} pattern, it's simply ignored in the list comprehension, whereas the = operator would have raised an exception.

We'll look at using one more basic data type in this chapter. It is a surprising feature that makes interpreting binary data easy as pie.

Working with Binary Data



Unlike most other languages, Erlang provides useful abstractions when dealing with binary values with pattern matching, instead of requiring the old-fashioned bit twiddling with special operators. It makes dealing with raw binary data fun and easy (no, really), which was necessary for the telecom applications it was created to help with. Bit manipulation has a unique syntax and idioms that may look kind of weird at first, but if you know how bits and bytes work generally, this should make sense to you. (You may want to skip the rest of this chapter if you're not familiar with binary operations.)

Bit Syntax

Erlang bit syntax encloses binary data between << and >> and splits it in readable segments; each segment is separated by a comma. A segment is a sequence of bits of a binary (not necessarily on a byte boundary, although this is the default behavior).

Suppose you want to store an orange pixel of true color (24 bits). If you've ever checked colors in Photoshop or in a CSS style sheet for the Web, you know the hexadecimal notation has the format *#RRGGBB*. A tint of orange is `#F09A29` in that notation, which could be expanded in Erlang to the following:

```
1> Color = 16#F09A29.
15768105
2> Pixel = <<Color:24>>.
<<240,154,41>>
```

This basically says, "Put the binary values of `#F09A29` on 24 bits of space (red on 8 bits, green on 8 bits, and blue also on 8 bits) in the variable `Pixel`." That value can then be written to a file or a socket later. This may not look like much, but once written to a file, this value will turn into a bunch of unreadable characters that, in the proper context, can be decoded as a picture.

This syntax is especially nice because you can use clean, readable text to write things that need to look messy to the naked eye in order to work. Without good abstractions, your code would also need to be messy. Even better: When you read the file back in, Erlang will interpret the binary value into the nice `<<240,151,41>>` format again! You can jump back and forth between representations, using only the one that's the most useful to you when you need it.

What's more interesting is the ability to pattern match with binaries to unpack content:

```
3> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
4> <<Pix1,Pix2,Pix3,Pix4>> = Pixels.
** exception error: no match of right hand side value <<213,45,132,64,76,32,76,0,0,234,32,15>>
5> <<Pix1:24, Pix2:24, Pix3:24, Pix4:24>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

On line 3, we declare what would be precisely 4 pixels of RGB colors in binary. On line 4, we tried to unpack four values from the binary content. It throws an exception, because we have more than 4 segments—in fact, we have 12. So we tell Erlang that each variable on the left side will hold 24 bits of data using `Pix1:24`, `Pix2:24`, and so on. We can then take the first pixel and unpack it further into single color values:

```
6> <<R:8, G:8, B:8>> = <<Pix1:24>>.
<<213,45,132>>
7> R.
213
```

"Yeah, that's dandy. But what if I only want the first color from the start? Will I need to unpack all these values all the time?" Don't worry—Erlang introduces more syntactic sugar and pattern matching to help you out:

```
8> <<R:8, Rest/binary>> = Pixels.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
9> R.
213
```

In this example, `Rest/binary` is a specific notation that lets you say that whatever is left in the binary, whatever length it is, is put into the `Rest` variable. So `<<`*`Pattern`*`, Rest/binary>>` is to binary pattern matching what `[`*`Head`*`|`*`Tail`*`]` is to list pattern matching.

Nice, huh? This works because Erlang allows more than one way to describe a binary segment. The following are all valid:

```
Value
Value:Size
Value/TypeSpecifierList
Value:Size/TypeSpecifierList
```

Here, *`Size`* is always in bits when no *`TypeSpecifierList`* is defined. *`TypeSpecifierList`* represents one or more of the following, separated by a hyphen (-):

Type

The possible values are `integer`, `float`, `binary`, `bytes`, `bitstring`, `bits`, `utf8`, `utf16`, and `utf32`. When no type is specified, Erlang assumes an `integer` type.

This represents the kind of binary data used. Note that `bytes` is shorthand for `binary`, and `bits` is shorthand for `bitstring`.

Signedness

The possible values are `signed` and `unsigned`. The default is `unsigned`.

This only matters for matching when the type is `integer`.

Endianness

The possible values are `big`, `little`, and `native`. By default, endianness is set to `big`, as it is the standard used in network protocol encodings.

Endianness only matters when the type is `integer`, `utf16`, `utf32`, or `float`. This has to do with how the system reads binary data. For example, the BMP image header format holds the size of its file as an integer stored in 4 bytes. For a file that has a size of 72 bytes, a little-endian system would represent this as `<<72,0,0,0>>`, and a big-endian system would represent it as `<<0,0,0,72>>`. The former will be read as 72, while the latter will be read as 1207959552, so make sure you use the correct endianness.

There is also the option to use `native`, which will choose at runtime if the CPU uses little-endianness or big-endianness natively.

Unit

This is written as `unit:Integer`.

The unit is the size of each segment. The allowed range is 1 to 256. It is set by default to 1 bit for `integer`, `float`, and `bitstring` types, and to 8 bits for `binary`. The `utf8`, `utf16`, and `utf32` types do not require a unit to be defined. The multiplication of size by unit is equal to the number of bits the segment will take, and must be evenly divisible by 8. The unit size is usually used to ensure byte alignment.

The default size of a data type can be changed by combining different parts of a binary. As an example, `<<25:4/unit:8>>` will encode the number 25 as a 4-byte integer, or `<<0,0,0,25>>` in its graphical representation. `<<25:2/unit:16>>` will give the same result, and so will `<<25:1/unit:32>>`. Erlang will generally accept `<<25:`*`Size`*`/unit:`*`Unit`*`>>` and multiply *`Size`* by *`Unit`* to figure out how much space it should take to represent the value. Again, the result of this should be divisible by 8.

Some examples may help you digest these definitions:

```
10> <<X1/unsigned>> =  <<-44>>.
<<"Ô">>
11> X1.
212
12> <<X2/signed>> =  <<-44>>.
<<"Ô">>
13> X2.
-44
14> <<X2/integer-signed-little>> =  <<-44>>.
<<"Ô">>
15> X2.
-44
16> <<N:8/unit:1>> = <<72>>.
<<"H">>
17> N.
72
18> <<N/integer>> = <<72>>.
<<"H">>
19> <<Y:4/little-unit:8>> = <<72,0,0,0>>.
<<72,0,0,0>>
20> Y.
72
```

You can see that there is more than one way to read, store, and interpret binary data. This is a bit confusing, but still much simpler than using the usual tools given by most languages.

Bitwise Binary Operations

The standard binary operations (shifting bits to left and right, and binary `and`, `or`, `xor`, and `not`) also exist in Erlang. Just use the operators `bsl` (bit shift left), `bsr` (bit shift right), `band`, `bor`, `bxor`, and `bnot`.

```
2#00100 = 2#00010 bsl 1.
2#00001 = 2#00010 bsr 1.
2#10101 = 2#10001 bor 2#00101.
```

With this notation and bit syntax in general, parsing and pattern matching binary data are a piece of cake. For example, you could parse TCP segments with code like this:

```
<<SourcePort:16, DestinationPort:16,AckNumber:32,
DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
CheckSum: 16, UrgentPointer:16,
Payload/binary>> = SomeBinary.
```

If `SomeBinary` does contain a TCP segment from some networking code, it can be extracted with a similar pattern. All values are in bits (except for the `Payload`, which is of arbitrary length), and well defined by a standard. Whichever part of the segment your program needs can then be referred to by its corresponding variable.

The same logic can then be applied to anything binary: video encoding, images, other protocol implementations, and so on.

DON'T DRINK TOO MUCH KOOL-AID

Erlang can be slow compared to languages like C or C++. Unless you are a patient person (or a prodigy), it would likely be a bad idea to do stuff like converting videos or images with it, even though the binary syntax makes it extremely interesting. Erlang is traditionally just not that great at heavy number-crunching.

Take note, however, that Erlang is usually mighty fast for applications that do not require number-crunching, such as reacting to events, message-passing (with the help of atoms being extremely light), and so on. It can deal with events in matters of milliseconds, and as such, is a great candidate for soft real-time applications.

Binary Strings

There's a whole other aspect to binary notation: *binary strings*. Binary strings are bolted on top of the language in the same way strings are with lists, but they're much more efficient in terms of space. This is because normal lists are similar to linked lists (one "node" per letter, and then a reference to the next part of the list), while binary strings are more like C arrays (a tightly packed block of memory).

Binary strings use the syntax <<`"this is a binary string!"`>>. The downside of binary strings compared to lists is a loss in simplicity when it comes to pattern matching and manipulation. Consequently, people tend to use binary strings when storing text that won't be manipulated too much or when space efficiency is a real issue.

NOTE *Even though binary strings are pretty light, you should avoid using them to tag values. It might be tempting to use string literals to say, for example, {<<"temperature">>,50}, but you should always use atoms in that case. Using atoms results in almost no overhead when comparing different values, and such comparisons are done in constant time regardless of length, while binaries are compared in linear time. Conversely, do not use atoms to replace strings because they are lighter. Strings can be manipulated (splitting, regular expressions, and so on), while atoms can only be compared and nothing else.*

Binary Comprehensions

Binary comprehensions are to bit syntax what list comprehensions are to lists: a way to make code short and concise when dealing with binaries. They can generally be used in the same manner as list comprehensions:

```
1> << <<X>> || <<X>> <= <<1,2,3,4,5>>, X rem 2 == 0>>.
<<2,4>>
```

The only change in syntax from regular list comprehensions is the <-, which becomes <= for binary generators, and using binaries (<<>>) instead of lists ([]).

Earlier in this chapter, you saw an example of using pattern matching to grab RGB values from a binary value that represented many pixels. That technique worked well in that example, but on larger structures, it could become harder to read and maintain. The same exercise can be done with a one-line binary comprehension, which is much cleaner:

```
2> Pixels = <<213,45,132,64,76,32,76,0,0,234,32,15>>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
3> RGB = [ {R,G,B} || <<R:8,G:8,B:8>> <= Pixels ].
[{213,45,132},{64,76,32},{76,0,0},{234,32,15}]
```

Changing <- to <= lets you use a binary as a generator. The complete binary comprehension basically changed binary data to integers inside tuples. Another binary comprehension syntax exists to let you do the exact opposite:

```
4> << <<R:8, G:8, B:8>> ||  {R,G,B} <- RGB >>.
<<213,45,132,64,76,32,76,0,0,234,32,15>>
```

Be careful, as the elements of the resulting binary require a clearly defined binary type if the generator returned binaries:

```
5> << <<Bin>> || Bin <- [<<3,7,5,4,7>>] >>.
** exception error: bad argument
6> << <<Bin/binary>> || Bin <- [<<3,7,5,4,7>>] >>.
<<3,7,5,4,7>>
```

By default, Erlang assumes that values you try to put into or extract from a binary are integers (unsigned, on 8 bits). When writing `<<Bin>>`, we're in fact declaring that we want a binary containing an integer that is stored in the variable `Bin`. The problem is that `Bin` holds another binary, and that just doesn't make sense to Erlang. We said we would give an integer, and we gave a binary. By specifying that the type is `binary` (as on line 6), Erlang is able to deal with the pattern because what we say `Bin` is and what `Bin` contains now make sense.

It's also possible to have a binary comprehension with a binary generator:

```
7> << <<(X+1)/integer>> || <<X>> <= <<3,7,5,4,7>> >>.
<<4,8,6,5,8>>
```

Note that specifying the type as `integer` is superfluous in this case, as Erlang assumes integers by default.

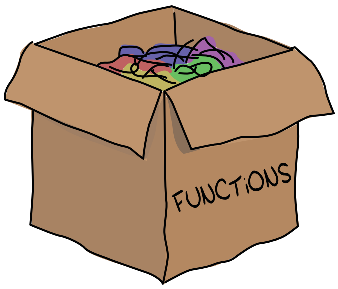
In this book, I won't go into much more detail on binaries and binary comprehensions. If you're interested in understanding more about bit syntax as a whole, you can read the white paper that defines their specification, at *http://user.it.uu.se/~pergu/papers/erlang05.pdf*.

2

MODULES

Working with the interactive shell is a vital part of using dynamic programming languages. It's useful to test all kinds of code and programs. In Chapter 1, we used the interactive shell to play with most of Erlang's basic data types without ever opening a text editor or saving a file. While you could stop reading here, go play ball outside, and call it a day, that would make you a terrible Erlang programmer. Code needs to be saved somewhere to be used! As you'll learn in this chapter, that's what modules are for.

What Are Modules?



A *module* is a bunch of functions grouped together in a single file, under a single name. All functions in Erlang must be defined in modules. You have already used modules, perhaps without realizing it. The

BIFs mentioned in Chapter 1, such as hd and tl, actually belong to the erlang module. All of the arithmetic, logic, and Boolean operators also are in the erlang module.

BIFs from the erlang module differ from other functions, as they are automatically imported when you use Erlang. Every other function defined in a module needs to be called with the form *Module*:*Function*(*Arguments*), as in this example:

```
1> erlang:element(2, {a,b,c}).
b
2> element(2, {a,b,c}).
b
3> lists:seq(1,4).
[1,2,3,4]
4> seq(1,4).
** exception error: undefined shell command seq/2
```

Here, the seq function from the lists module was not automatically imported, while element was. The error "undefined shell command" comes from the shell looking for a shell command like f() and not being able to find it. Some functions from the erlang module are not imported automatically, but they are not used very frequently.

Logically, you should put functions that deal with similar things inside a single module. Common operations on lists are kept in the lists module, while functions to do input and output (such as writing to the terminal or in a file) are grouped in the io module or the file module. One of the only modules you will encounter that doesn't respect that pattern is the erlang module, which has functions that do math, perform conversions, deal with multiprocessing, fiddle with the VM's settings, and so on. They have nothing in common except being BIFs. You should avoid creating modules like erlang, and instead focus on clean and logical separations.

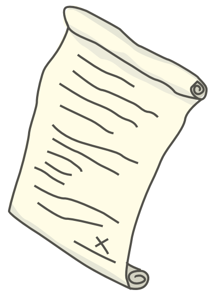
Creating Modules

When writing a module, you can declare two kinds of things: *functions* and *attributes*. Attributes are metadata describing the module itself, such as its name, the functions that should be visible to the outside world, the author of the code, and so on. This kind of metadata is useful because it gives hints to the compiler on how it should do its job, and also because it lets people retrieve information from compiled code without needing to consult the source.

A large variety of module attributes is currently used in Erlang code across the world. In fact, you can even declare your own attributes for whatever you please. However, some predefined attributes will appear more frequently than others in your code.

All module attributes follow the form `-Name(Attribute).`. Only one of them is necessary for your module to be compilable:

```
-module(Name).
```



This is always the first attribute (and statement) of a file, and for good reason: It's the name of the current module, where `Name` is an atom. This is the name you'll use to call functions from other modules. The calls are made with the form `M:F(A)`, where `M` is the module name, `F` the function, and `A` the arguments.

Note that the name of the module as defined in the `-module` attribute and the filename must match. For example, if the module name is `unimaginative_name`, then the file should be named *unimaginative_name.erl* (*.erl* is the standard Erlang source extension). If the names don't match, your module won't compile.

It's time to code already! Our first module will be very simple and useless. Open your text editor, type the following line, and then save the file as *useless.erl*.

```
-module(useless).
```

This line of text is actually a valid module. Really! Of course, it's useless without functions. Let's first decide which functions will be exported from our `useless` module. To do this, we will use another attribute:

```
-export([Function1/Arity, Function2/Arity, ..., FunctionN/Arity]).
```

This is used to define which functions of a module can be called by the outside world. It takes a list of functions with their respective arity. The *arity* of a function is an integer representing how many arguments can be passed to the function. This is critical information, because different functions defined within a module can share the same name if, and only if, they have a different arity. The functions `add(X,Y)` and `add(X,Y,Z)` would thus be considered different, and written in the form `add/2` and `add/3`, respectively.

NOTE *Exported functions represent a module's* interface. *It is important to define an interface that reveals only the bare minimum of what is necessary to use the module's functions. This lets you fiddle with the internal details of your implementations without breaking code that might depend on your module.*

Our `useless` module will first export a useful function named `add`, which will take two arguments. Add the following `-export` attribute after the module declaration:

```
-export([add/2]).
```

And now we can write the function:

```
add(A,B) ->
    A + B.
```

The syntax of a function follows the form `Name(Args) -> Body.`, where `Name` must be an atom, and `Body` can be one or more Erlang expressions separated by commas. The function is ended with a period. Note that Erlang doesn't use the `return` keyword as many imperative languages do. A `return` is useless! Instead, the last logical expression of a function to be executed will have its value returned to the caller automatically, without you needing to mention it.

Next, add the following function to the file. (Yes, every tutorial needs a "Hello, world" example!) Don't forget to add it to the `-export` attribute as well (the `-export` attribute should then look like `-export([add/2, hello/0]).`).

```
%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
    io:format("Hello, world!~n").
```

The first thing to notice in this listing is the comments. In Erlang, comments are single-line only and begin with a `%` sign. (In this case, we've used `%%`, but this is purely a question of style.) The `hello/0` function also demonstrates how to call functions from foreign modules inside your own module. In this case, `io:format/1` is the standard function to output text, as written in the comments.

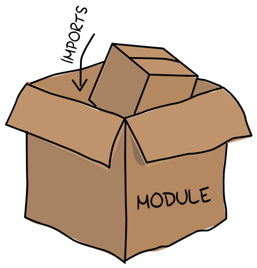
NOTE *The convention in the Erlang community is to use three percent signs (%%%) for comments that are general to a module (what the module is used for, licenses, and so on) and divisions of different sections of a module (public code, private code, helper functions, and so on). Two percent signs (%%) are used for all other comments that are alone on their own line and at the same level of indentation as the surrounding code. A single* `%` *is used for comments at the end of a line where there is code.*

Let's add one last function to the module, using both functions `add/2` and `hello/0`:

```
greet_and_add_two(X) ->
    hello(),
    add(X,2).
```

Again, don't forget to add `greet_and_add_two/1` to the exported function list. The calls to `hello/0` and `add/2` don't need to have the module name prepended to them, because they were declared in the module itself.

If you wanted to be able to call io:format/1 in the same manner as add/2, or any other function defined within the current module, you could have added the following module attribute at the beginning of the file: -import(io, [format/1]).. Then you could have called format("Hello, World!~n"). directly. More generally, the -import attribute follows this recipe:



```
-import(Module, [Function1/Arity, ..., FunctionN/Arity]).
```

Importing a function is a handy shortcut, although most programmers strongly discourage the use of the -import attribute, as it can reduce the readability of code. For example, in the case of io:format/2, there's another function in a different library with the same name: io_lib:format/2. Determining which one is used requires going to the top of the file to see from which module it was imported, if it was imported in the first place. Consequently, including the module name is considered good practice and will help the many Erlang users who love to use grep to find their way across projects. Usually, the only functions you'll see imported come from the lists module; its functions are used with a higher frequency than those from most other modules.

Your useless module should now look like the following:

```
-module(useless).
-export([add/2, hello/0, greet_and_add_two/1]).

add(A,B) ->
    A + B.

%% Shows greetings.
%% io:format/1 is the standard function used to output text.
hello() ->
    io:format("Hello, world!~n").

greet_and_add_two(X) ->
    hello(),
    add(X,2).
```

We are now finished with the useless module. Save your *useless.erl* file, and then we can try to compile it.

Compiling Code

Erlang code is compiled to bytecode so it can be used by the VM. You can call the compiler from many places. The most common way is to call it from the command line, like so:

```
$ erlc flags file.erl
```

When in the shell or in a module, you can compile it like this:

```
compile:file(Filename)
```

Another way, often used when developing code, is to compile from the shell:

```
c()
```

It's time to compile our `useless` module and try it out. But first we need to tell the Erlang shell where to find our module. Open the Erlang shell and type the following, filling in the full path where your file is saved.

```
1> cd("/path/to/where/you/saved/the-module/").
"Path Name to the directory you are in"
ok
```

By default, the shell will only look for files in the same directory it was started in and the standard library. The `cd/1` function is defined exclusively for the Erlang shell, telling it to change the directory to a new one, so it's less annoying to browse for files.

Next, enter the following:

```
2> c(useless).
{ok,useless}
```

If you get a different message—one that looks something like `useless.erl:`*`Line`*`: `*`Some Error Message`*—make sure the file is named correctly; that you are in the right directory; and that you've made no mistakes in your module, such as using unmatched parentheses, forgetting about full stops (`.`), and so on.

After you've successfully compiled your code, you'll notice that a *useless.beam* file has been added next to *useless.erl* in your working directory. This is the compiled module.

NOTE *The* .beam *filename extension stands for Bogdan/Björn's Erlang Abstract Machine, which is the VM itself. Other VMs for Erlang exist, but most are not used anymore. For example, Joe's Abstract Machine (JAM), inspired by Prolog's WAM and old BEAM, attempted to compile Erlang to C, and then to native code. Benchmarks demonstrated little benefit in this practice, and the concept was given up. More recently, there has been an effort to port Erlang to the JVM, giving the* Erjang *language. While the results are impressive, few developers have switched over to the Java platform for their Erlang development.*

Now let's try our first functions!

```
3> useless:add(7,2).
9
4> useless:hello().
Hello, world!
ok
5> useless:greet_and_add_two(-3).
Hello, world!
-1
6> useless:not_a_real_function().
** exception error: undefined function useless:not_a_real_function/0
```

The functions work as expected: `add/2` adds numbers, `hello/0` outputs `Hello, world!`, and `greet_and_add_two/1` does both. Of course, you might be asking why `hello/0` returns the atom `ok` after outputting text. This is because Erlang functions and expressions must *always* return something, even if they would not need to in other languages. As such, `io:format/1` returns `ok` to denote a normal condition: the absence of errors.

Line 6 shows an error being thrown because the function we tried to call doesn't exist in our module. If you forget to export a function, this is the kind of error message you will see when you try to call it.

Compiler Options

Erlang includes many compilation flags that can give you more control over how a module is compiled. You can get a list of all of them in the Erlang documentation. The following are the most common flags:

`-debug_info`
Erlang tools such as debuggers, code-coverage utilities, and static-analysis utilities will use the debug information of a module to do their work. In general, it is recommended to always turn on this option. You are more likely to need this option than the little bits of extra space you would save by not having it in your compiled code.

-{outdir,*Dir*}
By default, the Erlang compiler will create the *.beam* files in the current directory. This will let you choose where to put the compiled file.

-export_all
This flag causes the compiler to ignore the `-export` module attribute and instead export all functions defined. This is mainly useful when testing and developing new code, but should not be used in production.

-{d,*Macro*} or {d,*Macro*,*Value*}
This flag defines a macro to be used in the module, where *`Macro`* is an atom. This is most frequently used when unit testing, as it ensures that a module will have its testing functions created and exported only when they are explicitly wanted. By default, `Value` is `true` if it's not defined as the third element of the tuple.

To compile our `useless` module with some flags, we could do one of the following:

```
7> compile:file(useless, [debug_info, export_all]).
{ok,useless}
8> c(useless, [debug_info, export_all]).
{ok,useless}
```

You can also be sneaky and define compile flags from within a module, with a module attribute. To get the same results as from lines 7 and 8, you could add the following to the module:

```
-compile([debug_info, export_all]).
```

NOTE *Another option is to compile your Erlang module to native code. Native code compiling is* not *available for every platform and operating system, but on those that support this feature, it can make your programs go faster (about 20 percent faster, based on anecdotal evidence). To compile to native code, you need to use the* `hipe` *module and call it the following way:* `hipe:c(Module,OptionsList)`*. You could also use* `c(Module,[native]).` *when in the shell to achieve similar results. Note that the* .beam *file generated will no longer be portable across platforms. In general, compiling with* `hipe` *is seen as a last resort to get performance out of CPU-intensive operations.*

Defining Macros

Erlang *macros* are similar to C's `#define` statements, and are mainly used to define short functions and constants. They are simple expressions represented by text that will be replaced before the code is compiled for the VM. Such macros are mainly useful to avoid having "magic values" floating around your modules. For example, if you were to see code that compares some variable to a hard-coded number 3600, you'd have no idea if it represented 1 hour (3600 seconds), 60 hours (3600 minutes), some monetary amount, etc. However, if you encounter a value such as `?HOUR`, which is an Erlang macro, then you instantly have an idea of what you are dealing with. Even better, if you eventually switch your representation from seconds (3600) to, say, milliseconds (3,600,000), you need only change the macro definition in order to update all the instances of the macro in your code.

You can define such a macro as a module attribute in the following way:

```
-define(MACRO, some_value).
```

You can then use the macro as *`?MACRO`* inside any function defined in the module, and it will be replaced by *`some_value`* before the code is compiled. For the hour example above, we would define the macro as follows:

```
-define(HOUR, 3600). % in seconds
```

Defining a "function" macro is similar. Here's a simple macro used to subtract one number from another:

```
-define(sub(X,Y), X-Y).
```

To use this macro, simply call it in the same way that you would call any other macro. For example, if you called `?sub(23,47)`, this would be replaced with `23-47` by the compiler.

There are also a few predefined macros, such as the following:

- `?MODULE`, which is replaced by the current module name as an atom
- `?FILE`, which is replaced by the filename as a string
- `?LINE`, which returns the line number of wherever the macro is placed

You can also check whether particular macros are defined in your code and conditionally define other macros based on that result. To do this, use the attributes `-ifdef(`*`MACRO`*`).`, `-else.`, and `-endif.` as in this example:

```
-ifdef(DEBUGMODE).
-define(DEBUG(S), io:format("dbg: "++S)).
-else.
-define(DEBUG(S), ok).
-endif.
```

When used in code, the macro will look like `?DEBUG("entering some function")`, and will only output information if the module is compiled with a `DEBUGMODE` macro present. Otherwise, the atom `ok` is declared and does nothing at all.

As another example, you could also define tests to exist only if some test macro is first defined:

```
-ifdef(TEST).
my_test_function() ->
    run_some_tests().
-endif.
```

Then, using the compile flags mentioned previously, we can choose whether to define DEBUGMODE or TEST as c(*Module*, [{d,'TEST'},{d,'DEBUGMODE'}]).

More About Modules

Before we move on to writing more powerful functions and fewer useless snippets of code, we'll look at a few other miscellaneous bits of information about modules that might be useful to you in the future.

Metadata

As mentioned earlier in the chapter, module attributes are metadata describing properties of the module itself. Where can we find this metadata when we don't have an access to the source? Well, the compiler plays nice with us—when compiling a module, it will pick up most module attributes and store them (along with other information) in a module_info/0 function.

You can see the metadata of the useless module like this:

```
9> useless:module_info().
[{exports,[{add,2},
           {hello,0},
           {greet_and_add_two,1},
           {module_info,0},
           {module_info,1}]},
 {imports,[]},
 {attributes,[{vsn,[174839656007867314473085021121413256129]}]},
 {compile,[{options,[]},
 {version,"4.8"},
 {time,{2013,2,13,2,56,32}},
 {source,"/home/ferd/learn-you-some-erlang/useless.erl"}]}]
10> useless:module_info(attributes).
[{vsn,[174839656007867314473085021121413256129]}]
```

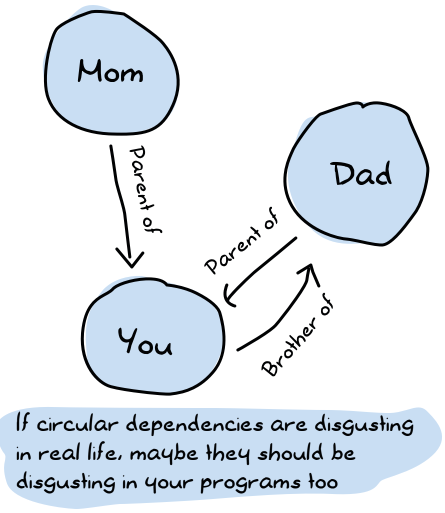
This snippet also shows an additional function, module_info/1, which will let you grab one specific piece of information. You can see exported functions, imported functions (none in this case), attributes (this is where your custom metadata would go), and compile options and information. Had you decided to add -author("An Erlang Champ"). to your module, it would have ended up in the same section as vsn.

NOTE *`vsn` is an automatically generated unique value that differentiates each version of your code, excluding comments. It is used in code hot-loading (upgrading an application while it runs, without stopping it) and by some tools related to release handling. You can also specify a `vsn` value yourself by adding `-vsn(VersionNumber)` to your module.*

There are limited uses for module attributes when it comes to production code, but they can be nice when doing little tricks to help yourself out. For example, I'm using them in my testing script for this book to annotate functions for which unit tests could be better. The script looks up module attributes, finds the annotated functions, and shows a warning about them. If you're interested in looking at this script, you can find it at *http://learnyousomeerlang.com/static/erlang/tester.erl*.

Circular Dependencies

Another point to keep in mind about module design is to avoid circular dependencies. A module `A` should not call a module `B` that also calls module `A`. Such dependencies usually end up making code maintenance difficult.


Mom
Dad
You
Parent of
Parent of
Brother of
If circular dependencies are disgusting in real life, maybe they should be disgusting in your programs too

In fact, code that depends on too many modules—even if they're not in a circular dependency—can make maintenance harder. The last thing you want is to wake up in the middle of the night only to find a maniac software engineer trying to gouge your eyes out because of terrible code you have written.

Well, that's enough of the pedantic moralizing. In Chapter 3, we'll continue our exploration of Erlang, focusing on functions.

3

SYNTAX IN FUNCTIONS

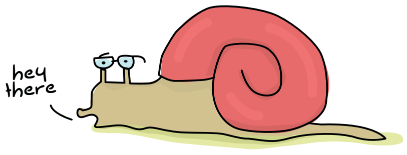
Now that we have the ability to store and compile our code, we can begin to write more advanced functions. The functions that we have written so far are extremely simple and a bit underwhelming. Now let's get to more interesting stuff. In this chapter, we'll work with functions that behave differently depending on the arguments passed to them and expressions that let us make decisions based on different conditions.

Pattern Matching

The first function we'll write will greet someone differently according to gender. To achieve this in most procedural languages, you would need to write something similar to the following pseudocode:

```
function greet(Gender,Name)
    if Gender == male then
        print("Hello, Mr. %s!", Name)
    else if Gender == female then
        print("Hello, Mrs. %s!", Name)
    else
```

```
        print("Hello, %s!", Name)
    end
```



Erlang can save you a whole lot of boilerplate code with pattern matching, which we used in Chapter 1. That chapter showed how we can compare and assign variables in structures like lists and tuples (remember patterns like {point,{X,Y}}).

Erlang lets us use similar patterns when defining functions. An Erlang version of the greet function looks like this:

```
greet(male, Name) ->
    io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
    io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
    io:format("Hello, ~s!", [Name]).
```

When we were in the shell and a given pattern could not be matched, Erlang would throw a fit and yell at us with an error message. When a pattern fails in a function (such as greet(male, Name)), Erlang just looks for the next part of the function with a different pattern (here, it would be greet(female, Name)) and runs that one if it matches.

The main difference between the two versions of greet is that in Erlang, we use pattern matching to define which parts of a function should be used and bind the values we need at the same time. There is no need to first bind the values and then compare them. So instead of this form:

```
function(Args)
    if X then
        Expression
    else if Y then
        Expression
    else
        Expression
```

we write this:

```
function(X) ->
    Expression;
function(Y) ->
    Expression;
function(_) ->
    Expression.
```

This allows us to get similar results, but in a much more declarative style.

Each of these function declarations is called a *function clause*. Function clauses must be separated by semicolons (;) and together form a *function* *declaration*. A function declaration counts as one larger statement, which is why the final function clause ends with a period. It's a strange use of tokens to determine workflow, but you'll get used to it. At least you'd better hope so, because there's no way out of it!

FORMATTING WITH IO:FORMAT

io:format's formatting is done with the help of tokens being replaced in a string. The tilde (~) character is used to denote a token. Some tokens are built in, such as ~n, which will be changed to a line break. Most other tokens denote a way to format data. For example, the function call io:format("~s!~n",["Hello"]). includes the token ~s, which accepts strings and binary strings as arguments. The final output message would be "Hello!\n". Another widely used token is ~p, which will print an Erlang term in the same way terms are output for you by the Erlang shell (adding indentation and everything).

We'll pick more uses of io:format as we go, but in the meantime, you can try the following calls to see what they do:

```
io:format("~s~n",[<<"Hello">>])
io:format("~p~n",[<<"Hello">>])
io:format("~~~n")
io:format("~f~n", [4.0])
io:format("~30f~n", [4.0])
```

This is just a small sample of io:format's possibilities. You can read the online documentation to find out more.

Fancier Patterns

Pattern matching in functions can be quite complex and powerful. As you may remember from Chapter 1, we can pattern match on lists to get their heads and tails. Let's do that!

Start a new module called functions:

```
-module(functions).
-compile(export_all). % Replace with -export() later, for sanity's sake!
```

In this module, we'll write a bunch of functions to explore many of the available pattern-matching avenues. The first function we'll write is head/1, which will act exactly like erlang:hd/1: It will take a list as an argument and return its first element. We'll do this with the help of the cons operator (|) and the "don't care" variable (_):

```
head([H|_]) -> H.
```

If you type functions:head([1,2,3,4]). in the shell (once the module is compiled), you can expect the value 1 to be returned. Consequently, to get the second element of a list, you would create this function:

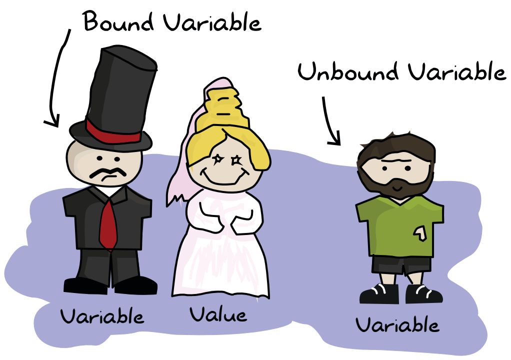
```
second([_,X|_]) -> X.
```

Erlang will be smart enough to look inside the list and fetch what it needs in order for the pattern match to succeed. Try it in the shell:

```
1> c(functions).
{ok, functions}
2> functions:head([1,2,3,4]).
1
3> functions:second([1,2,3,4]).
2
```

Retrieving values with pattern matching could be done for lists as long as you want, although it would be impractical to do it up to thousands of values. The smarter way to accomplish this is to use recursive functions, which are covered in Chapter 5. For now, let's concentrate on more pattern matching.

Variables in a Bind



The concept of free and bound variables discussed in Chapter 1 still holds true for functions. Let's review bound and unbound variables, using a wedding scenario.

Here, the bridegroom is sad because in Erlang, variables can never change value—no freedom! On the other hand, unbound variables don't have any values attached to them (like our little bum on the right). Binding a variable is simply attaching a value to an unbound variable. In the case of Erlang, when you want to assign a value to a variable that is already bound, an error occurs *unless the new value is the same as the old one*. Let's imagine our guy on the left has married one of two twins. If the second twin comes around, he won't differentiate them and will act normally. If a different woman comes around, he'll complain. (You can review "Invariable Variables" on page 11 if this concept is not clear to you.)

Using pattern matching and functions, we can compare and know if two parameters passed to a function are the same. For this, we'll create a function named `same/2` that takes two arguments and tells if they're identical:

```
same(X,X) ->
    true;
same(_,_) ->
    false.
```

And it's that simple.

When you call `same(a,a)`, the first `X` is seen as unbound; it automatically takes the value `a`. Then when Erlang goes to the second argument, it sees `X` is already bound. Erlang then compares the value to the `a` passed as the second argument and checks if it matches. The pattern matching succeeds, and the function returns `true`. If the two values aren't the same, pattern matching will fail and go to the second function clause, which doesn't care about its arguments (when you're the last to choose, you can't be picky!) and will instead return `false`. Note that this function can effectively take any kind of argument whatsoever. It works for any type of data, not just lists or single variables.

Now let's look at a more advanced example. The following function prints a date, but only if it is formatted correctly.

```
valid_time({Date = {Y,M,D}, Time = {H,Min,S}}) ->
    io:format("The Date tuple (~p) says today is: ~p/~p/~p,~n",[Date,Y,M,D]),
    io:format("The time tuple (~p) indicates: ~p:~p:~p.~n", [Time,H,Min,S]);
valid_time(_) ->
    io:format("Stop feeding me wrong data!~n").
```

Note that it is possible to use the `=` operator in the function head, allowing us to match both the content inside a tuple (`{Y,M,D}`) and the tuple as a whole (`Date`). We can test the function like this:

```
4> c(functions).
{ok, functions}
5> functions:valid_time({{2013,12,12},{09,04,43}}).
The Date tuple ({2013,9,6}) says today is: 2013/9/6,
The time tuple ({9,4,43}) indicates: 9:4:43.
ok
6> functions:valid_time({{2013,09,06},{09,04}}).
Stop feeding me wrong data!
ok
```

There is a problem though. This function could take anything for values, even text or atoms, as long as the tuples are in the form `{{A,B,C},{D,E,F}}`. This is one of the limits of pattern matching. It can either specify really precise values, such as a known number or atom, or abstract values, such as the head or tail of a list, a tuple of `N` elements, or anything (_ and unbound variables). To solve this problem, we use guards.

Guards, Guards!

Guards are additional clauses that can go in a function's head to make pattern matching more expressive. As mentioned earlier, pattern matching is somewhat limited, as it cannot express things like a range of values or certain types of data.

One concept that cannot be represented with pattern matching is counting: Is this 12-year-old basketball player too short to play with the pros? Is this distance too long to walk on your hands? Are you too old or too young to drive a car? You couldn't answer these questions with simple pattern matching. You could represent the driving question in a very impractical way like this:

```
old_enough(0) -> false;
old_enough(1) -> false;
old_enough(2) -> false;
...
old_enough(14) -> false;
old_enough(15) -> false;
old_enough(_) -> true.
```

You can do that if you want, but you'll be alone to work on your code forever. If you want to eventually make friends, start a new `guards` module, and then type in the following solution to the driving question:

```
old_enough(X) when X >= 16 -> true;
old_enough(_) -> false.
```

And you're finished! As you can see, this is much shorter and cleaner than the previous version.

A basic rule for guard expressions is that they must return `true` to succeed. The guard will fail if it returns `false` or if it raises an exception.

Suppose we now forbid people who are over 104 years old from driving. Our valid ages for drivers are from 16 years old up to 104 years old. We need to take care of that, but how? Let's just add a second guard clause:

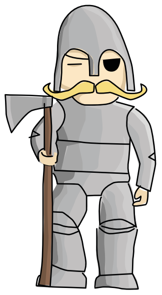
```
right_age(X) when X >= 16, X =< 104 ->
    true;
right_age(_) ->
    false.
```

In guard expressions, the comma (,) acts in a similar manner to the operator `andalso`, and the semicolon (;) acts a bit like `orelse` (described in Chapter 1). Because `right_age/1` uses the comma, both guard expressions need to succeed for the whole guard to pass. In fact, if you have any number of guards separated by commas, they all need to succeed for the entire guard to pass.

We could also represent the function in the opposite way:

```
wrong_age(X) when X < 16; X > 104 ->
    true;
wrong_age(_) ->
    false.
```

And we get correct results from this approach, too. Test it if you want (you should always test stuff!).



NOTE *I've compared `,` and `;` in guards to the operators `andalso` and `orelse`. They're not exactly the same, though. The former pair will catch exceptions as they happen, while the latter will not. What this means is that if there is an error thrown in the first part of the guard `X >= N; N >= 0`, the second part can still be evaluated, and the guard might succeed. If an error was thrown in the first part of `X >= N orelse N >= 0`, the second part will also be skipped, and the whole guard will fail. However (there is always a "however"), only `andalso` and `orelse` can be nested inside guards. This means `(A orelse B) andalso C` is a valid guard, while `(A; B), C` is not. Given their different use, the best strategy is often to mix them as necessary.*

In addition to using comparisons and Boolean evaluation in your guards, you can use math operations (for example, `A*B/C >= 0`) and functions about data types, such as `is_integer/1`, `is_atom/1`, and so on. (We'll talk more about these kinds of functions in Chapter 4.)

One negative point about guards is that they will not accept user-defined functions because of side effects. Erlang is not a purely functional programming language (like Haskell) because it relies on side effects a lot. You can do I/O, send messages between actors, or raise exceptions as you want and when you want. There is no trivial way to determine if a function you would use in a guard would print text or catch important errors every time it is tested over many function clauses. So instead, Erlang just doesn't trust you (and it may be right not to!).

That being said, you should now know enough to understand the basic syntax of guards and to understand them when you encounter them.

What the If?!

An `if` clause acts like a guard and shares the guard syntax, but outside a function clause's head. In fact, `if` clauses are called *guard patterns.*

Erlang's `if`s are different from the `if`s you'll ever encounter in most other languages. Compared to those other `if` clauses, Erlang's versions are weird creatures that might have been more accepted it they had a different name. When entering Erlang country, you should leave all you know about `if`s at the door.

To see how similar the if expression is to guards, enter the following examples in a module named what_the_if.erl:

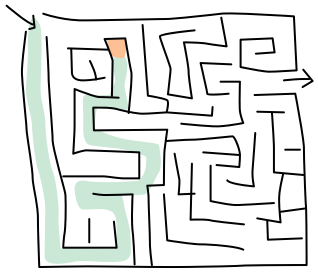
```
-module(what_the_if).
-export([heh_fine/0]).

heh_fine() ->
    if 1 =:= 1 ->
        works
    end,
    if 1 =:= 2; 1 =:= 1 ->
        works
    end,
  ❶ if 1 =:= 2, 1 =:= 1 ->
        fails
    end.
```

Save the module and let's try it:

```
1> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
2> what_the_if:heh_fine().
** exception error: no true branch found when evaluating an if expression
in function  what_the_if:heh_fine/0
```

Uh-oh! The compiler is warning us that no clause from the if at ❶ will ever match because its only guard evaluates to false. Remember that in Erlang, everything must return something, and if expressions are no exception to the rule. As such, when Erlang can't find a way to have a guard succeed, it will crash; it cannot *not* return something (this explains why the VM threw a "no true branch found" error when it got mad). We need to add a catchall branch that will always succeed no matter what. In most languages, this would be called an else. In Erlang, we use true, like this:



```
oh_god(N) ->
    if N =:= 2 -> might_succeed;
    true -> always_does  %% This is Erlang's if's 'else!'
end.
```

And now we can test this new function (the old one will keep spitting warnings; ignore them or take them as a reminder of what not to do):

```
3> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
4> what_the_if:oh_god(2).
might_succeed
5> what_the_if:oh_god(3).
always_does
```

Here's another function that shows how to use many guards in an `if` expression:

```
%% Note that this one would be better as a pattern match in function heads!
%% I'm doing it this way for the sake of the example.
help_me(Animal) ->
    Talk = if Animal == cat  -> "meow";
              Animal == beef -> "mooo";
              Animal == dog  -> "bark";
              Animal == tree -> "bark";
              true -> "fgdadfgna"
           end,
    {Animal, "says " ++ Talk ++ "!"}.
```

This function also demonstrates how any expression must return something. `Talk` has the result of the `if` expression bound to it, and is then concatenated in a string, inside a tuple. When reading the code, it's easy to see how the lack of a `true` branch would mess things up, considering Erlang has no such thing as a null value (such as Lisp's `nil`, C's `NULL`, and Python's `None`).

Let's try it:

```
6> c(what_the_if).
./what_the_if.erl:12: Warning: no clause will ever match
./what_the_if.erl:12: Warning: the guard for this clause evaluates to 'false'
{ok,what_the_if}
7> what_the_if:help_me(dog).
{dog,"says bark!"}
8> what_the_if:help_me("it hurts!").
{"it hurts!","says fgdadfgna!"}
```

You might be one of the many Erlang programmers wondering why `true` has taken over `else` as an atom to control flow—after all, `else` is much more familiar. Richard O'Keefe gave the following answer on the Erlang mailing lists, which I'll quote directly because I couldn't have put it better:

> It may be more FAMILIAR, but that doesn't mean 'else' is a good thing. I know that writing '; true ->' is a very easy way to get 'else' in Erlang, but we have a couple of decades of psychology-of-programming results to show that it's a bad idea. I have started to replace   by

```
if X > Y -> a()        if X > Y  -> a()
 ; true  -> b()         ; X =< Y -> b()
end                    end

if X > Y -> a()        if X > Y -> a()
 ; X < Y -> b()         ; X < Y -> b()
 ; true  -> c()         ; X ==Y -> c()
end                    end
```

> which I find mildly annoying when _writing_ the code but enormously helpful when _reading_ it.[1]

In other words, `else` or `true` branches should be avoided altogether. The `if` expressions are usually easier to read when you cover all logical ends, rather than relying on a catchall clause.

NOTE *All this horror expressed by the function names in* what_the_if.erl *is in regard to the* `if` *language construct when seen from the perspective of any other language's* `if`*. In Erlang, it turns out to be a perfectly logical construct with a confusing name.*

As mentioned earlier, only a limited set of functions can be used in guard expressions (we'll look at more of them in Chapter 4). This is where the real conditional powers of Erlang must be conjured. I present to you . . . the `case` expression!

In case ... of

If the `if` expression is like a guard, a `case ... of` expression is like the whole function head. You can have the complex pattern matching available for each argument of a function, and you can have guards, too.

For this example, we'll write the `insert` function for sets (a collection of unique values) that we will represent as an unordered list. This may be the

1. *http://erlang.org/pipermail/erlang-questions/2009-January/041229.html*

worst implementation possible in terms of efficiency, but what we want here is the syntax. Enter the following code in a file named cases.erl:

```
insert(X,[]) ->
    [X];
insert(X,Set) ->
    case lists:member(X,Set) of
        true  -> Set;
        false -> [X|Set]
    end.
```

If we send in an empty set (list) and a term X to be added, this code returns a list containing only X. Otherwise, the function lists:member/2 checks whether an element is part of a list, and returns true if it is or false if it is not. If we already have the element X in the set, we do not need to modify the list. Otherwise, we add X as the list's first element.

In this case, the pattern matching is really simple. However, it can get more complex, as in this example (still in the cases module):

```
beach(Temperature) ->
    case Temperature of
        {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
        {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically favorable';
        {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable in the US';
        _ ->
            'avoid beach'
    end.
```

Here, the answer to "Is it the right time to go to the beach?" is given in three different temperature systems: Celsius, Kelvin, and Fahrenheit. Pattern matching and guards are combined in order to return an answer satisfying all uses.

As pointed out earlier, case ... of expressions are pretty much the same thing as a bunch of function heads with guards. In fact, we could have written our code the following way:

```
beachf({celsius, N}) when N >= 20, N =< 45 ->
    'favorable';
...
beachf(_) ->
    'avoid beach'.
```

This raises the question of whether we should use if, case ... of, or functions for conditional expressions.

Which Should We Use?

Which of these three expressions—`if`, `case ... of`, or functions—to use is rather hard to answer. The differences between function calls and `case ... of` are minimal. In fact, they are represented the same way at a lower level, and they both effectively have the same performance cost. One obvious difference arises when more than one argument needs to be evaluated. For example, `function(A,B) -> ...` can have guards and values to match against `A` and `B`, but a `case` expression would need to be formulated a bit, like this:

```
case {A,B} of
    Pattern Guards -> ...
end.
```

This form might seem a bit surprising. In similar situations, using a function call might be more appropriate. On the other hand, the `insert/2` function we wrote earlier is arguably cleaner the way it is, rather than having an immediate function call to track down a simple `true` or `false` clause.

And why would you ever use `if`, given that `case` expressions and functions are flexible enough to even encompass `if` through guards? The rationale behind `if` is quite simple: It was added to the language as a short way to have guards without needing to write the whole pattern-matching part when it wasn't needed.

Of course, all of this is mostly a matter of personal preference. There is no good, solid answer. In fact, this topic is still debated by the Erlang community from time to time. No one is going to try to beat you up because of the method you've chosen, as long as it is easy to understand. As Ward Cunningham, inventor of the wiki, once put it, "Clean code is when you look at a routine and it's pretty much what you expected."

4

TYPES (OR LACK THEREOF)

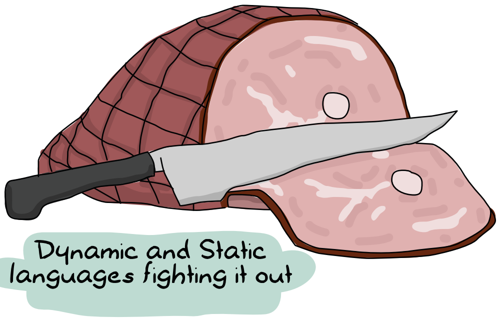
Modern functional languages are often known for their fancy type systems, which are powerful systems that let programmers obtain more safety and speed while doing less. Static type systems vary a lot—from C- and Java-like systems where annotations are provided to the compiler, to rather complex systems that depend on advanced mathematical concepts to guarantee the crash-free nature of a program. Other type systems are rather crude—not static at all, but dynamic. They give no guarantees about the safety of a piece of software, and just check everything while it runs.

This chapter introduces Erlang's type system, the reasons behind its use, and how that affects you, as a brand-new Erlang programmer.

Dynamite-Strong Typing

As you might have noticed when trying the examples in Chapter 1, and then creating modules and functions in Chapters 2 and 3, we never needed to specify the type of a variable or the type of a function. When pattern matching, the code we wrote didn't need to know what it would be matched against. The tuple `{X,Y}` could be matched with `{atom, 123}`, as well as `{"A string", <<"binary stuff!">>}`, `{2.0, ["strings","and",atoms]}`, or really anything at all.

When it didn't work, an error was thrown in your face, but only once you ran the code. This is because Erlang is *dynamically typed*. Every error is caught at runtime, and the compiler won't always yell at you when compiling modules where things may result in failure, as in the `5 + llama` example in Chapter 1.



One classic friction point between proponents of static and dynamic typing has to do with the safety of the software being written. Some programmers claim that good static type systems will catch most errors before you can even execute the code. As such, statically typed languages are typically seen as safer than their dynamic counterparts. While this might be true in comparison with many dynamic languages, Erlang begs to differ, and it has a track record to prove it.

The best example of Erlang's robustness is the often-reported nine nines (99.9999999 percent) of availability offered on the Ericsson AXD 301 ATM switches, which consist of more than a million lines of Erlang code. Please note that this is not an indication that none of the components in an Erlang-based system failed, but that a general switch system was available 99.9999999 percent of the time, planned outages included. This is partially because Erlang is built on the notion that a failure in one of the components should not affect the whole system. It accounts for errors coming from the programmer, hardware failures, and some network failures. The language includes features that allow distributing a program to different nodes. It can handle unexpected errors and never stop running.

To put it simply, while most languages and type systems aim to allow error-free programs, Erlang assumes that errors will happen and includes features that make those errors easier to handle smoothly and without unnecessary downtime. So Erlang's dynamic type system is not a barrier to the reliability and safety of programs. This sounds like a lot of prophetic talking, but you'll see the gritty details in later chapters.

NOTE *Dynamic typing was historically chosen for simple reasons. The programmers who first implemented Erlang mostly came from dynamically typed languages, and as such, making Erlang dynamic was a natural choice for them. Indirectly, this also proved to be the simplest way to allow hot-reloading (updating code without stopping it first). Doing static type checking on systems where any of its components might be replaced at any time proves to be quite difficult compared to doing it dynamically.*

Erlang is also *strongly typed*. A weakly typed language would do implicit type conversions between terms. For example, if Erlang were weakly typed, we could do the operation `6 = 5 + "1"`. But because of Erlang's strong typing, trying this operation raises an exception for bad arguments:

```
1> 6 + "1".
** exception error: bad argument in an arithmetic expression
in operator  +/2
called as 6 + "1"
```

Of course, there are times when you may want to convert one kind of data to another type. For example, you might want to change regular strings into binary strings to store them, or convert an integer to a floating-point number. The Erlang standard library provides a number of functions to do these conversions.

Type Conversions

Erlang, like many languages, changes the type of a term by casting it into another one. This is done with the help of BIFs, as many of the conversions could not be implemented in Erlang itself. Each of these functions takes the form *TypeA*`_to_`*TypeB*, and they are implemented in the `erlang` module. Here are a few of them:

```
1> erlang:list_to_integer("54").
54
2> erlang:integer_to_list(54).
"54"
3> erlang:list_to_integer("54.32").
** exception error: bad argument
in function  list_to_integer/1
called as list_to_integer("54.32")
4> erlang:list_to_float("54.32").
54.32
5> erlang:atom_to_list(true).
"true"
6> erlang:list_to_binary("hi there").
<<"hi there">>
7> erlang:binary_to_list(<<"hi there">>).
"hi there"
```

We're hitting on a language wart here: Because the scheme *Type*`_to_`*Type* is used, every time a new type is added to the language, a whole lot of conversion BIFs need to be added by the OTP team!

Here's the whole list already there:

```
atom_to_binary/2
atom_to_list/1
binary_to_atom/2
binary_to_existing_atom/2
binary_to_list/1
binary_to_term/1
binary_to_term/2
bitstring_to_list/1
float_to_list/1
fun_to_list/1
integer_to_list/1
integer_to_list/2
iolist_to_atom/1
iolist_to_binary/1
list_to_atom/1
list_to_binary/1
list_to_bitstring/1
list_to_existing_atom/1
list_to_float/1
list_to_integer/2
list_to_pid/1
list_to_tuple/1
pid_to_list/1
port_to_list/1
ref_to_list/1
term_to_binary/1
term_to_binary/2
tuple_to_list/1
```

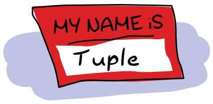
That's a lot of conversion functions. You'll see most, if not all, of these types in this book, although we probably won't need all of these functions in our code.

NOTE *The BIF* `binary_to_term/2` *lets you unserialize data the same way* `binary_to_term/1` *does. The big difference is that the second argument is an option list. If you pass in* `[safe]`*, the binary won't be decoded if it contains unknown atoms or anonymous functions, which could exhaust the memory of a node or represent a security risk. Use* `binary_to_term/2` *rather than* `binary_to_term/1` *if you are decoding data that could be unsafe.*

To Guard a Data Type

Erlang basic data types are easy to spot: Tuples have curly brackets, lists have square brackets, strings are enclosed in double quotation marks, and so on. So, we've been able to enforce a certain data type with pattern matching. For example, a function `head/1` taking a list could accept only lists because otherwise the matching (`[H|_]`) would fail.

However, we ran into a problem when pattern matching with numeric values because we couldn't specify ranges. So, in Chapter 3, we used guards in functions that needed to test for certain ranges, such as temperatures, ages, and so on. We're hitting another roadblock now. How could we write a guard that ensures that patterns match against data of a single specific type, like numbers, atoms, or binaries?



There are functions dedicated to the task of guarding data types. They take a single argument and return `true` if the type is correct; otherwise, they return `false`. They are part of the few functions allowed in guard expressions and are named the *type-test BIFs*. The following are the Erlang type-test BIFs:

`is_atom/1`	`is_function/1`	`is_port/1`
`is_binary/1`	`is_function/2`	`is_record/2`
`is_bitstring/1`	`is_integer/1`	`is_record/3`
`is_boolean/1`	`is_list/1`	`is_reference/1`
`is_builtin/3`	`is_number/1`	`is_tuple/1`
`is_float/1`	`is_pid/1`	

These functions can be used like any other guard expression, wherever guard expressions are allowed.

You might be wondering why there is no function that just gives the type of the term being evaluated (something akin to `type_of(X) -> Type`). The answer is simple: Erlang is about programming for the right cases. You program only for what you know will happen and what you expect, and everything else should cause an error as soon as possible. As such, having a single function `type_of(X)` would encourage people to write conditional branches to code, a bit like this:

```
my_function(Exp) ->
    case type_of(Exp) of
        binary -> Expression1;
        list -> Expression2
    end.
```

This code is equivalent to the following:

```
my_function(Exp) when is_binary(Exp) -> Expression1;
my_function(Exp) when is_list(Exp) -> Expression2.
```

The declarative nature of the language favors the latter form, where we do branching through function heads by specifying what we expect, rather than handling one of many types that a function like `type_of(X)` might return.

NOTE *Type-test BIFs constitute more than half of the functions allowed in guard expressions. The rest are also BIFs, but do not represent type tests. These include `abs(Number)`, `bit_size(Binary)`, `byte_size(Binary)`, `element(N, Tuple)`, `float(Term)`, `hd(List)`, `length(List)`, `node()`, `node(Pid|Ref|Port)`, `round(Number)`, `self()`, `tl(List)`, `trunc(Number)`, and `tuple_size(Tuple)`. The functions `node/1` and `self/0` are related to distributed Erlang and processes/actors.*

It may seem like Erlang data structures are relatively limited, but lists and tuples are usually enough to build other complex structures. For example, the basic node of a binary tree could be represented as `{node, Value, Left, Right}`, where `Left` and `Right` are either similar nodes or empty tuples. I could also represent myself as follows:

```
{person, {name, <<"Fred T-H">>},
{qualities, ["handsome", "smart", "honest", "objective"]},
{faults, ["liar"]},
{skills, ["programming", "bass guitar", "underwater breakdancing"]}}.
```

This shows that by nesting tuples and lists, and filling them with data, we can obtain complex data structures and build functions to operate on them.

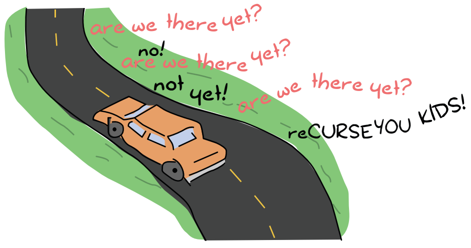
For Type Junkies

If you're a programmer who somehow can't live without a static type system, I invite you to jump to Chapter 30, which covers Dialyzer.

In that chapter, I will briefly describe tools used to do static type analysis in Erlang, allowing you to define custom types and get more safety that way. The types are entirely optional, and although useful, they are not necessary to make good Erlang programs.

5
HELLO RECURSION!

Some readers accustomed to imperative and object-oriented programming languages might be wondering why we haven't covered loops already. The answer to this is a question: What is a loop? The truth is that functional programming languages usually do not offer looping constructs like `for` and `while`. Instead, functional programmers rely on a silly concept called *recursion*, which is the topic of this chapter.



How Recursion Works

Recall how invariable variables were explained in Chapter 1 (if you can't, reread "Invariable Variables" on page 11). Recursion can also be explained with the help of mathematical concepts and functions.

A basic mathematical function such as the factorial of a value is a good example of a function that can be expressed recursively. The factorial of a number n is the product of the sequence $1 \times 2 \times 3 \times \ldots \times n$, or alternatively $n \times n - 1 \times n - 2 \times \ldots \times 1$. In mathematical notation, the factorial of a number is represented as the number followed by an exclamation point (!). To give some examples, the factorial of 3 is $3! = 3 \times 2 \times 1 = 6$, and the factorial of 4 is $4! = 4 \times 3 \times 2 \times 1 = 24$. Such a function can be expressed the following way in mathematical notation:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases}$$

This tells us that if the value of n is 0, we return the result 1. For any value above 0, we return n multiplied by the factorial of $n - 1$, which unfolds until it reaches 1:

$$4! = 4 \times 3!$$
$$4! = 4 \times 3 \times 2!$$
$$4! = 4 \times 3 \times 2 \times 1!$$
$$4! = 4 \times 3 \times 2 \times 1 \times 1$$

How can such a function be translated from mathematical notation to Erlang? The conversion is simple enough. Take a look at the parts of the notation: $n!$, 1, and $n((n - 1)!)$, and then the `if` expressions. What we have here is a function name (`n!`), guards (the `if`s), and a function body (`1` and `n((n-1)!)`). We'll rename `n!` to `fac(N)` to restrict our syntax a bit, and we get the following:

```
-module(recursive).
-export([fac/1]).

fac(N) when N == 0 -> 1;
fac(N) when N > 0  -> N*fac(N-1).
```

And this factorial function is now complete! It's pretty similar to the mathematical definition, really. With the help of pattern matching, we can shorten the definition a bit:

```
fac(0) -> 1;
fac(N) when N > 0 -> N*fac(N-1).
```

We looped by using a function that calls itself! And you know what? "A function that calls itself" is one way to define recursion.

However, having a function that calls itself is not enough. If the function just calls itself forever, it will, unsurprisingly, continue forever. What we need is a stopping condition (called a *base case*), which is a function clause where we return a value rather than calling the function again. In our case, the stopping condition is when `n` is equal to 0. At that point, we no longer tell our function to call itself, and it stops its execution right there by returning 1.

Length of a List

Let's try a slightly more practical example. We'll implement a function to count how many elements a list contains. So we know from the beginning that we will need the following:

- A base case
- A function that calls itself
- A list to test our function

With most recursive functions, I find it easier to write the base case first. What's the simplest input we'll need to find the length of? Surely an empty list, with a length of 0, is the simplest case. So let's make a mental note that `[] = 0` when dealing with lengths. Then the next simplest list has a length of 1: `[_] = 1`. This sounds like enough to get going with our definition. We can write this down:

```
len([]) -> 0;
len([_]) -> 1.
```

Awesome! We can calculate the length of lists, given the length is either 0 or 1. Very useful indeed! Of course, it's useless, because it's not yet recursive, which brings us to the hardest part: extending our function so it calls itself for lists longer than 1 or 0.

I mentioned in Chapter 1 that lists are defined recursively as `[1 | [2| ... [n | []]]]`. This means we can use the `[H|T]` pattern to match against lists of one or more elements, as a list of length 1 will be defined as `[X|[]]`, and a list of length 2 will be defined as `[X|[Y|[]]]`. Note that the second element is a list itself. This means that we need to count only the first one, and the function can call itself on the second element. Given each value in a list counts as a length of 1, the function can be rewritten the following way:

```
len([]) -> 0;
len([_|T]) -> 1 + len(T).
```

And now you have your own recursive function to calculate the length of a list. To see how `len/1` behaves when it runs, let's try it on a given list, say `[1,2,3,4]`:

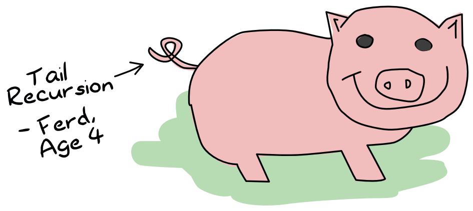
```
len([1,2,3,4]) = len([1 | [2,3,4])
               = 1 + len([2 | [3,4]])
               = 1 + 1 + len([3 | [4]])
               = 1 + 1 + 1 + len([4 | []])
               = 1 + 1 + 1 + 1 + len([])
               = 1 + 1 + 1 + 1 + 0
               = 1 + 1 + 1 + 1
               = 1 + 1 + 2
               = 1 + 3
               = 4
```

And we get the correct answer. Congratulations on your first useful recursive function in Erlang!

Length of a Tail Recursion

You might have noticed that for a list of four terms, we expanded our function call to a single chain of five additions. While this does the job fine for short lists, it can become problematic if your list has a few million values in it. You don't want to keep millions of numbers in memory for such a simple calculation. It's wasteful, and there's a better way. Enter *tail recursion*.

Tail recursion is a way to transform the preceding linear process (it grows as much as there are elements) to an iterative one (there is not really any growth). To make a function call tail recursive, it needs to be "alone," which requires a bit of explanation.



What made our previous calls grow is how the answer to the first part depends on evaluating the second part. The answer to `1 + len(Rest)` needs the answer to `len(Rest)` to be found. The function `len(Rest)` itself then needs the result of another function call to be found. The additions are stacked until the last one is found, and only then is the final result calculated. Tail recursion aims to eliminate this stacking of operations by reducing them as they happen.

To achieve this, we will need to hold an extra temporary variable as a parameter in our function. I'll illustrate the concept with the help of the factorial function, but this time defining it to be tail recursive. The aforementioned temporary variable is sometimes called an *accumulator*, and it acts as a place to store the results of our computations as they happen in order to limit the growth of our calls:

```
tail_fac(N) -> tail_fac(N,1).

tail_fac(0,Acc) -> Acc;
tail_fac(N,Acc) when N > 0 -> tail_fac(N-1,N*Acc).
```

Here, we define both `tail_fac/1` and `tail_fac/2`. This is necessary because Erlang doesn't allow default arguments in functions (different arity means different function), so we do that manually. In this specific case, `tail_fac/1` acts like an abstraction over the tail recursive `tail_fac/2` function. The details about the hidden accumulator of `tail_fac/2` don't interest anyone, so we would export only `tail_fac/1` from our module. When running this function, we can expand it to the following:

```
tail_fac(4)    = tail_fac(4,1)
tail_fac(4,1)  = tail_fac(4-1, 4*1)
tail_fac(3,4)  = tail_fac(3-1, 3*4)
tail_fac(2,12) = tail_fac(2-1, 2*12)
tail_fac(1,24) = tail_fac(1-1, 1*24)
tail_fac(0,24) = 24
```

Do you see the difference? Now we never need to hold more than two terms in memory, so the space usage is constant. It will take as much space to calculate the factorial of 4 as it will to calculate the factorial of 1,000,000 (that is, if we forget that 4! is a smaller number than 1,000,000! in its complete representation).

With an example of tail recursive factorials under your belt, you might be able to see how this pattern could be applied to our `len/1` function. We need to make our recursive call alone. If you like visual examples, just imagine you're going to put the `+1` part inside the function call by adding a parameter. So this:

```
len([]) -> 0;
len([_|T]) -> 1 + len(T).
```

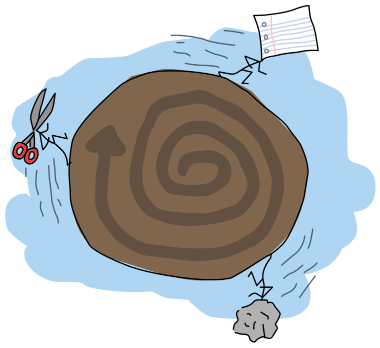
becomes the following:

```
tail_len(L) -> tail_len(L,0).

tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T,Acc+1).
```

And now our length function is tail recursive.

More Recursive Functions



We'll write a few more recursive functions, just to get in the habit. After all, since recursion is the only looping construct that exists in Erlang (except list comprehensions), it's one of the most important concepts to understand. It's also useful in every other functional programming language you'll try, so take notes!

A Duplicate Function

The first function we'll write is `duplicate/2`. This function takes an integer as its first parameter and any other term as its second parameter. It then creates a list of as many copies of the term as specified by the integer.

Again, thinking of the base case first might help us get going. For `duplicate/2`, asking to repeat something zero times is the most basic thing that can be done. All we need to do is return an empty list, no matter what the term is. Every other case needs to try to get to the base case by calling the function itself. We will also forbid negative values for the integer, because you can't duplicate something −*n* times. Here are these cases:

```
duplicate(0,_) ->
    [];
duplicate(N,Term) when N > 0 ->
    [Term|duplicate(N-1,Term)].
```

Once the basic recursive function is found, it becomes easier to transform it into a tail recursive one by moving the list construction into a temporary variable:

```
tail_duplicate(N,Term) ->
    tail_duplicate(N,Term,[]).

tail_duplicate(0,_,List) ->
    List;
tail_duplicate(N,Term,List) when N > 0 ->
   tail_duplicate(N-1, Term, [Term|List]).
```

Success!

A Reverse Function

There's also an interesting property that we can discover when we compare recursive and tail recursive functions by writing a `reverse/1` function, which will reverse a list of terms. For such a function, the base case is an empty

RECURSION IS LIKE A WHILE LOOP

Comparing tail recursion with a `while` loop helped me a lot when I was first learning about recursion. Our `tail_duplicate/2` function has all the usual parts of a `while` loop. If we were to imagine a `while` loop in a fictional language with Erlang-like syntax, our function could look a bit like this:

```
function(N, Term) ->
    while N > 0 ->
        List = [Term|List],
        N = N-1
    end,
    List.
```

Note that all the elements exist in both the fictional language and in Erlang; only their position changes. This demonstrates that a proper tail recursive function is similar to an iterative process, like a `while` loop. If recursion is confusing and you're used to `while` loops, translating them directly might help. Just be careful—once you've used Erlang for a while, you might end up thinking recursively and need to translate the other way around if you ever go back to imperative languages!

list, for which we have nothing to reverse. We can just return an empty list when that happens. Every other possibility should try to converge to the base case by calling itself, as with `duplicate/2`. Our function is going to iterate through the list by pattern matching `[H|T]`, and then putting `H` after the rest of the list:

```
reverse([]) -> [];
reverse([H|T]) -> reverse(T)++[H].
```

On long lists, this will be a true nightmare. Not only will we stack up all our append operations, but we will need to traverse the whole list for every single append operation until the last one! For visual readers, the many checks can be represented as follows:

```
reverse([1,2,3,4]) = [4]++[3]++[2]++[1]
                      ↑   ↵
                   = [4,3]++[2]++[1]
                      ↑ ↑   ↵
                   = [4,3,2]++[1]
                      ↑ ↑ ↑   ↵
                   = [4,3,2,1]
```

This is where tail recursion comes to the rescue. Because we will use an accumulator and will add a new head to it every time, our list will be reversed automatically.

Let's first see the implementation:

```
tail_reverse(L) -> tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).
```

If we represent this one in a similar manner as the normal version, we get the following:

```
tail_reverse([1,2,3,4]) = tail_reverse([2,3,4], [1])
                        = tail_reverse([3,4], [2,1])
                        = tail_reverse([4], [3,2,1])
                        = tail_reverse([], [4,3,2,1])
                        = [4,3,2,1]
```

This shows that the number of elements visited to reverse our list is now linear. Not only do we avoid growing the stack, we also do our operations much more efficiently!

A Sublist Function

Another function we'll implement is `sublist/2`, which takes a list *`L`* and an integer *`N`*, and returns the *`N`* first elements of the list. As an example, `sublist([1,2,3,4,5,6],3)` returns `[1,2,3]`.

Again, the base case is trying to obtain zero elements from a list. But we need to be careful, because `sublist/2` is a bit different. We have a second base case when the list passed is empty! If we do not check for empty lists, an error will be thrown when calling `recursive:sublist([1],2)`, when we want `[1]` instead. Once this is defined, the recursive part of the function only needs to cycle through the list, keeping elements as it goes, until it hits one of the base cases, as follows:

```
sublist(_,0) -> [];
sublist([],_) -> [];
sublist([H|T],N) when N > 0 -> [H|sublist(T,N-1)].
```

This can then be transformed to a tail recursive form in the same manner as before:

```
tail_sublist(L, N) -> tail_sublist(L, N, []).

tail_sublist(_, 0, SubList) -> SubList;
tail_sublist([], _, SubList) -> SubList;
tail_sublist([H|T], N, SubList) when N > 0 ->
    tail_sublist(T, N-1, [H|SubList]).
```

There's a flaw in this function—a *fatal* flaw! We use a list as an accumulator in exactly the same manner as we did to reverse our list. If you compiled this function as is, `sublist([1,2,3,4,5,6],3)` would not return `[1,2,3]` but instead would give you [3,2,1]. The only thing we can do is take the final result and reverse it ourselves. Just change the tail_sublist/2 call and leave all our recursive logic intact:

```
tail_sublist(L, N) -> reverse(tail_sublist(L, N, [])).
```

The final result will be ordered correctly. It might seem like reversing our list after a tail recursive call is a waste of time, and that's partially right (we still save memory doing this). On shorter lists, you might find your code is running faster with normal recursive calls than with tail recursive calls for this reason, but as your data sets grow, reversing the list will be comparatively lighter.

NOTE *Instead of writing your own* reverse/1 *function, you should use* lists:reverse/1*. It has been used so much for tail recursive calls that the maintainers and developers of Erlang decided to turn it into a BIF. Your lists can now benefit from extremely fast reversal (thanks to functions written in C), which will make the reversal disadvantage a lot less obvious. The rest of the code in this chapter will make use of our own reversal function, but after that, you should never use it again.*

A Zip Function

To push things a bit further, we'll write a zipping function. A zipping function takes two lists of the same length as parameters and joins them as a list of tuples, which all hold two terms. Our own zip/2 function will behave this way:

```
1> recursive:zip([a,b,c],[1,2,3]).
[{a,1},{b,2},{c,3}]
```

Given that we want both our parameters to have the same length, the base case will be zipping two empty lists:

```
zip([],[]) -> [];
zip([X|Xs],[Y|Ys]) -> [{X,Y}|zip(Xs,Ys)].
```

However, if we wanted a more lenient zipping function, we could decide to have it finish whenever one of the two lists is done. In this scenario, we have two base cases:

```
lenient_zip([],_) -> [];
lenient_zip(_,[]) -> [];
lenient_zip([X|Xs],[Y|Ys]) -> [{X,Y}|lenient_zip(Xs,Ys)].
```

Notice that no matter what our base cases are, the recursive part of the function remains the same.

LAST CALL OPTIMIZATION

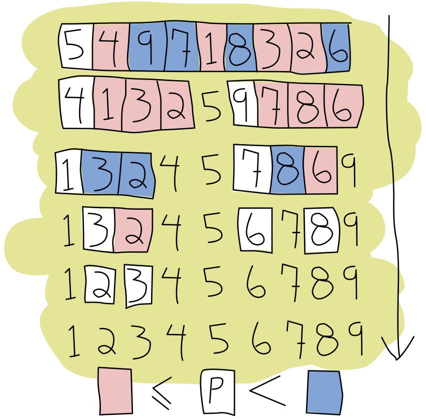
Tail recursion as seen here is not making the memory consumption grow, because when the VM sees a function calling itself in a tail position (the last expression to be evaluated in a function), it eliminates the current stack frame. This is called *tail call optimization (TCO)*, and it is a special case of a more general optimization named *last call optimization (LCO)*.

LCO is done whenever the last expression to be evaluated in a function body is another function call. When that happens, as with TCO, the Erlang VM avoids storing the stack frame. As such, tail recursion is also possible between multiple functions. As an example, the chain of functions `a() -> b(). b() -> c(). c() -> a().` will effectively create an infinite loop that won't run out of memory, as LCO avoids overflowing the stack. This principle, combined with the use of accumulators, is what makes tail recursion useful.

I suggest that you try to make your own tail recursive versions of `zip/2` and `lenient_zip/2`, just to make sure you fully understand how to make tail recursive functions. They will be one of the central concepts of larger applications, where the main loops will be made that way.

If you want to check your answers, take a look at my implementation of *recursive.erl* (*http://learnyousomeerlang.com/static/erlang/recursive.erl*), particularly the `tail_zip/2` and `tail_lenient_zip/3` functions.

Quick, Sort!



Just to ensure that recursion and tail recursion make sense to you, we'll push forward with a more complex example: quicksort. Yes, this is the canonical "Hey look, I can write short functional code" example.

A naive implementation of quicksort works by taking the first element of a list, the *pivot*, and then putting all the elements smaller than or equal to the pivot in a new list and all those larger than the pivot in another list. We then take each of these lists and do the same thing on them until each list gets smaller and smaller. This goes on until we have nothing but an empty list to sort, which will be our base case. This implementation is said to be naive because smarter versions of quicksort will try to pick optimal pivots to be faster. We don't really care about that for our example.

We will need two functions for this one: a function to partition the list into smaller and larger parts, and another function to apply the partition function on each of the new lists and to glue them together. First, we'll write the glue function (you can do this in recursive.erl):

```
quicksort([]) -> [];
quicksort([Pivot|Rest]) ->
    {Smaller, Larger} = partition(Pivot,Rest,[],[]),
    quicksort(Smaller) ++ [Pivot] ++ quicksort(Larger).
```

This shows the base case, a list already partitioned in larger and smaller parts by another function, and the use of a pivot with both lists quicksorted appended before and after it. So this should take care of assembling lists.

Next, we write the partitioning function:

```
partition(_,[], Smaller, Larger) -> {Smaller, Larger};
partition(Pivot, [H|T], Smaller, Larger) ->
    if H =< Pivot -> partition(Pivot, T, [H|Smaller], Larger);
       H >  Pivot -> partition(Pivot, T, Smaller, [H|Larger])
    end.
```

And you can now run our quicksort function.

If you've looked for Erlang examples on the Internet, you might have seen another implementation of quicksort—one that is simpler and easier to read, but makes use of list comprehensions. The easy-to-replace parts are the ones that create new lists, the partition/4 function:

```
lc_quicksort([]) -> [];
lc_quicksort([Pivot|Rest]) ->
    lc_quicksort([Smaller || Smaller <- Rest, Smaller =< Pivot])
    ++ [Pivot] ++
    lc_quicksort([Larger || Larger <- Rest, Larger > Pivot]).
```

DON'T DRINK TOO MUCH KOOL-AID

All this conciseness is good for educational purposes, but not for performance. Many functional programming tutorials never mention this! First of all, both implementations of quicksort shown here need to process values that are equal to the pivot more than once. We could have decided to instead return three lists—elements smaller, larger, and equal to the pivot—to make this more efficient.

Another problem relates to how we need to traverse all the partitioned lists more than once when attaching them to the pivot. It is possible to reduce the overhead a little by doing the concatenation while partitioning the lists in three parts. If you're curious about this, look at the last function (bestest_qsort/1) of *recursive.erl* for an example.

A nice point about all of these quicksort versions is that they will work on lists of any data type you have, even tuples of lists and whatnot. Try them, and you'll see that they work.

This version is much easier to read, but in exchange, it must traverse the list to partition it in two parts. This is a battle of clarity vs. performance, although the real loser here is you, because a `lists:sort/1` function already exists. Use that one instead.

More Than Lists

At this point, you might think that recursion in Erlang mainly concerns lists. While lists are a good example of a data structure that can be defined recursively, there's certainly more to recursion than working with lists. For the sake of diversity, we'll look at how to build binary trees and then read data from them.

First, it's important to define what a tree is. In our case, a tree has nodes all the way down. Nodes are tuples that contain a key, a value associated with the key, and then two other nodes. Of these two nodes, we need one that has a smaller key and one that has a larger key than the node holding them. So here's recursion! A tree is a node containing nodes, each of which contains nodes, which, in turn, also contain nodes. This can't keep going forever (we don't have infinite data to store), so we'll say that our nodes can also contain empty nodes.

To represent nodes, tuples are an appropriate data structure. For our implementation, we can then define these tuples as `{node, {Key, Value, Smaller, Larger}}` (a tagged tuple!), where `Smaller` and `Larger` can be another similar node or an empty node (`{node, nil}`). We don't need a concept more complex than that.

Let's start building a module for our very basic tree implementation. The first function, `empty/0`, returns an empty node. The empty node is the starting point of a new tree, also called the *root*.

```
-module(tree).
-export([empty/0, insert/3, lookup/2]).

empty() -> {node, 'nil'}.
```

By using that function and then encapsulating all representations of nodes the same way, we hide the implementation of the tree so people don't need to know how it's built. All that information can be contained by the module alone. If you ever decide to change the representation of a node, you can then do it without breaking external code.

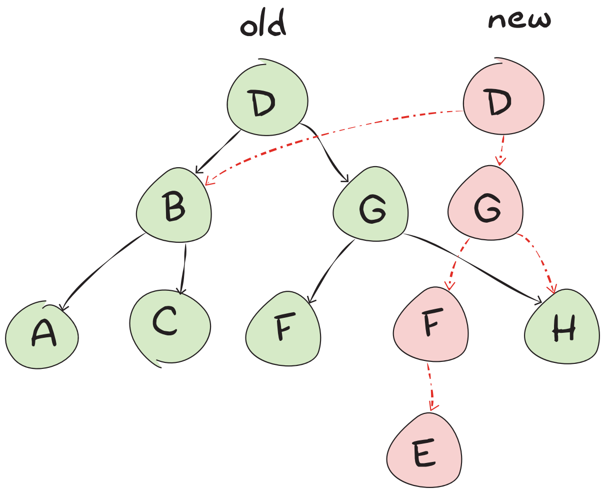
To add content to a tree, you must first understand how to recursively navigate through it. Let's proceed in the same way as we did for every other recursion example: by first trying to find the base case.

Given that an empty tree is an empty node, our base case is thus logically an empty node. So whenever we hit an empty node, that's where we can add our new key/value. The rest of the time, our code must go through the tree to try to find an empty node in which to put content.

To find an empty node starting from the root, we must use the fact that the presence of `Smaller` and `Larger` nodes lets us navigate by comparing the new key we have to insert to the current node's key. If the new key is smaller than the current node's key, we try to find the empty node inside `Smaller`; if it's larger, we look inside `Larger`. There is one last case, though: What if the new key is equal to the current node's key? We have two options there: let the program fail or replace the value with the new one. We'll take the latter option. Put into a function, all this logic works the following way:

```
insert(Key, Val, {node, 'nil'}) ->
    {node, {Key, Val, {node, 'nil'}, {node, 'nil'}}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when NewKey < Key ->
    {node, {Key, Val, insert(NewKey, NewVal, Smaller), Larger}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller, Larger}}) when NewKey > Key ->
    {node, {Key, Val, Smaller, insert(NewKey, NewVal, Larger)}};
insert(Key, Val, {node, {Key, _, Smaller, Larger}}) ->
    {node, {Key, Val, Smaller, Larger}}.
```

Note here that the function returns a completely new tree. This is typical of functional languages that have only single assignment. While this can be seen as inefficient, updating a tree or adding an element usually requires changing only the nodes that were modified up to the change. The other nodes can be shared between both versions of the tree, strongly reducing the memory overhead. In the following image, the node containing "E" is added, which requires updating all of its parents. However, the entire left side of the tree (starting with "B") doesn't change and can be kept the same across versions. This concept is more regularly known to functional programmers as using *persistent data structures.*



The last thing we need to do with our example tree implementation is to create a `lookup/2` function that will let us find a value from a tree by giving its key. The logic needed is extremely similar to the logic used to add new content to the tree: We step through the nodes, checking if the lookup key is equal to, smaller than, or larger than the current node's key. We have two base cases: one when the node is empty (the key isn't in the tree) and one when the key is found. Because we don't want our program to crash each time we look for a key that doesn't exist, we'll return the atom `undefined`. Otherwise, we'll return `{ok, Value}`. If we only returned `Value`, and the node contained the atom `undefined`, we would have no way to know if the tree did return the correct value or failed to find it. By wrapping successful cases in such a tuple, we make it easy to understand which is which. Here's the implemented function:

```
lookup(_, {node, 'nil'}) ->
    undefined;
lookup(Key, {node, {Key, Val, _, _}}) ->
    {ok, Val};
lookup(Key, {node, {NodeKey, _, Smaller, _}}) when Key < NodeKey ->
    lookup(Key, Smaller);
lookup(Key, {node, {_, _, _, Larger}}) ->
    lookup(Key, Larger).
```

And we're finished. Let's test it by making a little email address book. Compile the file and start the shell:

```
1> T1 = tree:insert("Jim Woodland", "jim.woodland@gmail.com", tree:empty()).
{node,{"Jim Woodland","jim.woodland@gmail.com",
       {node,nil},
       {node,nil}}}
2> T2 = tree:insert("Mark Anderson", "i.am.a@hotmail.com", T1).
{node,{"Jim Woodland","jim.woodland@gmail.com",
       {node,nil},
       {node,{"Mark Anderson","i.am.a@hotmail.com",
              {node,nil},
              {node,nil}}}}}
3> Addresses = tree:insert("Anita Bath", "abath@someuni.edu",
3>    tree:insert("Kevin Robert", "myfairy@yahoo.com",
3>       tree:insert("Wilson Longbrow", "longwil@gmail.com", T2))).
{node,{"Jim Woodland","jim.woodland@gmail.com",
       {node,{"Anita Bath","abath@someuni.edu",
              {node,nil},
              {node,nil}}},
       {node,{"Mark Anderson","i.am.a@hotmail.com",
              {node,{"Kevin Robert","myfairy@yahoo.com",
                     {node,nil},
                     {node,nil}}},
       {node,{"Wilson Longbrow","longwil@gmail.com",
              {node,nil},
              {node,nil}}}}}}}
```

And now you can look up email addresses with it:

```
4> tree:lookup("Anita Bath", Addresses).
{ok, "abath@someuni.edu"}
5> tree:lookup("Jacques Requin", Addresses).
undefined
```

That concludes our functional address book example built from a recursive data structure other than a list! *Anita Bath* now . . .

NOTE *Our tree implementation is very naive. We do not support common operations such as deleting nodes or rebalancing the tree to make the following lookups faster. If you're interested in implementing and/or exploring these, studying the implementation of Erlang's* `gb_trees` *module (*`YourErlangInstallPath/lib/stdlib/src/gb_trees.erl`*) is a good idea. This is also the module you should use when dealing with trees in your code, rather than reinventing the wheel.*

Thinking Recursively

If you've understood everything in this chapter, thinking recursively is probably becoming more intuitive. A different aspect of recursive definitions when compared to their imperative counterparts (usually in `while` or `for` loops) is that instead of taking a step-by-step approach ("do this, then that, then this, then you're finished"), our approach is more declarative ("if you get this input, do that; otherwise, do this"). This property is made more obvious with the help of pattern matching in function heads.

If you still haven't grasped how recursion works, maybe reading this sentence will help you.

Joking aside, recursion coupled with pattern matching is sometimes an optimal solution to the problem of writing concise algorithms that are easy to understand. By subdividing each part of a problem into separate functions until they can no longer be simplified, the algorithm becomes nothing but assembling a bunch of correct answers coming from short routines (that's a bit similar to what we did with quicksort). This kind of mental abstraction is also possible with your everyday loops, but I believe the practice is easier with recursion. Your mileage may vary.

AND NOW LADIES AND GENTLEMEN, A DISCUSSION: THE AUTHOR VS. HIMSELF

Self: Okay, I think I understand recursion. I get the declarative aspect of it. I get it has mathematical roots, like with invariable variables. I get that you find it easier in some cases. What else?

Author: It respects a regular pattern. Find the base cases and write them down. Then all the other cases should try to converge to these base cases to get your answer. It makes writing functions pretty easy.

Self: Yeah, I got that. You repeated it a bunch of times already. My loops can do the same thing.

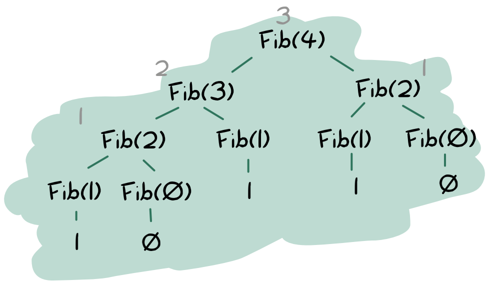
Author: Yes, they can. I can't deny that.

Self: And another thing: Why bother writing all these non-tail recursive versions if they're not as good as tail recursive ones?

Author: Oh, it's simply to make things easier to grasp. Moving from regular recursion, which is prettier and easier to understand, to tail recursion, which is theoretically more efficient, sounded like a good way to show all of the options.

Self: Right, so they're useless except for educational purposes. I get it.

Author: Not exactly. In practice, you'll see little difference in the performance between tail recursive and normal recursive calls. The areas to take care of are in functions that are supposed to loop infinitely, like main loops. There are also types of functions that will always generate very large stacks, be slow, and possibly crash early if you don't make them tail recursive. The best example of this is the Fibonacci function, which grows exponentially if it's not iterative or tail recursive. You should profile your code, see what slows it down, and fix it.



Self: But loops are always iterative and make this a nonissue.

Author: Yes, but . . . but . . . my beautiful Erlang . . .

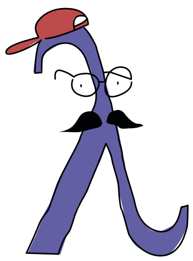
Self: Well isn't that great? All that learning because there is no `while` or `for` in Erlang. Thank you very much. I'm going back to programming my toaster in C!

Author: Not so fast there! Functional programming languages have other assets! If we've found a few basic common points between many recursive functions (accumulators, reversing at the end, and so on), a bunch of smart people found many more common points and patterns. In fact, they found enough of them that most frequent operations have been abstracted away in libraries. You'll rarely need to write recursive functions yourself. If you stay around, you'll see how such abstractions can be built. But for this, we will need more power. Let me tell you about higher-order functions . . .

6 HIGHER-ORDER FUNCTIONS

An important part of all functional programming languages is the ability to take a function you defined and then pass it as a parameter to another function. This binds that function parameter to a variable, which can be used like any other variable within the function. A function that can accept other functions transported around this way is called a *higher-order function*. As you'll learn in this chapter, higher-order functions are a powerful means of abstraction and one of the best tools to master in Erlang.

Let's Get Functional



The concept behind carrying functions around and passing them to higher-order functions is rooted in mathematics, mainly lambda calculus. Basically, in pure lambda calculus, everything is a function—even numbers, operators, and lists. Because everything is represented as a function, functions must accept other functions as parameters, and must be able to operate on them with even more functions! (If you want a good, quick introduction to lambda calculus, read the Wikipedia entry for it.)

This concept might be a bit hard to grasp, so let's start with an example (this is nowhere close to real lambda calculus, but it illustrates the point).

```
-module(hhfuns).
-compile(export_all).

one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

Now open the Erlang shell, compile the module, and get going:

```
1> c(hhfuns).
{ok, hhfuns}
2> hhfuns:add(one,two).
** exception error: bad function one
in function  hhfuns:add/2
3> hhfuns:add(1,2).
** exception error: bad function 1
in function  hhfuns:add/2
4> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
```

Confusing? Not so much, once you know how it works (isn't that always the case?). In line 2, the atoms `one` and `two` are passed to `add/2`, which then uses both atoms as function names (`X() + Y()`). If function names are written without a parameter list, then those names are interpreted as atoms, and atoms cannot be functions, so the call fails. This is why the expression on line 3 also fails: The values 1 and 2 cannot be called as functions, and functions are what we need!

To handle this issue, a new notation must be added to the language in order to pass functions from outside a module. This is the purpose of `fun Module:Function/Arity`:, which tells the VM to use that specific function and then bind it to a variable.

So what are the gains of using functions in that manner? Well, a little example might help answer that question. We'll add a few functions to `hhfuns` that work recursively over a list to add or subtract one from each integer of a list.

```
increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].
```

Do you see how similar these functions are? They basically do the same thing: cycle through a list, apply a function on each element (+ or -), and then call themselves again. Almost nothing changes in that code; only the applied function and the recursive call are different. The core of a recursive call on a list like that is always the same. We'll abstract all the similar parts in a single function (`map/2`) that will take another function as an argument.

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.
decr(X) -> X - 1.
```

Now let's test this in the shell.

```
1> c(hhfuns).
{ok, hhfuns}
2> L = [1,2,3,4,5].
[1,2,3,4,5]
3> hhfuns:increment(L).
[2,3,4,5,6]
4> hhfuns:decrement(L).
[0,1,2,3,4]
5> hhfuns:map(fun hhfuns:incr/1, L).
[2,3,4,5,6]
6> hhfuns:map(fun hhfuns:decr/1, L).
[0,1,2,3,4]
```

Here, the results are the same, but we have just created a very smart abstraction! Every time we want to apply a function to each element of a list, we only need to call `map/2` with our function as a parameter. However, it is a bit annoying to need to put every function we want to pass as a parameter to `map/2` in a module, name it, export it, compile it, and so on. In fact, it's plainly unpractical. What we need are functions that can be declared on the fly—the type of functions discussed next.

Anonymous Functions

Anonymous functions, or *funs*, address the problem of using functions as parameters by letting you declare a special kind of function inline, without naming that function. Anonymous functions can do pretty much everything normal functions can do, except call themselves recursively (how could they do that if they are anonymous?).

Anonymous functions have the following syntax:

```
fun(Args1) ->
     Expression1, Exp2, ..., ExpN;
   (Args2) ->
     Expression1, Exp2, ..., ExpN;
```

```
    (Args3) ->
      Expression1, Exp2, ..., ExpN
end
```

Here's an example of using an anonymous function:

```
7> Fn = fun() -> a end.
#Fun<erl_eval.20.67289768>
8> Fn().
a
9> hhfuns:map(fun(X) -> X + 1 end, L).
[2,3,4,5,6]
10> hhfuns:map(fun(X) -> X - 1 end, L).
[0,1,2,3,4]
```

And now you're seeing one of the things that make people like functional programming so much: the ability to make abstractions on a very low level of code. Basic concepts such as looping can thus be ignored, letting you focus on what is done, rather than how to do it.

More Anonymous Function Power

Anonymous functions are pretty dandy for such abstractions, but they have more hidden powers. Let's look at another example:

```
11> PrepareAlarm = fun(Room) ->
11>     io:format("Alarm set in ~s.~n",[Room]),
11>     fun() -> io:format("Alarm tripped in ~s! Call Batman!~n",[Room]) end
11> end.
#Fun<erl_eval.20.67289768>
12> AlarmReady = PrepareAlarm("bathroom").
Alarm set in bathroom.
#Fun<erl_eval.6.13229925>
13> AlarmReady().
Alarm tripped in bathroom! Call Batman!
ok
```

Hold the phone, Batman! What's going on here? Well, first of all, we declare an anonymous function assigned to `PrepareAlarm`. This function has not run yet. It is executed only when `PrepareAlarm("bathroom")` is called. At that point, the call to `io:format/2` is evaluated, and the "Alarm set" text is output. The second expression (another anonymous function) is returned to the caller and then assigned to `AlarmReady`. Note that in this function, the `Room` variable's value is taken from the "parent" function (`PrepareAlarm`). This is related to a concept called *closures*. But before we can talk about closures, we need to address the idea of scope.

Function Scope and Closures

A function's *scope* can be imagined as the place where all the variables and their values are stored. In the function `base(A) -> B = A + 1.`, for example, `A` and `B` are both defined to be part of `base/1`'s scope. This means that anywhere inside `base/1`, you can refer to `A` and `B` and expect a value to be bound to them. And when I say "anywhere," I ain't kidding, kid. This includes anonymous functions, too.

```
base(A) ->
    B = A + 1,
    F = fun() -> A * B end,
    F().
```

In this example, `B` and `A` are still bound to `base/1`'s scope, so the function `F` can still access them. This is because `F` inherits `base/1`'s scope. As with most kinds of real-life inheritance, the parents can't get what the children have.

```
base(A) ->
    B = A + 1,
    F = fun() -> C = A * B end,
    F(),
    C.
```

In this version of the function, `B` is still equal to `A + 1`, and `F` will still execute fine. However, the variable `C` is only in the scope of the anonymous function in `F`. When `base/1` tries to access `C`'s value on the last line, it finds only an unbound variable. In fact, if you tried to compile this function, the compiler would throw a fit. Inheritance goes only one way.

It is important to note that the inherited scope follows the anonymous function wherever it is, even when it is passed to another function. Here's an example:

```
a() ->
    Secret = "pony",
    fun() -> Secret end.

b(F) ->
    "a/0's password is "++F().
```

Now we can compile it.

```
14> c(hhfuns).
{ok, hhfuns}
15> hhfuns:b(hhfuns:a()).
"a/0's password is pony"
```

Who told a/0's password? Well, a/0 did. While the anonymous function has a/0's scope when it's declared in there, the function can still carry it when executed in b/1, as explained earlier. This is very useful because it lets us carry around parameters and content out of their original context, where the whole context itself is no longer needed (exactly as we did with Batman in the previous section).

You're most likely to use anonymous functions to carry state around when you have defined functions that take many arguments, but one of these arguments remains the same all the time, as in the following example.

```
16> math:pow(5,2).
25.0
17> Base = 2.
2
18> PowerOfTwo = fun(X) -> math:pow(Base,X) end.
#Fun<erl_eval.6.13229925>
17> hhfuns:map(PowerOfTwo, [1,2,3,4]).
[2.0,4.0,8.0,16.0]
```

By wrapping the call to math:pow/2 inside an anonymous function with the Base variable bound in that function's scope, we made it possible to have each of the calls to PowerOfTwo in hhfuns:map/2 use the integers from the list as the exponents of our base.

A little trap you might fall into when writing anonymous functions is when you try to redefine the scope, like this:

```
base() ->
    A = 1,
    (fun() -> A = 2 end)().
```

This will declare an anonymous function and then run it. As the anonymous function inherits base/0's scope, trying to use the = operator compares 2 with the variable A (bound to 1). This is guaranteed to fail. However, we can redefine the variable if we do that in the nested function's head:

```
base() ->
    A = 1,
    (fun(A) -> A = 2 end)(2).
```

And this works. If you try to compile it, you'll get a warning about shadowing: "Warning: variable 'A' shadowed in 'fun'." *Shadowing* is the term used to describe the act of defining a new variable that has the same name as one that was in the parent scope. This warning is there to prevent some mistakes (usually rightly so), so you might want to consider renaming your variables in these circumstances.

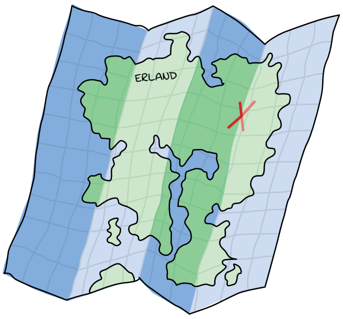
Now that we've covered scope, we can turn to closures. Closure is just the idea of having a function that references some environment along with it (the value's part of the scope). In other words, a closure is what happens when anonymous functions meet the concept of scope and carrying variables around.

We'll set the anonymous function theory aside for now and explore more common abstractions to avoid needing to write more recursive functions, as I promised at the end of Chapter 5.

Maps, Filters, Folds, and More

At the beginning of this chapter, we took a brief look at how to abstract away two similar functions to get a map/2 function:

```
map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].
```



Such a function can be used for any list where we want to act on each element. However, there are many other similar abstractions to build from commonly occurring recursive functions.

Filters

First, we'll look at filters. Consider the following functions:

```
%% Only keep even numbers.
even(L) -> lists:reverse(even(L,[])).

even([], Acc) -> Acc;
even([H|T], Acc) when H rem 2 == 0 ->
    even(T, [H|Acc]);
even([_|T], Acc) ->
    even(T, Acc).

%% Only keep men older than 60.
old_men(L) -> lists:reverse(old_men(L,[])).

old_men([], Acc) -> Acc;
old_men([Person = {male, Age}|People], Acc) when Age > 60 ->
    old_men(People, [Person|Acc]);
old_men([_|People], Acc) ->
    old_men(People, Acc).
```

The first of these functions takes a list of numbers and returns only those that are even. The second one goes through a list of people of the form {Gender, Age} and keeps only those that are males over 60.

The similarities are a bit harder to find here than in the previous examples, but we have some common points. Both functions operate on lists and have the same objective of keeping elements that succeed some test (also called a *predicate*) and then dropping the others. From this generalization, we can extract all the useful information we need and abstract them away, like this:

```
filter(Pred, L) -> lists:reverse(filter(Pred, L,[])).

filter(_, [], Acc) -> Acc;
filter(Pred, [H|T], Acc) ->
    case Pred(H) of
        true  -> filter(Pred, T, [H|Acc]);
        false -> filter(Pred, T, Acc)
    end.
```

To use the filtering function, we now only need to pass in a predicate outside of the function. Compile the `hhfuns` module and try it.

```
1> c(hhfuns).
{ok, hhfuns}
2> Numbers = lists:seq(1,10).
[1,2,3,4,5,6,7,8,9,10]
3> hhfuns:filter(fun(X) -> X rem 2 == 0 end, Numbers).
[2,4,6,8,10]
4> People = [{male,45},{female,67},{male,66},{female,12},{unkown,174},{male,74}].
[{male,45},{female,67},{male,66},{female,12},{unkown,174},{male,74}]
5> hhfuns:filter(fun({Gender,Age}) -> Gender == male andalso Age > 60 end, People).
[{male,66},{male,74}]
```

These two examples show that with the use of the `filter/2` function, the programmer needs to worry only about producing the predicate and the list. The act of cycling through the list to throw out unwanted items is no longer a consideration. This is one important thing about abstracting functional code: Try to get rid of what's always the same, and let the programmer supply the parts that change.

Fold Everything

In Chapter 5, we looked at another kind of recursive list manipulation, where we applied some operation to each element of a list successively to reduce the elements to a single value. This is called a *fold* and can be used to reduce the size of the following functions:

```
%% Find the maximum of a list.
max([H|T]) -> max2(T, H).

max2([], Max) -> Max;
max2([H|T], Max) when H > Max -> max2(T, H);
max2([_|T], Max) -> max2(T, Max).
```

```
%% Find the minimum of a list.
min([H|T]) -> min2(T,H).

min2([], Min) -> Min;
min2([H|T], Min) when H < Min -> min2(T,H);
min2([_|T], Min) -> min2(T, Min).

%% Find the sum of all the elements of a list.
sum(L) -> sum(L,0).

sum([], Sum) -> Sum;
sum([H|T], Sum) -> sum(T, H+Sum).
```

To find how the `fold` function should be used, we need to determine all the common points of the actions made by these functions, as well as what is different. As mentioned earlier, we always have a reduction from a list to a single value. Consequently, our `fold` function should consider iterating only while keeping a single item—no list building is needed. We need to ignore the guards, because they exist in only some of these functions, not all of them. The guards will need to be included in the function that we pass to `fold`. In this regard, our `fold` function will probably look a lot like `sum`.

A subtle element of all three functions is that every function needs to have an initial value to start counting with. In the case of `sum/2`, we use 0, as we're doing addition, and given `X = X + 0`, the value is neutral, so we can't mess up the calculation by starting there. If we were doing multiplication, we would use 1 given `X = X * 1`.

The functions `min/1` and `max/1` can't have a default starting value. If the list were only negative numbers and we started at 0, the answer would be wrong. So we need to use the first element of the list as a starting point. Sadly, we can't always decide the starting value this way, so we'll leave that decision to the programmer.

By taking all these elements, we can build the following abstraction:

```
fold(_, Start, []) -> Start;
fold(F, Start, [H|T]) -> fold(F, F(H,Start), T).
```

Let's try it.

```
6> c(hhfuns).
{ok, hhfuns}
7> [H|T] = [1,7,3,5,9,0,2,3].
[1,7,3,5,9,0,2,3]
8> hhfuns:fold(fun(A,B) when A > B -> A; (_,B) -> B end, H, T).
9
9> hhfuns:fold(fun(A,B) when A < B -> A; (_,B) -> B end, H, T).
0
```

```
10> hhfuns:fold(fun(A,B) -> A + B end, 0, lists:seq(1,6)).
21
```

Pretty much any function you can think of that reduces lists to one element can be expressed as a fold.

Strangely enough, you can represent an accumulator as a single element (or a single variable), and an accumulator can be a list. Therefore, we can use a fold to build a list. This means folding is universal in the sense that you can implement pretty much any other recursive function on lists with a fold, even maps and filters, like so:

```
reverse(L) ->
    fold(fun(X,Acc) -> [X|Acc] end, [], L).

map2(F,L) ->
    reverse(fold(fun(X,Acc) -> [F(X)|Acc] end, [], L)).

filter2(Pred, L) ->
    F = fun(X,Acc) ->
            case Pred(X) of
                true  -> [X|Acc];
                false -> Acc
            end
        end,
    reverse(fold(F, [], L)).
```

These all work in the same way as those written by hand before. How's that for powerful abstractions?

More Abstractions

Map, filters, and folds are only a few of many abstractions over lists provided by the Erlang standard library (see `lists:map/2`, `lists:filter/2`, `lists:foldl/3`, and `lists:foldr/3`). Other functions include `all/2` and `any/2`, which both take a predicate and test if all the elements return `true` or if at least one of them returns `true`, respectively.

Also available is `dropwhile/2`, which will ignore elements of a list until it finds one that fits a certain predicate. Its opposite, `takewhile/2`, will keep all elements until there is one that doesn't return `true` to the predicate. A complementary function to these is `partition/2`, which will take a list and return two lists: one that has the terms that satisfy a given predicate and one for the others.

Other frequently used list functions include `flatten/1`, `flatlength/1`, `flatmap/2`, `merge/1`, `nth/2`, `nthtail/2`, and `split/2`. You can look up all of these functions in the documentation if you want to learn more about them.

You'll also find other functions such as zipping functions (as shown in Chapter 5), unzipping functions, combinations of maps and folds, and so on. I encourage you to read the documentation on lists to see what can be done. You'll find yourself rarely needing to write recursive functions as long as you use what's already been abstracted away by smart people.

7
ERRORS AND EXCEPTIONS

There's no right place for a chapter such as this one. So far, you've seen plenty of errors but not much about the mechanisms for handling them. That's a bit because Erlang has two main paradigms: *functional* and *concurrent*. The functional subset is the one I've been explaining since the beginning of the book: referential transparency, recursion, higher-order functions, and so on. The concurrent subset is the one that makes Erlang famous: actors, thousands and thousands of concurrent processes, supervision trees, and more.

Although Erlang includes a few ways to handle errors in functional code, most of the time you'll be told to just let it crash. The error-handling mechanisms are in the concurrent part of the language. But because it's essential to understand the

functional part before moving on to the concurrent part, this chapter covers only the functional subset of the language. If we are to manage errors, we must first understand them.

A Compilation of Errors

There are many kinds of errors: compile-time errors, logical errors, and runtime errors. First, let's look at compile-time errors.

Compile-Time Errors

Compile-time errors are often syntactic mistakes. Check your function names, the tokens in the language (such as brackets, parentheses, periods, and commas), the arity of your functions, and so on.

Here's a list of some of the common compile-time error messages and potential resolutions in case you encounter them:

`module.beam: Module name 'madule' does not match file name 'module'`
The module name you've entered in the `-module` attribute doesn't match the filename.

`./module.erl:2: Warning: function some_function/0 is unused`
You have not exported a function, or the place where it's used has the wrong name or arity. It's also possible you've written a function that is no longer needed. Check your code!

`./module.erl:2: function some_function/1 undefined`
The function does not exist. You've written the wrong name or arity either in the `-export` attribute or when declaring the function. This error is also output when the given function could not be compiled, usually because of a syntax error like forgetting to end a function with a period.

`./module.erl:5: syntax error before: 'SomeCharacterOrWord'`
This happens for a variety of reasons. Common causes are unclosed parentheses, tuples, or wrong expression termination (like closing the last branch of a `case` with a comma). Other reasons include the use of a reserved atom in your code and Unicode characters not being converted correctly between different encodings (I've seen it happen!).

`./module.erl:5: syntax error before:`
This message is certainly not as descriptive as the previous one. It usually comes up when your line termination is not correct. This is a specific case of the previous error, so just keep an eye out.

`./module.erl:5: Warning: this expression will fail with a 'badarith' exception`
: Erlang is all about dynamic typing, but remember that the types are strong. In this case, the compiler is smart enough to find that one of your arithmetic expressions will fail (say, `llama + 5`). It won't find type errors much more complex than that, though.

`./module.erl:5: Warning: variable 'Var' is unused`
: You declared a variable and never used it. This might be a bug with your code, so double-check what you have written. Otherwise, you might want to switch the variable name to `_`, or just prefix it with an underscore if you feel the name helps make the code readable.

`./module.erl:5: Warning: a term is constructed, but never used`
: In one of your functions, you're doing something such as building a list, or declaring a tuple or an anonymous function without ever binding it to a variable or returning it. This warning tells you that you're doing something useless or have made some mistake.

`./module.erl:5: head mismatch`
: It's possible your function has more than one head, and each of them has a different arity. Don't forget that different arity means different functions, and you can't interleave function declarations that way. Similarly, this error is raised when you insert a function definition between the head clauses of another function.

`./module.erl:5: Warning: this clause cannot match because a previous clause at line 4 always matches`
: A function defined in the module has a specific clause defined after a catchall one. As such, the compiler can warn you that you'll never even need to go to the other branch.

`./module.erl:9: variable 'A' unsafe in 'case' (line 5)`
: You're using a variable declared within one of the branches of a `case ... of` outside of it. This is considered unsafe. If you want to use such variables, you're better off doing `MyVar = case ... of`.

This covers most of the errors you'll get at compile-time at this point. There aren't too many, and most of the time, the hardest part is finding which error caused a huge cascade of errors listed against other functions. It is better to resolve compiler errors in the order they were reported to avoid being misled by errors that may not actually be errors at all.

No, YOUR Logic Is Wrong!

Logical errors are the hardest kind of errors to find and debug. They're most likely errors coming from the programmer: branches of conditional statements such as `if`s and `case`s that don't consider all the cases, using a multiplication that should have been a division, and so on. They do not make your programs crash, but can lead to unseen bad data or your program working in an unintended manner.

You're most likely on your own when it comes to dealing with logical errors, but Erlang has many facilities to help you, such as test frameworks, the TypEr and Dialyzer tools, and a debugger and tracing module. Testing your code is likely your best defense. Sadly, there are enough of these kinds of errors in every programmer's career to write a few dozen books about them. Here, we'll focus on those that make your programs crash, because it happens right there and won't bubble up to 50 levels to search through. Note that this is pretty much the origin of the "let it crash" ideal I've mentioned previously.

Runtime Errors

Runtime errors are pretty destructive in the sense that they crash your code. While Erlang has ways to deal with them, recognizing these errors is always helpful. We'll look at some common runtime errors and examples of code that generate them.

Function Clause Errors

The most likely reasons you'll run into a function clause error is when you fail all guard clauses of a function or fail all pattern matches, as in this example:

```
1> lists:sort([3,2,1]).
[1,2,3]
2> lists:sort(fffffff).
** exception error: no function clause matching lists:sort(fffffff) (lists.erl, line 414)
```

Case Clause Errors

Case clause errors occur when you've forgotten a specific case, sent in the wrong kind of data, or need a catchall clause. Here's an example:

```
3> case "Unexpected Value" of
3>    expected_value -> ok;
3>    other_expected_value -> 'also ok'
3> end.
** exception error: no case clause matching "Unexpected Value"
```

If Clause Errors

If clause errors are similar to case clause errors. They arise when Erlang cannot find a branch that evaluates to `true`.

```
4> if 2 > 4 -> ok;
4>    0 > 1 -> ok
4> end.
** exception error: no true branch found when evaluating an if expression
```

Making sure you consider all cases or adding the catchall `true` clause might be what you need.

Bad Match Errors

Bad match errors happen whenever pattern matching fails. This most likely means you're trying to do impossible pattern matches (such as the following), trying to bind a variable for the second time, or just using anything that isn't equal on both sides of the `=` operator (which is pretty much what makes rebinding a variable fail!).

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

Note that this error sometimes happens because the programmer believes that a variable of the form `_MyVar` is the same as `_`. Variables with an underscore are normal variables, except the compiler won't complain if they're not used. It is not possible to bind them more than once.

Bad Argument Errors

Bad argument errors are similar to function clause errors, as they are about calling functions with incorrect arguments.

```
6> erlang:binary_to_list("heh, already a list").
** exception error: bad argument
     in function  binary_to_list/1
        called as binary_to_list("heh, already a list")
```

The main difference here is that this error is usually triggered by the programmer after validating the arguments from within the function, outside of the guard clauses. It is also the error of choice thrown by BIFs or any other function written in C. I'll show you how to raise such errors in "Raising Exceptions" on page 93.

Undefined Function Errors

An undefined function error happens when you call a function that doesn't exist.

```
7> lists:random([1,2,3]).
** exception error: undefined function lists:random/1
```

Make sure the function is exported from the module with the correct arity (if you're calling it from outside the module), and double-check that you typed the name of the function and the name of the module correctly.

You might also get this error message when the module is not in Erlang's search path. By default, Erlang's search path is set to be in the current directory. You can add paths to the list by using `code:add_patha("/some/path/")` or `code:add_pathz("some/path")`. If this still doesn't work, make sure you compiled the module to begin with!

Bad Arithmetic Errors

Bad arithmetic errors occur when you try to do arithmetic that doesn't exist, like divisions by zero or between atoms and numbers.

```
8> 5 + llama.
** exception error: bad argument in an arithmetic expression
     in operator  +/2
        called as 5 + llama
```

Bad Function Errors

The most frequent reason for bad function errors is when you use variables as functions, but the variable's value is not a function. The following example uses the `hhfuns` function from Chapter 6, with two atoms as functions. This doesn't work, and a bad function error is thrown.

```
9> hhfuns:add(one,two).
** exception error: bad function one
     in function  hhfuns:add/2 (hhfuns.erl, line 7)
```

Bad Arity Errors

The bad arity error is a specific case of a bad function error. It happens when you use higher-order functions, but you pass them more or fewer arguments than they can handle.

```
10> F = fun(_) -> ok end.
#Fun<erl_eval.6.13229925>
11> F(a,b).
** exception error: interpreted function with arity 1 called with two arguments
```

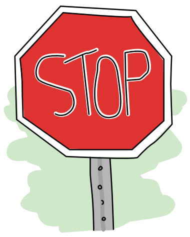
System Limit Errors

A system limit error may be raised for many reasons, including the following:

- Too many processes
- Atoms that are too long
- Too many arguments in a function
- Too many atoms
- Too many nodes connected

To get a full list and details of these errors, read the Erlang Efficiency Guide on system limits, at *http://www.erlang.org/doc/efficiency_guide/advanced.html#2265856*. Note that some of these errors are serious enough to crash the whole VM.

Raising Exceptions



In trying to monitor code's execution and protect against logical errors, it's often a good idea to provoke runtime crashes so problems will be spotted early.

There are three kinds of exceptions in Erlang: *errors*, *exits*, and *throws*. They all have different uses (kind of), as explained in the following sections.

Error Exceptions

Calling `erlang:error(Reason)` will end the execution in the current process and include a stack trace of the last functions called with their arguments when you catch the exception. These are the kind of exceptions that provoke runtime errors.

Errors are the means for a function to stop its execution when you can't expect the calling code to handle what just happened. If you get an if clause error, what can you do? Change the code and recompile—that's what you can do (other than just displaying a pretty error message).

When Not to Use Errors

An example of when *not* to use errors could be our `tree` module from Chapter 5. That module might not always be able to find a specific key in a tree when doing a lookup. In this case, it makes sense to expect the users to deal with unknown results. They could use a default value, check to insert a new one, delete the tree, or use some other approach. This is when it's appropriate to return a tuple of the form `{ok, Value}` or an atom like `undefined`, rather than raising errors.

Custom Errors

Errors aren't limited to the ones provided by Erlang. You can define your own kinds of errors, as in this example:

```
1> erlang:error(badarith).
** exception error: bad argument in an arithmetic expression
2> erlang:error(custom_error).
** exception error: custom_error
```

Here, `custom_error` is not recognized by the Erlang shell, and it has no custom translation, such as "bad argument in ...," but it's usable in the same way and can be handled by the programmer in an identical manner (as discussed in "Dealing with Exceptions" on page 96).

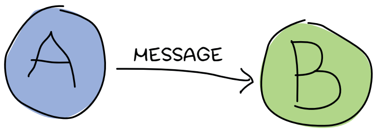
Exit Exceptions

Two kinds of exits exist in Erlang:

- *Internal exits* are triggered by calling the function `exit/1` and making the current process stop its execution.
- *External exits* are called with `exit/2` and have to do with multiple processes in the concurrent aspect of Erlang.

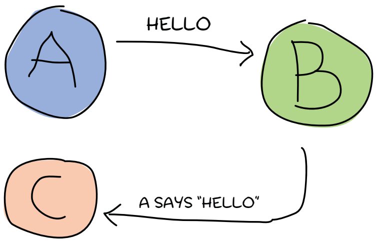
Here, we'll focus on internal exits. We will visit the external kind in Chapter 12.

Internal exits are similar to errors. In fact, historically speaking, they were the same, and only `exit/1` existed. Errors and exits have roughly the same use cases. So how do you choose which one to use? Well, the choice is not obvious. To decide when to use one or the other, you need to understand the most generic principles behind Erlang processes.

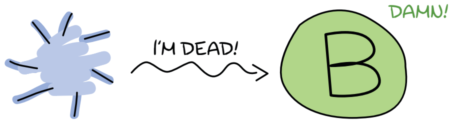


Processes can send each other messages. A process can also *listen* for messages, or wait for them.

You can also choose which messages to listen to. You can discard some messages, ignore others, give up listening after a certain time, and so on.



These basic concepts let the implementers of Erlang use a special kind of message (an *exit signal*) to communicate exceptions between processes. They act a bit like a process's last breath; they're sent right before a process dies, and the code it contains stops executing. Other processes that were listening for that specific kind of message can then know about the event and do whatever they please with it. This includes logging, restarting the process that died, and so on.



With this concept explained, the difference in using `erlang:error/1` and `exit/1` is easier to understand. While both can be used in an extremely similar manner, the real difference is intent. Is what you have simply an error, or is it a condition worthy of killing the current process? This point is made stronger by the fact `erlang:error/1` returns a stack trace and `exit/1` doesn't. If you had a pretty large stack trace or a lot of arguments to the current function, copying the exit message to every listening process would mean copying the data. In some cases, this could become impractical.

Throw Exceptions

A throw is a class of exceptions used for cases that the programmer can be expected to handle. Unlike exits and errors, throws don't really carry any "crash that process!" intent behind them but rather control flow.

To throw an exception, the syntax is as follows:

```
1> throw(permission_denied).
** exception throw: permission_denied
```

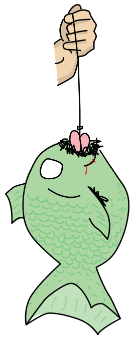
You can replace `permission_denied` with anything you want (including `'everything is fine'`, but that is not helpful, and you will lose friends).

NOTE *If you use throws while expecting the programmer to handle them, it's usually a good idea to document the throws within a module using them.*

Throws can also be used for nonlocal returns when in deep recursion. An example of this is the `ssl` module, which uses `throw/1` as a way to push `{error, Reason}` tuples back to a top-level function. That function then simply returns that tuple to the user. This lets the implementer write code only for the successful cases and have one function deal with the exceptions on top of it all.

Another example of using throws can be found in the `array` module, where there is a lookup function that can return a user-supplied default value if it can't find the element needed. When the element can't be found, the value `default` is thrown as an exception, and the top-level function handles that and replaces it with the user-supplied default value. This keeps the programmer of the module from needing to pass the default value as a parameter of every function of the lookup algorithm, again, focusing only on the successful cases.

As a rule of thumb, try to limit the use of your throws for nonlocal returns to a single module in order make it easier to debug your code. This approach will also let you change the innards of your module without requiring changes in its interface.



Dealing with Exceptions

As I've mentioned, throws, errors, and exits can be handled. The way to do this is by using a `try ... catch` expression.

A `try ... catch` is a way to evaluate an expression while letting you handle the successful case as well as the errors encountered. Here's the general syntax for such an expression:

```
try Expression of
    SuccessfulPattern1 [Guards] ->
        Expression1;
    SuccessfulPattern2 [Guards] ->
        Expression2
catch
    TypeOfError:ExceptionPattern1 ->
        Expression3;
    TypeOfError:ExceptionPattern2 ->
        Expression4
end.
```

NOTE *In the syntax shown here, the brackets around* `[Guards]` *only denote that the guards are optional. There is no need to put them in a list.*

The *`Expression`* between `try` and `of` is said to be *protected*. This means that any kind of exception happening within that call will be caught.

The patterns and expressions between the `try ... of` and `catch` behave in exactly the same manner as a `case ... of`. They are not protected, and allow pattern matching, variable bindings, and guards.

In the `catch` part, you can replace *`TypeOfError`* with `error`, `throw`, or `exit`, for each respective exception type. If no type is provided, `throw` is assumed. So let's put this in practice.

Handling Different Types of Exceptions

We'll start by creating a module named `exceptions` (we're going for simplicity here).

```
-module(exceptions).
-compile(export_all).
```

```
throws(F) ->
    try F() of
        _ -> ok
    catch
        Throw -> {throw, caught, Throw}
    end.
```

We can compile it and try it with different kinds of exceptions.

```
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
     in function  exceptions:throws/1 (exceptions.erl, line 5)
```

As you can see, this `try ... catch` is receiving only throws. This is because when no type is specified, a throw is assumed. We can add functions with `catch` clauses of each type.

```
errors(F) ->
    try F() of
        _ -> ok
    catch
        error:Error -> {error, caught, Error}
    end.

exits(F) ->
    try F() of
        _ -> ok
    catch
        exit:Exit -> {exit, caught, Exit}
    end.
```

Let's try this version.

```
4> c(exceptions).
{ok,exceptions}
5> exceptions:errors(fun() -> erlang:error("Die!") end).
{error,caught,"Die!"}
6> exceptions:exits(fun() -> exit(goodbye) end).
{exit,caught,goodbye}
```

The next example on the menu shows how to combine all types of exceptions in a single `try ... catch`. We'll first declare a function to generate all the exceptions we need.

```
sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
    try Attack() of
        _ -> "None shall pass."
    catch
        throw:slice -> "It is but a scratch.";
        error:cut_arm -> "I've had worse.";
        exit:cut_leg -> "Come on you pansy!";
        _:_ -> "Just a flesh wound."
    end.
```

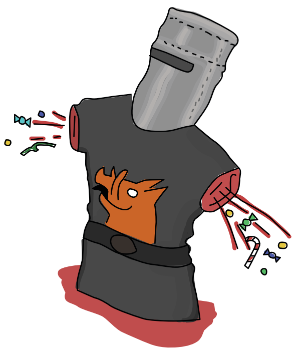
Here, `is_function/2` is a BIF that makes sure the variable `Attack` is a function of arity 0. Then we add this line for good measure:

```
talk() -> "blah blah".
```

And now for something completely different.

```
7> c(exceptions).
{ok,exceptions}
8> exceptions:talk().
"blah blah"
9> exceptions:black_knight(fun exceptions:talk/0).
"None shall pass."
10> exceptions:black_knight(fun() -> exceptions:sword(1) end).
"It is but a scratch."
11> exceptions:black_knight(fun() -> exceptions:sword(2) end).
"I've had worse."
12> exceptions:black_knight(fun() -> exceptions:sword(3) end).
"Come on you pansy!"
13> exceptions:black_knight(fun() -> exceptions:sword(4) end).
"Just a flesh wound."
14> exceptions:black_knight(fun() -> exceptions:sword(5) end).
"Just a flesh wound."
```

The expression on line 9 demonstrates normal behavior for the black knight, when normal function execution happens. Each line that follows that one demonstrates pattern matching on exceptions according to their class (`throw`, `error`, or `exit`) and the reason associated with them (`slice`, `cut_arm`, or `cut_leg`).



Lines 13 and 14 show a catchall clause for exceptions. You need to use the _:_ pattern to make sure to catch any exception from any type. In practice, you should be careful when using catchall patterns. Try to protect your code from what you can handle but not more. Erlang has other facilities in place to take care of the rest.

After the Catch

You can also add a clause after a `try ... catch` that will always be executed, as follows:

```
try Expression of
    Pattern -> Expr1
catch
    Type:Exception -> Expr2
after
    Expr3
end
```

This is equivalent to the `finally` block in many other languages. Whether or not there are errors, the expressions inside the `after` part are guaranteed to run.

However, you cannot get any return value out of the `after` construct. Therefore, `after` is mostly used to run code with side effects. The canonical use of this approach is when you want to make sure a file you were reading gets closed, whether or not exceptions were raised.

Trying Multiple Expressions

We've covered how to handle the three classes of exceptions in Erlang with `catch` blocks. However, I've hidden information from you: It's actually possible to have more than one expression between the `try` and the `of`!

```
whoa() ->
    try
        talk(),
        _Knight = "None shall pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    of
        tequila -> "Hey, this worked!"
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

By calling `exceptions:whoa()`, we'll get the obvious `{caught, throw, up}` because of `throw(up)`. So yeah, it's possible to have more than one expression between `try` and `of`.

What `exceptions:whoa/0` highlighted that you might not have noticed is that when we use many expressions in this manner, we might not always care what the return value is. So, the `of` part becomes a bit useless. Well, good news—you can just give it up:

```
im_impressed() ->
    try
        talk(),
        _Knight = "None shall pass!",
        _Doubles = [N*2 || N <- lists:seq(1,100)],
        throw(up),
        _WillReturnThis = tequila
    catch
        Exception:Reason -> {caught, Exception, Reason}
    end.
```

And now it's a bit leaner!

PROTECTING THE RIGHT THING

The protected part of an exception can't be tail recursive. The VM must always keep a reference there in case there's an exception popping up. Because the `try ... catch` construct without the `of` part has nothing but a protected part, calling a recursive function from there might be dangerous for programs that are supposed to run for a long time (which is Erlang's niche). After enough iterations, you'll run out of memory, or your program will get slower. When you put your recursive calls between the `of` and `catch`, they are not in a protected part, and you will benefit from last call optimization (discussed in Chapter 5). However, this effect is canceled if you use `after` in your `try` expression, as it needs to run *after* anything else, and thus needs to keep track of where it is in the list of function calls.

Some people use `try ... of ... catch` rather than `try ... catch` by default to avoid unexpected behaviors of that kind, except for obviously nonrecursive code with a result they don't care about. You're most likely able to make your own decision on what to do!

Wait, There's More!

As if the preceding constructs weren't already enough to put Erlang on par with most languages, it has yet another error-handling construct. This construct is defined as the keyword `catch` and basically captures all types of exceptions on top of the good results. It's a bit of a weird one because it displays a different representation of exceptions. Here's an example:

```
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{{'EXIT',{badarith,[{erlang,'/',[1,0],[]},
                    {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                    {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                    {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                    {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                    {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
4> catch 2+2.
4
```

As you can see, the throws remain the same, but exits and errors are both represented as `{'EXIT', Reason}`. That's consequent to errors being bolted to the language after exits (the Erlang implementers kept a similar representation for backward compatibility).

Let's try another example.

```
5> catch doesnt:exist(a,4).
{'EXIT',{undef,[{doesnt,exist,[a,4],[]},
                {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
```

The type of error is `undef`, which means the function you called is not defined.

The list immediately after the type of error is a stack trace. Here's how to read the stack trace:

- The tuple on top of the stack trace represents the last function to be called (`{Module, Function, Arguments}`). That's your undefined function.
- The tuples after that are the functions called before the error. This time, they're of the form `{Module, Function, Arity, Details}`.
- The `Details` field is a list of tuples containing the filename and the line within the file. In this case, the files are `erl_eval.erl` and `shell.erl` because they're in charge of interpreting the code you input in the Erlang shell.

That's all there is to it, really.

NOTE *Before the R15B release, Erlang didn't have the* `Details` *part of stack traces. For two decades, Erlang programmers found the origin of errors by using short functions and a strong sense of deduction.*

You can also manually get a stack trace by calling `erlang:get_stacktrace/0` in the process that crashed.

You'll often see `catch` written in the following manner (we're still in *exceptions.erl*):

```
catcher(X,Y) ->
    case catch X/Y of
        {'EXIT', {badarith,_}} -> "uh oh";
        N -> N
end.
```

And as expected, here's what happens when you run this:

```
6> c(exceptions).
{ok,exceptions}
7> exceptions:catcher(3,3).
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```

This sounds like a compact and easy way to catch exceptions, but there are a few problems with catch. This example shows one of them:

```
10> X = catch 4+2.
* 1: syntax error before: 'catch'
10> X = (catch 4+2).
6
```

We would expect the first case to behave exactly like the second one. Yet, it looks like Erlang can't cope with the way we declared things. That's because of the operator precedence defined by the language. The catch conflicts with =, and the only way to keep them from clashing is to wrap catch in parentheses. That's not exactly intuitive, given that most expressions do not need to be wrapped in parentheses this way.

Another problem with catch is that you can't see the difference between what looks like the underlying representation of an exception and a real exception, as in this example:

```
11> catch erlang:boat().
{'EXIT',{undef,[{erlang,boat,[],[]},
                {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
12> catch exit({undef,[{erlang,boat,[],[]},
12>                 {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
12>                 {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
12>                 {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
12>                 {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
12>                 {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}).
{'EXIT',{undef,[{erlang,boat,[],[]},
                {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
                {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,360}]},
                {shell,exprs,7,[{file,"shell.erl"},{line,668}]},
                {shell,eval_exprs,7,[{file,"shell.erl"},{line,623}]},
                {shell,eval_loop,3,[{file,"shell.erl"},{line,608}]}]}}
```

Also, you can't know the difference between an error and an actual exit, as both results are identical. You could also have used throw/1 to generate the preceding exception. In fact, a throw/1 in a catch might also be problematic in another scenario:

```
one_or_two(1) -> return;
one_or_two(2) -> throw(return).
```

And now the killer problem:

```
13> c(exceptions).
{ok,exceptions}
14> catch exceptions:one_or_two(1).
return
15> catch exceptions:one_or_two(2).
return
```

Because we're behind a catch, we can never know if the function threw an exception or it returned an actual value! This might not happen a whole lot in practice, but it's still a wart big enough to have warranted the addition of the try ... catch construct in the Erlang/OTP R10B release.

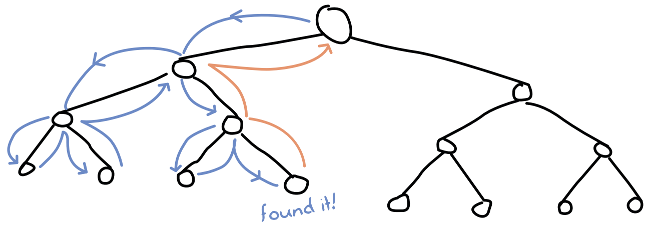
Try a try in a Tree

To put exceptions in practice, we'll do a little exercise requiring us to dig out our tree module from Chapter 5. We're going to add a function that lets us do a lookup in the tree to find out if a value is already present. Because the tree is ordered by its keys, and in this case we do not care about the keys, we'll need to traverse the whole thing until we find the value.

The traversal of the tree will be roughly similar to what we did in tree:lookup/2, except this time, we will always search down both the left branch and then the right branch. To write the function, you'll just need to remember that a tree node is either {node, {*Key*, *Value*, *NodeLeft*, *NodeRight*}} or {node, 'nil'} when empty. With this in mind, we can write a basic implementation without exceptions:

```
%% looks for a given value 'Val' in the tree
has_value(_, {node, 'nil'}) ->
    false;
has_value(Val, {node, {_, Val, _, _}}) ->
    true;
has_value(Val, {node, {_, _, Left, Right}}) ->
    case has_value(Val, Left) of
        true -> true;
        false -> has_value(Val, Right)
    end.
```

The problem with this implementation is that every node of the tree we branch at must test for the result of the previous branch.

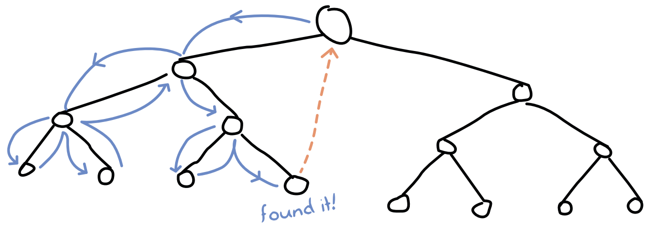


This is a bit annoying. With the help of throws, we can make something that will require fewer comparisons.

```
has_value(Val, Tree) ->
    try has_value1(Val, Tree) of
        false -> false
    catch
        true -> true
    end.

has_value1(_, {node, 'nil'}) ->
    false;
has_value1(Val, {node, {_, Val, _, _}}) ->
    throw(true);
has_value1(Val, {node, {_, _, Left, Right}}) ->
    has_value1(Val, Left),
    has_value1(Val, Right).
```

The execution of this code is similar to the previous version, except that we never need to check for the return value—we don't care about it at all. In this version, only a throw means the value was found. When this happens, the tree evaluation stops, and it falls back to the `catch` on top. Otherwise, the execution keeps going until the last `false` is returned, and that's what the user sees.



Of course, this implementation is longer than the previous one. However, it is possible to gain in speed and clarity by using nonlocal returns with a throw, depending on the operations you're doing. The current example is a simple comparison, and there's not much to see, but the practice still makes sense with more complex data structures and operations.

That being said, we're probably ready to solve real problems in sequential Erlang.