Programming Erlang

Software for a Concurrent World

Second Edition

Joe Armstrong

Edited by Susannah Davidson Pfalzer

Early Praise for
Programming Erlang, Second Edition

This second edition of Joe's seminal *Programming Erlang* is a welcome update, covering not only the core language and framework fundamentals but also key community projects such as rebar and cowboy. Even experienced Erlang programmers will find helpful tips and new insights throughout the book, and beginners to the language will appreciate the clear and methodical way Joe introduces and explains key language concepts.

➤ **Alexander Gounares**
   Former AOL CTO, advisor to Bill Gates, and founder/CEO of Concurix Corp.

A gem; a sensible, practical introduction to functional programming.

➤ **Gilad Bracha**
   Coauthor of the Java language and Java Virtual Machine specifications, creator of the Newspeak language, member of the Dart language team

Programming Erlang is an excellent resource for understanding how to program with Actors. It's not just for Erlang developers, but for anyone who wants to understand why Actors matters and why they are such an important tool in building reactive, scalable, resilient, and event-driven systems.

➤ **Jonas Bonér**
   Creator of the Akka Project and the AspectWerkz Aspect-Oriented Programming (AOP) framework, co-founder and CTO of Typesafe

Programming Erlang, Second Edition

Software for a Concurrent World

Joe Armstrong

Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Contents

Part III — Concurrent and Distributed Programs

Part IV — Programming Libraries and Frameworks

Part V — Building Applications

Introduction

New hardware is increasingly parallel, so new programming languages must support concurrency or they will die.

> "The way the processor industry is going is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it." —*Steve Jobs, Apple* [1]

Well, Steve was wrong; we do know how to program multicores. We program them in Erlang, and many of our programs just go faster as we add more cores.

Erlang was designed from the bottom up to program concurrent, distributed, fault-tolerant, scalable, soft, real-time systems. Soft real-time systems are systems such as telephone exchanges, banking systems, and so on, where rapid response times are important but it's not a disaster if the odd timing deadline is missed. Erlang systems have been deployed on a massive scale and control significant parts of the world's mobile communication networks.

If your problem is concurrent, if you are building a multiuser system, or if you are building a system that evolves with time, then using Erlang might save you a lot of work, since Erlang was explicitly designed for building such systems.

> "It's the mutable state, stupid." —*Brian Goetz, Java Concurrency in Practice*

Erlang belongs to the family of *functional programming languages*. Functional programming forbids code with side effects. Side effects and concurrency don't mix. In Erlang it's OK to mutate state within an individual process but not for one process to tinker with the state of another process. Erlang has no mutexes, no synchronized methods, and none of the paraphernalia of shared memory programming.

---

1. http://bits.blogs.nytimes.com/2008/06/10/apple-in-parallel-turning-the-pc-world-upside-down/

Processes interact by one method, and one method only, by exchanging messages. Processes share no data with other processes. This is the reason why we can easily distribute Erlang programs over multicores or networks.

When we write an Erlang program, we do not implement it as a single process that does everything; we implement it as large numbers of small processes that do simple things and communicate with each other.

What's This Book About?

It's about concurrency. It's about distribution. It's about fault tolerance. It's about functional programming. It's about programming a distributed concurrent system without locks and mutexes but using only pure message passing. It's about automatically speeding up your programs on multicore CPUs. It's about writing distributed applications that allow people to interact with each other. It's about design patterns for writing fault-tolerant and distributed systems. It's about modeling concurrency and mapping those models onto computer programs, a process I call *concurrency-oriented programming*.

Who Is This Book For?

The target audience for this book ranges from the experienced Erlang programmer who wants to learn more about Erlang internals and the philosophy behind Erlang to the absolute beginner. The text has been reviewed by programmers at all levels, from expert to beginner. One of the major differences between the second and first editions has been the addition of a large amount of explanatory material especially targeted at the beginner. Advanced Erlang programmers can skip over the introductory material.

A second goal has been to demystify functional, concurrent, and distributed programming and present it in a way that is appropriate to an audience that has no prior knowledge of concurrency or functional programming. Writing functional programs and parallel programs has long been regarded as a "black art"; this book is part of an ongoing attempt to change this.

While this book assumes no specific knowledge of either functional or concurrent programming, it is addressed to somebody who already is familiar with one or two programming languages.

When you come to a new programming language, it's often difficult to think of "problems that are suitable for solution in the new language." The exercises give you a clue. These are the kind of problems that are suitably solved in Erlang.

New in This Edition

First, the text has been brought up-to-date to reflect all the changes made to Erlang since the first edition of the book was published. We now cover all official language changes and describe Erlang version R17.

The second edition has been refocused to address the needs of beginners, with more explanatory text than in the first edition. Material intended for advanced users, or that might change rapidly, has been moved to online repositories.

The programming exercises proved so popular in the first edition that exercises now appear at the end of each chapter. The exercises vary in complexity, so there's something for both beginner users and advanced users.

In several completely new chapters, you'll learn about the Erlang type system and the Dialyzer, maps (which are new to Erlang, as of R17), websockets, programming idioms, and integrating third-party code. A new appendix describes how to build a minimal stand-alone Erlang system.

The final chapter, "Sherlock's Last Case," is a new chapter that gives you an exercise in processing and extracting meaning from a large volume of text. This is an open-ended chapter, and I hope that the exercises at the end of this chapter will stimulate future work.

Road Map

You can't run until you can walk. Erlang programs are made up from lots of small sequential programs running at the same time. Before we can write concurrent code, we need to be able to write sequential code. This means we won't get into the details of writing concurrent programs until .

- Part I has a short introduction to the central ideas of concurrent programming and a whirlwind tour of Erlang.

- Part II covers sequential Erlang programming in detail and also talks about types and methods for building Erlang programs.

- Part III is the core of the book where we learn about how to write concurrent and distributed Erlang programs.

- Part IV covers the major Erlang libraries, techniques for tracing and debugging, and techniques for structuring Erlang code.

- Part V covers applications. You'll learn how to integrate external software with the core Erlang libraries and how to turn your own code into open

source contributions. We'll talk about programming idioms and how to program multicore CPUs. And finally, Sherlock Holmes will analyze our thoughts.

At the end of each chapter, you'll find a selection of programming exercises. These are to test your knowledge of the chapter and to challenge you. The problems vary from easy to difficult. The most difficult problems would be suitable research projects. Even if you don't try to solve all the problems, just thinking about the problems and how you would solve them will enhance your understanding of the text.

The Code in This Book

Most of the code snippets come from full-length, running examples that you can download.[2] To help you find your way, if a code listing in this book can be found in the download, there'll be a bar above the snippet (just like the one here):

```erlang
shop1.erl
-module(shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([])            -> 0.
```

This bar contains the path to the code within the download. If you're reading the ebook version of this book and your ebook reader supports hyperlinks, you can click the bar, and the code should appear in a browser window.

Help! It Doesn't Work

Learning new stuff is difficult. You will get stuck. When you get stuck, rule 1 is to not silently give up. Rule 2 is to get help. Rule 3 is to ask Sherlock.

Rule 1 is important. There are people who have tried Erlang, gotten stuck and given up, and not told anybody. If we don't know about a problem, we can't fix it. End of story.

The best way to get help is to first try Google; if Google can't help, send mail to the Erlang mailing list.[3] You can also try #erlounge or #erlang at irc.freenode.net for a faster response.

Sometimes the answer to your question might be in an old posting to the Erlang mailing list but you just can't find it. In Chapter 27, *Sherlock's Last*

---

2. http://www.pragprog.com/titles/jaerlang2/source_code
3. erlang-questions@erlang.org

Case, on page 457, there's a program you can run locally that can perform sophisticated searches on all the old postings to the Erlang mailing list.

So, without further ado, I'll thank the good folks who helped me write this book (and the first edition), and you can skip on to Chapter 1, where we'll take a lightning tour of Erlang.

Acknowledgments

First Edition

Many people helped in the preparation of this book, and I'd like to thank them all here.

First, Dave Thomas, my editor: Dave taught me to write and subjected me to a barrage of never-ending questions. Why this? Why that? When I started the book, Dave said my writing style was like "standing on a rock preaching." He said, "I want you to talk to people, not preach." The book is better for it. Thanks, Dave.

Next, I had a little committee of language experts at my back. They helped me decide what to leave out. They also helped me clarify some of the bits that are difficult to explain. Thanks here (in no particular order) to Björn Gustavsson, Robert Virding, Kostis Sagonas, Kenneth Lundin, Richard Carlsson, and Ulf Wiger.

Thanks also to Claes Vikström who provided valuable advice on Mnesia, to Rickard Green who gave information on SMP Erlang, and to Hans Nilsson for the stemming algorithm used in the text-indexing program.

Sean Hinde and Ulf Wiger helped me understand how to use various OTP internals, and Serge Aleynikov explained active sockets to me so that I could understand.

Helen Taylor (my wife) proofread several chapters and provided hundreds of cups of tea at appropriate moments. What's more, she put up with my rather obsessive behavior for seven months. Thanks also to Thomas and Claire; and thanks to Bach and Handel, my cats Zorro and Daisy, and my Sat Nav Doris, who helped me stay sane, purred when stroked, and got me to the right addresses.

Finally, to all the readers of the beta book who filled in errata requests: I have cursed you and praised you. When the first beta went out, I was unprepared for the entire book to be read in two days and for you to shred every page with your comments. But the process has resulted in a much better book

than I had imagined. When (as happened several times) dozens of people said, "I don't understand this page," then I was forced to think again and rewrite the material concerned. Thanks for your help, everybody.

Second Edition

First, my new editor, Susannah Pfalzer, helped a lot in suggesting new ways to reorganize and refocus the book. It was great working with you; you've taught me a lot.

Kenneth Lundin and the guys in the OTP group worked hard to deliver the new language features described in the second edition.

Many readers of the first edition provided feedback on things they didn't understand, so I hope these are now rectified.

The design of maps is inspired by the work of Richard A. O'Keefe (who called them frames). Richard has championed the cause of frames on the Erlang mailing list for many years. Thanks, Richard, for all your comments and suggestions.

Kostis Sagonas provided lots of helpful feedback on the treatment of the type system.

I'd also like to thank Loïc Hoguin for his permission to use some examples from the cowboy web server from Nine Nines and the guys from Basho who wrote the code for BitLocker. I'd also like to thank Dave Smith for his work with rebar.

A number of people helped me by reviewing various drafts of the second edition. I'd like to thank all of them; they made this a better book. So, thanks to Erik Abefelt, Paul Butcher, Mark Chu-Carroll, Ian Dees, Henning Diedrich, Jeremy Frens, Loïc Hoguin, Andy Hunt, Kurt Landrus, Kenneth Lundin, Evan Miller, Patrik Nyblom, Tim Ottinger, Kim Shrier, and Bruce Tate for your help.

Helen Taylor (Twitter @mrsjoeerl) made countless cups of tea and cheered me up when I thought the book would never get finished.

Gustav Mahler, Sergei Rachmaninoff, Richard Wagner, and George Frideric Handel composed music (and Bob Dylan and few other guys…) that I played in the background while writing much of this book.

Part I

Why Erlang?

We introduce concurrency and talk about the difference between concurrency and parallelism. You'll learn about the benefits of writing concurrent programs and take a whirlwind tour of Erlang, introducing the main features of the language.

Introducing Concurrency

Let's forget about computers for a moment; I'm going to look out of my window and tell you what I see.

I see a woman taking a dog for a walk. I see a car trying to find a parking space. I see a plane flying overhead and a boat sailing by. All these things happen *in parallel*. In this book, we will learn how to describe parallel activities as sets of communicating parallel processes. We will learn how to write *concurrent programs*.

In everyday language, words like *concurrent*, *simultaneous*, and *parallel* mean almost the same thing. But in programming languages, we need to be more precise. In particular, we need to distinguish between concurrent and parallel programs.

If we have only a single-core computer, then we can never run a parallel program on it. This is because we have one CPU, and it can do only one thing at a time. We can, however, run concurrent programs on a single-core computer. The computer time-shares between the different tasks, maintaining the illusion that the different tasks run in parallel.

In the following sections, we'll start with some simple concurrency modeling, move on to see the benefits of solving problems using concurrency, and finally look at some precise definitions that highlight the differences between concurrency and parallelism.

1.1 Modeling Concurrency

We'll start with a simple example and build a concurrent model of an everyday scene. Imagine I see four people out for a walk. There are two dogs and a large number of rabbits. The people are talking to each other, and the dogs want to chase the rabbits.

To simulate this in Erlang, we'd make four modules called person, dog, rabbit, and world. The code for person would be in a file called person.erl and might look something like this:

```
-module(person). -
export([init/1]).

init(Name) -> ...
```

The first line, -module(person)., says that this file contains code for the module called person. This should be the same as the filename (excluding the .erl file-name extension). The module name *must* start with a small letter. Technically, the module name is an *atom*; we'll talk more about atoms in Section 3.5, *Atoms*, on page 33.

Following the module declaration is an *export declaration*. The export declarations tells which functions in the module can be called from *outside* the module. They are like *public* declarations in many programming languages. Functions that are not in an export declaration are private and cannot be called from outside the module.

The syntax -export([init/1]). means the function init with one argument (that's what /1 means; it does not mean divide by one) can be called from outside the module. If we want to export several functions, we'd use this syntax:

```
-export([FuncName1/N1, FuncName2/N2, .....]).
```

The square brackets […] mean "list of," so this declaration means we want to export a list of functions from the module.

We'd write similar code for dog and rabbit.

Starting the Simulation

To start the program, we'll call world:start(). This is defined in a module called world, which begins like this:

```
-module(world). -
export([start/0]).

start() ->
    Joe      = spawn(person, init, ["Joe"]),
    Susannah = spawn(person, init, ["Susannah"]),
    Dave     = spawn(person, init, ["Dave"]),
    Andy     = spawn(person, init, ["Andy"]),
    Rover    = spawn(dog,    init, ["Rover"]),
    ...
    Rabbit1  = spawn(rabbit, init, ["Flopsy"]),
  ...
```

spawn is an Erlang primitive that creates a concurrent process and returns a process identifier. spawn is called like this:

```
spawn(ModName, FuncName, [Arg1, Arg2, ..., ArgN])
```

When spawn is evaluated, the Erlang runtime system creates a new process (not an operating system process but a lightweight process that is managed by the Erlang system). Once the process has been created, it starts evaluating the code specified by the arguments. ModName is the name of the module that has the code we want to execute. FuncName is the name of the function in the module, and [Arg1, Arg2, …] is a list containing the arguments to the function that we want to evaluate. Thus, the following call means start a process that evaluates the function person:init("Joe"):

```
spawn(person, init, ["Joe"])
```

The return value of spawn is a *process identifier* (PID) that can be used to interact with the newly created process.

> ### Analogy with Objects
>
> Modules in Erlang are like classes in an object-oriented programming language (OOPL), and processes are like objects (or class instances) in an OOPL.
>
> In Erlang, spawn creates a new process by running a function defined in a module. In Java, new creates a new object by running a method defined in a class.
>
> In an OOPL we can have one class but several thousand class instances. Similarly, in Erlang we can have one module but thousands or even millions of processes that execute the code in the module. All the Erlang processes execute concurrently and independently and, if we had a million-core computer, might even run in parallel.

Sending Messages

Once our simulation has been started, we'll want to send messages between the different processes in the program. In Erlang, processes share no memory and can interact only with each other by sending messages. This is exactly how objects in the real world behave.

Suppose Joe wants to say something to Susannah. In the program we'd write a line of code like this:

```
Susannah ! {self(), "Hope the dogs don't chase the rabbits"}
```

The syntax Pid ! Msg means send the message Msg to the process Pid. The self() argument in the curly brackets identifies the process sending the message (in this case Joe).

Receiving Messages

For Susannah's process to receive the message from Joe, we'd write this:

```
receive
   {From, Message} ->
      ...
end
```

When Susannah's process receives a message, the variable From will be bound to Joe so that Susannah knows who the message came from, and the variable Message will contain the message.

We could imagine extending our model by having the dogs send "woof woof rabbits" messages to each other and the rabbits sending "panic go and hide" messages to each other.

The key point we should remember here is that our programming model is based on *observation* of the real world. We have three modules (person, dog, and rabbit) because there are three types of concurrent things in our example. The world module is needed for a top-level process to start everything off. We created two dog processes because there are two dogs, and we created four people processes because there were four people. The messages in the program reflect the observed messages in our example.

Rather than extending the model, we'll stop at this point, change gears, and look at some of the characteristics of concurrent programs.

1.2   Benefits of Concurrency

Concurrent programming can be used to improve performance, to create scalable and fault-tolerant systems, and to write clear and understandable programs for controlling real-world applications. The following are some of the reasons why this is true:

Performance

Imagine you have two tasks: A, which takes ten seconds to perform, and B, which takes fifteen seconds. On a single CPU doing both, A and B will take twenty-five seconds. On a computer with two CPUs that operate independently, doing A and B will take only fifteen seconds. To achieve this performance improvement, we have to write a concurrent program.

Until recently, parallel computers were rare and expensive, but today multicore computers are commonplace. A top-end processor has sixty-four cores, and we can expect the number of cores per chip to steadily increase in the foreseeable future. If you have a suitable problem and a

computer with sixty-four cores, your program might go sixty-four times faster when run on this computer, but only if you write a concurrent program.

One of the most pressing problems in the computer industry is caused by difficulties in parallelizing legacy sequential code so it can run on a multicore computer. There is no such problem in Erlang. Erlang programs written twenty years ago for a sequential machine now just run faster when we run them on modern multicores.

Scalability

Concurrent programs are made from small independent processes. Because of this, we can easily scale the system by increasing the number of processes and adding more CPUs. At runtime the Erlang virtual machine automatically distributes the execution of processes over the available CPUs.

Fault tolerance

Fault tolerance is similar to scalability. The keys to fault tolerance are independence and hardware redundancy. Erlang programs are made up of many small independent processes. Errors in one process cannot accidentally crash another process. To protect against the failure of an entire computer (or data center), we need to detect failures in remote computers. Both process independence and remote failure detection are built into the Erlang VM.

Erlang was designed for building fault-tolerant telecommunications systems, but the same technology can be applied equally well to building fault-tolerant scalable web systems or cloud services.

Clarity

In the real world things happen in parallel, but in most programming languages things happen sequentially. The mismatch between the parallelism in the real world and the sequentiality in our programming languages makes writing real-world control problems in a sequential language artificially difficult.

In Erlang we can map real-world parallelism onto Erlang concurrency in a straightforward manner. This results in clear and easily understood code.

Now that you've seen these benefits, we'll try to add some precision to the notion of concurrency and parallelism. This will give us a framework to talk about these terms in future chapters.

1.3 Concurrent Programs and Parallel Computers

I'm going to be pedantic here and try to give precise meanings to terms such as *concurrent* and *parallel*. We want to draw the distinction between a concurrent program, which is something that could potentially run faster if we had a parallel computer, and a parallel computer that really has more than one core (or CPU).

- A *concurrent program* is a program written in a concurrent programming language. We write concurrent programs for reasons of performance, scalability, or fault tolerance.

- A *concurrent programming language* is a language that has explicit language constructs for writing concurrent programs. These constructs are an integral part of the programming language and behave the same way on all operating systems.

- A *parallel computer* is a computer that has several processing units (CPUs or cores) that run at the same time.

Concurrent programs in Erlang are made from sets of communicating sequential processes. An Erlang process is a little virtual machine that can evaluate a single Erlang function; it should not be confused with an operating system process.

To write a concurrent program in Erlang, you must identify a set of processes that will solve your problem. We call this act of identifying the processes *modeling concurrency*. This is analogous to the art of identifying the objects that are needed to write an objected-oriented program.

Choosing the objects that are needed to solve a problem is recognized as being a hard problem in object-oriented design. The same is true in modeling concurrency. Choosing the correct processes can be difficult. The difference between a good and bad process model can make or break a design.

Having written a concurrent program, we can run it on a parallel computer. We can run on a multicore computer or on a set of networked computers or in the cloud.

Will our concurrent program actually run in parallel on a parallel computer? Sometimes it's hard to know. On a multicore computer, the operating system might decide to turn off a core to save energy. In a cloud, a computation might be suspended and moved to a new computer. These are things outside our control.

We've now seen the difference between a concurrent program and a parallel computer. Concurrency has to do with software structure; parallelism has to do with hardware. Next we'll look at the difference between sequential and concurrent programming languages.

1.4 Sequential vs. Concurrent Programming Languages

Programming languages fall into two categories: sequential and concurrent. Sequential languages are languages that were designed for writing sequential programs and have no linguistic constructs for describing concurrent computations. Concurrent programming languages are languages that were designed for writing concurrent programs and have special constructs for expressing concurrency in the language itself.

In Erlang, concurrency is provided by the Erlang virtual machine and not by the operating system or by any external libraries. In most sequential programming languages, concurrency is provided as an interface to the concurrency primitives of the host operating system.

The distinction between operating system– and language-based concurrency is important because if you use operating system–based concurrency, then your program will work in different ways on different operating systems. Erlang concurrency works the same way on all operating systems. To write concurrent programs in Erlang, you just have to understand Erlang; you don't have to understand the concurrency mechanisms in the operating system.

In Erlang, processes and concurrency are the tools we can use to shape and solve our problems. This allows fine-grained control of the concurrent structure of our program, something that is extremely difficult using operating system processes.

Wrapping Up

We've now covered the central themes of this book. We talked about concurrency as a means for writing performant, scalable, and fault-tolerant software, but we did not go into any details as to how this can be achieved. In the next chapter, we'll take a whirlwind tour through Erlang and write our first concurrent program.

A Whirlwind Tour of Erlang

In this chapter, we'll build our first concurrent program. We'll make a file server. The file server has two concurrent processes; one process represents the server, and the other represents the client.

We'll start with a small Erlang subset so we can show some broad principles without getting bogged down with the details. At a minimum we have to understand how to run code in the shell and compile modules. That's all we need to know to get started.

The best way to learn Erlang is to type in the examples into a live Erlang system and see whether you can reproduce what's in this book. To install Erlang, refer to http://joearms.github.com/installing.html. We try to keep the install instructions up-to-date. This is difficult since there are many different platforms configured in many different ways. If the instructions fail or are not up-to-date, please send a mail to the Erlang mailing list, and we'll try to help.

2.1 The Shell

The Erlang shell is where you'll spend most of your time. You enter an expression, and the shell evaluates the expression and prints the result.

```
$ erl
Erlang R16B ...
Eshell V5.9  (abort with ^G)
1> 123456 * 223344.
27573156864
```

So, what happened? $ is the operating system prompt. We typed the command erl, which started the Erlang shell. The Erlang shell responds with a banner and the numbered prompt 1>. Then we typed in an expression, which was evaluated and printed. Note that each expression *must* be finished with a dot followed by a whitespace character. In this context, whitespace means a space, tab, or carriage return character.

Beginners often forget to finish expressions with the dot whitespace bit. Think of a command as an English sentence. English sentences usually end with a dot, so this is easy to remember.

The = operator

We can assign values to variables using the = operator (technically this is called *binding* the variable to a value), like this:

```
2> X = 123.
123
3> X * 2.
246
```

If we try to change the value of a variable, something strange happens.

```
4> X = 999.
** exception error: no match of right hand side value 999
```

That's the first surprise. We can't *rebind* variables. Erlang is a functional language, so once we've said X = 123, then X is 123 *forever* and cannot be changed!

Don't worry, this is a benefit, not a problem. Programs where variables can't be changed once they are set are far easier to understand than programs where the same variable can acquire many different values during the life of the program.

When we see an expression like X = 123, it looks as if it means "assign the integer 123 to the variable X," but this interpretation is incorrect. = is not an assignment operator; it's actually a *pattern matching operator*. This is described in detail in *Variable Bindings and Pattern Matching,* on page 30.

As in functional programming languages, variables in Erlang can be bound only once. Binding a variable means giving a variable a value; once it has been bound, that value cannot be changed later.

This idea might seem strange to you if you're used to imperative languages. In an imperative language, variables are really a disguised way of referring to memory addresses. An X in a program is really the address of some data item somewhere in memory. When we say X=12, we are changing the value of memory location with address X, but in Erlang, a variable X represents a value that can never be changed.

Syntax of Variables and Atoms

Note that Erlang variables start with uppercase characters. So, X, This, and A_long_name are all variables. Names beginning with lowercase letters—for example, monday or friday—are not variables but are symbolic constants called *atoms*.

If you ever see or write an expression like x = 123 (Note: x here is written with a lowercase letter, in case you missed it), it's almost certainly a mistake. If you do this in the shell, the response is immediate.

```
1> abc=123.
** exception error: no match of right hand side value 123
```

But if a line like this was buried deeply in some code, it could crash your program, so be warned. Most editors such as Emacs and the Eclipse editor will color code atoms and variables with different colors, so the difference is easy to see.

Before you read the next section, try starting the shell and entering a few simple arithmetic expressions. At this stage, if things go wrong, just quit the shell by typing Control+C followed by a (for abort) and then restart the shell from the operating system prompt.

By now you should be familiar with starting and stopping the shell and using it to evaluate simple expressions. We also saw one of the fundamental differences between a functional programming language and an imperative programming language. In a functional language, variables can't change, but in an imperative language, variables can change.

2.2 Processes, Modules, and Compilation

Erlang programs are built from a number of parallel processes. Processes evaluate functions that are defined in modules. Modules are files with the extension .erl and must be compiled before they can be run. Having compiled a module, we can evaluate the functions in the module from the shell or directly from the command line in an operating system environment.

In the next sections, we'll look at compiling modules and evaluating functions in the shell and from the OS command line.

Compiling and Running "Hello World" in the Shell

Make a file called hello.erl with the following content:

```
hello.erl
-module(hello).
-export([start/0]).

start() ->
    io:format("Hello world~n").
```

To compile and run this, we start the Erlang shell in the directory where we stored hello.erl and do the following:

```
$ erl
Erlang R16B ...
1> c(hello).
{ok,hello}
2> hello:start().
Hello world
ok
3> halt().
$
```

The command c(hello) compiles the code in the file hello.erl. {ok, hello} means the compilation succeeded. The code is now ready to be run. In line 2, we evaluated the function hello:start(). In line 3, we stopped the Erlang shell.

The advantage of working in the shell is that this method of compiling and running programs is known to work on all platforms where Erlang is supported. Working from the operating system command line may not work identically on all platforms.

Compiling Outside the Erlang Shell

Using the same code as before, we can compile and run our code from the OS command line, as follows:

```
$ erlc hello.erl
$ erl -noshell -s hello start -s init stop
Hello world
```

erlc evokes the Erlang compiler from the command line. The compiler compiles the code in hello.erl and produces an object code file called hello.beam.

The $erl -noshell … command loads the module hello and evaluates the function hello:start(). After this, it evaluates the expression init:stop(), which terminates the Erlang session.

Running the Erlang compiler (erlc) outside the Erlang shell is the preferred way of compiling Erlang code. We can compile modules inside the Erlang shell, but to do so, we first have to start the Erlang shell. The advantage of

using erlc is automation. We can run erlc inside rakefile or makefiles and automate the build process.

When you start learning Erlang, it is advisable to use the Erlang shell for everything; that way, you'll get familiar with the details of compiling and running code. More advanced users will want to automate compilation and make lesser use of the Erlang shell.

2.3 Hello, Concurrency

We've seen how to compile a simple module. But what about writing a concurrent program? The basic unit of concurrency in Erlang is the *process*. A process is a lightweight virtual machine that can communicate with other processes only by sending and receiving messages. If you want a process to do something, you send it a message and possibly wait for a reply.

The first concurrent program we'll write is a file server. To transfer files between two machines, we need two programs: a client that runs on one machine and a server that runs on a second machine. To implement this, we'll make two modules called afile_client and afile_server.

The File Server Process

The file server is implemented in a module called afile_server. Just to remind you, processes and modules are like objects and classes. The code for a process is contained in a module, and to create a process, we call the primitive spawn(...), which actually creates the process.

```
afile_server.erl
-module(afile_server).
-export([start/1, loop/1]).

start(Dir) -> spawn(afile_server, loop, [Dir]).

loop(Dir) ->
    receive
        {Client, list_dir} ->
            Client ! {self(), file:list_dir(Dir)};
        {Client, {get_file, File}} ->
            Full = filename:join(Dir, File),
            Client ! {self(), file:read_file(Full)}
    end,
    loop(Dir).
```

This code has a very simple structure. If we omit most of the detail, it looks like this:

```
loop(Dir) ->
    %% wait for a command
    receive
        Command ->
            ... do something ...
    end,
    loop(Dir).
```

This is how we write an infinite loop in Erlang. The variable Dir contains the current working directory of the file server. In the loop we wait to receive a command; when we receive a command, we obey the command and then call ourselves again to get the next command.

Just for the curious: Don't worry about the fact that the last thing we do is to call ourselves; we're not going to run out of stack space. Erlang applies a so-called tail-call optimization to the code, which means that this function will run in constant space. This is the standard way of writing a loop in Erlang. Just call yourself as the last thing you do.

Another point to note is that loop is a function that never returns. In a sequential programming language, we have to be extremely careful to avoid infinite loops; we have only one thread of control, and if this thread gets stuck in a loop, we're in trouble. In Erlang, there is no such problem. A server is just a program that services requests in an infinite loop and that runs in parallel with any other tasks that we want to perform.

Now let's stare hard at the receive statement; to remind you, it looks like this:

afile_server.erl
```
receive
    {Client, list_dir} ->
        Client ! {self(), file:list_dir(Dir)};
    {Client, {get_file, File}} ->
        Full = filename:join(Dir, File),
        Client ! {self(), file:read_file(Full)}
end,
```

This code says that if we receive the message {Client, list_dir}, we should reply with a list of files, or if we receive the message {Client, {get_file, File}}, then reply with the file. The variable Client becomes bound as part of the pattern matching process that occurs when a message is received.

This code is very compact, so it's easy to miss the details of what's going on. There are three significant points that you should note about this code.

Who to reply to

All the received messages contained the variable Client; this is the process identifier of the process that sent the request and to whom the reply should be sent.

If you want a reply to a message, you'd better say who the reply is to be sent to. This is like including your name and address in a letter; if you don't say who the letter came from, you won't ever get a reply.

Use of self()

The reply sent by the server contains the argument self() (in this case self() is the process identifier of the server). This identifier is added to the message so that the client can check that the message the client received came from the server and not some other process.

Pattern matching is used to select the message

The inside of the receive statement has two *patterns*. We just write them like this:

```
receive
    Pattern1 ->
      Actions1;
    Pattern2 ->
      Actions2 ->
    ...
end
```

The Erlang compiler and runtime system will correctly figure out how to run the appropriate code when a message is received. We don't have to write any if-then-else or switch statements to work out what to do. This is one of the joys of pattern matching, which will save you lots of work.

We can compile and test this code in the shell as follows:

```
1> c(afile_server).
{ok,afile_server}
2> FileServer = afile_server:start(".").
<0.47.0>
3> FileServer ! {self(), list_dir}.
{<0.31.0>,list_dir}
4> receive X -> X end.
{<0.47.0>,
 {ok,["afile_server.beam","processes.erl","attrs.erl","lib_find.erl",
    "dist_demo.erl","data1.dat","scavenge_urls.erl","test1.erl",
    ...]}}
```

Let's look at the details.

```
1> c(afile_server).
{ok,afile_server}
```

We compile the module afile_server contained in the file afile_server.erl. Compilation succeeds, so the return value of the "compile" function c is {ok, afile_server}.

```
2> FileServer = afile_server:start(".").
<0.47.0>
```

afile_server:start(Dir) calls spawn(afile_server, loop, [Dir]). This creates a new parallel process that evaluates the function afile_server:loop(Dir) and returns a *process identifier* that can be used to communicate with the process.

<0.47.0> is the process identifier of the file server process. It is displayed as three integers separated by periods and contained within angle brackets. *Note:* Every time you run this program, the process identifiers will change. So, the numbers like <0.47.0> will differ from session to session.

```
3> FileServer ! {self(), list_dir}.
{<0.31.0>,list_dir}
```

This sends a {self(), list_dir} message to the file server process. The return value of Pid ! Message is *defined* to be Message, so the shell prints out the value of {self(), list_dir}, which is {<0.31.0>, list_dir}. <0.31.0> is the process identifier of the Erlang shell itself; this is included in the message so that the file server knows who to reply to.

```
4> receive X -> X end.
{<0.47.0>,
{ok,["afile_server.beam","processes.erl","attrs.erl","lib_find.erl",
     "dist_demo.erl","data1.dat","scavenge_urls.erl","test1.erl",
     ...]}
```

receive X -> X end receives the reply sent by the file server. It returns the tuple {<0.47.0>, {ok, ...}. The first element in the tuple is <0.47.0>, which is the process identifier of the file server. The second argument is the return value of the function file:list_dir(Dir), which was evaluated inside the receive loop of the file server process.

The Client Code

The file server is accessed through a client module called afile_client. The main purpose of this module is to hide the details of the underlying communication protocol. The user of the client code can transfer files by calling the functions ls and get_file that are exported from the client module. This gives us the freedom to change the underlying protocols without changing the details of the client code API.

```
afile_client.erl
-module(afile_client).
-export([ls/1, get_file/2]).

ls(Server) ->
    Server ! {self(), list_dir},
    receive
        {Server, FileList} ->
            FileList
    end.

get_file(Server, File) ->
    Server ! {self(), {get_file, File}},
    receive
        {Server, Content} ->
            Content
    end.
```

If you compare the code for afile_client with afile_server, you'll see a beautiful symmetry. Where there is a send statement in the client Server ! ..., there is a receive pattern in the server, and vice versa.

```
receive
    {Client, Pattern} ->
        ...
end
```

Now we'll restart the shell, recompile everything, and show the client and server working together.

```
1> c(afile_server).
{ok,afile_server}
2> c(afile_client).
{ok,afile_client}
3> FileServer = afile_server:start(".").
<0.43.0>
4> afile_client:get_file(FileServer,"missing").
{error,enoent}
5> afile_client:get_file(FileServer,"afile_server.erl").
{ok,<<"-module(afile_server).\n-export([start/1])....}
```

The only difference between the code we ran in the shell and the previous code is that we have abstracted out the interface routines and put them into a separate module. We hide the details of the message passing between the client and server, since no other program is interested in them.

What you've seen so far is the basis of a fully blown file server, but it is not yet complete. There are a lot of details associated with starting and stopping the server, connecting to a socket, and so on, which are not covered here.

From the Erlang point of view, how we start and stop servers, connect to sockets, recover from errors, and so on, are uninteresting details. The *essence* of the problem has to do with creating parallel processes and sending and receiving messages.

In Erlang we use processes to structure the solutions to our problems. Thinking about the process structure (in other words, which processes know about each other) and thinking about the messages that are sent between processes and what information the messages contain are central to our way of thinking and our way of programming.

Improving the File Server

The file server that we have developed involves two communicating processes running on the same machine and illustrates several of the building blocks needed to write concurrent programs. In a real server, the client and server would run on different machines, so somehow we have to arrange that inter-process messages can pass not only between processes in the same Erlang node but between Erlang processes located on physically separated machines.

In Chapter 17, *Programming with Sockets,* on page 263, we'll see how to use a TCP transport layer for process communication, and in *The File Server Revisited,* on page 221, we'll see how to implement the file server directly in distributed Erlang.

In this chapter we saw how to perform some simple operations in the shell, compile a module, and create a simple concurrent program with two processes using three primitives: spawn, send, and receive.

This concludes Part I of the book. In Part II we'll go through sequential programming in a lot more detail, and we'll return to concurrent programming in Chapter 12, *Concurrent Programming,* on page 181. In the next chapter, we'll start learning sequential programming by looking at the shell, pattern matching, and the primitive Erlang data types in a lot of detail.

Exercises

Now might be a good time to check your understanding of what we have done so far.

1. Start and stop the Erlang shell.

2. Type in a few commands in the Erlang shell. Remember to finish the commands with dot whitespace.

3. Make some small modifications to hello.erl. Compile and run them in the shell. If things go wrong, abort from the Erlang shell and restart the shell.

4. Run the file client and server code. Add a command called put_file. What messages do you need to add? Find out how to consult manual pages. Consult the manual pages for the file module.

Part II

Sequential Programming

In this part, you'll learn to write sequential Erlang programs. We'll cover all of sequential Erlang and also talk about ways of compiling and running your programs and using the type system to describe the types of Erlang functions and to statically detect programming errors.

Basic Concepts

This chapter sets the scene for Erlang programming. All Erlang programs, parallel or sequential, use pattern matching, single-assignment variables, and the basic types that Erlang uses to represent data.

In this chapter, we'll use the Erlang shell to experiment with the system and see how it behaves. We'll start with the shell.

3.1 Starting and Stopping the Erlang Shell

On a Unix system (including Mac OS X), you start the Erlang shell from a command prompt; on a Windows system, click the Erlang icon in the Windows Start menu.

```
$ erl
Erlang R16B (erts-5.10.1) [source] [64-bit] [smp:4:4] [async-threads:10]
    [hipe] [kernel-poll:false]
Eshell V5.10.1  (abort with ^G)
1>
```

This is the Unix command to start the Erlang shell. The shell responds with a banner telling you which version of Erlang you are running. The easiest way to stop the system is just to press Ctrl+C (Windows Ctrl+Break) followed by a (short for abort), as follows:

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
$
```

Typing a will immediately stop the system and may result in some data corruption. For a controlled shutdown, you can type q() (short for quit).

```
1> q().
ok
$
```

This stops the system in a controlled manner. All open files are flushed and closed, databases are stopped (if running), and all applications are closed down in an ordered manner. q() is a shell alias for the command init:stop().

To immediately stop the system, evaluate the expression erlang:halt().

If none of these methods works, read *Stopping Erlang, on page 169*.

Evaluating Commands in the Shell

When the shell is ready to accept an expression, it prints the command prompt.

```
1> X = 20.
20
```

You'll see that this dialogue starts at command 1 (that is, the shell printed 1>). This means we have started a new Erlang shell. Every time you see a dialogue in this book that starts with 1>, you'll have to start a new shell if you want to *exactly* reproduce the examples in the book. When an example starts with a prompt number that is greater than 1, this implies that the shell session is continued from the previous examples, so you don't have to start a new shell.

At the prompt we typed an expression. The shell evaluated the expression and printed the result.

```
2> X + 20. % and this is a comment
40
```

The shell printed out another prompt, this time for expression 2 (because the command number increases each time a new expression is entered).

In line 2, the percent (%) character indicates the start of a comment. All the text from the percent sign to the end of line is treated as a comment and is ignored by the shell and the Erlang compiler.

Now might be a good time to experiment with the shell. Enter the expressions in the examples exactly as they appear in text, and check you get the same results as in the book. Some command sequences can be entered several times, but others can be only once since they depend upon previous commands. If anything goes wrong, the best approach is to abort the shell and try again with a freshly started shell.

Things That Can Go Wrong

You can't type everything you read in this book into the shell. *The syntactic forms in an Erlang module are not expressions and are not understood by the*

shell. In particular, you can't enter annotations into the shell; these are things that start with a hyphen (such as -module, -export, and so on).

Something that might have gone wrong is that you've started to type something that is quoted (that is, starts with a single or double quote mark) but have not yet typed a matching closing quote mark that should be the same as the open quote mark.

If any of these happen, then the best thing to do is type an extra closing quote, followed by dot whitespace to complete the command.

Advanced: You can start and stop multiple shells. For details, see *The Shell Isn't Responding,* on page 170.

Command Editing in the Erlang Shell

The Erlang shell contains a built-in line editor. It understands a subset of the line-editing commands used in the popular Emacs editor. Previous lines can be recalled and edited in a few keystrokes. The available commands are shown next (note that ^Key means you should press Ctrl+*Key*):

Command	Description
^A	Beginning of line
^D	Deletes current character
^E	End of line
^F or right arrow	Forward character
^B or left arrow	Backward character
^P or up arrow	Previous line
^N or down arrow	Next line
^T	Transposes last two characters
Tab	Tries to expand current module or function name

As you get more experienced, you'll learn that the shell is a really powerful tool. Best of all, when you start writing distributed programs, you will find that you can attach a shell to a running Erlang system on a different Erlang node in a cluster or even make a secure shell (ssh) connection directly to an Erlang system running on a remote computer. Using this, you can interact with any program on any node in a system of Erlang nodes.

3.2 Simple Integer Arithmetic

Let's evaluate some arithmetic expressions.

```
1> 2 + 3 * 4.
14
2> (2 + 3) * 4.
20
```

You'll see that Erlang follows the normal rules for arithmetic expressions, so 2 + 3 * 4 means 2 + (3 * 4) and not (2 + 3) * 4.

Erlang uses arbitrary-sized integers for performing integer arithmetic. In Erlang, integer arithmetic is exact, so you don't have to worry about arithmetic overflows or not being able to represent an integer in a certain word size.

Why not try it? You can impress your friends by calculating with very large numbers.

```
3> 123456789 * 987654321 * 112233445566778899 * 998877665544332211.
13669560260321809985966198898925761696613427909935341
```

You can enter integers in a number of ways (for details, see *Integers*, on page 132). Here's an expression that uses base 16 and base 32 notation:

```
4> 16#cafe * 32#sugar.
1577682511434
```

3.3 Variables

We can store the result of a command in a variable.

```
1> X = 123456789.
123456789
```

In the first line we gave a value to the variable X; in the next line the shell prints the value of the variable.

Note that all variable names *must* start with an uppercase letter.

If you want to see the value of a variable, just enter the variable name.

```
2> X.
123456789
```

Now that X has a value, you can use it.

```
3> X*X*X*X.
232305722798259244150093798251441
```

However, if you try to assign a different value to the variable X, you'll get an error message.

```
4> X = 1234.
** exception error: no match of right hand side value 1234
```

> ## Single Assignment Is Like Algebra
>
> When I went to school, my math teacher said, "If there's an *X* in several different parts in the same equation, then all the *X*s mean the same thing." That's how we can solve equations: if we know that X+Y=10 and X-Y=2, then *X* will be 6 and *Y* will be 4 in both equations.
>
> But when I learned my first programming language, we were shown stuff like this:
>
> ```
> X = X + 1
> ```
>
> Everyone protested, saying "You can't do that!" But the teacher said we were wrong, and we had to unlearn what we learned in math class.
>
> In Erlang, variables are just like they are in math. When you associate a value with a variable, you're making an assertion—a statement of fact. This variable has that value. And that's that.

To explain what happened here, I'm going to have to shatter two assumptions you have about the simple statement X = 1234.

- First, X is not a variable, not in the sense that you're used to in languages such as Java and C.

- Second, = is not an assignment operator; it's a pattern matching operator.

This is probably one of the trickiest areas when you're new to Erlang, so let's dig deeper.

Erlang Variables Do Not Vary

Erlang has *single-assignment variables*. As the name suggests, they can be given a value only once. If you try to change the value of a variable once it has been set, then you'll get an error (in fact, you'll get the badmatch error we just saw). A variable that has had a value assigned to it is called a *bound* variable; otherwise, it is called an *unbound* variable.

When Erlang sees a statement such as X = 1234 and X has not been bound before, then it binds the variable X to the value 1234. Before being bound, X could take any value: it's just an empty slot waiting to be filled. However, once it gets a value, it keeps it forever.

At this point, you're probably wondering why we use the name *variables*. This is for two reasons.

- They are variables, but their value can be changed only once (that is, they change from being unbound to having a value).

- They look like variables in conventional programming languages, so when we see a line of code that starts like this:

```
X = ...  %% '...' means 'Code I'm not showing'
```

then our brains say, "Aha, I know what this is; X is a variable, and = is an assignment operator." And our brains are almost right: X is almost a variable, and = is almost an assignment operator.

In fact, = is a pattern matching operator, which behaves like assignment when X is an unbound variable.

Finally, the *scope* of a variable is the lexical unit in which it is defined. So, if X is used inside a single function clause, its value does not "escape" to outside the clause. There are no such things as global or private variables shared by different clauses in the same function. If X occurs in many different functions, then all the values of X are unrelated.

Variable Bindings and Pattern Matching

In Erlang, variables acquire values as the result of a successful pattern matching operation.

In most languages, = denotes an assignment statement. In Erlang, however, = is a *pattern matching* operation. Lhs = Rhs really means this: evaluate the right side (Rhs), and then match the result against the pattern on the left side (Lhs).

A variable, such as X, is a simple form of pattern. As we said earlier, variables can be given a value only once. The *first* time we say X = SomeExpression, Erlang says to itself, "What can I do to make this statement true?" Because X doesn't yet have a value, it can bind X to the value of SomeExpression, the statement becomes valid, and everyone is happy.

If at a later stage we say X = AnotherExpression, the match will succeed only if SomeExpression and AnotherExpression are identical. Here are some examples of this:

```
1> X = (2+4).
6
```

Before this statement X had no value, so the pattern match succeeds and X is bound to 6.

```
2> Y = 10.
10
```

Similarly, Y is bound to 10.

```
3> X = 6.
6
```

This is subtly different from line 1; before this expression was evaluated, X was 6, so the match succeeds, and the shell prints out the value of the expression, which is 6.

```
4> X = Y.
** exception error: no match of right hand side value 10
```

Before this expression is evaluated, X is 6 and Y is 10. 6 is not equal to 10, so an error message is printed.

```
5> Y = 10.
10
```

The pattern match succeeds because Y is 10.

```
6> Y = 4.
** exception error: no match of right hand side value 4
```

This fails since Y is 10.

At this stage, it may seem that I am belaboring the point. All the patterns to the left of the = are just variables, either bound or unbound, but as we'll see later, we can make arbitrarily complex patterns and match them with the = operator. I'll be returning to this theme after we have introduced tuples and lists, which are used for storing compound data items.

> ### Absence of Side Effects Means We Can Parallelize Our Programs
>
> The technical term for memory areas that can be modified is *mutable state*. Erlang is a functional programming language and has immutable state.
>
> Later in the book we'll look at how to program multicore CPUs and see that the consequences of having immutable state are enormous.
>
> If you use a conventional programming language such as C or Java to program a multicore CPU, then you will have to contend with the problem of *shared memory*. In order not to corrupt shared memory, the memory has to be locked while it is accessed. Programs that access shared memory must not crash while they are manipulating the shared memory.
>
> In Erlang, there is no mutable state, there is no shared memory, and there are no locks. This makes it easy to parallelize our programs.

Why Single Assignment Improves Our Programs

In Erlang a variable is just a reference to a value—in the Erlang implementation, a bound variable is represented by a pointer to an area of storage that contains the value. This value cannot be changed.

The fact that we cannot change a variable is extremely important and is unlike the behavior of variables in imperative languages such as C or Java.

Using immutable variables simplifies debugging. To understand why this is true, we must ask ourselves what an error is and how an error makes itself known.

One rather common way that we discover that a program is incorrect is when we find that a variable has an unexpected value. Once we know which variable is incorrect, we just have to inspect the program to find the place where the variable was bound. Since Erlang variables are immutable, the code that produced the variable must be incorrect. In an imperative language, variables can be changed many times, so every place where the variable was changed might be the place where the error occurred. In Erlang there is only one place to look.

At this point, you might be wondering how it's possible to program *without* mutable variables. How can we express something like X = X + 1 in Erlang? The Erlang way is to invent a new variable whose name hasn't been used before (say X1) and to write X1 = X + 1.

3.4 Floating-Point Numbers

Let's try doing some arithmetic with floating-point numbers.

```
1> 5/3.
1.6666666666666667
```

In line 1 the number at the end of the line is the integer 3. The period signifies the end of the expression and is not a decimal point. If I had wanted a floating-point number here, I'd have written 3.0.

When you divide two integers with /, the result is automatically converted to a floating-point number; thus, 5/3 evaluates to 1.6666666666666667.

```
2> 4/2.
2.0
```

Even though 4 is exactly divisible by 2, the result is a floating-point number and not an integer. To obtain an integer result from division, we have to use the operators div and rem.

```
3> 5 div 3.
1
4> 5 rem 3.
2
5> 4 div 2.
2
```

N div M divides N by M and discards the remainder. N rem M is the remainder left after dividing N by M.

Internally, Erlang uses 64-bit IEEE 754-1985 floats, so programs using floats will have the same kind of rounding or precision problems associated with floats that you would get in a language like C.

3.5 Atoms

In Erlang, atoms are used to represent constant values.

If you are used to enumerated types in C or Java, or symbols in Scheme or Ruby, then you will have already used something very similar to atoms.

C programmers will be familiar with the convention of using symbolic constants to make their programs self-documenting. A typical C program will define a set of global constants in an include file that consists of a large number of constant definitions; for example, there might be a file glob.h containing this:

```
#define OP_READ 1
#define OP_WRITE 2
#define OP_SEEK 3
...
#define RET_SUCCESS 223
...
```

Typical C code using such symbolic constants might read as follows:

```
#include "glob.h"
int ret;
ret = file_operation(OP_READ, buff);
if( ret == RET_SUCCESS ) { ... }
```

In a C program, the values of these constants are not interesting; they're interesting here only because they are all different and they can be compared for equality. The Erlang equivalent of this program might look like this:

```
Ret = file_operation(op_read, Buff),
if
    Ret == ret_success ->
        ...
```

In Erlang, atoms are global, and this is achieved without the use of macro definitions or include files.

Suppose we want to write a program that manipulates the days of the week. To do this, we'd represent the days using the atoms monday, tuesday, ….

Atoms start with lowercase letters, followed by a sequence of alphanumeric characters or the underscore (_) or at (@) sign, for example, red, december, cat, meters, yards, joe@somehost, and a_long_name.

Atoms can also be quoted with a single quotation mark ('). Using the quoted form, we can create atoms that start with uppercase letters (which otherwise would be interpreted as variables) or that contain nonalphanumeric characters, for example, 'Monday', 'Tuesday', '+', '*', 'an atom with spaces'. You can even quote atoms that don't need to be quoted, so 'a' means exactly the same as a. In some languages, single and double quotes can be used interchangeably. This is not the case in Erlang. Single quotes are used as shown earlier; double quotes delimit string literals.

The value of an atom is just the atom. So, if we give a command that is just an atom, the Erlang shell will print the value of that atom.

```
1> hello.
hello
```

It may seem slightly strange to talk about the value of an atom or the value of an integer. But because Erlang is a functional programming language, every expression must have a value. This includes integers and atoms that are just extremely simple expressions.

3.6 Tuples

Suppose we want to group a fixed number of items into a single entity. For this we'd use a *tuple*. We can create a tuple by enclosing the values we want to represent in curly brackets and separating them with commas. So, for example, if we want to represent someone's name and height, we might use {joe, 1.82}. This is a tuple containing an atom and a floating-point number.

Tuples are similar to structs in C, with the difference that they are anonymous. In C, a variable P of type point might be declared as follows:

```
struct point {
  int x;
  int y;
} P;
```

We'd access the fields in a C struct using the dot operator. So, to set the x and y values in the point, we might say this:

```
P.x = 10; P.y = 45;
```

Erlang has no type declarations, so to create a "point," we might just write this:

```
P = {10, 45}
```

This creates a tuple and binds it to the variable P. Unlike C structs, the fields of a tuple have no names. Since the tuple itself contains just a couple of integers, we have to remember what it's being used for. To make it easier to remember what a tuple is being used for, it's common to use an atom as the first element of the tuple, which describes what the tuple represents. So, we'd write {point, 10, 45} instead of {10, 45}, which makes the program a lot more understandable. This way of tagging a tuple is not a language requirement but is a recommended style of programming.

Tuples can be nested. Suppose we want to represent some facts about a person—their name, height, foot size, and eye color. We could do this as follows:

```
1> Person = {person, {name, joe}, {height, 1.82},
    {footsize, 42}, {eyecolour, brown}}.
{person, {name, joe}, {height, 1.82}, {footsize, 42}, {eyecolour, brown}}
```

Note how we used atoms both to identify the field and (in the case of name and eyecolour) to give the field a value.

Creating Tuples

Tuples are created automatically when we declare them and are destroyed when they can no longer be used.

Erlang uses a garbage collector to reclaim all unused memory, so we don't have to worry about memory allocation.

If we use a variable in building a new tuple, then the new tuple will share the value of the data structure referenced by the variable. Here's an example:

```
2> F = {firstName, joe}.
{firstName,joe}
3> L = {lastName, armstrong}.
{lastName,armstrong}
4> P = {person, F, L}.
{person,{firstName,joe},{lastName,armstrong}}
```

If we try to create a data structure with an undefined variable, then we'll get an error.

```
5> {true, Q, 23, Costs}.
** 1: variable 'Q' is unbound **
```

This just means that the variable Q is undefined.

Extracting Values from Tuples

Earlier, we said that =, which looks like an assignment statement, was not actually an assignment statement but was really a pattern matching operator. You might wonder why we were being so pedantic. Well, it turns out that pattern matching is fundamental to Erlang and it's used for lots of different tasks. It's used for extracting values from data structures, and it's also used for flow of control within functions and for selecting which messages are to be processed in a parallel program when we send messages to a process.

If we want to extract some values from a tuple, we use the pattern matching operator =.

Let's go back to our tuple that represents a point.

```
1> Point = {point, 10, 45}.
{point, 10, 45}.
```

Supposing we want to extract the fields of Point into the two variables X and Y, we do this as follows:

```
2> {point, X, Y} = Point.
{point,10,45}
3> X.
10
4> Y.
45
```

In command 2, X is bound to 10 and Y to 45. The value of the expression Lhs = Rhs is defined to be Rhs, so the shell prints {point,10,45}.

As you can see, the tuples on both sides of the equal sign must have the same number of elements, and the corresponding elements on both sides must bind to the same value.

Now suppose we had entered something like this:

```
5> {point, C, C} = Point.
** exception error: no match of right hand side value {point,10,45}
```

The pattern {point, C, C} does not match {point, 10, 45} since C cannot be simultaneously 10 and 45. Therefore, the pattern matching fails, and the system prints an error message.

Here's an example where the pattern {point, C, C} does match:

```
6> Point1 = {point,25,25}.
{point,25,25}
7> {point, C, C} = Point1.
{point,25,25}
8> C.
25
```

If we have a complex tuple, then we can extract values from the tuple by writing a pattern that is the same shape (structure) as the tuple and that contains unbound variables at the places in the pattern where we want to extract values.

To illustrate this, we'll first define a variable Person that contains a complex data structure.

```
1> Person={person,{name,joe,armstrong},{footsize,42}}.
{person,{name,joe,armstrong},{footsize,42}}
```

Now we'll write a pattern to extract the first name of the person.

```
2> {_,{_,Who,_},_} = Person.
{person,{name,joe,armstrong},{footsize,42}}
```

And finally we'll look at the value of Who.

```
3> Who.
joe
```

Note that in the previous example we wrote _ as a placeholder for variables that we're not interested in. The symbol _ is called an *anonymous variable*. Unlike regular variables, several occurrences of _ in the same pattern don't have to bind to the same value.

3.7   Lists

Lists are used to store arbitrary numbers of things. We create a list by enclosing the list elements in square brackets and separating them with commas.

Suppose we want to represent a drawing. If we assume that the drawing is made up of triangles and squares, we could represent the drawing as a list.

```
1> Drawing =  [{square,{10,10},10}, {triangle,{15,10},{25,10},{30,40}},
 ...]
```

Each of the individual elements in the drawing list are fixed-size tuples (for example, {square, Point, Side} or {triangle, Point1, Point2, Point3}), but the drawing itself can contain an arbitrary number of things and so is represented by a list.

The individual elements of a list can be of any type, so, for example, we could write the following:

```
2> [1+7,hello,2-2,{cost, apple, 30-20},3].
[8,hello,0,{cost,apple,10},3]
```

Terminology

We call the first element of a list the *head* of the list. If you imagine removing the head from the list, what's left is called the *tail* of the list.

For example, if we have a list [1,2,3,4,5], then the head of the list is the integer 1, and the tail is the list [2,3,4,5]. Note that the head of a list can be anything, but the tail of a list is usually also a list.

Accessing the head of a list is a very efficient operation, so virtually all list-processing functions start by extracting the head of a list, doing something to the head of the list, and then processing the tail of the list.

Defining Lists

If T is a list, then [H|T] is also a list with head H and tail T. The vertical bar (|) separates the head of a list from its tail. [ ] is the empty list.

Note for LISP programmers: [H|T] is a CONS cell with CAR H and CDR T. In a pattern, this syntax unpacks the CAR and CDR. In an expression, it constructs a CONS cell.

Whenever we construct a list using a [...|T] constructor, we should make sure that T is a list. If it is, then the new list will be "properly formed." If T is not a list, then the new list is said to be an "improper list." Most of the library functions assume that lists are properly formed and won't work for improper lists.

We can add more than one element to the beginning of T by writing [E1,E2,..,En|T].

So, for example, if we start by defining ThingsToBuy as follows:

```
3> ThingsToBuy = [{apples,10},{pears,6},{milk,3}].
{apples,10},{pears,6},{milk,3}]
```

then we can extend the list by writing this:

```
4> ThingsToBuy1 = [{oranges,4},{newspaper,1}|ThingsToBuy].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Extracting Elements from a List

As with everything else, we can extract elements from a list with a pattern matching operation. If we have the nonempty list L, then the expression [X|Y] = L, where X and Y are unbound variables, will extract the head of the list into X and the tail of the list into Y.

So, we're at the store, and we have our shopping list ThingsToBuy1—the first thing we do is unpack the list into its head and tail.

```
5> [Buy1|ThingsToBuy2] = ThingsToBuy1.
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

This succeeds with bindings Buy1 = {oranges,4} and ThingsToBuy2 = [{newspaper,1}, {apples,10}, {pears,6}, {milk,3}]. We go and buy the oranges, and then we could extract the next couple of items.

```
6> [Buy2,Buy3|ThingsToBuy3] = ThingsToBuy2.
[{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

This succeeds with Buy2 = {newspaper,1}, Buy3 = {apples,10}, and ThingsToBuy3 = [{pears,6},{milk,3}].

3.8 Strings

Strictly speaking, there are no strings in Erlang. To represent a string in Erlang, we can choose between representing the string as a list of integers or as a binary (for a discussion of binaries, see Section 7.1, *Binaries*, on page 99). When a string is represented as a list of integers, each element in the list represents a Unicode codepoint.

We can create such a list by using a string literal. *A string literal* is just a sequence of characters enclosed in double quotation marks ("), so, for example, we can write this:

```
1> Name = "Hello".
"Hello"
```

"Hello" is just shorthand for the list of integer character codes that represent the individual characters in that string.

Note: In some programming languages, strings can be quoted with either single or double quotes. In Erlang, we must use double quotes.

When the shell prints the value of a list, it prints it as a string literal if all the integers in the list represent printable characters; otherwise, it prints it in list notation (for character set issues, see Section 8.8, *Character Set*, on page 122).

```
2> [1,2,3].
[1,2,3]
3> [83,117,114,112,114,105,115,101].
"Surprise"
4> [1,83,117,114,112,114,105,115,101].
[1,83,117,114,112,114,105,115,101].
```

In expression 2, the list [1,2,3] is printed without any conversion. This is because 1, 2, and 3 are not printable characters.

In expression 3, all the items in the list are printable characters, so the list is printed as a string literal.

Expression 4 is just like expression 3, except that the list starts with a 1, which is not a printable character. Because of this, the list is printed without conversion.

We don't need to know which integer represents a particular character. We can use the "dollar syntax" for this purpose. So, for example, $a is actually the integer that represents the character a, and so on.

```
5> I = $s.
115
6> [I-32,$u,$r,$p,$r,$i,$s,$e].
"Surprise"
```

When we use lists to represent strings, the individual integers in the list represent Unicode characters. We have to use a special syntax to enter some of the characters and choose the correct formatting conventions when we print the list. This is best explained with an example.

```
1> X = "a\x{221e}b".
[97,8734,98].
2> io:format("~ts~n",[X]).
a∞b
```

In the line 1, we created a list of three integers. The first integer, 97, is the ASCII and Unicode code for the character a. The notation \x{221e} was used to input a hexadecimal integer (8734) that represents the Unicode INFINITY character. Finally, 98 is the ASCII and Unicode code for the character b. The shell prints this in list notation ([97,8734,98]); this is because 8734 is not a printable Latin1 character code. In line 2 we used a formatted I/O statement to print the string using the correct character glyph for the infinity character.

If the shell prints a list of integers as a string but you really wanted it printed as a list of integers, then you'll have to use a formatted write statement, as in the following:

```
1> X = [97,98,99].
"abc"
2> io:format("~w~n",["abc"]).
[97,98,99]
```

3.9 Pattern Matching Again

To round off this chapter, we'll go back to pattern matching one more time.

The following table has some examples of patterns and terms; all the variables in the patterns are assumed unbound. A *term* is just an Erlang data structure. The third column of the table, marked *Result*, shows whether the pattern matched the term and, if so, the variable bindings that were created. Read through the examples, and make sure you really understand them.

Pattern	=	Term	Result
{X,abc}	=	{123,abc}	*Succeeds* with X = 123
{X,Y,Z}	=	{222,def,"cat"}	*Succeeds* with X = 222, Y = def, and Z = "cat"
{X,Y}	=	{333,ghi,"cat"}	*Fails*—the tuples have different shapes
X	=	true	*Succeeds* with X = true
{X,Y,X}	=	{{abc,12},42,{abc,12}}	*Succeeds* with X = {abc,12} and Y = 42
{X,Y,X}	=	{{abc,12},42,true}	*Fails*—X cannot be both {abc,12} and true
[H|T]	=	[1,2,3,4,5]	*Succeeds* with H = 1 and T = [2,3,4,5]
[H|T]	=	"cat"	*Succeeds* with H = 99 and T = "at"
[A,B,C|T]	=	[a,b,c,d,e,f]	*Succeeds* with A = a, B = b, C = c, and T = [d,e,f]

If you're unsure about any of these, then try entering a Pattern = Term expression into the shell to see what happens.

Here's an example:

```
1> {X, abc} = {123, abc}.
{123,abc}.
2> X.
123
3> f().
ok
4> {X,Y,Z} = {222,def,"cat"}.
{222,def,"cat"}.
5> X.
222
6> Y.
def
...
```

Note: The command f() tells the shell to *forget* any bindings it has. After this command, all variables become unbound, so the X in line 4 has nothing to do with the X in lines 1 and 2.

Now that we're comfortable with the basic data types and with the ideas of single assignment and pattern matching, we can step up the tempo and see how to define modules and functions. We'll do this in the next chapter.

Exercises

1. Take a quick look at *Command Editing in the Erlang Shell,* on page 27; then test and memorize the line-editing commands.

2. Give the command help() in the shell. You'll see a long list of commands. Try some of the commands.

3. Try representing a house using a tuple and a street using a list of houses. Make sure you can pack and unpack the data in the representations.

Modules and Functions

Modules and functions are the basic units from which sequential and parallel programs are built. Modules contain functions, and the functions can be run sequentially or in parallel.

This chapter builds upon the ideas of pattern matching from the previous chapter and introduces all the control statements we need for writing code. We'll be talking about higher-order functions (called *funs*) and how they can be used to create your own control abstractions. Also, we'll talk about list comprehensions, guards, records, and case expressions and show how they are used in small fragments of code.

Let's get to work.

4.1 Modules Are Where We Store Code

Modules are the basic units of code in Erlang. Modules are contained in files with .erl extensions and must be compiled before the code in the modules can be run. Compiled modules have the extension .beam.

Before we write our first module, we'll remind ourselves about pattern matching. All we're going to do is create a couple of data structures representing a rectangle and a square. Then we're going to unpack these data structures and extract the sides from the rectangle and the square. Here's how:

```
1> Rectangle = {rectangle, 10, 5}.
{rectangle, 10, 5}.
2> Square = {square, 3}.
{square, 3}
3> {rectangle, Width, Height} = Rectangle.
{rectangle,10,5}
4> Width.
10
5> Height.
5
```

```
6> {square, Side} = Square.
{square,3}
7> Side.
3
```

In lines 1 and 2 we *created* a rectangle and square. In lines 3 and 6 we *unpacked* the fields of the rectangle and square using pattern matching. In lines 4, 5, and 7 we printed the variable bindings that were created by the pattern matching expressions. After line 7 the variable bindings in the shell are Width = 10, Height = 5, and Side = 3.

Going from pattern matching in the shell to pattern matching in functions is an extremely small step. Let's start with a function called area that computes the areas of rectangles and squares. We'll put this in a module called geometry and store the module in the file called geometry.erl. The entire module looks like this:

geometry.erl
```
-module(geometry).
-export([area/1]).

area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

The first line in the file is a *module declaration*. The name of the module in the declaration must be the same as the base name of the file where the module is stored.

The second line is an *export declaration*. The notation Name/N means a function called Name with N arguments; N is called the *arity* of the function. The argument to export is a list of Name/N items. Thus, -export([area/1]) means that the function area with one argument can be called from outside this module.

Functions that are not exported from a module can be called only from within a module. Exported functions are equivalent to public methods in an object-oriented programming language (OOPL); nonexported functions are equivalent to private methods in an OOPL.

The function area consists of two *clauses*. The clauses are separated by a semicolon, and the final clause is terminated by dot whitespace. Each clause has a *head* and a *body* separated by an arrow (->). The head consists of a function name followed by zero or more patterns, and the body consists of a sequence of *expressions* (expressions are defined in Section 8.13, *Expressions and Expression Sequences*, on page 127), which are evaluated if the pattern in the head is successfully matched against the calling arguments. The clauses are tried in the order they appear in the function definition.

Note how the patterns that we used in the shell example have become part of the area function definition. Each pattern corresponds to exactly one clause. The first clause of the area function:

```
area({rectangle, Width, Height}) -> Width * Height;
```

tells us how to compute the area of a rectangle. When we evaluate the function geometry:area({rectangle, 10, 5}), the first clause in area/1 matches with bindings Width = 10 and Height = 5. Following the match, the code following the arrow -> is evaluated. This is just Width * Height, which is 10*5, or 50. Note that the function has no explicit return statement; the return value of the function is simply the value of the last expression in the body of the clause.

Now we'll compile the module and run it.

```
1> c(geometry).
{ok,geometry}
2> geometry:area({rectangle, 10, 5}).
50
3> geometry:area({square, 3}).
9
```

In line 1 we give the command c(geometry), which compiles the code in the file geometry.erl. The compiler returns {ok,geometry}, which means that the compilation succeeded and that the module geometry has been compiled and loaded. The compiler will create an object code module called geometry.beam in the current directory. In lines 2 and 3 we call the functions in the geometry module. Note how we need to include the module name together with the function name in order to identify exactly which function we want to call.

Common Errors

A word of warning: commands like c(geometry). (used earlier) work only in the shell and cannot be put into modules. Some readers have mistakenly typed into the shell fragments of code contained in the source code listings. These are not valid shell commands, and you'll get some very strange error message if you try to do this. So, don't do this.

If you accidentally choose a module name that collides with one of the system modules, then when you compile your module, you'll get a strange message saying that you can't load a module that resides in a sticky directory. Just rename the module, and delete any .beam file that you might have made when compiling your module.

Directories and Code Paths

If you download the code examples in this book or want to write your own examples, you have to make sure that when you run the compiler from the shell, you are in the right directory so that the system can find your files.

The Erlang shell has a number of built-in commands to see and change the current working directory.

- pwd() prints the current working directory.
- ls() lists the names of the files in the current working directory.
- cd(Dir) changes the current working directory to Dir.

Adding Tests to Your Code

At this stage, we can add some simple tests to our module. Let's rename the module to geometry1.erl and add some test code.

```
geometry1.erl
-module(geometry1).
-export([test/0, area/1]).

test() ->
    12  = area({rectangle, 3, 4}),
    144 = area({square, 12}),
    tests_worked.

area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

```
1> c(geometry1).
{ok,geometry1}
2> geometry1:test().
tests_worked
```

The line of code 12 = area({rectangle, 3, 4}) is a test. If area({rectangle, 3, 4}) had not returned 12, the pattern match would fail and we'd get an error message. When we evaluate geometry1:test() and see the result tests_worked, we can conclude that all the tests in the body of test/0 succeeded.

We can easily add tests and perform test-driven development without any additional tools. All we need is pattern matching and =. While this is sufficient for quick-and-dirty testing, for production code it is better to use a fully featured test framework, such as the common or unit test framework; for details, read the test section of the Erlang documentation.[1]

---

1. http://www.erlang.org/doc

Extending the Program

Now suppose we want to extend our program by adding a circle to our geometric objects. We could write this:

```
area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side;
area({circle, Radius})           -> 3.14159 * Radius * Radius.
```

or this:

```
area({rectangle, Width, Height}) -> Width * Height;
area({circle, Radius})           -> 3.14159 * Radius * Radius;
area({square, Side})             -> Side * Side.
```

Note that in this example, the order of the clauses doesn't matter; the program means the same no matter how the clauses are ordered. This is because the patterns in the clause are mutually exclusive. This makes writing and extending programs very easy—we just add more patterns. In general, though, clause order does matter. When a function is entered, the clauses are pattern matched against the calling arguments in the order they are presented in the file.

Before going any further, you should note the following about the way the area function is written:

- The function area consists of several different clauses. When we call the function, execution starts in the first clause that matches the call arguments.

- Our function does not handle the case where no pattern matches—our program will fail with a runtime error. This is deliberate. This is the way we program in Erlang.

Many programming languages, such as C, have only one entry point per function. If we had written this in C, the code might look like this:

```
enum ShapeType { Rectangle, Circle, Square };

struct Shape {
    enum ShapeType kind;

    union {
        struct { int width, height; } rectangleData;
        struct { int radius; }        circleData;
        struct { int side;}           squareData;
    } shapeData;
};
```

```
double area(struct Shape* s) {
    if( s->kind == Rectangle ) {
      int width, ht;
      width = s->shapeData.rectangleData.width;
      ht    = s->shapeData.rectangleData.height;
      return width * ht;
    } else if ( s->kind == Circle ) {
      ...
```

The C code performs what is essentially a pattern matching operation on the argument to the function, but the programmer has to write the pattern matching code and make sure that it is correct.

In the Erlang equivalent, we merely write the patterns, and the Erlang compiler generates optimal pattern matching code, which selects the correct entry point for the program.

The following shows what the equivalent code would look like in Java:[2]

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double ht;
    final double width;
    Rectangle(double width, double height) {
        this.ht = height;
        this.width  = width;
    }

    double area() { return width * ht; }
}

class Square extends Shape {
    final double side;
    Square(double side) {
        this.side = side;
    }

    double area() { return side * side; }
}
```

---

2. http://java.sun.com/developer/Books/shiftintojava/page1.html

If you compare the Erlang code with Java code, you'll see that in the Java program the code for area is in three places. In the Erlang program, all the code for area is in the same place.

Where to Put the Semicolons

Before we leave our geometry example, we'll take one more look at the code, this time looking at the punctuation. This time stare hard at the code and look for the placement of commas, semicolons, and periods.

geometry.erl
```
-module(geometry).
-export([area/1]).

area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

You'll see the following:

- *Commas* (,) separate arguments in function calls, data constructors, and patterns.

- *Semicolons* (;) separate *clauses*. We find clauses in several contexts, namely, in function definitions and in case, if, try..catch, and receive expressions.

- *Periods* (.) (followed by whitespace) separate entire functions and expressions in the shell.

There's an easy way to remember this—*think of English*. Full stops separate sentences, semicolons separate clauses, and commas separate subordinate clauses. A comma is a short-range symbol, a semicolon is a medium-range symbol, and a period a long-range symbol.

Whenever we see sets of patterns followed by expressions, we'll see semicolons as separators. Here's an example:

```
case f(...) of
    Pattern1 ->
       Expressions1;
    Pattern2 ->
       Expressions2;

    ...

    LastPattern ->
       LastExpression
end
```

Note that the last expression (that which immediately precedes the end keyword) has no semicolon.

That's enough theory for now. Let's continue with some code; we'll get back to control structures later.

4.2 Back to Shopping

In *Defining Lists, on page 38*, we had a shopping list that looked like this:

```
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Now suppose that we'd like to know what our shopping costs. To work this out, we need to know how much each item in the shopping list costs. Let's assume that this information is computed in a module called shop, defined as follows:

```
shop.erl
-module(shop).
-export([cost/1]).

cost(oranges)   -> 5;
cost(newspaper) -> 8;
cost(apples)    -> 2;
cost(pears)     -> 9;
cost(milk)      -> 7.
```

The function cost/1 is made up from five *clauses*. The head of each clause contains a pattern (in this case a very simple pattern that is just an atom). When we evaluate shop:cost(X), then the system will try to match X against each of the patterns in these clauses. If a match is found, the code to the right of the -> is evaluated.

Let's test this. We'll compile and run the program in the Erlang shell.

```
1> c(shop).
{ok,shop}
2> shop:cost(apples).
2
3> shop:cost(oranges).
5
4> shop:cost(socks).
** exception error: no function clause matching shop:cost(socks)
   (shop.erl, line 4)
```

In line 1 we compiled the module in the file shop.erl. In lines 2 and 3, we asked how much apples and oranges cost (the results 2 and 5 are in cost units). In line 4 we asked what socks cost, but no clause matched, so we got a pattern

matching error, and the system printed an error message containing the filename and line number where the error occurred.

Back to the shopping list. Suppose we have a shopping list like this:

```
1> Buy = [{oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

and want to calculate the total value of all the items in the list. One way to do this is to define a function shop1:total/1 as follows:

```
shop1.erl
-module(shop1).
-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);
total([])            -> 0.
```

Let's experiment with this:

```
2> c(shop1).
{ok,shop1}
3> shop1:total([]).
0
```

This returns 0 because the second clause of total/1 is total([]) -> 0.

Here's a more complex query:

```
4> shop1:total([{milk,3}]).
21
```

This works as follows. The call shop1:total([{milk,3}]) matches the following clause with bindings What = milk, N = 3, and T = []:

```
total([{What,N}|T]) -> shop:cost(What) * N + total(T);
```

Following this, the code in the body of the function is evaluated, so we have to evaluate the expression.

```
shop:cost(milk) * 3 + total([]);
```

shop:cost(milk) is 7 and total([]) is 0, so the final return value is 21.

We can test this with an even more complex argument.

```
5> shop1:total([{pears,6},{milk,3}]).
75
```

Again, line 5 matches the first clause of total/1 with bindings What = pears, N = 6, and T = [{milk,3}].

```
total([{What,N}|T]) -> shop:cost(What) * N + total(T);
```

The variables What, N, and T are substituted into the body of the clause, and shop:cost(pears) * 6 + total([{milk,3}]) is evaluated, which reduces to 9 * 6 + total([{milk,3}]).

But we worked out before that total([{milk,3}]) was 21, so the final result is 9*6 + 21 = 75.

Finally:

```
6> shop1:total(Buy).
123
```

Before we leave this section, we should take a more detailed look at the function total. total(L) works by a case analysis of the argument L. There are two possible cases; L is a nonempty list, or L is an empty list. We write one clause for each possible case, like this:

```
total([Head|Tail]) ->
    some_function_of(Head)  + total(Tail);
total([]) ->
    0.
```

In our case, Head was the pattern {What,N}. When the first clause matches a nonempty list, it picks out the head from the list, does something with the head, and then calls itself to process the tail of the list. The second clause matches when the list has been reduced to an empty list ([]).

The function total/1 actually did two different things. It looked up the prices of each of the elements in the list, and then it summed all the prices multiplied by the quantities of items purchased. We can rewrite total in a way that separates looking up the values of the individual items and summing the values. The resulting code will be clearer and easier to understand. To do this, we'll write two small list-processing functions called sum and map. To write map, we have to introduce the idea of funs. After this, we'll write an improved version of total in the module shop2.erl that you can find near the end of Section 4.4, *Simple List Processing*, on page 57.

4.3   Funs: The Basic Unit of Abstraction

Erlang is a functional programming language. Among other things this means that functions can be used as arguments to functions and that functions can return functions. Functions that manipulate functions are called *higher-order functions*, and the data type that represents a function in Erlang is called a *fun*.

Higher-order functions are the very essence of functional programming languages—not only can functional programs manipulate regular data structures,

they can also manipulate the functions that transform the data. Once you've learned to use them, you'll love them. We'll see a lot more of them in the future.

Funs can be used in the following ways:

- To perform the same operation on every element of a list. In this case, we pass funs as arguments to functions like lists:map/2, lists:filter/2, and so on. This usage of funs is extremely common.

- To create our own control abstractions. This technique is extremely useful. Erlang has, for example, no for loop. But we can easily create our own for loop. The advantage of creating our own control abstractions is that we can make them do exactly what we want them to do rather than rely on a predefined set of control abstractions that might not behave exactly as we want.

- To implement things like reentrant parsing code, parser combinators, or lazy evaluators. In this case, we write functions, which return funs. This is a very powerful technique but can lead to programs that are difficult to debug.

funs are "anonymous" functions. They are called this because they have no name. You might see them referred to as *lambda abstractions* in other programming languages. Let's start experimenting; first we'll define a fun and assign it to a variable.

```
1> Double = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
```

When we define a fun, the Erlang shell prints #Fun<...> where the ... is some weird number. Don't worry about this now.

There's only one thing we can do with a fun, and that is to apply it to an argument, like this:

```
2> Double(2).
4
```

Funs can have any number of arguments. We can write a function to compute the hypotenuse of a right-angled triangle, like this:

```
3> Hypot = fun(X, Y) -> math:sqrt(X*X + Y*Y) end.
#Fun<erl_eval.12.115169474>
4> Hypot(3,4).
5.0
```

If the number of arguments is incorrect, you'll get an error.

```
5> Hypot(3).
** exception error: interpreted function with arity 2 called with one argument
```

The error message tells us that Hypot expects two arguments but we have supplied only one. Remember that *arity* is the number of arguments a function accepts.

Funs can have several different clauses. Here's a function that converts temperatures between Fahrenheit and Centigrade:

```
6> TempConvert = fun({c,C}) -> {f, 32 + C*9/5};
6>                  ({f,F}) -> {c, (F-32)*5/9}
6>              end.
#Fun<erl_eval.6.56006484>
7> TempConvert({c,100}).
{f,212.0}
8> TempConvert({f,212}).
{c,100.0}
9> TempConvert({c,0}).
{f,32.0}
```

Note: The expression in line 6 spans several lines. As we enter this expression, the shell repeats the prompt 6> every time we enter a new line. This means the expression is incomplete and the shell wants more input.

Functions That Have Funs As Their Arguments

The module lists, which is in the standard libraries, exports several functions whose arguments are funs. The most useful of all these is lists:map(F, L). This is a function that returns a list made by applying the fun F to every element in the list L.

```
10> L = [1,2,3,4].
[1,2,3,4]
11> lists:map(fun(X) -> 2*X end, L).
[2,4,6,8]
```

Another useful function is lists:filter(P, L), which returns a new list of all the elements E in L such that P(E) is true.

Let's define a function Even(X) that is true if X is an even number.

```
12> Even = fun(X) -> (X rem 2) =:= 0 end.
#Fun<erl_eval.6.56006484>
```

Here X rem 2 computes the remainder after X has been divided by 2, and =:= is a test for equality. Now we can test Even, and then we can use it as an argument to map and filter.

```
13> Even(8).
true
14> Even(7).
false
15> lists:map(Even, [1,2,3,4,5,6,8]).
[false,true,false,true,false,true,true]
16> lists:filter(Even, [1,2,3,4,5,6,8]).
[2,4,6,8]
```

We refer to operations such as map and filter that do something to an entire list in one function call as *list-at-a-time* operations. Using list-at-a-time operations makes our programs small and easy to understand; they are easy to understand because we can regard each operation on the entire list as a single conceptual step in our program. Otherwise, we have to think of each individual operation on the elements of the list as single steps in our program.

Functions That Return Funs

Not only can funs be used as arguments to functions (such as map and filter), but functions can also *return* funs.

Here's an example—suppose I have a list of something, say fruit:

```
1> Fruit = [apple,pear,orange].
[apple,pear,orange]
```

Now I can define a function MakeTest(L) that turns a list of things (L) into a test function that checks whether its argument is in the list L.

```
2> MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.
#Fun<erl_eval.6.56006484>
3> IsFruit = MakeTest(Fruit).
#Fun<erl_eval.6.56006484>
```

lists:member(X, L) returns true if X is a member of the list L; otherwise, it returns false. Now that we have built a test function, we can try it.

```
4> IsFruit(pear).
true
5> IsFruit(apple).
true
6> IsFruit(dog).
false
```

We can also use it as an argument to lists:filter/2.

```
7> lists:filter(IsFruit, [dog,orange,cat,apple,bear]).
[orange,apple]
```

The notation for funs that return funs takes a little getting used to, so let's dissect the notation to make what's going on a little clearer. A function that returns a "normal" value looks like this:

```
1> Double = fun(X) -> ( 2 * X ) end.
#Fun<erl_eval.6.56006484>
2> Double(5).
10
```

The code inside the parentheses (in other words, 2 * X) is clearly the "return value" of the function. Now let's try putting a *fun* inside the parentheses.

Remember the thing inside the parentheses *is* the return value.

```
3> Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.
#Fun<erl_eval.6.56006484>
```

The fun inside the parentheses is fun(X) -> X * Times end; this is just a function of X. Times is the argument of the "outer" fun.

Evaluating Mult(3) *returns* fun(X) -> X * 3 end, which is the body of the inner fun with Times substituted with 3. Now we can test this.

```
4> Triple = Mult(3).
#Fun<erl_eval.6.56006484>
5> Triple(5).
15
```

So, Mult is a *generalization* of Double. Instead of computing a value, *it returns a function, which when called will compute the required value.*

Defining Your Own Control Abstractions

So far, we haven't seen any if statements, switch statements, for statements, or while statements, and yet this doesn't seem to matter. Everything is written using pattern matching and higher-order functions.

If we want additional control structures, we can make our own. Here's an example; Erlang has no for loop, so let's make one:

lib_misc.erl
```
for(Max, Max, F) -> [F(Max)];
for(I, Max, F)   -> [F(I)|for(I+1, Max, F)].
```

So, for example, evaluating for(1,10,F) creates the list [F(1), F(2), ..., F(10)].

Now we have a simple for loop. We can use it to make a list of the integers from 1 to 10.

```
1> lib_misc:for(1,10,fun(I) -> I end).
[1,2,3,4,5,6,7,8,9,10]
```

Or we can compute the squares of the integers from 1 to 10.

```
2> lib_misc:for(1,10,fun(I) -> I*I end).
[1,4,9,16,25,36,49,64,81,100]
```

As you become more experienced, you'll find that being able to create your own control structures can dramatically decrease the size of your programs and sometimes make them a lot clearer. This is because you can create exactly the right control structures that are needed to solve your problem and because you are not restricted by a small and fixed set of control structures that came with your programming language.

4.4 Simple List Processing

Now that we've introduced funs, we can get back to writing sum and map, which we'll need for our improved version of total (which I'm sure you haven't forgotten about!).

We'll start with sum, which computes the sum of the elements in a list.

```
mylists.erl
❶ sum([H|T]) -> H + sum(T);
❷ sum([])    -> 0.
```

Note that the order of the two clauses in sum is unimportant. This is because the first clause matches a nonempty list and the second an empty list, and these two cases are mutually exclusive. We can test sum as follows:

```
1> c(mylists).  %% <-- Last time I do this
{ok, mylists}
2> L = [1,3,10].
[1,3,10]
3> mylists:sum(L).
14
```

Line 1 compiled the module mylists. From now on, I'll often omit the command to compile the module, so you'll have to remember to do this yourself. It's pretty easy to understand how this works. Let's trace the execution.

1. sum([1,3,10])
2. sum([1,3,10]) = 1 + sum([3,10]) (by ❶)
3. = 1 + 3 + sum([10]) (by ❶)
4. = 1 + 3 + 10 + sum([]) (by ❶)
5. = 1 + 3 + 10 + 0 (by ❷)
6. = 14

Finally, let's look at map/2, which we met earlier. Here's how it's defined:

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

❶ The first clause says what to do with an empty list. Mapping any function over the elements of an empty list (there are none!) just produces an empty list.

❷ The second clause is a rule for what to do with a list with a head H and tail T. That's easy. Just build a new list whose head is F(H) and whose tail is map(F, T).

Note: The definition of map/2 is copied from the standard library module lists to mylists. You can do anything you like to the code in mylists.erl. Do not under any circumstance try to make your own module called lists—if you make any mistakes in lists, you could easily seriously damage the system.

We can run map using a couple of functions that double and square the elements in a list, as follows:

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]
2> mylists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
3> mylists:map(fun(X) -> X*X end, L).
[1,4,9,16,25]
```

Later, we'll show an even shorter version of map written using list comprehensions, and in Section 26.3, *Parallelizing Sequential Code,* on page 445, we'll show how we can compute all the elements of the map *in parallel* (which will speed up our program on a multicore computer)—but this is jumping too far ahead. Now that we know about sum and map, we can rewrite total using these two functions:

```
shop2.erl
-module(shop2).
-export([total/1]).
-import(lists, [map/2, sum/1]).

total(L) ->
    sum(map(fun({What, N}) -> shop:cost(What) * N end, L)).
```

We can see how this function works by looking at the steps involved.

```
1> Buy = [{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
2> L1=lists:map(fun({What,N}) -> shop:cost(What) * N end, Buy).
[20,8,20,54,21]
3> lists:sum(L1).
123
```

How I Write Programs

When I'm writing a program, my approach is to "write a bit" and then "test a bit." I start with a small module with a few functions, and then I compile it and test it with a few commands in the shell. Once I'm happy with it, I write a few more functions, compile them, test them, and so on.

Often I haven't really decided what sort of data structures I'll need in my program, and as I run small examples, I can see whether the data structures I have chosen are appropriate.

I tend to "grow" programs rather than think them out completely before writing them. This way I don't tend to make large mistakes before I discover that things have gone wrong. Above all, it's fun, I get immediate feedback, and I see whether my ideas work as soon as I have typed in the program.

Once I've figured out how to do something in the shell, I usually then go and write a makefile and some code that reproduces what I've learned in the shell.

Note also the use of the -import and -export declarations in the module.

- The declaration -import(lists, [map/2, sum/1]). means the function map/2 is *imported* from the module lists, and so on. This means we can write map(Fun, ...) instead of lists:map(Fun, ...). cost/1 was not declared in an import declaration, so we had to use the "fully qualified" name shop:cost.

- The declaration -export([total/1]) means the function total/1 can be called from outside the module shop2. Only functions that are exported from a module can be called from outside the module.

By this time you might think that our total function cannot be further improved, but you'd be wrong. Further improvement is possible. To do so, we'll use a list comprehension.

4.5 List Comprehensions

List comprehensions are expressions that create lists without having to use funs, maps, or filters. This makes our programs even shorter and easier to understand.

We'll start with an example. Suppose we have a list L.

```
1> L = [1,2,3,4,5].
[1,2,3,4,5]
```

And say we want to double every element in the list. We've done this before, but I'll remind you.

```
2> lists:map(fun(X) -> 2*X end, L).
[2,4,6,8,10]
```

But there's a much easier way that uses a list comprehension.

```
4> [2*X || X <- L ].
[2,4,6,8,10]
```

The notation [F(X) || X <- L] means "the list of F(X) where X is taken from the list L." Thus, [2*X || X <- L] means "the list of 2*X where X is taken from the list L."

To see how to use a list comprehension, we can enter a few expressions in the shell to see what happens. We start by defining Buy.

```
1> Buy=[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Now let's double the number of every item in the original list.

```
2> [{Name, 2*Number} || {Name, Number} <- Buy].
[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]
```

Note that the tuple {Name, Number} to the right side of the (||) sign is a *pattern* that matches each of the elements in the list Buy. The tuple to the left side, {Name, 2*Number}, is a *constructor*.

Suppose we want to compute the total cost of all the elements in the original list; we could do this as follows. First replace the name of every item in the list with its price.

```
3> [{shop:cost(A), B} || {A, B} <- Buy].
[{5,4},{8,1},{2,10},{9,6},{7,3}]
```

Now multiply the numbers.

```
4> [shop:cost(A) *  B || {A, B} <- Buy].
[20,8,20,54,21]
```

Then sum them.

```
5> lists:sum([shop:cost(A) *  B || {A, B} <- Buy]).
123
```

Finally, if we wanted to make this into a function, we would write the following:

```
total(L) ->
    lists:sum([shop:cost(A) *  B || {A, B} <- L]).
```

List comprehensions will make your code really short and easy to read. For example, we can define an even shorter version of map.

```
map(F, L) -> [F(X) || X <- L].
```

The most general form of a list comprehension is an expression of the following form:

```
[X || Qualifier1, Qualifier2, ...]
```

X is an arbitrary expression, and each qualifier is either a generator, a bitstring generator, or a filter.

- Generators are written as `Pattern <- ListExpr` where `ListExpr` must be an expression that evaluates to a list of terms.

- Bitstring generators are written as `BitStringPattern <= BitStringExpr` where `BitStringExpr` must be an expression that evaluates to a bitstring. More information about bitstring patterns and generators can be found in the Erlang Reference Manual.[3]

- Filters are either predicates (functions that return `true` or `false`) or boolean expressions.

Note that the generator part of a list comprehension works like a filter; here's an example:

```
1> [ X || {a, X} <- [{a,1},{b,2},{c,3},{a,4},hello,"wow"]].
[1,4]
```

We'll finish this section with a few short examples.

Quicksort

Here's how to write a sort algorithm using two list comprehensions:

lib_misc.erl
```
qsort([]) -> [];
qsort([Pivot|T]) ->
        qsort([X || X <- T, X < Pivot])
        ++ [Pivot] ++
        qsort([X || X <- T, X >= Pivot]).
```

Note that ++ is the infix append operator. This code is shown for its elegance rather than its efficiency. Using ++ in this way is not generally considered good programming practice. See Section 4.9, *Building Lists in Natural Order, on page 70*, for more information.

```
1> L=[23,6,2,9,27,400,78,45,61,82,14].
[23,6,2,9,27,400,78,45,61,82,14]
2> lib_misc:qsort(L).
[2,6,9,14,23,27,45,61,78,82,400]
```

---

3. http://www.erlang.org/doc/pdf/otp-system-documentation.pdf

To see how this works, we'll step through the execution. We start with a list L and call qsort(L). The following matches the second clause of qsort with bindings Pivot → 23 and T → [6,2,9,27,400,78,45,61,82,14]:

```
3> [Pivot|T] = L.
[23,6,2,9,27,400,78,45,61,82,14]
```

Now we split T into two lists, one with all the elements in T that are less than Pivot, and the other with all the elements greater than or equal to Pivot.

```
4> Smaller = [X || X <- T, X < Pivot].
[6,2,9,14]
5> Bigger  = [X || X <- T, X >= Pivot].
[27,400,78,45,61,82]
```

Now we sort Smaller and Bigger and combine them with Pivot.

```
qsort( [6,2,9,14] ) ++ [23] ++ qsort( [27,400,78,45,61,82] )
= [2,6,9,14] ++ [23] ++ [27,45,61,78,82,400]
= [2,6,9,14,23,27,45,61,78,82,400]
```

Pythagorean Triplets

Pythagorean triplets are sets of integers {A,B,C} where $A^2 + B^2 = C^2$.

The function pythag(N) generates a list of all integers {A,B,C} where $A^2 + B^2 = C^2$ and where the sum of the sides is less than or equal to N.

```
lib_misc.erl
pythag(N) ->
    [ {A,B,C} ||
        A <- lists:seq(1,N),
        B <- lists:seq(1,N),
        C <- lists:seq(1,N),
        A+B+C =< N,
        A*A+B*B =:= C*C
    ].
```

Just a few words of explanation: lists:seq(1, N) returns a list of all the integers from 1 to N. Thus, A <- lists:seq(1, N) means that A takes all possible values from 1 to N. So, our program reads, "Take all values of A from 1 to N, all values of B from 1 to N, and all values of C from 1 to N such that A + B + C is less than or equal to N and A*A + B*B = C*C."

```
1> lib_misc:pythag(16).
[{3,4,5},{4,3,5}]
2> lib_misc:pythag(30).
[{3,4,5},{4,3,5},{5,12,13},{6,8,10},{8,6,10},{12,5,13}]
```

Anagrams

If you're interested in English-style crossword puzzles, you'll often find yourself figuring out anagrams. Let's use Erlang to find all the permutations of a string using the beautiful little function perms.

```
perms([]) -> [[]];
perms(L)  -> [[H|T] || H <- L, T <- perms(L--[H])].
```

```
1> lib_misc:perms("123").
["123","132","213","231","312","321"]
2> lib_misc:perms("cats").
["cats", "cast", "ctas", "ctsa", "csat", "csta", "acts", "acst",
"atcs", "atsc", "asct", "astc", "tcas", "tcsa", "tacs", "tasc",
"tsca", "tsac", "scat", "scta", "sact", "satc", "stca", "stac"]
```

X -- Y is the list subtraction operator. It subtracts the elements in Y from X; there's a more precise definition in Section 8.16, *List Operations ++ and - -,* on page 129.

perms is pretty neat. It works as follows: Assume we want to compute all permutations of the string "cats". First we isolate the first character of the string, which is c, and compute all permutations of the string with the character c removed. "cats" with c removed is the string "ats", and all the permutations of "ats" are the strings ["ats", "ast", "tas", "tsa", "sat", "sta"]. Next we append the c to the beginning of each of these strings, forming ["cats", "cast", "ctas", "ctsa", "csat", "csta"]. Then we repeat the algorithm isolating the second character, and so on.

This is exactly what the perms function does.

```
[ [H|T] || H <- L, T <- perms(L -- [H]) ]
```

This means take H from L in all possible ways and then take T from perms(L--[H]) (that is, all permutations of the list L with H removed) in all possible ways and return [H|T].

4.6 BIFs

A *BIF* is a *built-in function*; BIFs are functions that are defined as part of the Erlang language. Some BIFs are implemented in Erlang, but most are implemented as primitive operations in the Erlang virtual machine.

BIFs provide interfaces to the operating system or perform operations that are impossible or very inefficient to program in Erlang. For example, it's impossible to turn a list into a tuple or to find the current time and date. To perform such an operation, we call a BIF.

For example, the BIF list_to_tuple/1 converts a list to a tuple, and time/0 returns the current time of day in hours, minutes, and seconds.

```
1> list_to_tuple([12,cat,"hello"]).
{12,cat,"hello"}
2> time().
{20,0,3}
```

All BIFs behave as if they belong to the module erlang, though the most common BIFs (such as list_to_tuple) are *autoimported*, so we can call list_to_tuple(...) instead of erlang:list_to_tuple(...).

You'll find a full list of all BIFs in the erlang manual page in your Erlang distribution or online at http://www.erlang.org/doc/man/erlang.html. Throughout the remainder of the book I'll introduce only the BIFs that are necessary to understand a particular section in the book. There are actually more BIFs in the system than those that I describe in the book, so I recommend printing the manual page and trying to learn what all the BIFs are.

4.7 Guards

Guards are constructs that we can use to increase the power of pattern matching. Using guards, we can perform simple tests and comparisons on the variables in a pattern. Suppose we want to write a function max(X, Y) that computes the max of X and Y. We can write this using a guard as follows:

```
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.
```

The first clause matches when X is greater than Y and the result is X.

If the first clause doesn't match, then the second clause is tried. The second clause always returns the second argument Y. Y must be greater than or equal to X; otherwise, the first clause would have matched.

You can use guards in the heads of function definitions where they are introduced by the when keyword, or you can use them at any place in the language where an expression is allowed. When they are used as expressions, they evaluate to one of the atoms true or false. If the guard evaluates to true, we say that the evaluation *succeeded*; otherwise, it *fails*.

Guard Sequences

A *guard sequence* is either a single guard or a series of guards, separated by semicolons (;). The guard sequence G1; G2; ...; Gn is true if at least one of the guards—G1, G2, ...—evaluates to true.

A *guard* is a series of *guard expressions*, separated by commas (,). The guard GuardExpr1, GuardExpr2, …, GuardExprN is true if all the guard expressions—GuardExpr1, GuardExpr2, …—evaluate to true.

The set of valid guard expressions is a subset of all valid Erlang expressions. The reason for restricting guard expressions to a subset of Erlang expressions is that we want to guarantee that evaluating a guard expression is free from side effects. Guards are an extension of pattern matching, and since pattern matching has no side effects, we don't want guard evaluation to have side effects.

In addition, guards cannot call user-defined functions, since we want to guarantee that they are side effect free and terminate.

The following syntactic forms are legal in a guard expression:

- The atom true
- Other constants (terms and bound variables); these all evaluate to false in a guard expression
- Calls to the guard predicates in Table 1, *Guard predicates*, on page 66 and to the BIFs in Table 2, *Guard built-in functions*, on page 66
- Term comparisons (Table 6, *Term comparisons*, on page 137)
- Arithmetic expressions (Table 3, *Arithmetic expressions*, on page 116)
- Boolean expressions (Section 8.7, *Boolean Expressions*, on page 121)
- Short-circuit boolean expressions (Section 8.23, *Short-Circuit Boolean Expressions*, on page 135)

Note: When reading *Guard predicates* and *Guard built-in functions*, you will find references to data types that we have not yet discussed. They are included in these tables for completeness.

When evaluating a guard expression, the precedence rules described in Section 8.20, *Operator Precedence*, on page 133 are used.

Guard Examples

We've talked about the syntax of guards, which can be fairly complex; here are a few examples:

```
f(X,Y) when is_integer(X), X > Y, Y < 6 -> ...
```

This means "When X is an integer, X is greater than Y, and Y is less than 6." The comma, which separates the test in the guard, means "and."

Predicate	*Meaning*
is_atom(X)	X is an atom.
is_binary(X)	X is a binary.
is_constant(X)	X is a constant.
is_float(X)	X is a float.
is_function(X)	X is a fun.
is_function(X, N)	X is a fun with N arguments.
is_integer(X)	X is an integer.
is_list(X)	X is a list.
is_map(X)	X is a map.
is_number(X)	X is an integer or a float.
is_pid(X)	X is a process identifier.
is_pmod(X)	X is an instance of a parameterized module.
is_port(X)	X is a port.
is_reference(X)	X is a reference.
is_tuple(X)	X is a tuple.
is_record(X,Tag)	X is a record of type Tag.
is_record(X,Tag,N)	X is a record of type Tag and size N.

Table 1—Guard predicates

Function	*Meaning*
abs(X)	Absolute value of X.
byte_size(X)	The number of bytes in X. X must be a bitstring or a binary.
element(N, X)	Element N of X. Note X must be a tuple.
float(X)	Converts X, which must be a number, to a float.
hd(X)	The head of the list X.
length(X)	The length of the list X.
node()	The current node.
node(X)	The node on which X was created. X can be a process, an identifier, a reference, or a port.
round(X)	Converts X, which must be a number, to an integer.
self()	The process identifier of the current process.
size(X)	The size of X. X can be a tuple or a binary.
trunc(X)	Truncates X, which must be a number, to an integer.
tl(X)	The tail of the list X.
tuple_size(T)	The size of the tuple T.

Table 2—Guard built-in functions

```
is_tuple(T), tuple_size(T) =:= 6, abs(element(3, T)) > 5
element(4, X) =:= hd(L)
...
```

The first line means T is a tuple of six elements, and the absolute value of the third element of T is greater than 5. The second line means that element 4 of the tuple X is identical to the head of the list L.

```
X =:= dog; X =:= cat
is_integer(X), X > Y ; abs(Y) < 23
...
```

The first guard means X is either a cat or a dog, and the semicolon (;) in the guard means "or." The second guard means that X is an integer and is greater than Y or the absolute value of Y is less than 23.

Here are some examples of guards that use short-circuit boolean expressions:

```
A >= -1.0 andalso A+1 > B
is_atom(L) orelse (is_list(L) andalso length(L) > 2)
```

The reason for allowing boolean expressions in guards is to make guards syntactically similar to other expressions. The reason for the orelse and andalso operators is that the boolean operators and/or were originally defined to evaluate both their arguments. In guards, there can be differences between (and and andalso) or between (or and orelse). For example, consider the following two guards:

```
f(X) when (X == 0) or (1/X > 2) ->
    ...

g(X) when (X == 0) orelse (1/X > 2) ->
    ...
```

The guard in f(X) fails when X is zero but succeeds in g(X).

In practice, few programs use complex guards, and simple (,) guards suffice for most programs.

Use of the true Guard

You might wonder why we need the true guard at all. The reason is that atom true can be used as a "catchall" guard at the end of an if expression, like this:

```
if
    Guard -> Expressions;
    Guard -> Expressions;
    ...
    true  -> Expressions
end
```

if will be discussed in *if Expressions, on page 69*.

The following tables list all the guard predicates (that is, guards that return booleans) and all the guard functions.

4.8 case and if Expressions

So far, we've used pattern matching for *everything*. This makes Erlang code small and consistent. But sometimes defining separate function clauses for everything is rather inconvenient. When this happens, we can use case or if expressions.

case Expressions

case has the following syntax:

```
case Expression of
    Pattern1 [when Guard1] -> Expr_seq1;
    Pattern2 [when Guard2] -> Expr_seq2;
    ...
end
```

case is evaluated as follows: First, Expression is evaluated; assume this evaluates to Value. Thereafter, Value is matched in turn against Pattern1 (with the optional guard Guard1), Pattern2, and so on, until a match is found. As soon as a match is found, then the corresponding expression sequence is evaluated—the result of evaluating the expression sequence is the value of the case expression. If no pattern matches, then an exception is raised.

Earlier, we used a function called filter(P, L); it returns a list of all those elements X in L for which P(X) is true. Using case we can define filter as follows:

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

Strictly speaking, case is unnecessary. This is how filter would have been defined using pure pattern matching:

```
filter(P, [H|T]) ->  filter1(P(H), H, P, T);
filter(P, [])    ->  [].

filter1(true, H, P, T)  -> [H|filter(P, T)];
filter1(false, H, P, T) -> filter(P, T).
```

> **Obsolete Guard Functions**
>
> If you come across some old Erlang code written a few years ago, the names of the guard tests were different. Old code used guard tests called atom(X), constant(X), float(X), integer(X), list(X), number(X), pid(X), port(X), reference(X), tuple(X), and binary(X). These tests have the same meaning as the modern tests named is_atom(X).... The use of old names in modern code is frowned upon.

This definition is rather ugly; we have to invent an additional function (called filter1) and pass it all of the arguments of filter/2.

if Expressions

A second conditional primitive, if, is also provided. Here is the syntax:

```
if
  Guard1 ->
    Expr_seq1;
  Guard2 ->
    Expr_seq2;
  ...
end
```

This is evaluated as follows: First Guard1 is evaluated. If this evaluates to true, then the value of if is the value obtained by evaluating the expression sequence Expr_seq1. If Guard1 does not succeed, Guard2 is evaluated, and so on, until a guard succeeds. At least one of the guards in the if expression must evaluate to true; otherwise, an exception will be raised.

Often the final guard in an if expression is the atom true, which guarantees that the last form in the expression will be evaluated if all other guards have failed.

One point that can lead to confusion is the use of a final true guard in an if expression. If you come from a language like C, you can write an if statement that does not have an else part, like this:

```
if( a > 0) {
    do_this();
}
```

So, you might be tempted to write the following in Erlang:

```
if
   A > 0 ->
       do_this()
end
```

This might lead to a problem in Erlang because if is an expression, and all expressions are supposed to have values. In the case where A is less or equal to zero, the if expression has no value. This would be an error in Erlang and cause the program to crash. But it would not be an error in C.

To avoid a possible exception, the Erlang programmer will often add a true guard at the end of an if expression. Or course, if they want an exception to be generated, then they omit the additional true guard.

4.9 Building Lists in Natural Order

The most efficient way to build a list is to add the elements to the head of an existing list, so we often see code with this kind of pattern:

```
some_function([H|T], ..., Result, ...) ->
    H1 = ... H ...,
    some_function(T, ..., [H1|Result], ...);
some_function([], ..., Result, ...) ->
    {..., Result, ...}.
```

This code walks down a list extracting the head of the list H and computes some value based on this function (we can call this H1); it then adds H1 to the output list Result. When the input list is exhausted, the final clause matches, and the output variable Result is returned from the function.

The elements in Result are in the opposite order as the elements in the original list, which may or may not be a problem, but if they are in the wrong order, they can easily be reversed in the final step.

The basic idea is fairly simple.

1. Always add elements to a list head.

2. Taking the elements from the head of an *InputList* and adding them head first to an *OutputList* results in the *OutputList* having the reverse order of the *InputList*.

3. If the order matters, then call lists:reverse/1, which is highly optimized.

4. Avoid going against these recommendations.

Note: Whenever you want to reverse a list, you should call lists:reverse and nothing else. If you look in the source code for the module lists, you'll find a definition of reverse. However, this definition is simply used for illustration. The compiler, when it finds a call to lists:reverse, calls a more efficient internal version of the function.

If you ever see code like the following, it should set warning bells sounding in your brain—this is very inefficient and acceptable only if List is short:

```
List ++ [H]
```

Even though ++ might lead to inefficient code, there is a trade-off between clarity and performance. Using ++ might lead to a clearer program without performance problems. The best thing to do is first write your programs as clearly as possible and then, if there are performance problems, measure before making any optimizations.

4.10 Accumulators

Often we want to return two lists from a function. For example, we might want to write a function that splits a list of integers into two lists that contain the even and odd integers in the original list. Here's one way of doing it:

lib_misc.erl
```
odds_and_evens1(L) ->
    Odds  = [X || X <- L, (X rem 2) =:= 1],
    Evens = [X || X <- L, (X rem 2) =:= 0],
    {Odds, Evens}.

5> lib_misc:odds_and_evens1([1,2,3,4,5,6]).
{[1,3,5],[2,4,6]}
```

The problem with this code is that we traverse the list *twice*—this doesn't matter when the list is short, but if the list is very long, it might be a problem.

To avoid traversing the list twice, we can re-code this as follows:

lib_misc.erl
```
odds_and_evens2(L) ->
    odds_and_evens_acc(L, [], []).

odds_and_evens_acc([H|T], Odds, Evens) ->
    case (H rem 2) of
        1 -> odds_and_evens_acc(T, [H|Odds], Evens);
        0 -> odds_and_evens_acc(T, Odds, [H|Evens])
    end;
odds_and_evens_acc([], Odds, Evens) ->
    {Odds, Evens}.
```

Now this traverses the list only once, adding the odd and even arguments onto the appropriate output lists (which are called *accumulators*). This code also has an additional benefit, which is less obvious; the version with an accumulator is more *space efficient* than the version with the [H || filter(H)] type construction.

If we run this, we get *almost* the same result as before.

```
1> lib_misc:odds_and_evens2([1,2,3,4,5,6]).
{[5,3,1],[6,4,2]}
```

The difference is that the order of the elements in the odd and even lists is reversed. This is a consequence of the way that the list was constructed. If we want the list elements in the same order as they were in the original, all we have to do is reverse the lists in the final clause of the function by changing the second clause of odds_and_evens2 to the following:

```
odds_and_evens_acc([], Odds, Evens) ->
    {lists:reverse(Odds), lists:reverse(Evens)}.
```

You now know enough to write and understand a significant amount of Erlang code. We've covered the basic structure of modules and functions and most of the control structures and programming techniques we need to write sequential programs.

Erlang has two more data types called *records* and *maps*. Both are used for storing complex data types. Records are used to give names to the elements of a tuple. This is useful when the number of elements in a tuple is large. Records and maps are the subject of the next chapter.

Exercises

Find the manual page for the erlang module. You'll see it lists a large number of BIFs (far more than we've covered here). You'll need this information to solve some of the following problems:

1. Extend geometry.erl. Add clauses to compute the areas of circles and right-angled triangles. Add clauses for computing the perimeters of different geometric objects.

2. The BIF tuple_to_list(T) converts the elements of the tuple T to a list. Write a function called my_tuple_to_list(T) that does the same thing only not using the BIF that does this.

3. Look up the definitions of erlang:now/0, erlang:date/0, and erlang:time/0. Write a function called my_time_func(F), which evaluates the fun F and times how long it takes. Write a function called my_date_string() that neatly formats the current date and time of day.

4. *Advanced:* Look up the manual pages for the Python datetime module. Find out how many of methods in the Python datetime class can be implemented using the time-related BIFs in the erlang module. Search the erlang manual pages for equivalent routines. Implement any glaring omissions.

5. Write a module called math_functions.erl, exporting the functions even/1 and odd/1. The function even(X) should return true if X is an even integer and otherwise false. odd(X) should return true if X is an odd integer.

6. Add a higher-order function to math_functions.erl called filter(F, L), which returns all the elements X in L for which F(X) is true.

7. Add a function split(L) to math_functions.erl, which returns {Even, Odd} where Even is a list of all the even numbers in L and Odd is a list of all the odd numbers in L. Write this function in two different ways using accumulators and using the function filter you wrote in the previous exercise.

Records and Maps

So far we have talked about two containers for data, namely, *tuples* and *lists*. Tuples are used to store a fixed number of elements, and lists are used for a variable number of elements.

This chapter introduces *records* and *maps*. Records are really just tuples in disguise. Using records we can associate a name with each element in a tuple.

Maps are associative collections of key-value pairs. The key can be any Erlang term. In Perl and Ruby they are called hashes; in C++ and Java they are called maps, in Lua they are called tables, and in Python they are called dictionaries.

Using records and maps makes programming easier; instead of remembering where a data item is stored in a complex data structure, we just use the name of the item and the system figures out where the data is stored. Records use a fixed and predefined set of names; maps can add new names dynamically.

5.1 When to Use Maps or Records

Records are just tuples in disguise, so they have the same storage and performance characteristics as tuples. Maps use more storage than tuples and have slower lookup properties. On the other hand, maps are far more flexible than tuples.

Records should be used in the following cases:

- When you can represent your data using a fixed number of predetermined atoms

- When the number of elements in the record and the names of the elements will not change with time

- When storage is an issue, typically when you have a large array of tuples and each tuple has the same structure

Maps are appropriate for the following cases:

- Representing key-value data structures where the keys are not known in advance

- Representing data with large numbers of different keys

- As a ubiquitous data structure where efficiency is not important but convenience of use is

- For "self-documenting" data structures, that is, data structures where the user can make a good guess at the meaning of the value of a key from the key name

- For representing key-value parse trees such as XML or configuration files

- For communication with other programming languages, using JSON

5.2 Naming Tuple Items with Records

In a small tuple, remembering what the individual elements represent is rarely a problem, but when there are a large number of elements in the tuple, it becomes convenient to name the individual elements. Once we have named the elements, we will be able to refer to them using the name and not have to remember what position they had in the tuple.

To name the elements in a tuple, we use a record declaration that has the following syntax:

```erlang
-record(Name, {
            %% the next two keys have default values
            key1 = Default1,
            key2 = Default2,
            ...
            %% The next line is equivalent to
            %% key3 = undefined
            key3,
            ...

        }).
```

Warning: record is not a shell command (use rr in the shell; see the description that comes later in this section). Record declarations can be used only in Erlang source code modules and *not* in the shell.

In the previous example, Name is the name of the record. key1, key2, and so on, are the names of the fields in the record; they must always be atoms. Each field in a record can have a default value that is used if no value for this particular field is specified when the record is created.

For example, suppose we want to manipulate a to-do list. We start by defining a todo record and storing it in a file (record definitions can be included in Erlang source code files or put in files with the extension .hrl, which are then included by Erlang source code files).

Note that file inclusion is the only way to ensure that several Erlang modules use the same record definitions. This is similar to the way common definitions are defined in .h files in C and included by source code files. Details of the include directive can be found in .

records.hrl
```
-record(todo, {status=reminder,who=joe,text}).
```

Once a record has been defined, instances of the record can be created.

To do this in the shell, we have to read the record definitions into the shell before we can define a record. We use the shell function rr (short for *read records*) to do this.

```
1> rr("records.hrl").
[todo]
```

Creating and Updating Records

Now we're ready to define and manipulate records.

```
2> #todo{}.
#todo{status = reminder,who = joe,text = undefined}
3> X1 = #todo{status=urgent, text="Fix errata in book"}.
#todo{status = urgent,who = joe,text = "Fix errata in book"}
4> X2 = X1#todo{status=done}.
#todo{status = done,who = joe,text = "Fix errata in book"}
```

In lines 2 and 3 we *created* new records. The syntax #todo{key1=Val1, …, keyN=ValN} is used to create a new record of type todo. The keys are all atoms and must be the same as those used in the record definition. If a key is omitted, then a default value is assumed for the value that comes from the value in the record definition.

In line 4 we *copied* an existing record. The syntax X1#todo{status=done} says to create a copy of X1 (which must be of type todo), changing the field value status to done. Remember, this makes a *copy* of the original record; the original record is not changed.

Extracting the Fields of a Record

To extract several fields of a record in one operation, we use pattern matching.

```
5> #todo{who=W, text=Txt} = X2.
#todo{status = done,who = joe,text = "Fix errata in book"}
6> W.
joe
7> Txt.
"Fix errata in book"
```

On the left side of the match operator (=), we write a record pattern with the unbound variables W and Txt. If the match succeeds, these variables get bound to the appropriate fields in the record. If we want just one field of a record, we can use the "dot syntax" to extract the field.

```
8> X2#todo.text.
"Fix errata in book"
```

Pattern Matching Records in Functions

We can write functions that pattern match on the fields of a record and that create new records. We usually write code like this:

```
clear_status(#todo{status=S, who=W} = R) ->
    %% Inside this function S and W are bound to the field
    %% values in the record
    %%
    %% R is the *entire* record
    R#todo{status=finished}
    %% ...
```

To match a record of a particular type, we might write the function definition.

```
do_something(X) when is_record(X, todo) ->
    %% ...
```

This clause matches when X is a record of type todo.

Records Are Tuples in Disguise

Records are just tuples.

```
9> X2.
#todo{status = done,who = joe,text = "Fix errata in book"}
```

Now let's tell the shell to forget the definition of todo.

```
10> rf(todo).
ok
11> X2.
{todo,done,joe,"Fix errata in book"}
```

In line 10 the command rf(todo) told the shell to forget the definition of the todo record. So, now when we print X2, the shell displays X2 as a tuple. Internally

there are only tuples. Records are a syntactic convenience, so you can refer to the different elements in a tuple by name and not position.

5.3 Maps: Associative Key-Value Stores

Maps were made available from version R17 of Erlang.

Maps have the following properties:

- The syntax of maps is similar to that of records, the difference being that the record name is omitted and the key-value separator is either => or :=.

- Maps are associative collections of key-value pairs.

- The keys in a map can be any *fully ground* Erlang term (fully grounded means that there are no unbound variables in the term).

- The elements in a map are ordered by the keys.

- Updating a map where the keys are not changed is a space-efficient operation.

- Looking up the value of a key in a map is an efficient operation.

- Maps have a well-defined order.

We'll look at maps in more detail in the following sections.

The Semantics of Maps

Map literals are written with the following syntax:

```
#{ Key1 Op Val1, Key2 Op Val2, ..., KeyN Op ValN }
```

This has a similar syntax to records, but there is no record name following the hash symbol, and Op is one of the symbols => or :=.

The keys and values can be any valid Erlang terms. For example, suppose we want to create a map with two keys, a and b.

```
1> F1 = #{ a => 1, b => 2 }.
#{ a => 1, b => 2 }.
```

Or suppose we want to create a map with nonatomic keys.

```
2> Facts = #{ {wife,fred} => "Sue", {age, fred} => 45,
   {daughter,fred} => "Mary",
   {likes, jim} => [...]}.
#{ {age, fred} => 45, {daughter,fred} => "Mary", ...}
```

Internally the map is stored as an ordered collection and will be always printed using the sort order of the keys, irrespective of how the map was created. Here's an example:

```
3> F2 = #{ b => 2, a => 1 }.
#{ a => 1, b => 2 }.
4> F1 = F2.
#{ a => 1, b => 2 }.
```

To update a map based on an existing map, we use the following syntax where Op (the update operator) is => or :=:

```
NewMap = OldMap # { K1 Op V1,...,Kn Op Vn }
```

The expression K => V is used for two purposes, either to update the value of an existing key K with a new value V or to add a completely new K-V pair to the map. This operation always succeeds.

The expression K := V is used to update the value of an existing key K with a new value V. *This operation fails if the map being updated does not contain the key K.*

```
5> F3 = F1#{ c => xx }.
#{ a => xx, b => 2 , c => xx}
6> F4 = F1#{ c := 3}
** exception error: bad argument
key c does not exist in old map
```

There are two good reasons for using the := operator. First, if we misspell the name of the new key, we want an error to occur. If we create a map Var = #{keypos => 1, ...} and later update it with Var #{key_pos := 2 }, then we have almost certainly spelled the keyname incorrectly and we want to know about it. The second reason has to do with efficiency. If we use only the := operator in a map update operation, then we know that the old and new maps have an identical set of keys and thus can share the same key descriptor. If we had, for example, a list with a few million maps, all with the same keys, then the space savings would be significant.

The best way to use maps is to always use Key => Val the first time a key is defined and use Key := Val each time the value of a specific key is changed.

Pattern Matching the Fields of a Map

The => syntax we used in a map literal can also be used as a map pattern. As before, the keys in a map pattern cannot contain any unbound variables, but the value can now contain variables that become bound if the pattern match succeeds.

Maps in Other Languages

Note that maps in Erlang work in a very different manner than the equivalent constructs in many other programming languages. To illustrate this, we can take a look at what happens in JavaScript.

Suppose we do the following in JavaScript:

```javascript
var x = {status:'old', task:'feed cats'};
var y = x;
y.status = 'done';
```

The value of y is the object {status:'done', task:'feed cats'}. No surprises here. But surprise, surprise, x has changed to {status:'done', task:'feed cats'}. This comes as a great surprise to an Erlang programmer. We managed to change the value of one of the fields of the variable x, not by referring to x but by assigning a value to a field of the variable y. Changing x through an aliased pointer leads to many kinds of subtle errors that can be very difficult to debug.

The logically equivalent Erlang code is as follows:

```erlang
D1 = {status=>old, task=>'feed cats'},
D2 = D1#{status := done},
```

In the Erlang code, the variables D1 and D2 never change their initial values. D2 behaves exactly as if it were a deep copy of D1. In fact, a deep copy is not made; the Erlang system copies only those parts of the internal structures necessary to maintain the illusion that a copy has been created, so creating what appears to be deep copies of an object is an extremely lightweight operation.

```erlang
1> Henry8 = #{ class => king, born => 1491, died => 1547 }.
#{ born => 1491, class=> king, died => 1547 }.
2> #{ born => B } = Henry8.
#{ born => 1491, class=> king, died => 1547 }.
3> B.
1491
4> #{ D => 1547 }.
* 4: variable 'D' unbound
```

In line 1 we create a new map containing information about Henry VIII. In line 2 we create a pattern to extract the value associated with the born key from the map. The pattern matching succeeds and the shell prints the value of the entire map. In line 3 we print the value of the variable B.

In line 4 we tried to find some unknown key (D) whose value was 1547. But the shell prints an error since all keys in a map must be fully ground terms and D is undefined.

Note that the number of keys in the map pattern can be less than the number of keys in the map being matched.

We can use maps containing patterns in function heads, provided that all the keys in the map are known. For example, we can define a function count_characters(Str) that returns a map of the number of times a particular character occurs in a string.

```erlang
count_characters(Str) ->
   count_characters(Str, #{}).

count_characters([H|T], #{ H => N }=X) ->
   count_characters(T, X#{ H := N+1 });
count_characters([H|T], X) ->
   count_characters(T, X#{ H => 1 });
count_characters([], X) ->
   X.
```

Here's an example:

```erlang
1> count_characters("hello").
#{101=>1,104=>1,108=>2,111=>1}
```

So, the character h (ASCII, 101) occurred once, and so on. There are two things to note about count_characters/2. In the first clause, the variable H inside the map is also defined *outside* the map and thus is bound (as required). In the second clause, we used map_extend to add a new key to the map.

BIFs That Operate on Maps

A number of additional functions operate on maps. They are some of the functions in the module maps.

maps:new() -> #{}

> Return a new empty map.

erlang:is_map(M) -> bool()

> Return true if M is a map; otherwise, return false. This can be used as a guard test or in a function body.

maps:to_list(M) -> [{K1,V1},..., {Kn,Vn}]

> Convert the keys and values in the map M to a list of keys and values. The keys in the resulting list are in strict ascending order.

maps:from_list([{K1,V1},...,{Kn,Vn}]) -> M

> Convert a lists of pairs to a map M. If the same key occurs more than once, then the value associated with first key in the list will be used, and any subsequent values will be ignored.

maps:map_size(Map) -> NumberOfEntries

> Return the number of entries in the map.

maps:is_key(Key, Map) -> bool()

> Return true if the map contains an item with key Key; otherwise, return false.

maps:get(Key, Map) -> Val

> Return the value associated with Key from the map; otherwise, raise an exception.

maps:find(Key, Map) -> {ok, Value} | error

> Return the value associated with Key from the map; otherwise, return error.

maps:keys(Map) -> [Key1,..KeyN]

> Return a list of keys, in ascending order, that are in the map.

maps:remove(Key, M) -> M1

> Return a new map M1 that is the same as M except that the item with key Key (if present) has been removed.

maps:without([Key1, ..., KeyN], M) -> M1

> Return a new map M1 that is a copy of M but with any elements having keys in the list [Key1,...,KeyN] removed.

maps:difference(M1, M2) -> M3

> M3 is equivalent to M1 with any elements having the same keys as the elements in M2 removed.

> This behaves as if it had been defined as follows:

```
maps:difference(M1, M2) ->
    maps:without(maps:keys(M2), M1).
```

Ordering of Maps

Maps are compared by comparing first their size and then their keys and values in the sort order of their keys.

If A and B are maps, then A < B if maps:size(A) < maps:size(B).

If A and B are maps of equal size, then A < B if maps:to_list(A) < maps:to_list(B).

So, for example, A = #{age => 23, person => "jim"} is less than B = # {email => "sue@somplace.com", name => "sue"}. This is because the smallest key in A (age) is smaller than the smallest key in B (email).

When comparing maps with other Erlang terms, maps are viewed as being "more complex" than lists or tuples, and thus a map is always considered greater than a list or tuple.

Maps can be output with the ~p option in io:format and read with io:read or file:consult.

The JSON Bridge

Those of you who are familiar with JSON will notice the similarity between maps and JSON terms. Two BIFS convert between maps and JSON terms.

maps:to_json(Map) -> Bin

> Converts a map to a binary containing the JSON representation of the map. Binaries are discussed in Chapter 7, *Binaries and the Bit Syntax, on page 99*. Note that not all maps can be converted to JSON terms. All the values in the map must be objects that can be represented in JSON. So, for example, values cannot include objects such as funs, PIDs, references, and so on. maps:to_json fails if any of the keys or values cannot be represented in JSON.

maps:from_json(Bin) -> Map

> Converts a binary containing a JSON term to a map.

maps:safe_from_json(Bin) -> Map

> Converts a binary containing a JSON term to a map. Any atoms in Bin must exist before the BIF is called; otherwise, an exception will be raised. The reason for this is to avoid creating large numbers of new atoms. For reasons of efficiency, Erlang does not garbage collect atoms, so continuously adding new atoms will, after a very long time, kill the Erlang VM.

In both the previous definitions Map must be an instance of the type json_map(), which is defined as follows (type definitions will be introduced later in Chapter 9, *Types*, on page 141):

```
-type json_map() = [{json_key(), json_value()}].
```

Where:

```
-type json_key() =
    atom() | binary() | io_list()
```

and:

```
-type json_value() =
    integer() | binary() | float() | atom() | [json_value()] | json_map()
```

The mapping between JSON objects and Erlang values is as follows:

- *JSON numbers* are represented as Erlang integers or floats.

- *JSON strings* are represented as Erlang binaries.

- *JSON lists* are represented as Erlang lists.

- *JSON true* and *false* are represented as Erlang atoms true and false.

- *JSON objects* are represented as Erlang maps, with the restriction that the keys in the map must be atoms, strings, or binaries, and the values must be representable as JSON terms.

When we convert to and from JSON terms, we should be aware of certain limitations of the conversion. Erlang provides integers with unlimited precision. So, Erlang will happily convert a bignum in a map into a bignum in a JSON term; this may or may not be understandable by the program that decodes the JSON term.

In Chapter 18, *Browsing with Websockets and Erlang*, on page 287, you'll find out how to use maps combined with JSON terms and websockets to provide a simple method of communicating with an application running inside a web browser.

We've now covered all the ways there are of creating compound data structures in Erlang. We know about lists as containers for a variable number of items and tuples as containers for a fixed number of items. Records are used to add symbolic names to the elements of a tuple, and maps are used as associative arrays.

In the next chapter, we'll look at error handling. After this, we'll get back to sequential programming and then look at binaries and the bit syntax that we've omitted up to now.

Exercises

1. Configuration files can be conveniently represented as JSON terms. Write some functions to read configuration files containing JSON terms and turn them into Erlang maps. Write some code to perform sanity checks on the data in the configuration files.

2. Write a function map_search_pred(Map, Pred) that returns the first element {Key,Value} in the map for which Pred(Key, Value) is true.

3. *Advanced:* Look up the manual pages for the Ruby hash class. Make a module of the methods in the Ruby class that you think would be appropriate to Erlang.

Error Handling in Sequential Programs

Erlang was originally designed for programming fault-tolerant systems, systems that in principle should never stop. This means that dealing with errors at runtime is crucially important. We take error handling very seriously in Erlang. When errors occur, we need to detect them, correct them, and continue.

Typical Erlang applications are composed of dozens to millions of concurrent processes. Having large numbers of processes changes how we think about error handling. In a sequential language with only one process, it is crucially important that this process does not crash. If we have large numbers of processes, it is not so important if a process crashes, provided some other process can detect the crash and take over whatever the crashed process was supposed to be doing.

To build really fault-tolerant systems, we need more than one computer; after all, the entire computer might crash. So, the idea of detecting failure and resuming the computation elsewhere has to be extended to networked computers.

To fully understand error handling, we first need to look at error handling in sequential programs and then, having understood this, see how to handle errors in collections of parallel processes. This chapter looks at the former problem. Handling errors in parallel processes is dealt with in Chapter 13, *Errors in Concurrent Programs, on page 199*, and building sets of processes that collaborate to correct errors is the subject of Section 23.5, *The Supervision Tree, on page 396*.

6.1 Handling Errors in Sequential Code

Every time we call a function in Erlang, one of two things will happen: either the function returns a value or something goes wrong. We saw examples of this in the previous chapter. Remember the cost function?

```
shop.erl
cost(oranges)   -> 5;
cost(newspaper) -> 8;
cost(apples)    -> 2;
cost(pears)     -> 9;
cost(milk)      -> 7.
```

This is what happened when we ran it:

```
1> shop:cost(apples).
2
2> shop:cost(socks).
** exception error: no function clause matching
   shop:cost(socks) (shop.erl, line 5)
```

When we called cost(socks), the function crashed. This happened because none of the clauses that define the function matched the calling arguments.

Calling cost(socks) is pure nonsense. There is no sensible value that the function can return, since the price of socks is undefined. In this case, instead of returning a value, the system *raises an exception*—this is the technical term for "crashing."

We don't try to repair the error because this is not possible. We don't know what socks cost, so we can't return a value. It is up to the *caller* of cost(socks) to decide what to do if the function crashes.

Exceptions are raised by the system when internal errors are encountered or explicitly in code by calling throw(Exception), exit(Exception), or error(Exception). When we evaluated cost(socks), a pattern matching error occurred. There was no clause defining the cost of socks, so the system automatically generated an error.

Typical internal errors that raise exceptions are pattern matching errors (no clauses in a function match) or calling BIFs with incorrectly typed arguments (for example, calling atom_to_list with an argument that is an integer) or calling a BIF with an incorrect value of an argument (for example, trying to divide a number by zero).

Note: Many languages say you should use *defensive programming* and check the arguments to all functions. In Erlang, defensive programming is built-in. You should describe the behavior of functions only for valid input arguments;

all other arguments will cause internal errors that are automatically detected. You should never return values when a function is called with invalid arguments. You should always raise an exception. This rule is called "Let it crash."

We can explicitly generate an error by calling one of the following BIFs:

exit(Why)

> This is used when you really want to terminate the current process. If this exception is not caught, the signal {'EXIT',Pid,Why} will be broadcast to all processes that are linked to the current process. We haven't met signals yet, but we'll say a lot more about this in Section 13.3, *Creating Links, on page 203*. Signals are almost like error messages, but I won't dwell on the details here.

throw(Why)

> This is used to throw an exception that a caller might want to catch. In this case, we *document* that our function might throw this exception. The user of this function has two alternatives: you can program for the common case and blissfully ignore exceptions, or you can enclose the call in a try…catch expression and handle the errors.

error(Why)

> This is used for denoting "crashing errors." That is, something rather nasty has happened that callers are not really expected to handle. This is on par with internally generated errors.

Erlang has two methods of *catching* an exception. One is to enclose the call to the function that raised the exception within a try…catch expression. The other is to enclose the call in a catch expression.

6.2 Trapping an Exception with try…catch

If you're familiar with Java, then you'll have no difficulties understanding the try…catch expression. Java can trap an exception with the following syntax:

```
try {
    block
} catch (exception type identifier) {
    block
} catch (exception type identifier) {
    block
} ...
finally {
    block
}
```

Erlang has a remarkably similar construct, which looks like this:

```
try FuncOrExpressionSeq of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
catch
    ExceptionType1: ExPattern1 [when ExGuard1] -> ExExpressions1;
    ExceptionType2: ExPattern2 [when ExGuard2] -> ExExpressions2;
    ...
after
    AfterExpressions
end
```

try…catch Has a Value

Remember, everything in Erlang is an expression, and all expressions have values. We talked about this earlier in *if Expressions*, on page 69, when discussing why the if expression didn't have an else part. This means the expression try…end also has a value. So, we might write something like this:

```
f(...) ->
...
X = try ... end,
Y = g(X),
...
```

More often, we don't need the value of the try…catch expression. So, we just write this:

```
f(...) ->
    ...
    try ... end,
    ...
    ...
```

Notice the similarity between the try…catch expression and the case expression.

```
case Expression of
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
end
```

try…catch is like a case expression on steroids. It's basically a case expression with catch and after blocks at the end.

try…catch works as follows: First FuncOrExpessionSeq is evaluated. If this finishes without raising an exception, then the return value of the function is pattern matched against the patterns Pattern1 (with optional guard Guard1), Pattern2, and so on, until a match is found. If a match is found, then the value of the entire

try…catch is found by evaluating the expression sequence following the matching pattern.

If an exception is raised within FuncOrExpressionSeq, then the catch patterns ExPattern1, and so on, are matched to find which sequence of expressions should be evaluated. ExceptionType is an atom (one of throw, exit, or error) that tells us how the exception was generated. If ExceptionType is omitted, then the value defaults to throw.

Note: Internal errors that are detected by the Erlang runtime system always have the tag error.

The code following the after keyword is used for cleaning up after FuncOrExpressionSeq. This code is guaranteed to be executed, even if an exception is raised. The code in the after section is run immediately after any code in Expressions in the try or catch section of the expression. The return value of AfterExpressions is lost.

If you're coming from Ruby, all of this should seem very familiar. In Ruby, we'd write a similar pattern.

```
begin
    ...
rescue
    ...
ensure
    ...
end
```

The keywords are different, but the behavior is similar.

Shortcuts

We can omit several of the parts of a try…catch expression. This:

```
try F
catch
    ...
end
```

means the same as this:

```
try F of
  Val -> Val
catch
    ...
end
```

Also, the after section can be omitted.

Programming Idioms with try…catch

When we design applications, we often make sure that the code that catches an error can catch all the errors that a function can produce.

Here's a pair of functions illustrating this. The first function generates three different types of an exception and has two ordinary return values.

try_test.erl
```erlang
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> error(a).
```

Now we'll write a wrapper function to call generate_exception in a try…catch expression.

try_test.erl
```erlang
demo1() ->
    [catcher(I) || I <- [1,2,3,4,5]].

catcher(N) ->
   try generate_exception(N) of
      Val -> {N, normal, Val}
   catch
      throw:X -> {N, caught, thrown, X};
      exit:X  -> {N, caught, exited, X};
      error:X -> {N, caught, error, X}
   end.
```

Running this we obtain the following:

```erlang
> try_test:demo1().
[{1,normal,a},
 {2,caught,thrown,a},
 {3,caught,exited,a},
 {4,normal,{'EXIT',a}},
 {5,caught,error,a}]
```

This shows that we can trap and distinguish all the forms of exception that a function can raise.

6.3 Trapping an Exception with catch

The other way to trap an exception is to use the primitive catch. The catch primitive is not the same as the catch block in the try…catch statement (this is because the catch statement was part of the language long before try…catch was introduced).

When an exception occurs within a catch statement, it is converted into an {'EXIT', ...} tuple that describes the error. To demonstrate this, we can call generate_exception within a catch expression.

try_test.erl
```erlang
demo2() ->
    [{I, (catch generate_exception(I))} || I <- [1,2,3,4,5]].
```

Running this we obtain the following:

```erlang
2> try_test:demo2().
[{1,a},
 {2,a},
 {3,{'EXIT',a}},
 {4,{'EXIT',a}},
 {5,{'EXIT',
      {a,[{try_test,generate_exception,1,
                  [{file,"try_test.erl"},{line,9}]},
        {try_test,'-demo2/0-lc$^0/1-0-',1,
                  [{file,"try_test.erl"},{line,28}]},
        {try_test,'-demo2/0-lc$^0/1-0-',1,
                  [{file,"try_test.erl"},{line,28}]},
        {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
        {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}]}]}}}]
```

If you compare this with the output from the try…catch section, you'll see that the two methods provide differing amounts of debug information. The first method summarized the information. The second provided a detailed stack trace.

6.4 Programming Style with Exceptions

Handling exceptions is not rocket science; the following sections contain some frequently occurring code patterns that we can reuse in our programs.

Improving Error Messages

One use of the error/1 BIF is to improve the quality of error messages. If we call math:sqrt(X) with a negative argument, we'll see the following:

```erlang
1> math:sqrt(-1).
** exception error: bad argument in an arithmetic expression
in function  math:sqrt/1
   called as math:sqrt(-1)
```

We can write a wrapper for this, which improves the error message.

```
lib_misc.erl
sqrt(X) when X < 0 ->
    error({squareRootNegativeArgument, X});
sqrt(X) ->
    math:sqrt(X).

2> lib_misc:sqrt(-1).
** exception error: {squareRootNegativeArgument,-1}
    in function  lib_misc:sqrt/1
```

Code Where Error Returns Are Common

If your function does not really have a "common case," you should probably return something like {ok, Value} or {error, Reason}, but remember that this forces all callers to do *something* with the return value. You then have to choose between two alternatives; you either write this:

```
...
case f(X) of
    {ok, Val} ->
        do_some_thing_with(Val);

    {error, Why} ->
        %% ... do something with the error ...
end,
...
```

which takes care of both return values, or write this:

```
...
{ok, Val} = f(X),
do_some_thing_with(Val);
...
```

which raises an exception if f(X) returns {error, ...}.

Code Where Errors Are Possible but Rare

Typically you should write code that is expected to handle errors, as in this example:

```
try my_func(X)
catch
  throw:{thisError, X} -> ...
  throw:{someOtherError, X} -> ...
end
```

And the code that detects the errors should have matching throws as follows:

```
my_func(X) ->
    case ... of
       ...
       ... ->
             ... throw({thisError, ...})
       ... ->
             ... throw({someOtherError, ...})
```

Catching Every Possible Exception

If we want to catch every possible error, we can use the following idiom (which uses the fact that _ matches anything):

```
try Expr
catch
    _:_ -> ... Code to handle all exceptions  ...
end
```

If we omit the tag and write this:

```
try Expr
catch
    _ -> ... Code to handle all exceptions  ...
    end
```

then we *won't* catch all errors, since in this case the default tag throw is assumed.

6.5 Stack Traces

When an exception is caught, we can find the latest stack trace by calling erlang:get_stacktrace(). Here's an example:

try_test.erl
```
demo3() ->
    try generate_exception(5)
    catch
       error:X ->
            {X, erlang:get_stacktrace()}
    end.
```

```
1> try_test:demo3().
{a,[{try_test,generate_exception,1,[{file,"try_test.erl"},{line,9}]},
    {try_test,demo3,0,[{file,"try_test.erl"},{line,33}]},
    {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,576}]},
    {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 previous trace shows what happened when we tried to evaluate try_test:demo3(). It shows that our program crashed in the function generate_exception/1, which was defined in line 9 of the file try_test.erl.

The stack trace contains information about where the current function (which crashed) would have returned to had it succeeded. The individual tuples in the stack trace are of the form {Mod,Func,Arity,Info}. Mod, Func, and Arity denote a function, and Info contains the filename and line number of the item in the stack trace.

So, try_test:generate_exception/1 would have returned to try_test:demo3(), which would have returned to erl_eval:do_apply/6, and so on. If a function was called from the middle of a sequence of expressions, then the site of the call and the place to which the function will return are almost the same. If the function that was called was the last function in a sequence of expressions, then information about where the function was called from is not retained on the stack. Erlang applies a last-call optimization to such code, so the stack trace will not record where the function was called from, only where it will return to.

Examining the stack trace gives us a good indication of where the program was executing at the time when the error occurred. Normally the top two entries on the stack trace give you enough information to locate the place where the error occurred.

Now we know about handling errors in sequential programs. The important thing to remember is to *let it crash*. Never return a value when a function is called with an incorrect argument; raise an exception. Assume that the caller will fix the error.

6.6 Fail Fast and Noisily, Fail Politely

We need to consider two key principles when coding for errors. First, we should fail as soon as an error occurs, and we should fail noisily. Several programming languages adopt the principle of failing silently, trying to fix up the error and continuing; this results in code that is a nightmare to debug. In Erlang, when an error is detected internally by the system or is detected by program logic, the correct approach is to crash immediately and generate a meaningful error message. We crash immediately so as not to make matters worse. The error message should be written to a permanent error log and be sufficiently detailed so that we can figure out what went wrong later.

Second, fail politely means that only the programmer should see the detailed error messages produced when a program crashes. A user of the program should never see these messages. On the other hand, the user should be

alerted to the fact that an error has occurred and be told what action they can take to remedy the error.

Error messages are gold dust for programmers. They should never scroll up the screen to vanish forever. They should go to a permanent log file that can be read later.

At this point, we have covered errors only in sequential programs. In Chapter 13, *Errors in Concurrent Programs,* on page 199, we'll look at how errors can be managed in concurrent programs, and in Section 23.2, *The Error Logger,* on page 384, we'll see how to log errors permanently so we never lose them.

In the next chapter, we'll look at binaries and the bit syntax. The bit syntax is unique to Erlang and extends pattern matching over bit fields, which simplifies writing programs that manipulate binary data.

Exercises

1. file:read_file(File) returns {ok, Bin} or {error, Why}, where File is the filename and Bin contains the contents of the file. Write a function myfile:read(File) that returns Bin if the file can be read and raises an exception if the file cannot be read.

2. Rewrite the code in try_test.erl so that it produces two error messages: a polite message for the user and a detailed message for the developer.

Binaries and the Bit Syntax

A *binary* is a data structure designed for storing large quantities of raw data in a space-efficient manner. The Erlang VM is optimized for the efficient input, output, and message passing of binaries.

Binaries should be used whenever possible for storing the contents of large quantities of unstructured data, for example large strings or the contents of files.

In most circumstances, the number of bits in a binary will be exactly divisible by 8 and thus corresponds to a sequence of bytes. If the number of bits is not exactly divisible by 8, we use the name *bitstring* to refer to the data. When we say bitstring, it is to emphasize the fact that the number of bits in the data is not an exact multiple of 8.

Binaries, bitstrings, and bit-level pattern matching were introduced in Erlang to simplify network programming where we often want to probe into the bit- and byte-level structure of protocol packets.

In this chapter, we'll first take a detailed look at binaries. Most of the operations on binaries work in the same way on bitstrings, so after understanding binaries, we'll look at bitstrings emphasizing where they differ from binaries.

7.1 Binaries

Binaries are written and printed as sequences of integers or strings, enclosed in double less-than and greater-than brackets. Here's an example:

```
1> <<5,10,20>>.
<<5,10,20>>
2> <<"hello">>.
<<"hello">>
3> <<65,66,67>>
<<"ABC">>
```

When you use integers in a binary, each must be in the range 0 to 255. The binary <<"cat">> is shorthand for <<99,97,116>>; that is, the binary is made from the ASCII character codes of the characters in the string.

As with strings, if the content of a binary is a printable string, then the shell will print the binary as a string; otherwise, it will be printed as a sequence of integers.

We can build a binary and extract the elements of a binary using a BIF, or we can use the bit syntax (see Section 7.2, *The Bit Syntax*, on page 101). In this section, we'll look only at the BIFs that manipulate binaries.

Working with Binaries

We can manipulate binaries using BIFs or with functions from the binary module. Many of the functions exported from binary are implemented as native code. Here are some of the most important:

list_to_binary(L) -> B

list_to_binary returns a binary constructed by flattening (*flattening* means removing all the list parentheses) all the elements in the *iolist* L. An *iolist* is defined recursively as a list whose elements are integers in 0..255, binaries, or iolists.

```
1> Bin1 = <<1,2,3>>.
<<1,2,3>>
2> Bin2 = <<4,5>>.
<<4,5>>
3> Bin3 = <<6>>.
<<6>>
4> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

Note: The space surrounding the equals sign in line 1 is necessary. Without this space, the second symbol seen by the Erlang tokenizer would be the atom '=<', which is the equal-to-or-less-than operator. Sometimes we have to put spaces or parentheses around binary literals to avoid syntax errors.

split_binary(Bin, Pos) -> {Bin1, Bin2}

This splits the binary Bin into two parts at position Pos.

```
1> split_binary(<<1,2,3,4,5,6,7,8,9,10>>, 3).
{<<1,2,3>>,<<4,5,6,7,8,9,10>>}
```

term_to_binary(Term) -> Bin

This converts any Erlang term into a binary.

The binary produced by term_to_binary is represented in the so-called external term format. Terms that have been converted to binaries using term_to_binary can be stored in files, sent in messages over a network, and so on, and the original term from which they were made can be reconstructed later. This is extremely useful for storing complex data structures in files or sending complex data structures to remote machines.

binary_to_term(Bin) -> Term

This is the inverse of term_to_binary.

```
1> B = term_to_binary({binaries,"are", useful}).
<<131,104,3,100,0,8,98,105,110,97,114,105,101,115,107,
0,3,97,114,101,100,0,6,117,115,101,102,117,108>>
2> binary_to_term(B).
{binaries,"are",useful}
```

byte_size(Bin) -> Size

This returns the number of bytes in the binary.

```
1> byte_size(<<1,2,3,4,5>>).
5
```

Of all these, term_to_binary and binary_to_term are my absolute favorites. They are incredibly useful. term_to_binary turns any term into a binary. Inside the binary (if you peeked), you'll find data stored in "the Erlang external term format" (defined in the Erlang documentation).[1] Once we have converted a term to a binary, we can send the binary in a message over a socket or store it in a file. This is the basic underlying method used for implementing distributed Erlang and is used internally in many databases.

7.2 The Bit Syntax

The bit syntax is a notation used for extracting and packing individual bits or sequences of bits in binary data. When you're writing low-level code to pack and unpack binary data at a bit level, you'll find the bit syntax incredibly useful. The bit syntax was developed for protocol programming (something that Erlang excels at) and produces highly efficient code for manipulating binary data.

Suppose we have three variables—X, Y, and Z—that we want to pack into a 16-bit memory area. X should take 3 bits in the result, Y should take 7 bits, and Z should take 6. In most languages this involves some messy low-level operations involving bit shifting and masking. In Erlang, we just write the following:

M = <<X:3, Y:7, Z:6>>

---

1. http://erlang.org/doc/apps/erts/erl_ext_dist.html

This creates a binary and stores it in the variable M. Note: M is of type `binary` since the total bit length of the data is 16 bits, which is exactly divisible by 8. If we change the size of X to 2 bits and write this:

```
M = <<X:2, Y:7, Z:6>>
```

then the total number of bits in M is 15, so the resulting data structure is of type `bitstring`.

The full bit syntax is slightly more complex, so we'll go through it in small steps. First we'll look at some simple code to pack and unpack RGB color data into 16-bit words. Then we'll dive into the details of bit syntax expressions. Finally we'll look at three examples taken from real-world code that uses the bit syntax.

Packing and Unpacking 16-Bit Colors

We'll start with a very simple example. Suppose we want to represent a 16-bit RGB color. We decide to allocate 5 bits for the red channel, 6 bits for the green channel, and 5 bits for the blue channel. (We use one more bit for the green channel because the human eye is more sensitive to green light.)

We can create a 16-bit memory area Mem containing a single RGB triplet as follows:

```
1> Red = 2.
2
2> Green = 61.
61
3> Blue = 20.
20
4> Mem = <<Red:5, Green:6, Blue:5>>.
<<23,180>>
```

Note in expression 4 we created a 2-byte binary containing a 16-bit quantity. The shell prints this as <<23,180>>.

To pack the memory, we just wrote the expression <<Red:5, Green:6, Blue:5>>.

To unpack the binary into integer variables, R1, G1, and B1, we write a pattern.

```
5> <<R1:5, G1:6, B1:5>> = Mem.
<<23,180>>
6> R1.
2
7> G1.
61
8> B1.
20
```

That was easy. If you don't believe me, try doing that using bitshifts and logical ands and ors in your favorite programming language.

We can actually do far more with the bit syntax than this simple example suggests, but first we need to master a rather complex syntax. Once we've done this, we'll be able to write remarkably short code to pack and unpack complex binary data structures.

Bit Syntax Expressions

Bit syntax expressions are used to construct binaries or bitstrings. They have the following form:

```
<<>>
<<E1, E2, ..., En>>
```

Each element `Ei` specifies a single *segment* of the binary or bitstring. Each element `Ei` can have one of four possible forms.

```
Ei = Value |
     Value:Size |
     Value/TypeSpecifierList |
     Value:Size/TypeSpecifierList
```

If the total number of bits in the expression is evenly divisible by 8, then this will construct a binary; otherwise, it will construct a bitstring.

When you construct a binary, `Value` must be a bound variable, a literal string, or an expression that evaluates to an integer, a float, or a binary. When used in a pattern matching operation, `Value` can be a bound or unbound variable, integer, literal string, float, or binary.

`Size` must be an expression that evaluates to an integer. In pattern matching, `Size` must be an integer or a bound variable whose value is an integer. `Size` must be a bound variable, at the point in the pattern where the value is needed. The value of the `Size` can be obtained from earlier pattern matches in the binary. For example, the following:

```
<<Size:4, Data:Size/binary, ...>>
```

is a legal pattern, since the value of `Size` is unpacked from the first four bits of the binary and then used to denote the size of the next segment in the binary.

The value of `Size` specifies the size of the segment. The default value depends on the type. For an integer it is 8, for a float it is 64, and for a binary it is the size of the binary. In pattern matching, this default value is valid only for the very last element. If the size of a segment is not specified, a default value will be assumed.

TypeSpecifierList is a hyphen-separated list of items of the form End-Sign-Type-Unit. Any of the previous items can be omitted, and the items can occur in any order. If an item is omitted, then a default value for the item is used.

The items in the specifier list can have the following values:

End is big | little | native

This specifies the endianess of the machine. native is determined at runtime, depending upon the CPU of your machine. The default is big, which is also known as *network byte order*. The only significance of this has to do with packing and unpacking integers and floats from binaries. When packing and unpacking integers from binaries on different endian machines, you should take care to use the correct endianess.

When writing bit syntax expressions, some experimentation may be necessary. To assure yourself that you are doing the right thing, try the following shell command:

```
1> {<<16#12345678:32/big>>,<<16#12345678:32/little>>,
<<16#12345678:32/native>>,<<16#12345678:32>>}.
{<<18,52,86,120>>,<<120,86,52,18>>,
<<120,86,52,18>>,<<18,52,86,120>>}
```

The output shows you exactly how integers are packed in a binary using the bit syntax.

In case you're worried, term_to_binary and binary_to_term "do the right thing" when packing and unpacking integers. So, you can, for example, create a tuple containing integers on a big-endian machine. Then use term_to_binary to convert the term to a binary and send this to a little-endian machine. On the little-endian, you do binary_to_term, and all the integers in the tuple will have the correct values.

Sign is signed | unsigned

This parameter is used only in pattern matching. The default is unsigned.

Type is integer | float | binary | bytes | bitstring | bits | utf8 | utf16 | utf32

The default is integer.

Unit is written unit:1 | 2 | ... 256

The default value of Unit is 1 for integer, float, and bitstring and is 8 for binary. No value is required for types utf8, utf16, and utf32.

The total size of the segment is Size x Unit bits long. A segment of type binary must have a size that is evenly divisible by 8.

If you've found the bit syntax description a bit daunting, don't panic. Getting the bit syntax patterns right can be pretty tricky. The best way to approach this is to experiment in the shell with the patterns you need until you get it right, then cut and paste the result into your program. That's how I do it.

Real-World Bit Syntax Examples

Learning the bit syntax is a bit of extra effort, but the benefits are enormous. This section has three examples from real life. All the code here is cut and pasted from real-world programs.

The first example looks for synchronization points in MPEG audio data. This example shows the power of bit syntax pattern matching; the code is very easy to understand and has a clear correspondence to the MPEG header frame specification. The second example was used to build binary data files in the Microsoft Common Object File Format (COFF) format. Packing and unpacking binary data files (like COFF) is typically performed using binaries and binary pattern matching. The final example shows how to unpack an IPv4 datagram.

Finding the Synchronization Frame in MPEG Data

Suppose we want to write a program that manipulates MPEG audio data. We might want to write a streaming media server in Erlang or extract the data tags that describe the content of an MPEG audio stream. To do this, we need to identify and synchronize with the data frames in an MPEG stream.

MPEG audio data is made up from a number of frames. Each frame has its own header followed by audio information—there is no file header, and in principle, you can cut an MPEG file into pieces and play any of the pieces. Any software that reads an MPEG stream is supposed to find the header frames and thereafter synchronize the MPEG data.

An MPEG header starts with an 11-bit *frame sync* consisting of eleven consecutive 1 bits followed by information that describes the data that follows:

```
AAAAAAAA AAABBCCD EEEEFFGH IIJJKLMM
```

AAAAAAAAAAA The sync word (11 bits, all 1s).

BB   2 bits is the MPEG Audio version ID.

CC   2 bits is the layer description.

D   1 bit, a protection bit.

And so on....

The exact details of these bits need not concern us here. Basically, given knowledge of the values of A to M, we can compute the total length of an MPEG frame.

To find the sync point, we first assume that we are correctly positioned at the start of an MPEG header. We then try to compute the length of the frame. Then one of the following can happen:

- Our assumption was correct, so when we skip forward by the length of the frame, we will find another MPEG header.

- Our assumption was incorrect; either we are not positioned at a sequence of 11 consecutive 1 bits that marks the start of a header or the format of the word is incorrect so that we cannot compute the length of the frame.

- Our assumption was incorrect, but we are positioned at a couple of bytes of music data that happen to look like the start of a header. In this case, we can compute a frame length, but when we skip forward by this length, we cannot find a new header.

To be really sure, we look for three consecutive headers. The synchronization routine is as follows:

```
mp3_sync.erl
find_sync(Bin, N) ->
    case is_header(N, Bin) of
        {ok, Len1, _} ->
            case is_header(N + Len1, Bin) of
                {ok, Len2, _} ->
                    case is_header(N + Len1 + Len2, Bin) of
                        {ok, _, _} ->
                            {ok, N};
                        error ->
                            find_sync(Bin, N+1)
                    end;
                error ->
                    find_sync(Bin, N+1)
            end;
        error ->
            find_sync(Bin, N+1)
    end.
```

find_sync tries to find three consecutive MPEG header frames. If byte N in Bin is the start of a header frame, then is_header(N, Bin) will return {ok, Length, Info}. If is_header returns error, then N cannot point to the start of a correct frame.

We can do a quick test in the shell to make sure this works.

```
1> c(mp3_sync).
{ok, mp3_sync}
2> {ok, Bin} = file:read_file("/home/joe/music/mymusic.mp3").
{ok,<<73,68,51,3,0,0,0,0,33,22,84,73,84,50,0,0,0,28, ...>>
3> mp3_sync:find_sync(Bin, 1).
{ok,4256}
```

This uses file:read_file to read the entire file into a binary (see *Reading the Entire File into a Binary,* on page 248). Now for is_header:

mp3_sync.erl
```
is_header(N, Bin) ->
    unpack_header(get_word(N, Bin)).

get_word(N, Bin) ->
    {_,<<C:4/binary,_/binary>>} = split_binary(Bin, N),
    C.

unpack_header(X) ->
    try decode_header(X)
    catch
        _:_ -> error
    end.
```

This is slightly more complicated. First we extract 32 bits of data to analyze (this is done by get_word); then we unpack the header using decode_header. Now decode_header is written to crash (by calling exit/1) if its argument is not at the start of a header. To catch any errors, we wrap the call to decode_header in a try...catch statement (read more about this in Section 6.1, *Handling Errors in Sequential Code,* on page 88). This will also catch any errors that might be caused by incorrect code in framelength/4. decode_header is where all the fun starts.

mp3_sync.erl
```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
    Vsn = case B of
              0 -> {2,5};
              1 -> exit(badVsn);
              2 -> 2;
              3 -> 1
          end,
    Layer = case C of
                0 -> exit(badLayer);
                1 -> 3;
                2 -> 2;
                3 -> 1
            end,
    %% Protection = D,
    BitRate = bitrate(Vsn, Layer, E) * 1000,
    SampleRate = samplerate(Vsn, F),
```

```
    Padding = G,
    FrameLength = framelength(Layer, BitRate, SampleRate, Padding),
    if
        FrameLength < 21 ->
            exit(frameSize);
        true ->
            {ok, FrameLength, {Layer,BitRate,SampleRate,Vsn,Bits}}
    end;
decode_header(_) ->
    exit(badHeader).
```

The magic lies in the amazing expression in the first line of the code.

```
decode_header(<<2#11111111111:11,B:2,C:2,_D:1,E:4,F:2,G:1,Bits:9>>) ->
```

2#11111111111 is a base 2 integer, so this pattern matches eleven consecutive 1 bits, 2 bits into B, 2 bits into C, and so on. Note that the code exactly follows the bit-level specification of the MPEG header given earlier. More beautiful and direct code would be difficult to write. This code is beautiful and also highly efficient. The Erlang compiler turns the bit syntax patterns into highly optimized code that extracts the fields in an optimal manner.

Unpacking COFF Data

A few years ago I decided to write a program to make stand-alone Erlang programs that would run on Windows—I wanted to build a Windows executable on any machine that could run Erlang. Doing this involved understanding and manipulating the Microsoft Common Object File Format (COFF)–formatted files. Finding out the details of COFF was pretty tricky, but various APIs for C++ programs were documented. The C++ programs used the type declarations DWORD, LONG, WORD, and BYTE; these type declarations will be familiar to programmers who have programmed Windows internals.

The data structures involved were documented, but only from a C or C++ programmer's point of view. The following is a typical C typedef:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD  MajorVersion;
    WORD  MinorVersion;
    WORD  NumberOfNamedEntries;
    WORD  NumberOfIdEntries;
    } IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

To write my Erlang program, I first defined four macros that must be included in the Erlang source code file.

```
-define(DWORD, 32/unsigned-little-integer).
-define(LONG,  32/unsigned-little-integer).
-define(WORD,  16/unsigned-little-integer).
-define(BYTE,   8/unsigned-little-integer).
```

Note: Macros are explained in Section 8.17, *Macros*, on page 129. To expand these macros, we use the syntax ?DWORD, ?LONG, and so on. For example, the macro ?DWORD expands to the literal text 32/unsigned-little-integer.

These macros deliberately have the same names as their C counterparts. Armed with these macros, I could easily write some code to unpack image resource data into a binary.

```
unpack_image_resource_directory(Dir) ->
<<Characteristics      : ?DWORD,
  TimeDateStamp        : ?DWORD,
  MajorVersion         : ?WORD,
  MinorVersion         : ?WORD,
  NumberOfNamedEntries : ?WORD,
  NumberOfIdEntries    : ?WORD, _/binary>> = Dir,
  ...
```

If you compare the C and Erlang code, you'll see that they are pretty similar. So, by taking care with the names of the macros and the layout of the Erlang code, we can minimize the semantic gap between the C code and the Erlang code, something that makes our program easier to understand and less likely to have errors.

The next step was to unpack data in Characteristics, and so on.

Characteristics is a 32-bit word consisting of a collection of flags. Unpacking these using the bit syntax is extremely easy; we just write code like this:

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> =
<<Characteristics:32>>
```

The code <<Characteristics:32>> converted Characteristics, which was an integer, into a binary of 32 bits. Then the following code unpacked the required bits into the variables ImageFileRelocsStripped, ImageFileExecutableImage, and so on:

```
<<ImageFileRelocsStripped:1, ImageFileExecutableImage:1, ...>> = ...
```

Again, I kept the same names as in the Windows API in order to keep the semantic gap between the specification and the Erlang program to a minimum.

Using these macros made unpacking data in the COFF format...well, I can't really use the word *easy*, but the code was reasonably understandable.

Unpacking the Header of an IPv4 Datagram

This example illustrates parsing an Internet Protocol version 4 (IPv4) datagram in a single pattern-matching operation:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN, 5).

...
DgramSize = byte_size(Dgram),
case Dgram of
  <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
    ID:16, Flags:3, FragOff:13,
    TTL:8, Proto:8, HdrChkSum:16,
    SrcIP:32,
    DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize ->
        OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
        <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
        ...
```

This code matches an IP datagram in a single pattern-matching expression. The pattern is complex and illustrates how data that does not fall on byte boundaries can easily be extracted (for example, the Flags and FragOff fields that are 3 and 13 bits long, respectively). Having pattern matched the IP datagram, the header and data part of the datagram are extracted in a second pattern matching operation.

We've now covered bit field operations on binaries. Recall that binaries must be a multiple of eight bits long. The next section covers bitstrings, which are used to store sequences of bits.

7.3   Bitstrings: Processing Bit-Level Data

Pattern matching on bitstrings works at a bit level, so we can pack and unpack sequences of bits into a bitstring in a single operation. This is extremely useful when writing code that needs to manipulate bit-level data, such as with data that is not aligned to 8-bit boundaries, or variable-length data, where the data length is expressed in bits rather than bytes.

We can illustrate bit-level processing in the shell.

```
1> B1 = <<1:8>>.
<<1>>
2> byte_size(B1).
1
3> is_binary(B1).
true
4> is_bitstring(B1).
true
```

```
5> B2 = <<1:17>>.
<<0,0,1:1>>
6> is_binary(B2).
false
7> is_bitstring(B2).
true
8> byte_size(B2).
3
9> bit_size(B2).
17
```

Bit-Level Storage

In most programming languages, the least addressable unit of storage is typically 8 bits wide. Most C compilers, for example, define a char (the least addressable unit of storage) to be 8 bits wide. Manipulating bits within a char is complicated, since to access individual bits, they have to masked out and shifted into registers. Writing such code is tricky and error-prone.

In Erlang the least addressable unit of storage is a bit, and individual sequences of bits within a bitstring can be accessed directly without any shifting and masking operations.

In the previous example, B1 is a binary, but B2 is a bitstring since it is 17 bits long. We construct B2 with the syntax <<1:17>>, and it is printed as <<0,0,1:1>>, that is, as a binary literal whose third segment is a bitstring of length 1. The bit size of B2 is 17, and the byte size is 3 (this is actually the size of the binary that contains the bitstring).

Working with bitstrings is tricky. We can't, for example, write a bitstring to a file or socket (which we can do with a binary), since files and sockets work in units of bytes.

We'll conclude this section with a single example, which extracts the individual bits of a byte. To do so, we'll make use of a new construct called a *bit comprehension*. Bit comprehensions are to binaries what list comprehensions are to lists. List comprehensions iterate over lists and return lists. Bit comprehensions iterate over binaries and produce lists or binaries.

This example shows how to extract the bits from a byte:

```
1> B = <<16#5f>>.
<<"_">>
2> [ X || <<X:1>> <= B].
[0,1,0,1,1,1,1,1]
3> << <<X>> || <<X:1>> <= B >>.
<<0,1,0,1,1,1,1,1>>
```

In line 1 we made a binary that contains a single byte. 16#5f is a hexadecimal constant. The shell prints this as <<"_">> since 16#f5 is the ASCII code for the _ character. In line 2, the syntax <<<X:1>> is a pattern representing one bit. The result is a list of the bits in the byte. Line 3 is similar to line 2, only we construct a binary from the bits instead of a list.

The syntax of bit comprehensions is not described here but can be found in the *Erlang Reference Manual.*[2] More examples of bitstring processing can be found in the paper "Bit-Level Binaries and Generalized Comprehensions in Erlang."[3]

Now we know about binaries and bitstrings. Binaries are used internally in the Erlang system whenever we want to manage large amounts of unstructured data. In later chapters we'll see how binaries can be sent in messages over sockets and stored in files.

We're almost done with sequential programming. What remains are a number of small topics; there's nothing really fundamental or exciting, but they're useful subjects to know.

Exercises

1. Write a function that reverses the order of bytes in a binary.

2. Write a function term_to_packet(Term) -> Packet that returns a binary consisting of a 4-byte length header N followed by N bytes of data produced by calling term_to_binary(Term).

3. Write the inverse function packet_to_term(Packet) -> Term that is the inverse of the previous function.

4. Write some tests in the style of *Adding Tests to Your Code*, on page 46, to test that the previous two functions can correctly encode terms into packets and recover the original terms by decoding the packets.

5. Write a function to reverse the bits in a binary.

---

2. http://www.erlang.org/doc/reference_manual/users_guide.html
3. http://user.it.uu.se/~pergu/papers/erlang05.pdf

The Rest of Sequential Erlang

What remains to sequential Erlang are a number of small odds and ends that you have to know but that don't fit into any of the other topics. There's no particular logical order to these topics, so they are just presented in alphabetic order for ease of reference. The topics covered are as follows:

Apply
  This computes the value of a function from its name and arguments when the function and module name are computed dynamically.

Arithmetic expressions
  All legal arithmetic expressions are defined here.

Arity
  The arity of a function is a number of arguments that a function accepts.

Attributes
  This section covers the syntax and interpretation of the Erlang module attributes.

Block expressions
  These are expressions using begin and end.

Booleans
  These are things represented by the atoms true or false.

Boolean expressions
  This section covers all the boolean expressions.

Character set
  This is the character set that Erlang uses.

Comments
  This section covers the syntax of comments.

Dynamic code loading

This section covers how dynamic code loading works.

The Erlang preprocessor

This section covers what happens before Erlang is compiled.

Escape sequences

This section covers the syntax of the escape sequences used in strings and atoms.

Expressions and expression sequences

This section covers how expressions and expression sequences are defined.

Function references

This section covers how to refer to functions.

Include files

This section covers how to include files at compile time.

List addition and subtraction operators

These are ++ and - -.

Macros

This section covers the Erlang macro processor.

Match operator in patterns

This section covers how the match operator = can be used in patterns.

Numbers

This section covers the syntax of numbers.

Operator precedence

This section covers the priority and associativity of all the Erlang operators.

The process dictionary

Each Erlang process has a local area of destructive storage, which can be useful sometimes.

References

References are unique symbols.

Short-circuit boolean expressions

These are boolean expressions that are not fully evaluated.

Term comparisons

This section covers all the term comparison operators and the lexical ordering of terms.

Tuple modules

> These provide a method of creating "stateful" modules.

Underscore variables

> These are variables that the compiler treats in a special way.

I suggest you just skim through these topics, not reading them in detail; just put the information into the back of your mind for later reference.

8.1 apply

The BIF apply(Mod, Func, [Arg1, Arg2, …, ArgN]) applies the function Func in the module Mod to the arguments Arg1, Arg2, … ArgN. It is equivalent to calling this:

```
Mod:Func(Arg1, Arg2, ..., ArgN)
```

apply lets you call a function in a module, passing it arguments. What makes it different from calling the function directly is that the module name and/or the function name can be computed dynamically.

All the Erlang BIFs can also be called using apply by assuming that they belong to the module erlang. So, to build a dynamic call to a BIF, we might write the following:

```
1> apply(erlang, atom_to_list, [hello]).
"hello"
```

Warning: The use of apply should be avoided if possible. When the number of arguments to a function is known in advance, it is much better to use a call of the form M:F(Arg1, Arg2, … ArgN) than apply. When calls to functions are built using apply, many analysis tools cannot work out what is happening, and certain compiler optimizations cannot be made. So, use apply sparingly and only when absolutely needed.

The Mod argument to apply does not have to be an atom; it can also be a tuple. If we call this:

```
{Mod, P1, P2, ..., Pn}:Func(A1, A2, ..., An)
```

then what actually gets called is the following function:

```
Mod:Func(A1, A2, ..., An, {Mod, P1, P2, ..., Pn})
```

This technique is discussed in detail in Section 24.3, *Stateful Modules*, on page 418.

8.2 Arithmetic Expressions

All the possible arithmetic expressions are shown in the following table. Each arithmetic operation has one or two arguments—these arguments are shown in the table as Integer or Number (*Number* means the argument can be an integer or a float).

Op	*Description*	*Arg. Type*	*Priority*
+ X	+ X	Number	1
- X	- X	Number	1
*X * Y*	X * Y	Number	2
X / Y	X / Y (floating-point division)	Number	2
bnot X	Bitwise not of X	Integer	2
X div Y	Integer division of X and Y	Integer	2
X rem Y	Integer remainder of X divided by Y	Integer	2
X band Y	Bitwise and of X and Y	Integer	2
X + Y	X + Y	Number	3
X - Y	X - Y	Number	3
X bor Y	Bitwise or of X and Y	Integer	3
X bxor Y	Bitwise xor of X and Y	Integer	3
X bsl N	Arithmetic bitshift left of X by N bits	Integer	3
X bsr N	Arithmetic bitshift right of X by N bits	Integer	3

Table 3—Arithmetic expressions

Associated with each operator is a *priority*. The order of evaluation of a complex arithmetic expression depends upon the priority of the operator: all operations with priority 1 operators are evaluated first, then all operators with priority 2, and so on.

You can use parentheses to change the default order of evaluation—any parenthesized expressions are evaluated first. Operators with equal priorities are treated as left associative and are evaluated from left to right.

8.3 Arity

The *arity* of a function is the number of arguments that the function has. In Erlang, two functions with the same name and different arity in the same module represent *entirely* different functions. They have *nothing* to do with each other apart from a coincidental use of the same name.

By convention Erlang programmers often use functions with the same name and different arities as auxiliary functions. Here's an example:

lib_misc.erl
```
sum(L) -> sum(L, 0).

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

What you see here are two different functions, one with arity 1 and the second with arity 2.

The function sum(L) sums the elements of a list L. It makes use of an auxiliary routine called sum/2, but this could have been called anything. You could have called the auxiliary routine hedgehog/2, and the meaning of the program would be the same. sum/2 is a better choice of name, though, since it gives the reader of your program a clue as to what's going on and since you don't have to invent a new name (which is always difficult).

Often we "hide" auxiliary functions by not exporting them. So, a module defining the sum(L) would export only sum/1 and not sum/2.

8.4 Attributes

Module attributes have the syntax -AtomTag(...) and are used to define certain properties of a file. (*Note:* -record(...) and -include(...) have a similar syntax but are not considered module attributes.) There are two types of module attributes: predefined and user-defined.

Predefined Module Attributes

The following module attributes have predefined meanings and must be placed before any function definitions:

-module(modname).

   The module declaration. modname must be an atom. This attribute must be the first attribute in the file. Conventionally the code for modname should be stored in a file called modname.erl. If you do not do this, then automatic code loading will not work correctly; see Section 8.10, *Dynamic Code Loading*, on page 122 for more details.

-import(Mod, [Name1/Arity1, Name2/Arity2,...]).

   The import declaration specifies which functions are to be imported into a module. The previous declaration means that the functions Name1 with Arity1 arguments, Name2 with Arity2 arguments, and so on, are to be imported from the module Mod.

Once a function has been imported from a module, then calling the function can be achieved *without* specifying the module name. Here's an example:

```erlang
-module(abc).
-import(lists, [map/2]).

f(L) ->
    L1 = map(fun(X) -> 2*X end, L),
    lists:sum(L1).
```

The call to map needs no qualifying module name, whereas to call sum we need to include the module name in the function call.

-export([Name1/Arity1, Name2/Arity2, ...]).

Export the functions Name1/Arity1, Name2/Arity2, and so on, from the current module. Only exported functions can be called from outside a module. Here's an example:

```erlang
abc.erl
-module(abc).
-export([a/2, b/1]).

a(X, Y) -> c(X) + a(Y).
a(X) -> 2 * X.
b(X) -> X * X.
c(X) -> 3 * X.
```

The export declaration means that only a/2 and b/1 can be called from outside the module abc. So, for example, calling abc:a(5) from the shell (which is outside the module) will result in an error because a/1 is not exported from the module.

```erlang
1> abc:a(1,2).
7
2> abc:b(12).
144
3> abc:a(5).
** exception error: undefined function abc:a/1
```

The error message might cause confusion here. The call to abc:a(5) failed because the function concerned is undefined. It is actually defined in the module, but it is not exported.

-compile(Options).

Add Options to the list of compiler options. Options is a single compiler option or a list of compiler options (these are described in the manual page for the module compile).

Note: The compiler option -compile(export_all). is often used while debugging programs. This exports all functions from the module without having to explicitly use the -export annotation.

-vsn(Version).

Specify a module version. Version is any literal term. The value of Version has no particular syntax or meaning, but it can be used by analysis programs or for documentation purposes.

User-Defined Attributes

The syntax of a user-defined attribute is as follows:

```
-SomeTag(Value).
```

SomeTag must be an atom, and Value must be a literal term. The values of the module attributes are compiled into the module and can be extracted at runtime. Here's an example of a module containing some user-defined attributes:

```
attrs.erl
-module(attrs).
-vsn(1234).
-author({joe,armstrong}).
-purpose("example of attributes").
-export([fac/1]).

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

We can extract the attributes as follows:

```
1> attrs:module_info().
[{exports,[{fac,1},{module_info,0},{module_info,1}]},
{imports,[]},
{attributes,[{vsn,[1234]},
            {author,[{joe,armstrong}]},
            {purpose,"example of attributes"}]},
            {compile,[{options,[]},
            {version,"4.8"},
            {time,{2013,5,3,7,36,55}},
            {source,"/Users/joe/jaerlang2/code/attrs.erl"}]}]
```

The user-defined attributes contained in the source code file reappear as a subterm of {attributes, ...}. The tuple {compile, ...} contains information that was added by the compiler. The value {version,"4.5.5"} is the version of the compiler and should not be confused with the vsn tag defined in the module attributes. In the previous example, attrs:module_info() returns a property list of all the

metadata associated with a compiled module. attrs:module_info(X), where X is one of exports, imports, attributes, or compile, returns the individual attribute associated with the module.

Note that the functions module_info/0 and module_info/1 are automatically created every time a module is compiled.

To run attrs:module_info, we have to load the beam code for the module attrs into the Erlang VM. We can extract the same information *without* loading the module by using the module beam_lib.

```
3> beam_lib:chunks("attrs.beam",[attributes]).
{ok,{attrs,[{attributes,[{author,[{joe,armstrong}]},
                         {purpose,"example of attributes"},
                         {vsn,[1234]}]}]}}
```

beam_lib:chunks extracts the attribute data from a module without loading the code for the module.

8.5   Block Expressions

Block expressions are used when the Erlang syntax requires a single expression, but we want to have a sequence of expressions at this point in the code. For example, in a list comprehension of the form [E || ...], the syntax requires E to be a single expression, but we might want to do several things in E.

```
begin
    Expr1,
    ...,
    ExprN
end
```

You can use block expressions to group a sequence of expressions, similar to a clause body. The value of a begin … end block is the value of the last expression in the block.

8.6   Booleans

There is no distinct boolean type in Erlang; instead, the atoms true and false are given a special interpretation and are used to represent boolean literals.

Sometimes we write functions that return one of two possible atomic values. When this happens, it's good practice to make sure they return a boolean. It's also a good idea to name your functions to make it clear that they return a boolean.

For example, suppose we write a program that represents the state of some file. We might find ourselves writing a function file_state(File) that returns open or closed. When we write this function, we could think about renaming the function and letting it return a boolean. With a little thought we could rewrite our program to use a function called is_file_open(File) that returns true or false.

The reason for using booleans instead of choosing two different atoms to represent the status is simple. There are a large number of functions in the standard libraries that work on functions that return booleans. So, if we make sure all our functions return booleans, then we'll be able to use them together with the standard library functions.

For example, suppose we have a list of files L and we want to partition this into a list of open files and a list of closed files. Using the standard libraries, we could write the following:

```
lists:partition(fun is_file_open/1, L)
```

But using our file_state/1 function, we'd have to write a conversion function before we call the library routine.

```
lists:partition(fun(X) ->
                 case file_state(X) of
                     open   -> true;
                     closed -> false
                 end, L)
```

8.7 Boolean Expressions

There are four possible boolean expressions.

- not B1: Logical not
- B1 and B2: Logical and
- B1 or B2: Logical or
- B1 xor B2: Logical xor

In all of these, B1 and B2 must be boolean literals or expressions that evaluate to booleans. Here are some examples:

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

8.8 Character Set

Since Erlang version R16B, Erlang source code files are assumed to be encoded in the UTF-8 character set. Prior to this, the ISO-8859-1 (Latin-1) character set was used. This means all UTF-8 printable characters can be used in source code files without using any escape sequences.

Internally Erlang has no character data type. Strings don't really exist but instead are represented by lists of integers. Unicode strings can be represented by lists of integers without any problems.

8.9 Comments

Comments in Erlang start with a percent character (%) and extend to the end of line. There are no block comments.

Note: You'll often see double percent characters (%%) in code examples. Double percent marks are recognized in the Emacs erlang-mode and enable automatic indentation of commented lines.

```erlang
% This is a comment
my_function(Arg1, Arg2) ->
    case f(Arg1) of
        {yes, X} ->    % it worked
          ..
```

8.10 Dynamic Code Loading

Dynamic code loading is one of the most surprising features built into the heart of Erlang. The nice part is that it just works without you really being aware of what's happening in the background.

The idea is simple: every time we call someModule:someFunction(…), we'll always call the latest version of the function in the latest version of the module, *even if we recompile the module while code is running in this module.*

If a calls b in a loop and we recompile b, then a will automatically call the new version of b the next time b is called. If many different processes are running and all of them call b, then all of them will call the new version of b if b is recompiled. To see how this works, we'll write two little modules: a and b. b is very simple.

```erlang
b.erl
-module(b).
-export([x/0]).

x() -> 1.
```

Now we'll write a.

```
a.erl
-module(a).
-compile(export_all).

 start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn1 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

Now we can compile a and b and start a couple of a processes.

```
1> c(b).
{ok, b}
2> c(a).
{ok, a}
3> a:start(one).
<0.41.0>
Vsn1 (one) b:x() = 1
4> a:start(two).
<0.43.0>
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
Vsn1 (one) b:x() = 1
Vsn1 (two) b:x() = 1
```

The a processes sleep for three seconds, wake up and call b:x(), and then print the result. Now we'll go into the editor and change the module b to the following:

```
-module(b).
-export([x/0]).

x() -> 2.
```

Then we recompile b in the shell. This is what happens:

```
5> c(b).
{ok,b}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
```

The two original versions of a are still running, but now they call the *new* version of b. So, when we call b:x() from within the module a, we really call "the latest version of b." We can change and recompile b as many times as we want, and all the modules that call it will automatically call the new version of b without having to do anything special.

Now we've recompiled b, but what happens if we change and recompile a? We'll do an experiment and change a to the following:

```erlang
-module(a).
-compile(export_all).

start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn2 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

Now we compile and start a.

```erlang
6> c(a).
{ok,a}
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
...
7> a:start(three).
<0.53.0>
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
Vsn1 (one) b:x() = 2
Vsn1 (two) b:x() = 2
Vsn2 (three) b:x() = 2
...
```

Something funny is going on here. When we start the new version of a, we see that new version running. However, the existing processes running the first version of a are still running that old version of a without any problems.

Now we could try changing b yet again.

```
-module(b).
-export([x/0]).

x() -> 3.
```

We'll recompile b in the shell. Watch what happens.

```
8> c(b).
{ok,b}
Vsn1 (one) b:x() = 3
Vsn1 (two) b:x() = 3
Vsn2 (three) b:x() = 3
...
```

Now both the old and new versions of a call the latest version of b.

Finally, we'll change a again (this is the third change to a).

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

start(Tag) ->
    spawn(fun() -> loop(Tag) end).

loop(Tag) ->
    sleep(),
    Val = b:x(),
    io:format("Vsn3 (~p) b:x() = ~p~n",[Tag, Val]),
    loop(Tag).

sleep() ->
    receive
        after 3000 -> true
    end.
```

Now when we recompile a and start a new version of a, we see the following:

```
9> c(a).
{ok,a}
Vsn2 (three) b:x() = 3
...
10> a:start(four).
<0.106.0>
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
Vsn2 (three) b:x() = 3
Vsn3 (four) b:x() = 3
...
```

The output contains strings generated by the last two versions of a (versions 2 and 3); the processes running version 1 of a's code have died.

Erlang can have two versions of a module running at any one time, the current version and an old version. When you recompile a module, any process running code in the old version is killed, the current version becomes the old version, and the newly compiled module becomes the current version. Think of this as a shift register with two versions of the code. As we add new code, the oldest version is junked. Some processes can run old versions of the code while other processes can simultaneously run new versions of the code.

Read the purge_module documentation[1] for more details.

8.11 Erlang Preprocessor

Before an Erlang module is compiled, it is automatically processed by the Erlang preprocessor. The preprocessor expands any macros that might be in the source file and inserts any necessary include files.

Ordinarily, you won't need to look at the output of the preprocessor, but in exceptional circumstances (for example, when debugging a faulty macro), you might want to save the output of the preprocessor. To see the result of preprocessing the module some_module.erl, give the OS shell command.

```
$ erlc -P some_module.erl
```

This produces a listing file called some_module.P.

8.12 Escape Sequences

Within strings and quoted atoms, you can use escape sequences to enter any nonprintable characters. All the possible escape sequences are shown in Table 4, *Escape sequences*, on page 127.

Let's give some examples in the shell to show how these conventions work. (Note: ~w in a format string prints the list without any attempt to pretty print the result.)

```
%% Control characters
1> io:format("~w~n", ["\b\d\e\f\n\r\s\t\v"]).
[8,127,27,12,10,13,32,9,11]
ok
%% Octal characters in a string
2> io:format("~w~n", ["\123\12\1"]).
[83,10,1]
ok
%% Quotes and escapes in a string
3> io:format("~w~n", ["\'\"\\"]).
[39,34,92]
```

---

1. http://www.erlang.org/doc/man/erlang.html#purge_module/1

```
ok
%% Character codes
4> io:format("~w~n", ["\a\z\A\Z"]).
[97,122,65,90]
ok
```

Escape Sequence	Meaning	Integer Code
\b	Backspace	8
\d	Delete	127
\e	Escape	27
\f	Form feed	12
\n	New line	10
\r	Carriage return	13
\s	Space	32
\t	Tab	9
\v	Vertical tab	11
\x{...}	Hexadecimal characters (... are hexadecimal characters)	
\^a..\^z or \^A..\^Z	Ctrl+A to Ctrl+Z	1 to 26
\'	Single quote	39
\"	Double quote	34
\\	Backslash	92
\C	The ASCII code for C (C is a character)	(An integer)

Table 4—Escape sequences

8.13 Expressions and Expression Sequences

In Erlang, anything that can be evaluated to produce a value is called an *expression*. This means things such as catch, if, and try…catch are expressions. Things such as record declarations and module attributes cannot be evaluated, so they are not expressions.

Expression sequences are sequences of expressions separated by commas. They are found all over the place immediately following an -> arrow. The value of the expression sequence E1, E2, …, En is defined to be the value of the last expression in the sequence. This is computed using any bindings created when computing the values of E1, E2, and so on. This is equivalent to progn in LISP.

8.14 Function References

Often we want to refer to a function that is defined in the current module or in some external module. You can use the following notation for this:

fun LocalFunc/Arity

> This is used to refer to the local function called LocalFunc with Arity arguments in the current module.

fun Mod:RemoteFunc/Arity

> This is used to refer to an external function called RemoteFunc with Arity arguments in the module Mod.

Here's an example of a function reference in the current module:

```erlang
-module(x1).
-export([square/1, ...]).

square(X) -> X * X.
...
double(L) -> lists:map(fun square/1, L).
```

If we wanted to call a function in a remote module, we could refer to the function as in the following example:

```erlang
-module(x2).
...
double(L) -> lists:map(fun x1:square/1, L).
```

fun x1:square/1 means the function square/1 in the module x1.

Note that function references that include the module name provide switchover points for dynamic code upgrade. For details, read Section 8.10, *Dynamic Code Loading,* on page 122.

8.15 Include Files

Files can be included with the following syntax:

```erlang
-include(Filename).
```

In Erlang, the convention is that include files have the extension .hrl. The FileName should contain an absolute or relative path so that the preprocessor can locate the appropriate file. Library header files can be included with the following syntax:

```erlang
-include_lib(Name).
```

Here's an example:

```erlang
-include_lib("kernel/include/file.hrl").
```

In this case, the Erlang compiler will find the appropriate include files. (kernel, in the previous example, refers to the application that defines this header file.)

Include files usually contain record definitions. If many modules need to share common record definitions, then the common record definitions are put into include files that are included by all the modules that need these definitions.

8.16 List Operations ++ and - -

++ and -- are infix operators for list addition and subtraction.

A ++ B adds (that is, appends) A and B.

A -- B subtracts the list B from the list A. Subtraction means that every element in B is removed from A. Note that if some symbol X occurs only K times in B, then only the first K occurrences of X in A will be removed.

Here are some examples:

```
1> [1,2,3] ++ [4,5,6].
[1,2,3,4,5,6]
2> [a,b,c,1,d,e,1,x,y,1] -- [1].
[a,b,c,d,e,1,x,y,1]
3> [a,b,c,1,d,e,1,x,y,1] -- [1,1].
[a,b,c,d,e,x,y,1]
4> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1].
[a,b,c,d,e,x,y]
5> [a,b,c,1,d,e,1,x,y,1] -- [1,1,1,1].
[a,b,c,d,e,x,y]
```

++ can also be used in patterns. When matching strings, we can write patterns such as the following:

```
f("begin" ++ T) -> ...
f("end" ++ T) -> ...
...
```

The pattern in the first clause is expanded into [$b,$e,$g,$i,$n|T].

8.17 Macros

Erlang macros are written as shown here:

```
-define(Constant, Replacement).
-define(Func(Var1, Var2,.., Var), Replacement).
```

Macros are expanded by the Erlang preprocessor epp when an expression of the form ?MacroName is encountered. Variables occurring in the macro definition match complete forms in the corresponding site of the macro call.

```
-define(macro1(X, Y), {a, X, Y}).

foo(A) ->
    ?macro1(A+10, b)
```

That expands into this:

```
foo(A) ->
    {a,A+10,b}.
```

In addition, a number of predefined macros provide information about the current module. They are as follows:

- ?FILE expands to the current filename.
- ?MODULE expands to the current module name.
- ?LINE expands to the current line number.

Control Flow in Macros

Inside a module, the following directives are supported; you can use them to control macro expansion:

-undef(Macro).

Undefines the macro; after this you cannot call the macro.

-ifdef(Macro).

Evaluates the following lines only if Macro has been defined.

-ifndef(Macro).

Evaluates the following lines only if Macro is undefined.

-else.

Allowed after an ifdef or ifndef statement. If the condition was false, the statements following else are evaluated.

-endif.

Marks the end of an ifdef or ifndef statement.

Conditional macros must be properly nested. They are conventionally grouped as follows:

```
-ifdef(<FlagName>).
-define(...).
-else.
-define(...).
-endif.
```

We can use these macros to define a DEBUG macro. Here's an example:

```
m1.erl
-module(m1).
-export([loop/1]).

-ifdef(debug_flag).
-define(DEBUG(X), io:format("DEBUG ~p:~p ~p~n",[?MODULE, ?LINE, X])).
-else.
-define(DEBUG(X), void).
-endif.

loop(0) ->
    done;
loop(N) ->
    ?DEBUG(N),
    loop(N-1).
```

Note: io:format(String, [Args]) prints the variables in [Args] in the Erlang shell according to the formatting information in String. The formatting codes are preceded by a ~ symbol. ~p is short for *pretty print*, and ~n produces a newline. io:format understands an extremely large number of formatting options; for more information, see *Writing a List of Terms to a File,* on page 251.

To enable the macro, we set the debug_flag when we compile the code. This is done with an additional argument to c/2 as follows:

```
1> c(m1, {d, debug_flag}).
{ok,m1}
2> m1:loop(4).
DEBUG m1:13 4
DEBUG m1:13 3
DEBUG m1:13 2
DEBUG m1:13 1
done
```

If debug_flag is not set, the macro just expands to the atom void. This choice of name has no significance; it's just a reminder to you that nobody is interested in the value of the macro.

8.18 Match Operator in Patterns

Let's suppose we have some code like this:

```
Line 1  func1([{tag1, A, B}|T]) ->
     2      ...
     3      ... f(..., {tag1, A, B}, ...)
     4      ...
```

In line 1, we pattern match the term {tag1, A, B}, and in line 3, we call f with an argument that is {tag1, A, B}. When we do this, the system rebuilds the term

{tag1, A, B}. A much more efficient and less error-prone way to do this is to assign the pattern to a temporary variable, Z, and pass this into f, like this:

```
func1([{tag1, A, B}=Z|T]) ->
    ...
    ... f(... Z, ...)
    ...
```

The match operator can be used at any point in the pattern, so if we have two terms that need rebuilding, such as in this code:

```
func1([{tag, {one, A}, B}|T]) ->
    ...
    ...  f(..., {tag, {one,A}, B}, ...),
    ...  g(..., {one, A}), ...)
    ...
```

then we could introduce two new variables, Z1 and Z2, and write the following:

```
func1([{tag, {one, A}=Z1, B}=Z2|T]) ->
    ..,.
    ...  f(..., Z2, ...),
    ...  g(..., Z1, ...),
    ...
```

8.19 Numbers

Numbers in Erlang are either integers or floats.

Integers

Integer arithmetic is exact, and the number of digits that can be represented in an integer is limited only by available memory.

Integers are written with one of three different syntaxes.

Conventional syntax

Here integers are written as you expect. For example, 12, 12375, and -23427 are all integers.

Base K integers

Integers in a number base other than ten are written with the syntax K#Digits; thus, we can write a number in binary as 2#00101010 or a number in hexadecimal as 16#af6bfa23. For bases greater than ten, the characters abc... (or ABC...) represent the numbers 10, 11, 12, and so on. The highest number base is 36.

$ syntax

The syntax $C represents the integer code for the ASCII character C. Thus, $a is short for 97, $1 is short for 49, and so on.

Immediately after the $ we can also use any of the escape sequences described in Table 4, *Escape sequences*, on page 127. Thus, $\n is 10, $\^c is 3, and so on.

Here are some examples of integers:

```
0 -65 2#010001110 -8#377 16#fe34 16#FE34 36#wow
```

(Their values are 0, -65, 142, -255, 65076, 65076, and 42368, respectively.)

Floats

A floating-point number has five parts: an optional sign, a whole number part, a decimal point, a fractional part, and an optional exponent part.

Here are some examples of floats:

```
1.0 3.14159 -2.3e+6 23.56E-27
```

After parsing, floating-point numbers are represented internally in IEEE 754 64-bit format. Real numbers with absolute value in the range 10^{-323} to 10^{308} can be represented by an Erlang float.

8.20 Operator Precedence

Table 5, *Operator precedence*, on page 133 shows all the Erlang operators in order of descending priority together with their associativity. Operator precedence and associativity are used to determine the evaluation order in unparenthesized expressions.

Operators	*Associativity*
:	
#	
(unary) +, (unary) -, bnot, not	
/, *, div, rem, band, and	Left associative
+, -, bor, bxor, bsl, bsr, or, xor	Left associative
++, --	Right associative
==, /=, =<, <, >=, >, =:=, =/=	
andalso	
orelse	
= !	Right associative
catch	

Table 5—Operator precedence

Expressions with higher priority (higher up in the table) are evaluated first, and then expressions with lower priority are evaluated. So, for example, to evaluate 3+4*5+6, we first evaluate the subexpression 4*5, since (*) is higher up in the table than (+). Now we evaluate 3+20+6. Since (+) is a left-associative operator, we interpret this as meaning (3+20)+6, so we evaluate 3+20 first, yielding 23; finally we evaluate 23+6.

In its fully parenthesized form, 3+4*5+6 means ((3+(4*5))+6). As with all programming languages, it is better to use parentheses to denote scope than to rely upon the precedence rules.

8.21 The Process Dictionary

Each process in Erlang has its own private data store called the *process dictionary*. The process dictionary is an associative array (in other languages this might be called a *map*, *hashmap*, or *hash table*) composed of a collection of keys and values. Each key has only one value.

The dictionary can be manipulated using the following BIFs:

put(Key, Value) -> OldValue.
> Add a Key, Value association to the process dictionary. The value of put is OldValue, which is the previous value associated with Key. If there was no previous value, the atom undefined is returned.

get(Key) -> Value.
> Look up the value of Key. If there is a Key, Value association in the dictionary, return Value; otherwise, return the atom undefined.

get() -> [{Key,Value}].
> Return the entire dictionary as a list of {Key,Value} tuples.

get_keys(Value) -> [Key].
> Return a list of keys that have the values Value in the dictionary.

erase(Key) -> Value.
> Return the value associated with Key or the atom undefined if there is no value associated with Key. Finally, erase the value associated with Key.

erase() -> [{Key,Value}].
> Erase the entire process dictionary. The return value is a list of {Key,Value} tuples representing the state of the dictionary before it was erased.

Here's an example:

```
1> erase().
[]
2> put(x, 20).
undefined
3> get(x).
20
4> get(y).
undefined
5> put(y, 40).
undefined
6> get(y).
40
7> get().
[{y,40},{x,20}]
8> erase(x).
20
9> get().
[{y,40}]
```

As you can see, variables in the process dictionary behave pretty much like conventional mutable variables in imperative programming languages. If you use the process dictionary, your code will no longer be side effect free, and all the benefits of using nondestructive variables that we discussed in *Erlang Variables Do Not Vary*, on page 29, do not apply. For this reason, you should use the process dictionary sparingly.

Note: I rarely use the process dictionary. Using the process dictionary can introduce subtle bugs into your program and make it difficult to debug. One form of usage that I do approve of is to use the processes dictionary to store "write-once" variables. If a key acquires a value exactly once and does not change the value, then storing it in the process dictionary is sometimes acceptable.

8.22 References

References are globally unique Erlang terms. They are created with the BIF erlang:make_ref(). References are useful for creating unique tags that can be included in data and then at a later stage compared for equality. For example, a bug-tracking system might add a reference to each new bug report in order to give it a unique identity.

8.23 Short-Circuit Boolean Expressions

Short-circuit boolean expressions are boolean expressions whose arguments are evaluated only when necessary.

There are two "short-circuit" boolean expressions.

Expr1 orelse Expr2

    This first evaluates Expr1. If Expr1 evaluates to true, Expr2 is not evaluated. If Expr1 evaluates to false, Expr2 is evaluated.

Expr1 andalso Expr2

    This first evaluates Expr1. If Expr1 evaluates to true, Expr2 is evaluated. If Expr1 evaluates to false, Expr2 is not evaluated.

Note: In the corresponding boolean expressions (A or B; A and B), both the arguments are always evaluated, even if the truth value of the expression can be determined by evaluating only the first expression.

8.24 Term Comparisons

There are eight possible term comparison operations, shown in Table 6, *Term comparisons*, on page 137.

For the purposes of comparison, a total ordering is defined over all terms. This is defined so that the following is true:

number < atom < reference < fun < port < pid < tuple (and record) < map < list < binary

This means that, for example, a number (any number) is defined to be smaller than an atom (any atom), that a tuple is greater than an atom, and so on. (Note that for the purposes of ordering, ports and PIDs are included in this list. We'll talk about these later.)

Having a total order over all terms means we can sort lists of any type and build efficient data access routines based on the sort order of the keys.

All the term comparison operators, with the exception of =:= and =/=, behave in the following way if their arguments are numbers:

- If one argument is a integer and the other is a float, then the integer is converted to a float before the comparison is performed.

- If both arguments are integers or if both arguments are floats, then the arguments are used "as is," that is, without conversion.

You should also be really careful about using == (especially if you're a C or Java programmer). In 99 out of 100 cases, you should be using =:=. == is useful *only* when comparing floats with integers. =:= is for testing whether two terms are *identical*.

Identical means having the same value (like the Common Lisp EQUAL). Since values are immutable, this does not imply any notion of pointer identity. If in doubt, use =:=, and be suspicious if you see ==. Note that a similar comment

applies to using /= and =/=, where /= means "not equal to" and =/= means "not identical."

Note: In a lot of library and published code, you'll see == used when the operator should have been =:=. Fortunately, this kind of error does not often result in an incorrect program, since if the arguments to == do not contain any floats, then the behaviors of the two operators are the same.

You should also be aware that function clause matching always implies exact pattern matching, so if you define a fun F = fun(12) -> … end, then trying to evaluate F(12.0) will fail.

Operator	*Meaning*
X > Y	X is greater than Y.
X < Y	X is less than Y.
X =< Y	X is equal to or less than Y.
X >= Y	X is greater than or equal to Y.
X == Y	X is equal to Y.
X /= Y	X is not equal to Y.
X =:= Y	X is identical to Y.
X =/= Y	X is not identical to Y.

Table 6—Term comparisons

8.25 Tuple Modules

When we call M:f(Arg1, Arg2, …, ArgN), we have assumed that M is a module name. But M can also be a tuple of the form {Mod1, X1, X2, … Xn}, in which case the function Mod1:f(Arg1, Arg2, …, Arg3, M) is called.

This mechanism can be used to create "stateful modules," which is discussed in Section 24.3, *Stateful Modules*, on page 418, and to create "adapter patterns," discussed in Section 24.4, *Adapter Patterns*, on page 419.

8.26 Underscore Variables

There's one more thing to say about variables. The special syntax _VarName is used for a normal variable, not an anonymous variable. Normally the compiler will generate a warning if a variable is used only once in a clause since this is usually the sign of an error. If the variable is used only once but starts with an underscore, the warning message will not be generated.

Since _Var is a normal variable, very subtle bugs can be caused by forgetting this and using it as a "don't care" pattern. In a complicated pattern match,

it can be difficult to spot that, for example, _Int is repeated when it shouldn't have been, causing the pattern match to fail.

There are two main uses of underscore variables.

- To name a variable that we don't intend to use. That is, writing open(File, _Mode) makes the program more readable than writing open(File, _).

- For debugging purposes. For example, suppose we write this:

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    io:format("Q = ~p~n", [Q]),
    P.
```

This compiles without an error message.

Now comment out the following format statement:

```
some_func(X) ->
    {P, Q} = some_other_func(X),
    %% io:format("Q = ~p~n", [Q]),
    P.
```

If we compile this, the compiler will issue a warning that the variable Q is not used.

If we rewrite the function like this:

```
some_func(X) ->
    {P, _Q} = some_other_func(X),
    io:format("_Q = ~p~n", [_Q]),
    P.
```

then we can comment out the format statement, and the compiler will not complain.

Now we're actually through with sequential Erlang.

In the next two chapters we'll round off Part II of the book. We'll start with the type notation that is used to describe the types of Erlang functions and talk about a number of tools that can be used to type check Erlang code. In the final chapter of Part II, we'll look at different ways to compile and run your programs.

Exercises

1. Reread the section about Mod:module_info() in this chapter. Give the command dict:module_info(). How many functions does this module return?

2. The command code:all_loaded() returns a list of {Mod,File} pairs of all modules that have been loaded into the Erlang system. Use the BIF Mod:module_info() to find out about these modules. Write functions to determine which module exports the most functions and which function name is the most common. Write a function to find all unambiguous function names, that is, function names that are used in only one module.

Types

Erlang has a type notation that we can use to define new data types and add type annotations to our code. The type annotations make the code easier to understand and maintain and can be used to detect errors at compile time.

In this chapter we'll introduce the type notation and talk about two programs that can be used to find errors in our code.

The programs we'll discuss are called dialyzer and typer and are included in the standard Erlang distribution. Dialyzer stands for "DIscrepancy AnaLYZer for ERlang programs," and it does precisely that which is implied by its name: it finds discrepancies in Erlang code. typer provides information about the types used in your programs. Both the dialyzer and typer work perfectly well with no type annotations at all, but if you add type annotations to your program, the quality of the analysis performed by these tools will be improved.

This is a fairly complex chapter, so we'll start with a simple example, and then we'll go a bit deeper and look at the type grammar; following this, we'll have a session with the dialyzer. We'll talk about a workflow we should use with the dialyzer and about the kind of errors that the dialyzer cannot find. We'll wrap up with a little theory of how the dialyzer works, which will help us understand the errors that the dialyzer finds.

9.1 Specifying Data and Function Types

We are going on a walking tour and are lucky enough to have a module that we can use to plan our walks. The module starts like this:

```
walks.erl
-module(walks).
-export([plan_route/2]).

-spec plan_route(point(), point()) -> route().
```

```
-type direction() :: north | south | east | west.
-type point()     :: {integer(), integer()}.
-type route()     :: [{go,direction(),integer()}].
```

...

This module exports a function called plan_route/2. The input and return types for the function are specified in a *type specification*, and three new types are defined using *type declarations*. These are interpreted as follows:

-spec plan_route(point(), point()) -> route().

> Means that if the function plan_route/2 is called with two input arguments, both of type point(), then it will return an object of type route().

-type direction() :: north | south | east | west.

> Introduces a new type called direction() whose value is one of the atoms north, south, east, or west.

-type point() :: {integer(), integer()}.

> Means that the type point() is a tuple of two integers (integer() is a predefined type).

-type route() :: [{go, direction(), integer()}].

> Defines the type route() to be a list of 3-tuples, where each tuple contains the atom go, an object of type direction, and an integer. The notation [X] means a list of type X.

From the type annotations alone we can imagine evaluating plan_route and seeing something like this:

```
> walks:plan_route({1,10}, {25, 57}).
[{go, east, 24},
 {go, north, 47},
 ...
 ]
```

Of course, we have no idea if the function plan_route will return anything at all; it might just crash and not return a value. But if it does return a value, the returned value should be of type route() if the input arguments were of type point(). We also have no idea what the numbers in the previous expression mean. Are they miles, kilometers, centimeters, and so on? All we know is what the type declaration tells us, that is, that they are integers.

To add expressive power to the types, we can annotate them with descriptive variables. For example, we could change the specification of plan_route to the following:

```
-spec plan_route(From:: point(), To:: point()) -> ...
```

The names `From` and `To` in the type annotation give the user some idea as to the role these arguments play in the function. They are also used to link the names in the documentation to the variables in the type annotations. The official Erlang documentation uses strict rules for writing type annotations so that the names in the type annotations correspond to the names used in the corresponding documentation.

Saying that our route starts at `From` and that `From` is a pair of integers may or may not be sufficient to document the function; it depends upon the context. We could easily refine the type definitions by adding more information. For example, by writing this:

```
-type angle()      :: {Degrees::0..360, Minutes::0..60, Seconds::0..60}.
-type position()   :: {latitude | longitude, angle()}.
-spec plan_route1(From::position(), To::position()) -> ...
```

the new form gives a lot more information but again invites guesswork. We might guess that units of the angles are in degrees, since the range of allowed values is 0 to 360, but they might just be in radians, and we would have guessed wrongly.

As the type annotations become longer, we might end up being more precise at the expense of increased verbosity. The increased size of the annotations might make the code more difficult to read. Writing good type annotations is as much of an art as writing good clear code—something that is very difficult and takes years of practice. It's a form of zen meditation: the more you do it, the easier it becomes and the better you get!

We've seen a simple example of how to define types; the next section formalizes the type notation. Once we're happy with the type notation, we'll have a session with the dialyzer.

9.2   Erlang Type Notation

So far, we have introduced types through informal descriptions. To make full use of the type system, we need to understand the type grammar so we can read and write more precise type descriptions.

The Grammar of Types

Types are defined informally using the following syntax:

```
T1 :: A | B | C ...
```

This means that `T1` is defined to be one of `A`, `B`, or `C`.

Using this notation, we can define a subset of Erlang types as follows:

```
Type :: any() | none() | pid() | port() | reference() | []
      | Atom | binary() | float() | Fun | Integer | [Type] |
      | Tuple | Union | UserDefined

Union :: Type1 | Type2 | ...

Atom :: atom() | Erlang_Atom

Integer :: integer() | Min .. Max

Fun :: fun() | fun((...) -> Type)

Tuple :: tuple() | {T1, T2, ... Tn}
```

In the previous example, `any()` means any Erlang term, `X()` means an Erlang object of type `X`, and the token `none()` is used to denote the type of a function that never returns.

The notation `[X]` denotes a list of type `X`, and `{T1, T2, …, Tn}` denotes a tuple of size `n` whose arguments are of type `T1, T2, … Tn`.

New types can be defined with the following syntax:

```
-type NewTypeName(TVar1, TVar2, ... TVarN) :: Type.
```

`TVar1` to `TVarN` are optional type variables, and `Type` is a type expression.

Here are some examples:

```
-type onOff()       :: on | off.
-type person()      :: {person, name(), age()}.
-type people()      :: [person()].
-type name()        :: {firstname, string()}.
-type age()         :: integer().
-type dict(Key,Val) :: [{Key,Val}].
...
```

These rules say that, for example, `{firstname, "dave"}` is of type `name()`, and `[{person, {firstname,"john"}, 35}, {person, {firstname,"mary"}, 26}]` is of type `people()`, and so on. The type `dict(Key,Val)` shows the use of type variables and defines a dictionary type to be a list of `{Key, Val}` tuples.

Predefined Types

In addition to the type grammar, the following type aliases are predefined:

```
-type term() :: any().
-type boolean() :: true | false.
-type byte() :: 0..255.
-type char() :: 0..16#10ffff.
-type number() :: integer() | float().
```

```
-type list() :: [any()].
-type maybe_improper_list() :: maybe_improper_list(any(), any()).
-type maybe_improper_list(T) :: maybe_improper_list(T, any()).
-type string() :: [char()].
-type nonempty_string() :: [char(),...].
-type iolist() :: maybe_improper_list(byte() | binary() | iolist(),
                                       binary() | []).
-type module() :: atom().
-type mfa() :: {atom(), atom(), atom()}.
-type node() :: atom().
-type timeout() :: infinity | non_neg_integer().
-type no_return() :: none().
```

maybe_improper_list is used to specify the types of lists whose ultimate tail is non-nil. Such lists are rarely used, but it is possible to specify their types!

There are also a small number of predefined types. non_neg_integer() is a non-negative integer, pos_integer() is a positive integer, and neg_integer() is a negative integer. Finally, the notation [X,...] means a non-empty list of type X.

Now that we can define types, let's move on to function specifications.

Specifying the Input and Output Types of a Function

Function specifications say what the types of the arguments to a function are and what the type of the return value of the function is. A function specification is written like this:

```
-spec functionName(T1, T2, ..., Tn) -> Tret when
    Ti :: Typei,
    Tj :: Typej,
    ...
```

Here T1, T2, ..., Tn describe the types of the arguments to a function, and Tret describes the type of the return value of the function. Additional type variables can be introduced if necessary after the optional when keyword.

We'll start with an example. The following type specification:

```
-spec file:open(FileName, Modes) -> {ok, Handle} | {error, Why} when
    FileName  :: string(),
    Modes     :: [Mode],
    Mode      :: read | write | ...
    Handle    :: file_handle(),
    Why       :: error_term().
```

says that if we open the file FileName, we should get a return value that is either {ok, Handle} or {error, Why}. FileName is a string, Modes is a list of Mode, and Mode is one of read, write, and so on.

The previous function specification could have been written in a number of equivalent ways; for example, we might have written the following and not used a `when` qualifier:

```
-spec file:open(string(), [read|write|...] -> {ok, Handle} | {error, Why}
```

The problems with this are, first, that we lose the descriptive variables `FileName` and `Modes`, and so on, and, second, that the type specification becomes a lot longer and is consequently more difficult to read and format in printed documentation. In the documentation that ideally follows the program we have no way to refer to the arguments of the function if they are not named.

In the first way of writing the specification, we wrote the following:

```
-spec file:open(FileName, Modes) -> {ok, Handle} | {error, Why} when
    FileName  :: string(),
    ...
```

So, any documentation of this function could unambiguously refer to the file that was being opened by using the name `FileName`. If we said this:

```
-spec file:open(string(), [read|write|...) -> {ok, Handle} | {error, Why}.
```

and dropped the `when` qualifier, then the documentation would have to refer to the file that was being opened as "the first argument of the open function," a circumlocution that is unnecessary in the first way of writing the function specification.

Type variables can be used in arguments, as in the following examples:

```
-spec lists:map(fun((A) -> B), [A]) -> [B].
-spec lists:filter(fun((X) -> bool()), [X]) -> [X].
```

This means that `map` takes a function from type `A` to `B` and list of objects of type `A` and returns a list of type `B` objects, and so on.

Exported and Local Types

Sometimes we want the definition of a type to be local to the module where the type is defined; in other circumstances, we want to export the type to another module. Imagine two modules `a` and `b`. Module `a` produces objects of type `rich_text`, and module `b` manipulates these objects. In module `a`, we make the following annotations:

```
-module(a).
-type rich_text() :: [{font(), char()}].
-type font()      :: integer().
-export_type([rich_text/0, font/0]).
...
```

Not only do we declare a rich text and font type, we also export them using an -export_type(...) annotation.

Suppose module b manipulates instances of rich text; there might be some function rich_text_length that computes the length of a rich-text object. We could write the type specification for this function as follows:

```
-module(b).
...
-spec rich_text_length(a:rich_text()) -> integer().
...
```

The input argument to rich_text_length uses the fully qualified type name, a:rich_text(), which means the type rich_text() exported from the module a.

Opaque Types

In the previous section, two modules, a and b, cooperate by manipulating the internal structure of the object that represents rich text. We may, however, want to hide the internal details of the rich-text data structure so that only the module that creates the data structure knows the details of the type. This is best explained with an example.

Assume module a starts like this:

```
-module(a).
-opaque rich_text() :: [{font(), char()}].
-export_type([rich_text/0]).

-export([make_text/1, bounding_box/1]).
-spec make_text(string()) -> rich_text().
-spec bounding_box(rich_text()) -> {Height::integer(), Width::integer()}.
...
```

The following statement:

```
-opaque rich_text() :: [{font(), char()}].
```

creates an opaque types called rich_text(). Now let's look at some code that tries to manipulate rich-text objects:

```
-module(b).
...

do_this() ->
   X = a:make_text("hello world"),
   {W, H} = a:bounding_box(X)
```

The module b never needs to know anything about the internal structure of the variable X. X is created inside the module a and is passed back into a when we call bounding_box(X).

Now suppose we write code that makes use of some knowledge about the shape of the rich_text object. For example, suppose we create a rich-text object and then ask what fonts are needed to render the object. We might write this:

```erlang
-module(c).
...

fonts_in(Str) ->
    X = a:make_text(Str),
    [F || {F,_} <- X].
```

In the list comprehension we "know" that X is list of 2-tuples. In the module a we declared the return type of make_text to be an opaque type, which means we are not supposed to know anything about the internal structure of the type. Making use of the internal structure of the type is called an *abstraction violation* and can be detected by the dialyzer if we correctly declare the visibility of the types in the functions involved.

9.3 A Session with the Dialyzer

The first time you run the dialyzer you need to build a cache of all the types in the standard libraries that you intend to use. This is a once-only operation. If you launch the dialyzer, it tells you what to do.

```
$ dialyzer
Checking whether the PLT /Users/joe/.dialyzer_plt is up-to-date...
dialyzer: Could not find the PLT: /Users/joe/.dialyzer_plt
Use the options:
--build_plt   to build a new PLT; or
--add_to_plt  to add to an existing PLT

For example, use a command like the following:
dialyzer --build_plt --apps erts kernel stdlib mnesia
...
```

PLT is short for *persistent lookup table*. The PLT should contain a cache of all the types in the standard system. Building the PLT takes a few minutes. The first command we give builds the PLT for erts, stdlib, and kernel.

```
$ dialyzer --build_plt --apps erts kernel stdlib
Compiling some key modules to native code... done in 0m59.78s
Creating PLT /Users/joe/.dialyzer_plt ...
Unknown functions:
compile:file/2
compile:forms/2
```

```
compile:noenv_forms/2
compile:output_generated/1
crypto:des3_cbc_decrypt/5
crypto:start/0
Unknown types:
compile:option/0
done in 4m3.86s
done (passed successfully)
```

Now that we have built the PLT, we are ready to run the dialyzer. The warnings about unknown functions occur because the functions referred to are in applications outside the three we have chosen to analyze.

The dialyzer is *conservative*. If it complains, then there really is an inconsistency in the program. One of the goals of the project that produced the dialyzer was to eliminate false warning messaging, that is, messages that warn for errors that are not real errors.

In the following sections, we'll give examples of incorrect programs; we'll run the dialyzer on these programs and illustrate what kind of errors we can expect to be reported by the dialyzer.

Incorrect Use of a BIF Return Value

dialyzer/test1.erl
```erlang
-module(test1).
-export([f1/0]).

f1() ->
    X = erlang:time(),
    seconds(X).

seconds({_Year, _Month, _Day, Hour, Min, Sec}) ->
    (Hour * 60 + Min)*60 + Sec.
```

```
> dialyzer test1.erl
Checking whether the PLT /Users/joe/.dialyzer_plt is up-to-date... yes
Proceeding with analysis...
test1.erl:4: Function f1/0 has no local return
test1.erl:6: The call test1:seconds(X::
    {non_neg_integer(),non_neg_integer(),non_neg_integer()})
    will never return since it differs in the 1st argument
    from the success typing arguments: ({_,_,_,number(),number(),number()})
test1.erl:8: Function seconds/1 has no local return
test1.erl:8: The pattern {_Year, _Month, _Day, Hour, Min, Sec} can never
    match the type {non_neg_integer(),non_neg_integer(),non_neg_integer()}
done in 0m0.41s
```

This rather scary error message is because erlang:time() returns a 3-tuple called {Hour, Min, Sec} and not a 6-tuple as we expected. The message "Function f1/0 has no local return" means f1/0 will crash. The dialyzer knows that the return value of erlang:time() is an instance of the type {non_neg_integer(), non_neg_integer(), non_neg_integer()}) and so will never match the 6-tuple pattern, which is the argument to seconds/1.

Incorrect Arguments to a BIF

We can use the dialyzer to tell us when we call a BIF with incorrect arguments. Here is an example of this:

dialyzer/test2.erl
```
-module(test2).
-export([f1/0]).

f1() ->
    tuple_size(list_to_tuple({a,b,c})).
```

```
$ dialyzer test2.erl
test2.erl:4: Function f1/0 has no local return
test2.erl:5: The call erlang:list_to_tuple({'a','b','c'})
will never return since it differs in the 1st argument from the
success typing arguments: ([any()])
```

This tells us that list_to_tuple expects an argument of type [any()] and not {'a','b','c'}.

Incorrect Program Logic

The dialyzer can also detect faulty program logic. Here's an example:

dialyzer/test3.erl
```
-module(test3).
-export([test/0, factorial/1]).

test() -> factorial(-5).

factorial(0) -> 1;
factorial(N) -> N*factorial(N-1).
```

```
$ dialyzer test3.erl
test3.erl:4: Function test/0 has no local return
test3.erl:4: The call test3:factorial(-5) will never return since
it differs in the 1st argument from the success typing
arguments: (non_neg_integer())
```

This is actually pretty remarkable. The definition of factorial is incorrect. If factorial is called with a negative argument, the program will enter an infinite loop, eating up stack space, and eventually Erlang will run out of memory

and die. The dialyzer has deduced that the argument to factorial is of type non_neg_integer(), and therefore, the call to factorial(-5) is an error.

The dialyzer does not print out the inferred types of the function, so we'll ask typer what the types were.

```
$ typer test3.erl
-spec test() -> none().
-spec factorial(non_neg_integer()) -> pos_integer().
```

Typer has deduced that the type of factorial is (non_neg_integer()) -> pos_integer() and that the type of test() is none().

The programs have reasoned as follows: the base case of the recursion is factorial(0), and so for the argument of factorial to be zero, the call factorial(N-1) must eventually reduce to zero; therefore N must be greater or equal to one, which is the reason for the type of factorial. This is very clever.

Working with the Dialyzer

Using the dialyzer to check your programs for type errors involves a particular workflow. What you should not do is write the entire program with no type annotations and then, when you think that it is ready, go back and add type annotations to everything and then run the dialyzer. If you do this, you will probably get a large number of confusing errors and not know where to start looking to fix the errors.

The best way to work with the dialyzer is to use it at every stage of development. When you start writing a new module, *think about the types first* and declare them before you write your code. Write type specifications for all the exported functions in your module. Do this first before you start writing the code. You can comment out the type specs of the functions that you have not yet implemented and then uncomment them as you implement the functions.

Now write your functions, one at a time, and check after you have written each new function to see whether the dialyzer can find any errors in your program. Add type specifications if the function is exported. If the function is not exported, then add type specifications if you think this will help the type analysis or help us understand the program (remember, type annotations provide good documentation of the program). If the dialyzer finds any errors, then stop and think and find out exactly what the error means.

Things That Confuse the Dialyzer

The dialyzer can get easily confused. We can help prevent this by following a few simple rules.

- Avoid using -compile(export_all). If you export all functions in the module, the dialyzer might not be able to reason about some of the arguments to some of your exported functions; they could be called from anywhere and have any type. The values of these arguments can propagate to other functions in the module and give confusing errors.

- Provide detailed type specifications for all the arguments to the *exported* functions in the module. Try to tightly constrain the arguments to exported functions as much as possible. For example, at first sight you might reason that an argument to a function is an integer, but after a little more thought, you might decide that the argument is a positive integer or even a bounded integer. The more precise you can be about your types, the better results you will get with the dialyzer. Also, add precise guard tests to your code if possible. This will help with the program analysis and will often help the compiler generate better-quality code.

- Provide default arguments to all elements in a record definition. If you don't provide a default, the atom undefined is taken as the default, and this type will start propagating through the program and might produce strange type errors.

- Using anonymous variables in arguments to a function often results in types that are far less specific than you had intended; try to constrain variables as much as possible.

9.4 Type Inference and Success Typing

Some of the errors the dialyzer produces are pretty strange. To understand these errors, we have to understand the process by which the dialyzer derives the types of Erlang functions. Understanding this will help us interpret these cryptic error messages.

Type inference is the process of deriving the types of a function by analyzing the code. To do this, we analyze the program looking for *constraints*; from the constraints, we build a set of constraint equations, and then we solve the equations. The result is a set of types that we call the *success typing* of the program. Let's look at a simple module and see what it tells us.

```
dialyzer/types1.erl
-module(types1).
-export([f1/1, f2/1, f3/1]).

f1({H,M,S}) ->
    (H+M*60)*60+S.
```

```erlang
f2({H,M,S}) when is_integer(H) ->
    (H+M*60)*60+S.

f3({H,M,S}) ->
    print(H,M,S),
    (H+M*60)*60+S.

print(H,M,S) ->
    Str = integer_to_list(H) ++ ":" ++ integer_to_list(M) ++ ":" ++
        integer_to_list(S),
    io:format("~s", [Str]).
```

Before reading the next section, take a moment to stare hard at the code and try to work out the types of the variables that occur in the code.

This is what happens when we run the dialyzer:

```
$ dialyzer types1.erl
Checking whether the PLT /Users/joe/.dialyzer_plt is up-to-date... yes
Proceeding with analysis... done in 0m0.41s
done (passed successfully)
```

The dialyzer found no type errors in the code. But this does not mean that the code is correct; it means only that all the data types in the program are used consistently. When converting hours, minutes, and seconds to seconds, I wrote (H+M*60)*60+S, which is plain wrong—it should have been (H*60+M)*60+S. No type system will detect this. Even if you have a well-typed program, you still have to provide test cases.

Running the typer on the same program produces the following:

```
$ typer types1.erl
%% File: "types1.erl"
%% ------------------
-spec f1({number(),number(),number()}) -> number().
-spec f2({integer(),number(),number()}) -> number().
-spec f3({integer(),integer(),integer()}) -> integer().
-spec print(integer(),integer(),integer()) -> 'ok'.
```

typer reports the types of all the functions in the module that it analyzes. typer says that the type of the function f1 is as follows:

```
-spec f1({number(),number(),number()}) -> number().
```

This is derived by looking at the definition of f1, which was as follows:

```erlang
f1({H,M,S}) ->
    (H+M*60)*60+S.
```

This function provides us with five different constraints. First the argument to f1 must be a tuple of three elements. Each arithmetic operator provides an

additional constraint. For example, the subexpression M*60 tells us that M must be of type number() since both the arguments of a multiplication operator must be numbers. Similarly, …+S tells us that S must be an number.

Now consider the function f2. Here is the code and the inferred type of the function:

```
f2({H,M,S}) when is_integer(H) ->
    (H+M*60)*60+S.
```

```
-spec f2({integer(),number(),number()}) -> number().
```

The addition of the is_integer(H) guard added the additional constraint that H must be an integer, and this constraint changes the type of the first element of the tuple argument to f2 from number() to the more precise type integer().

Note that to be strictly correct we should really say "added the additional constraint that if the function succeeds, then H must have been an integer." This is why we call the inferred type of the function the *success typing*—it literally means "the types that the arguments in a function had to have in order for the function evaluation to succeed."

Now let's move to the final function in types1.erl.

```
f3({H,M,S}) ->
    print(H,M,S),
    (H+M*60)*60+S.

print(H,M,S) ->
    Str = integer_to_list(H) ++ ":" ++ integer_to_list(M) ++ ":" ++
        integer_to_list(S),
    io:format("~s", [Str]).
```

The inferred types were as follows:

```
-spec f3({integer(),integer(),integer()}) -> integer().
-spec print(integer(),integer(),integer()) -> 'ok'.
```

Here you can see how calling integer_to_list constrains the type of its argument to be an integer. This constraint that occurs in the function print propagates into the body of the function f3.

As we have seen, type analysis proceeds in two stages. First we derive a set of constraint equations; then we solve these equations. When the dialyzer finds no errors, it is saying that the set of constraint equations is solvable, and the typer prints out the solutions to these equations. When the equations are inconsistent and cannot be solved, the dialyzer reports an error.

Now we'll make a small change to the previous program and introduce an error to see what effect it has on the analysis.

```
dialyzer/types1_bug.erl
-module(types1_bug).
-export([f4/1]).

f4({H,M,S}) when is_float(H) ->
    print(H,M,S),
    (H+M*60)*60+S.

print(H,M,S) ->
    Str = integer_to_list(H) ++ ":" ++ integer_to_list(M) ++ ":" ++
        integer_to_list(S),
    io:format("~s", [Str]).
```

We'll run typer first.

```
$ typer types1_bug.erl
-spec f4(_) -> none().
-spec print(integer(),integer(),integer()) -> 'ok'.
```

Typer says that the return type of f4 is none(). This is a special type that means "this function will never return."

When we run the dialyzer, we see the following:

```
$ dialyzer types1_bug.erl
types1_bug.erl:4: Function f4/1 has no local return
types1_bug.erl:5: The call types1_bug:print(H::float(),M::any(),S::any())
   will never return since it differs in the 1st argument from the
   success typing arguments: (integer(),integer(),integer())
types1_bug.erl:8: Function print/3 has no local return
types1_bug.erl:9: The call erlang:integer_to_list(H::float())
   will never return since it differs in the 1st argument from the
   success typing arguments: (integer())
```

Now look back at the code for a moment. The guard test is_float(H) tells the system that H must be a float. But H gets propagated into the function print, and inside print the function call integer_to_list(H) tells the system that H must be an integer. Now the dialyzer has no idea which of these two statements is correct, so it assumes that both are wrong. This is why it says "Function print/3 has no local return value." This is one of the limitations of type systems; all they can say is that the program is inconsistent and then leave it up to the programmer to figure out why.

9.5   Limitations of the Type System

Let's look at what happens when we add type specifications to code. We'll start with the well-known boolean and function. and is true if both its arguments

are true, and it is false if any of its arguments are false. We'll define a function myand1 (which is supposed to work like and) as follows:

```
types1.erl
myand1(true, true) -> true;
myand1(false, _)   -> false;
myand1(_, false)   -> false.
```

Running typer on this, we see the following:

```
$typer types1.erl
-spec myand1(_,_) -> boolean().
...
```

The inferred type of myand1 is (_,_) -> boolean(), which means that each of the arguments to myand1 can be anything you like, and the return type will be boolean. It infers that the arguments to myand1 can be anything because of the underscores in the argument positions. For example, the second clause of myand1 is myand1(false, _) -> false, from which it infers that the second argument can be anything.

Now suppose we add an incorrect function bug1 to the module as follows:

```
types1.erl
bug1(X, Y) ->
    case myand1(X, Y) of
        true ->
            X + Y
    end.
```

Then we ask typer to analyze the module.

```
$ typer types1.erl
-spec myand1(_,_) -> boolean().
-spec bug1(number(), number()) -> number().
```

typer knows that + takes two numbers as arguments and returns a number, so it infers that both X and Y are numbers. It also has inferred that the arguments to myand1 can be anything, which is consistent with both X and Y being numbers. If we run the dialyzer on this module, no errors will be returned. typer thinks that calling bug1 with two number arguments will return a number, but it won't. It will crash. This example shows how under-specification of the types of the arguments (that is, using _ as a type instead of boolean()) led to errors that could not be detected when the program was analyzed.

We now know all we need to know about types. In the next chapter, we'll wrap up Part II of the book by looking at a number of ways to compile and run your programs. A lot of what we can do in the shell can be automated, and we'll

look at ways of doing this. By the time you have finished the next chapter, you'll know all there is to know about building and running sequential Erlang code. After that, we can turn to concurrent programming, which is actually the main subject of the book, but you have to learn to walk before you can run and to write sequential programs before you can write concurrent programs.

Exercises

1. Write some very small modules that export a single function. Write type specifications for the exported functions. In the functions make some type errors; then run the dialyzer on these programs and try to understand the error messages. Sometimes you'll make an error but the dialyzer will not find the error; stare hard at the program to try to work out why you did not get the error you expected.

2. Look at the type annotations in the code in the standard libraries. Find the source code for the module lists.erl and read all the type annotations.

3. Why is it a good idea to think about the types of a function in a module before you write the module? Is this always a good idea?

4. Experiment with opaque types. Create two modules; the first should export an opaque type. The second module should use the internal data structures of the opaque type exported by the first module in such a way as to cause an abstraction violation. Run the dialyzer on the two modules and make sure you understand the error messages.

Compiling and Running Your Program

In the previous chapters, we didn't say much about compiling and running your programs—we just used the Erlang shell. This is fine for small examples, but as your programs become more complex, you'll want to automate the process in order to make life easier. That's where makefiles come in.

There are actually three different ways to run your programs. In this chapter, we'll look at all three so you can choose the best method for any particular occasion.

Sometimes things will go wrong: makefiles will fail, environment variables will be wrong, and your search paths will be incorrect. We'll help you deal with these issues by looking at what to do when things go wrong.

10.1 Modifying the Development Environment

When you start programming in Erlang, you'll probably put all your modules and files in the same directory and start Erlang from this directory. If you do this, then the Erlang loader will have no trouble finding your code. However, as your applications become more complex, you'll want to split them into manageable chunks and put the code into different directories. And when you include code from other projects, this external code will have its own directory structure.

Setting the Search Paths for Loading Code

The Erlang runtime system makes use of a code autoloading mechanism. For this to work correctly, you must set a number of search paths in order to find the correct version of your code.

The code-loading mechanism is actually programmed in Erlang—we talked about this earlier in Section 8.10, *Dynamic Code Loading,* on page 122. Code loading is performed "on demand."

When the system tries to call a function in a module that has not been loaded, an exception occurs, and the system tries to find an object code file for the missing module. If the missing module is called myMissingModule, then the code loader will search for a file called myMissingModule.beam in all the directories that are in the current load path. The search stops at the first matching file, and the object code in this file is loaded into the system.

You can find the value of the current load path by starting an Erlang shell and giving the command code:get_path(). Here's an example:

```
1> code:get_path().
[".",
 "/usr/local/lib/erlang/lib/kernel-2.15/ebin",
 "/usr/local/lib/erlang/lib/stdlib-1.18/ebin",
 "/home/joe/installed/proper/ebin",
 "/usr/local/lib/erlang/lib/xmerl-1.3/ebin",
 "/usr/local/lib/erlang/lib/wx-0.99.1/ebin",
 "/usr/local/lib/erlang/lib/webtool-0.8.9.1/ebin",
 "/usr/local/lib/erlang/lib/typer-0.9.3/ebin",
 "/usr/local/lib/erlang/lib/tv-2.1.4.8/ebin",
 "/usr/local/lib/erlang/lib/tools-2.6.6.6/ebin",
 ...]
```

The two most common functions that we use to manipulate the load path are as follows:

-spec code:add_patha(Dir) => true | {error, bad_directory}

Add a new directory, Dir, to the start of the load path.

-spec code:add_pathz(Dir) => true | {error, bad_directory}

Add a new directory, Dir, to the end of the load path.

Usually it doesn't matter which you use. The only thing to watch out for is if using add_patha and add_pathz produces different results. If you suspect an incorrect module was loaded, you can call code:all_loaded() (which returns a list of all loaded modules) or code:clash() to help you investigate what went wrong.

There are several other routines in the module code for manipulating the path, but you probably won't ever need to use them, unless you're doing some strange system programming.

The usual convention is to put these commands in a file called .erlang in your home directory.

Alternatively, you can start Erlang with a command like this:

```
$ erl -pa Dir1 -pa Dir2 ... -pz DirK1 -pz DirK2
```

The -pa Dir flag adds Dir to the beginning of the code search path, and -pz Dir adds the directory to the end of the code path.

Executing a Set of Commands When the System Is Started

We saw how you can set the load path in your .erlang file in your home directory. In fact, you can put any Erlang code in this file—when you start Erlang, it first reads and evaluates all the commands in this file.

Suppose your .erlang file is as follows:

```
io:format("Hi, I'm in your .erlang file~n").
...
```

Then when we start the system, we'll see the following output:

```
$ erl
...
Hi, I'm in your .erlang file
Eshell V5.9  (abort with ^G)
1>
```

If there is a file called .erlang in the current directory when Erlang is started, then it will take precedence over the .erlang in your home directory. This way, you can arrange that Erlang will behave in different ways depending upon where it is started. This can be useful for specialized applications. In this case, it's probably a good idea to include some print statements in the startup file; otherwise, you might forget about the local startup file, which could be very confusing.

Tip: In some systems, it's not clear where your home directory is, or it might not be where you think it is. To find out where Erlang thinks your home directory is, do the following:

```
1> init:get_argument(home).
{ok,[["/home/joe"]]}
```

From this we can infer that Erlang thinks that my home directory is /home/joe.

10.2 Different Ways to Run Your Program

Erlang programs are stored in modules. Once you have written your program, you have to compile it before you can run it. Alternatively, you can run your program directly without compiling it by running an *escript*.

The next sections show how to compile and run a couple of programs in a number of ways. The programs are slightly different, and the ways in which we start and stop them differ.

The first program, hello.erl, just prints "Hello world." It's not responsible for starting or stopping the system, and it does not need to access any command-line arguments. By way of contrast, the second program, fac, needs to access the command-line arguments.

Here's our basic program. It writes the string containing "Hello world" followed by a newline (~n is interpreted as a newline in the Erlang io and io_lib modules).

```erlang
hello.erl
-module(hello).
-export([start/0]).

start() ->
    io:format("Hello world~n").
```

Let's compile and run it three ways.

Compile and Run in the Erlang Shell

We begin by starting the Erlang shell.

```
$ erl
...
1> c(hello).
{ok,hello}
2> hello:start().
Hello world
ok
```

Quick Scripting

Often we want to be able to execute an arbitrary Erlang function from the OS command line. The -eval argument is very handy for quick scripting.

Here's an example:

```
erl -eval 'io:format("Memory: ~p~n", [erlang:memory(total)]).'\
    -noshell -s init stop
```

Compile and Run from the Command Prompt

Compiling a program can be done directly from the command prompt. This is the easiest way to do things if you just want to compile some code but not run it. This is done as follows:

```
$ erlc hello.erl
$ erl -noshell -s hello start -s init stop
Hello world
$
```

Note: All shell commands in this chapter assume that the user has installed a suitable shell on their system and that commands erl, erlc, and so on, can be executed directly in the shell. Details of how to configure the system are system specific and change with time. Up-to-date details can be found on the Erlang website[1] and in the Readme files on the main development archive.[2]

The first line, erlc hello.erl, compiles the file hello.erl, producing an object code file called hello.beam. The second command has three options.

-noshell Starts Erlang without an interactive shell (so you don't get the Erlang "banner," which ordinarily greets you when you start the system).

-s hello start Runs the function hello:start(). *Note*: When using the -s Mod ... option, the Mod must have been compiled.

-s init stop Stops the system by evaluating the function init:stop() after the previous command has finished.

The command erl -noshell ... can be put in a shell script, so typically we'd make a shell script to run our program that sets the path (with -pa *Directory*) and launches the program.

In our example, we used two -s ... commands. We can have as many functions as we like on the command line. Each -s ... command is evaluated with an apply statement, and when it has run to completion, the next command is evaluated.

Here's an example that launches hello.erl:

hello.sh
```sh
#!/bin/sh
erl -noshell -pa /home/joe/2012/book/JAERLANG/Book/code\
            -s hello start -s init stop
```

Note: This script needs an absolute path that points to the directory containing the file hello.beam. So although this script works on my machine, you'll have to edit it to get it to run on your machine.

To run the shell script, we chmod the file (only once), and then we can run the script.

```
$ chmod u+x hello.sh
$ ./hello.sh
Hello world
```

---

1. http://www.erlang.org
2. https://github.com/erlang/otp

Run As an Escript

Using an escript, you can run your programs directly as scripts—there's no need to compile them first. To run *hello* as an escript, we create the following file:

```
hello
#!/usr/bin/env escript

main(Args) ->
    io:format("Hello world~n").
```

The file must contain a function main(Args). When called from an operating system shell, Args will contain a list of the command-line arguments represented as atoms. On a Unix system, we can run this immediately and without compilation as follows:

```
$ chmod u+x hello
$ ./hello
Hello world
```

Note: The file mode for this file must be set to "executable" (on a Unix system, give the command chmod u+x File)—you have to do this only once, not every time you run the program.

Exporting Functions During Development

When you're developing code, it can be a bit of a pain to have to be continually adding and removing export declarations to your program just so that you can run the exported functions in the shell.

The special declaration -compile(export_all). tells the compiler to export every function in the module. Using this makes life much easier when you're developing code.

When you're finished developing the code, you should comment out the export_all declaration and add the appropriate export declarations. This is for two reasons. First, when you come to read your code later, you'll know that the only important functions are the exported functions. All the other functions cannot be called from outside the module, so you can change them in any way you like, provided the interfaces to the exported functions remain the same. Second, the compiler can produce much better code if it knows exactly which functions are exported from the module.

Note that using -compile(export_all). will make analyzing code with the dialyzer a lot more difficult.

Programs with Command-Line Arguments

"Hello world" had no arguments. Let's repeat the exercise with a program that computes factorials. It takes a single argument.

First, here's the code:

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

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

We can compile fac.erl and run it in the Erlang shell like this:

```
$ erl
...
1> c(fac).
{ok,fac}
2> fac:fac(25).
15511210043330985984000000
```

If we want to be able to run this program from the command line, we'll need to modify it to take command-line arguments.

```
fac1.erl
-module(fac1).
-export([main/1]).

main([A]) ->
    I = list_to_integer(atom_to_list(A)),
    F = fac(I),
    io:format("factorial ~w = ~w~n",[I, F]),
    init:stop().

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

We can then compile and run it.

```
$ erlc fac1.erl
$ erl -noshell -s fac1 main 25
factorial 25 = 15511210043330985984000000
```

Note: The fact that the function is called main has no significance; it can be called anything. The important thing is that the function name and the name on the command line agree.

Finally, we can run it as an escript.

factorial
```
#!/usr/bin/env escript
main([A]) ->
    I = list_to_integer(A),
    F = fac(I),
    io:format("factorial ~w = ~w~n",[I, F]).

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

No compilation is necessary; just run it, like so:

```
$ ./factorial 25
factorial 25 = 15511210043330985984000000
```

10.3 Automating Compilation with Makefiles

When I'm writing a large program, I like to automate as much as possible. There are two reasons for this. First, in the long run, it saves typing—typing the same old commands over and over again as I test and retest my program takes a lot of keystrokes, and I don't want to wear out my fingers.

Second, I often suspend what I'm working on and go work on some other project. It can be months before I return to a project that I have suspended, and when I return to the project, I've usually forgotten how to build the code in my project. make to the rescue!

make is *the* utility for automating my work—I use it for compiling and distributing my Erlang code. Most of my makefiles[3] are extremely simple, and I have a simple template that solves most of my needs.

I'm not going to explain makefiles in general. Instead, I'll show the form that I find useful for compiling Erlang programs. In particular, we'll look at the makefiles accompanying this book so you'll be able to understand them and build your own makefiles.

A Makefile Template

Here's the template that I base most of my makefiles on:

Makefile.template
```
# leave these lines alone
.SUFFIXES: .erl .beam .yrl

.erl.beam:
        erlc -W $<
```

----

3.   http://en.wikipedia.org/wiki/Make

```
.yrl.erl:
        erlc -W $<

ERL = erl -boot start_clean

# Here's a list of the erlang modules you want compiling
# If the modules don't fit onto one line add a \ character
# to the end of the line and continue on the next line

# Edit the lines below
MODS = module1 module2 \
        module3 ... special1 ...\
        ...
        moduleN

# The first target in any makefile is the default target.
# If you just type "make" then "make all" is assumed (because
#    "all" is the first target in this makefile)

all: compile

compile: ${MODS:%=%.beam} subdirs

## special compilation requirements are added here

special1.beam: special1.erl
        ${ERL} -Dflag1 -W0 special1.erl

## run an application from the makefile

application1: compile
        ${ERL} -pa Dir1  -s application1 start Arg1 Arg2

# the subdirs target compiles any code in
# sub-directories

subdirs:
        cd dir1; $(MAKE)
        cd dir2; $(MAKE)
        ...

# remove all the code

clean:
        rm -rf *.beam erl_crash.dump
        cd dir1; $(MAKE) clean
        cd dir2; $(MAKE) clean
```

The makefile starts with some rules to compile Erlang modules and files with the extension .yrl (these are files containing parser definitions for the Erlang parser generator program). The Erlang parser generator is called yecc (an Erlang version of yacc, which is short for *yet another compiler compiler*; see the online tutorial[4] for more details).

The important part is the line starting like this:

```
MODS = module1 module2
```

This is a list of all the Erlang modules that I want to compile.

Any module in the MODS list will be compiled with the Erlang command erlc *Mod*.erl. Some modules might need special treatment (for example the module special1 in the template file), so there is a separate rule to handle this.

Inside a makefile there are a number of *targets*. A target is an alphanumeric string starting in the first column and terminated by a colon (:). In the makefile template, all, compile, and special1.beam are all targets. To run the makefile, you give the shell command.

```
$ make [Target]
```

The argument Target is optional. If Target is omitted, then the first target in the file is assumed. In the previous example, the target all is assumed if no target is specified on the command line.

If I wanted to build all my software and run application1, then I'd give the command make application1. If I wanted this to be the default behavior, which happens when I just give the command make, then I'd move the lines defining the target application1 so that they were the first target in the makefile.

The target clean removes all compiled Erlang object code files and the file erl_crash.dump. The crash dump contains information that can help debug an application. See *Erlang Has Crashed and You Want to Read the Crash Dump, on page 172*, for details.

Specializing the Makefile Template

I'm not a fan of clutter in my software, so what I usually do is start with the template makefile and remove all lines that aren't relevant to my application. This results in makefiles that are shorter and easier to read. Alternatively, you could have a common makefile that is included by all makefiles and that is parameterized by the variables in the makefiles.

---

4.   http://www.erlang.org/contrib/parser_tutorial-1.0.tgz

Once I'm through with this process, I'll end up with a much simplified makefile, something like the following:

```
.SUFFIXES: .erl .beam

.erl.beam:
    erlc -W $<

ERL = erl -boot start_clean

MODS = module1 module2 module3

all: compile
    ${ERL} -pa '/home/joe/.../this/dir' -s module1 start

compile: ${MODS:%=%.beam}

clean:
    rm -rf *.beam erl_crash.dump
```

10.4 When Things Go Wrong

This section lists some common problems (and their solutions).

Stopping Erlang

Erlang can sometimes be difficult to stop. Here are a number of possible reasons:

- The shell is not responding.

- The Ctrl+C handler has been disabled.

- Erlang has been started with the -detached flag, so you may not be aware that it is running.

- Erlang has been started with the -heart Cmd option. This option causes an OS monitor process to be set up that watches over the Erlang OS process. If the Erlang OS process dies, then Cmd is evaluated. Often Cmd will simply restart the Erlang system. This is one of the tricks we use when making fault-tolerant nodes—if Erlang itself dies (which should never happen), it just gets restarted. The trick here is to find the heartbeat process (use ps on Unix-like systems and the Task Manager on Windows) and kill it before you kill the Erlang process.

- Something might have gone seriously wrong and left you with a detached zombie Erlang process.

Undefined (Missing) Code

If you try to run code in a module that the code loader cannot find (because the code search path was wrong), you'll be met with an undef error message. Here's an example:

```
1> glurk:oops(1,23).
** exception error: undefined function glurk:oops/2
```

Actually, there is no module called glurk, but that's not the issue here. The thing you should be concentrating on is the error message. The error message tells us that the system tried to call the function oops with two arguments in the module glurk. So, one of four things could have happened.

- There really is no module glurk—nowhere, not anywhere. This is probably because of a spelling mistake.

- There is a module glurk, but it hasn't been compiled. The system is looking for a file called glurk.beam somewhere in the code search path.

- There is a module glurk and it has been compiled, but the directory containing glurk.beam is not one of the directories in the code search path. To fix this, you'll have to change the search path.

- There are several different versions of glurk in the code load path, and we've chosen the wrong one. This is a rare error, but it can happen.

  If you suspect this has happened, you can run the code:clash() function, which reports all duplicated modules in the code search path.

Has Anybody Seen My Semicolons?

If you forget the semicolons between the clauses in a function or put periods there instead, you'll be in trouble—real trouble.

If you're defining a function foo/2 in line 1234 of the module bar and put a period instead of a semicolon, the compiler will say this:

bar.erl:1234 function foo/2 already defined.

Don't do it. Make sure your clauses are always separated by semicolons.

The Shell Isn't Responding

If the shell is not responding to commands, then a number of things might have happened. The shell process itself might have crashed, or you might have issued a command that will never terminate. You might even have

forgotten to type a closing quote mark or forgotten to type dot-carriage-return at the end of your command.

Regardless of the reason, you can interrupt the current shell by pressing Ctrl+G and proceeding as in the following example:

```
❶ 1> receive foo -> true end.
   ^G
   User switch command
❷ --> h
   c [nn]    - connect to job
   i [nn]    - interrupt job
   k [nn]    - kill job
   j         - list all jobs
   s         - start local shell
   r [node] - start remote shell
   q         - quit erlang
   ? | h     - this message
❸ --> j
   1* {shell,start,[init]}
❹ --> s
   --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
❺ --> c 2
   Eshell V5.5.1  (abort with ^G)
   1> init:stop().
   ok
   2> $
```

❶ Here we told the shell to receive a foo message. But since nobody ever sends the shell this message, the shell goes into an infinite wait. We entered the shell by pressing Ctrl+G.

❷ The system enters "shell JCL" (Job Control Language) mode. We typed h for some help.

❸ Typing j listed all jobs. Job 1 is marked with a star, which means it is the default shell. All the commands with an optional argument [nn] use the default shell unless a specific argument is supplied.

❹ Typing the command s started a new shell, followed by j again. This time we can see there are two shells marked 1 and 2, and shell 2 has become the default shell.

❺ We typed c 2, which connected us to the newly started shell 2; after this, we stopped the system.

As we can see, we can have many shells in operation and swap between them by pressing Ctrl+G and then the appropriate commands. We can even start a shell on a remote node with the r command.

My Makefile Doesn't Make

What can go wrong with a makefile? Well, lots, actually. But this isn't a book about makefiles, so I'll deal only with the most common errors. Here are the two most common errors that I make:

- *Blanks in the makefile*: Makefiles are extremely persnickety. Although you can't see them, each of the indented lines in the makefile (with the exception of continuation lines, where the *previous* line ends with a \ character) should begin with a tab character. If there are any spaces there, make will get confused, and you'll start seeing errors.

- *Missing erlang file*: If one of the modules declared in MODS is missing, you'll get an error message. To illustrate this, assume that MODS contains a module name glurk but that there is no file called glurk.erl in the code directory. In this case, make will fail with the following message:

```
$ make
make: *** No rule to make target `glurk.beam',
         needed by `compile'.  Stop.
```

Alternatively, there is no missing module, but the module name is spelled incorrectly in the makefile.

Erlang Has Crashed and You Want to Read the Crash Dump

If Erlang crashes, it leaves behind a file called erl_crash.dump. The contents of this file might give you a clue as to what has gone wrong. To analyze the crash dump, there is a web-based crash analyzer. To start the analyzer, give the following command:

```
1> crashdump_viewer:start().
WebTool is available at http://localhost:8888/
Or  http://127.0.0.1:8888/
ok
```

Then point your browser at http://localhost:8888/. You can then happily surf the error log.

10.5 Getting Help

On a Unix system, we can access the man pages as follows:

```
$ erl -man erl
NAME
erl - The Erlang Emulator

DESCRIPTION
The erl program starts the Erlang runtime system.
The exact details (e.g. whether erl is a script
or a program and which other programs it calls) are system-dependent.
...
```

We can also get help about individual modules as follows:

```
$ erl -man lists
MODULE
lists - List Processing Functions

DESCRIPTION
This  module  contains  functions  for  list  processing.
The functions are organized in two groups:
...
```

Note: On a Unix system, the manual pages are not installed by default. If the command erl -man ... does not work, then you need to install the manual pages. All the manual pages are in a single compressed archive.[5] The manual pages should be unpacked in the root of the Erlang installation directory (usually /usr/local/lib/erlang).

The documentation is also downloadable as a set of HTML files. On Windows the HTML documentation is installed by default and accessible through the Erlang section of the Start menu.

10.6 Tweaking the Environment

The Erlang shell has a number of built-in commands. You can see them all with the shell command help().

```
1> help().
** shell internal commands **
b()        -- display all variable bindings
e(N)       -- repeat the expression in query <N>
f()        -- forget all variable bindings
f(X)       -- forget the binding of variable X
h()        -- history
...
```

All these commands are defined in the module shell_default.

_____

5.   http://www.erlang.org/download.html

If you want to define your own commands, just create a module called user_default. Here's an example:

```erlang
user_default.erl
-module(user_default).

-compile(export_all).

hello() ->
    "Hello Joe how are you?".

away(Time) ->
    io:format("Joe is away and will be back in ~w minutes~n",
              [Time]).
```

Once this has been compiled and is placed somewhere in your load path, then you can call any of the functions in user_default without giving a module name.

```erlang
1> hello().
"Hello Joe how are you?"
2> away(10).
Joe is away and will be back in 10 minutes
ok
```

Now we're through with the nuts-and-bolts stuff, so we can begin to look at concurrent programs. This is where the fun really starts.

Exercises

1. Create a new directory and copy the makefile template in the chapter to this directory. Write a small Erlang program and save it in this directory. Add commands to the makefile and to the Erlang code to automatically run a set of unit tests (see *Adding Tests to Your Code,* on page 46) on the code when you type make.

Part III

Concurrent and Distributed Programs

This part covers concurrent and distributed Erlang. Building on sequential Erlang, you'll learn how to write concurrent programs and how to run these on distributed networks of computers.

Real-World Concurrency

Let's forget about programming for a while and think about what happens in the real world.

We understand concurrency.

A deep understanding of concurrency is hardwired into our brains. We react to stimulation extremely quickly, using a part of the brain called the *amygdala*. Without this reaction, we would die. Conscious thought is just too slow; by the time the thought "hit the brakes" has formed itself, we have already done it.

While driving on a major road, we mentally track the positions of dozens, or perhaps hundreds, of cars. This is done without conscious thought. If we couldn't do this, we would probably be dead.

The world is parallel.

If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

This is why we should program in a concurrent programming language.

And yet most often we program real-world applications in sequential programming languages. This is unnecessarily difficult.

Use a language that was designed for writing concurrent applications, and concurrent development becomes a lot easier.

Erlang programs model how we think and interact.

We don't have shared memory. I have my memory. You have yours. We have two brains, one each. They are not joined. To change your memory, I send you a message: I talk, or I wave my arms.

You listen, you see, and your memory changes; however, without asking you a question or observing your response, I do not know that you have received my messages.

This is how it is with Erlang processes. Erlang processes have no shared memory. Each process has its own memory. To change the memory of some other process, you must send it a message and hope that it receives and understands the message.

To confirm that another process has received your message and changed its memory, you must ask it (by sending it a message). This is exactly how we interact.

> **Sue:** Hi, Bill, my telephone number is 345-678-1234.
>
> **Sue:** Did you hear me?
>
> **Bill:** Sure, your number is 345-678-1234.

These interaction patterns are well known to us. From birth onward we learn to interact with the world by observing it and by sending it messages and observing the responses.

People function as independent entities who communicate by sending messages.

That's how Erlang processes work, and that's how we work, so it's easy to understand an Erlang program.

An Erlang program consists of dozens, thousands, or even hundreds of thousands of small processes. All these processes operate independently. They communicate with each other by sending messages. Each process has a private memory. They behave like a huge room of people all chattering away to each other.

This makes Erlang programs inherently easy to manage and scale. Suppose we have ten people (processes) and they have too much work to do. What can we do? Get more people. How can we manage these groups of people? It's easy—just shout instructions at them (broadcasting).

Erlang processes don't share memory, so there is no need to lock the memory while it is being used. Where there are locks, there are keys that can get lost. What happens when you lose your keys? You panic and don't know what to do. That's what happens in software systems when you lose your keys and your locks go wrong.

Distributed software systems with locks and keys always go wrong.

Erlang has no locks and no keys.

If somebody dies, other people will notice.

If I'm in a room and suddenly keel over and die, somebody will probably notice (well, at least I hope so). Erlang processes are just like people—they can on occasion die. Unlike people, when they die, they shout out in their last breath exactly what they have died from.

Imagine a room full of people. Suddenly one person keels over and dies. Just as they die, they say "I'm dying of a heart attack" or "I'm dying of an exploded gastric wobbledgog." That's what Erlang processes do. One process might die saying "I'm dying because I was asked to divide by zero." Another might say, "I'm dying because I was asked what the last element in an empty list was."

Now in our room full of people, we might imagine there are specially assigned people whose job it is to clear away the bodies. Let's imagine two people, Jane and John. If Jane dies, then John will fix any problems associated with Jane's death. If John dies, then Jane will fix the problems. Jane and John are linked with an invisible agreement that says that if one of them dies, the other will fix up any problems caused by the death.

That's how error detection in Erlang works. Processes can be linked. If one of the processes dies, the other process gets an error message saying why the first process died.

That's basically it.

That's how Erlang programs work.

Here's what we've learned so far:

- Erlang programs are made of lots of processes. These processes can send messages to each other.

- These messages may or may not be received and understood. If you want to know whether a message was received and understood, you must send the process a message and wait for a reply.

- Pairs of processes can be linked. If one of the processes in a linked pair dies, the other process in the pair will be sent a message containing the reason why the first process died.

This simple model of programming is part of a model I call *concurrency-oriented programming.*

In the next chapter, we'll start writing concurrent programs. We need to learn three new primitives: spawn, send (using the ! operator), and receive. Then we can write some simple concurrent programs.

When processes die, some other process notices if they are linked. This is the subject of Chapter 13, *Errors in Concurrent Programs*, on page 199.

As you read the next two chapters, think of people in a room. The people are the processes. The people in the room have individual private memories; this is the state of a process. To change your memory, I talk to you, and you listen. This is sending and receiving messages. We have children; this is spawn. We die; this is a process exit.

Concurrent Programming

Writing concurrent programs is easy once we know sequential Erlang. All we need are three new primitives: spawn, send, and receive. spawn creates a parallel process. send sends a message to a process, and receive receives messages.

Erlang concurrency is based on *processes*. These are small, self-contained virtual machines that can evaluate Erlang functions.

I'm sure you've met processes before, but only in the context of operating systems. *In Erlang, processes belong to the programming language and not the operating system.* This means that Erlang processes will have the same logical behavior on any operating system, so we can write portable concurrent code that can run on any operating system that supports Erlang.

In Erlang:

- Creating and destroying processes is very fast.

- Sending messages between processes is very fast.

- Processes behave the same way on all operating systems.

- We can have very large numbers of processes.

- Processes share no memory and are completely independent.

- The only way for processes to interact is through message passing.

For these reasons Erlang is sometimes called a *pure message passing language.*

If you haven't programmed with processes before, you might have heard rumors that it is rather difficult. You've probably heard horror stories of memory violations, race conditions, shared-memory corruption, and the like. In Erlang, programming with processes is easy.

12.1 The Concurrency Primitives

Everything we've learned about sequential programming is still true for concurrent programming. All we have to do is to add the following primitives:

Pid = spawn(Mod, Func, Args)

Creates a new concurrent process that evaluates apply(Mod, Func, Args). The new process runs in parallel with the caller. spawn returns a Pid (short for *process identifier*). You can use a Pid to send messages to the process. Note that the function Func with arity length(Args) must be exported from the module Mod.

When a new process is created, the *latest* version of the module defining the code is used.

Pid = spawn(Fun)

Creates a new concurrent process that evaluates Fun(). This form of spawn always uses the current value of the fun being evaluated, and this fun does not have to be exported from the module.

The essential difference between the two forms of spawn has to do with dynamic code upgrade. How to choose between the two forms of spawn is discussed later in Section 12.8, *Spawning with MFAs or Funs*, on page 197.

Pid ! Message

Sends Message to the process with identifier Pid. Message sending is asynchronous. The sender does not wait but continues with what it was doing. ! is called the *send* operator.

Pid ! M is defined to be M. Because of this, Pid1 ! Pid2 ! ... ! Msg means send the message Msg to all the processes Pid1, Pid2, and so on.

receive ... end

Receives a message that has been sent to a process. It has the following syntax:

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    ...
end
```

When a message arrives at the process, the system tries to match it against Pattern1 (with possible guard Guard1); if this succeeds, it evaluates Expressions1.

If the first pattern does not match, it tries Pattern2, and so on. If no pattern matches, the message is saved for later processing, and the process waits for the next message. This is described in more detail in Section 12.5, *Selective Receive, on page 193*.

The patterns and guards used in a receive statement have exactly the same syntactic form and meaning as the patterns and guards that we use when we define a function.

That's it. You don't need threading and locking and semaphores and artificial controls.

So far we have glossed over exactly how spawn, send, and receive work. When a spawn command is executed, the system creates a new process. Each process has an associated mailbox that is also created when the process is created.

When you send a message to a process, the message is put into the mailbox of the process. The only time the mailbox is examined is when your program evaluates a receive statement.

Using these three primitives, we can recast the area/1 function in Section 4.1, *Modules Are Where We Store Code, on page 43* into a process. Just to remind you, the code that defined the area/1 function looked like this:

geometry.erl
```erlang
area({rectangle, Width, Height}) -> Width * Height;
area({square, Side})             -> Side * Side.
```

Now we'll rewrite the same function as a *process*. To do this, we take the two patterns that were the arguments to the area function and rearrange them to form the patterns in a receive statement.

area_server0.erl
```erlang
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {square, Side} ->
            io:format("Area of square is ~p~n", [Side * Side]),
            loop()
    end.
```

We can create a process that evaluates loop/0 in the shell.

```
1> Pid = spawn(area_server0, loop, []).
<0.36.0>
2> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
{rectangle,6,10}
3> Pid ! {square, 12}.
Area of square is 144
{square, 144}
```

In line 1 we created a new parallel process. spawn(area_server, loop, []) creates a parallel process that evaluates area_server:loop(); it returns Pid, which is printed as <0.36.0>.

In line 2 we sent a message to the process. This message matches the first pattern in the receive statement in loop/0:

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

Having received a message, the process prints the area of the rectangle. Finally, the shell prints {rectangle, 6, 10}. This is because the value of Pid ! Msg is defined to be Msg.

12.2 Introducing Client-Server

Client-server architectures are central to Erlang. Traditionally, client-server architectures have involved a network that separates a client from a server. Most often there are multiple instances of the client and a single server. The word *server* often conjures up a mental image of some rather heavyweight software running on a specialized machine.

In our case, a much lighter-weight mechanism is involved. The client and server in a client-server architecture are separate processes, and normal Erlang message passing is used for communication between the client and the server. Both client and server can run on the same machine or on two different machines.

The words *client* and *server* refer to the roles that these two processes have; the client always initiates a computation by sending a *request* to the server. The server computes a reply and sends a *response* to the client.

Let's write our first client-server application. We'll start by making some small changes to the program we wrote in the previous section.

In the previous program, all that we needed was to send a request to a process that received and printed that request. Now, what we want to do is send a response to the process that sent the original request. The trouble is we do not know to whom to send the response. To send a response, the client has to include an address to which the server can reply. This is like sending a letter to somebody—if you want to get a reply, you had better include your address in the letter!

So, the sender must include a reply address. This can be done by changing this:

```
Pid ! {rectangle, 6, 10}
```

to the following:

```
Pid ! {self(),{rectangle, 6, 10}}
```

self() is the PID of the client process.

To respond to the request, we have to change the code that receives the requests from this:

```
loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop()
        ...
```

to the following:

```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        ...
```

Note how we now send the result of our calculation back to the process identified by the From parameter. Because the client set this parameter to its own process ID, it will receive the result.

The process that sends requests is usually called a *client*. The process that receives requests and replies to the client is called a *server*.

In addition, it's good practice to make sure every message sent to a process is actually received. If we send a message to the process that doesn't match one of the two patterns in the original receive statement, then this message will end up in the mailbox of the process and never be received. To deal with

this, we add a clause at the end of the receive statement that is guaranteed
to match any message that is sent to the process.

Finally, we add a small utility function called rpc (short for *remote procedure call*)
that encapsulates sending a request to a server and waiting for a response.

```
area_server1.erl
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
```

Putting all of this together, we get the following:

```
area_server1.erl
-module(area_server1).
-export([loop/0, rpc/2]).
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response ->
            Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Width * Ht,
            loop();
        {From, {circle, R}} ->
            From !  3.14159 * R * R,
            loop();
        {From, Other} ->
            From ! {error,Other},
            loop()
    end.
```

We can experiment with this in the shell.

```
1>  Pid = spawn(area_server1, loop, []).
<0.36.0>
2>  area_server1:rpc(Pid, {rectangle,6,8}).
48
3>  area_server1:rpc(Pid, {circle,6}).
113.097
4>  area_server1:rpc(Pid, socks).
{error,socks}
```

There's a slight problem with this code. In the function rpc/2, we send a request
to the server and then wait for a response. *But we do not wait for a response*

from the server; we wait for any message. If some other process sends the client a message while it is waiting for a response from the server, it will misinterpret this message as a response from the server. We can correct this by changing the form of the receive statement to this:

```erlang
loop() ->
    receive
        {From, ...} ->
            From ! {self(), ...}
            loop()
        ...
    end.
```

and by changing rpc to the following:

```erlang
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
```

When we call the rpc function, Pid is bound to some value, so in the pattern {Pid, Response}, Pid is bound, and Response is unbound. This pattern will match only a message containing a two-element tuple where the first element is Pid. All other messages will be queued. (receive provides what is called *selective receive*, which I'll describe after this section.) With this change, we get the following:

```erlang
area_server2.erl
-module(area_server2).
-export([loop/0, rpc/2]).
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

This works as expected.

```
1> Pid = spawn(area_server2, loop, []).
<0.37.0>
2> area_server2:rpc(Pid, {circle, 5}).
78.5397
```

There's one final improvement we can make. We can *hide* the spawn and rpc *inside* the module. Note that we also have to export the argument of spawn (that is, loop/0) from the module. This is good practice because we will be able to change the internal details of the server without changing the client code. Finally, we get this:

```
area_server_final.erl
-module(area_server_final).
-export([start/0, area/2, loop/0]).

start() -> spawn(area_server_final, loop, []).

area(Pid, What) ->
    rpc(Pid, What).
rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        {Pid, Response} ->
            Response
    end.
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

To run this, we call the functions start/0 and area/2 (where before we called spawn and rpc). These are better names that more accurately describe what the server does.

```
1> Pid = area_server_final:start().
<0.36.0>
2> area_server_final:area(Pid, {rectangle, 10, 8}).
80
3> area_server_final:area(Pid, {circle, 4}).
50.2654
```

So now we've built a simple client-server module. All we needed were the three primitives, spawn, send, and receive. This pattern will repeat over and over again in major and minor variations, but the underlying ideas are always the same.

12.3 Processes Are Cheap

At this point, you might be worried about performance. After all, if we're creating hundreds or thousands of Erlang processes, we must be paying some kind of penalty. Let's find out how much.

We'll do a bunch of spawns, create loads of processes, and time how long it takes. Here's the program; note that here we use spawn(Fun) and that the function being spawned does not have to be exported from the module:

```
processes.erl
-module(processes).

-export([max/1]).

%% max(N)

%%    Create N processes then destroy them
%%    See how much time this takes

max(N) ->
    Max = erlang:system_info(process_limit),
    io:format("Maximum allowed processes:~p~n",[Max]),
    statistics(runtime),
    statistics(wall_clock),
    L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
    {_, Time1} = statistics(runtime),
    {_, Time2} = statistics(wall_clock),
    lists:foreach(fun(Pid) -> Pid ! die end, L),
    U1 = Time1 * 1000 / N,
    U2 = Time2 * 1000 / N,
    io:format("Process spawn time=~p (~p) microseconds~n",
              [U1, U2]).

wait() ->
    receive
        die -> void
    end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

Here are the results I obtained on the computer I'm using at the moment, a 2.90GHz Intel Core i7 dual core with 8GB memory running Ubuntu:

```
1> processes:max(20000).
Maximum allowed processes:262144
Process spawn time=3.0 (3.4) microseconds
2> processes:max(300000).
Maximum allowed processes:262144

=ERROR REPORT==== 14-May-2013::09:32:56 ===
Too many processes

** exception error: a system limit has been reached
...
```

Spawning 20,000 processes took an average of 3.0 μs/process of CPU time and 3.4 μs of elapsed (wall-clock) time.

Note that I used the BIF erlang:system_info(process_limit) to find the maximum allowed number of processes. Some of these processes are reserved, so your program cannot actually use this number. When we exceed the system limit, the system refuses to start any more processes and produces an error report (command 2).

The system limit is set to 262,144 processes; to exceed this limit, you have to start the Erlang emulator with the +P flag as follows:

```
$ erl +P 3000000
1> processes:max(500000).
Maximum allowed processes:4194304
Process spawn time=2.52 (2.896) microseconds
ok
2> processes:max(1000000).
Maximum allowed processes:4194304
Process spawn time=3.65 (4.095) microseconds
ok
3> processes:max(2000000).
Maximum allowed processes:4194304
Process spawn time=4.02 (8.0625) microseconds
ok
6> processes:max(3000000).
Maximum allowed processes:4194304
Process spawn time=4.048 (8.624) microseconds
ok
```

In the previous example, the actual value chosen is the next highest power of two that is greater than the supplied argument. The actual value can be obtained by calling erlang:system_info(process_limit). We can see that the process spawn time increases as we increase the number of processes. If we continue to increase the number of processes, we will reach a point where we run out

of physical memory, and the system will start swapping physical memory to disk and run dramatically slower.

If you're writing a program that uses a large number of processes, it's a good idea to find out how many processes can fit into physical memory before the system starts swapping memory to disk and to make sure that your program will run in physical memory.

As you can see, creating large numbers of processes is pretty fast. If you're a C or Java programmer, you might hesitate to use a large number of processes, and you would have to take care managing them. In Erlang, creating processes simplifies programming instead of complicating it.

12.4 Receive with a Timeout

Sometimes a receive statement might wait forever for a message that never comes. This could be for a number of reasons. For example, there might be a logical error in our program, or the process that was going to send us a message might have crashed before it sent the message. To avoid this problem, we can add a timeout to the receive statement. This sets a maximum time that the process will wait to receive a message. The syntax is as follows:

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    ...
after Time ->
    Expressions
end
```

If no matching message has arrived within Time milliseconds of entering the receive expression, then the process will stop waiting for a message and evaluate Expressions.

Receive with Just a Timeout

You can write a receive consisting of only a timeout. Using this, we can define a function sleep(T), which suspends the current process for T milliseconds.

```
lib_misc.erl
sleep(T) ->
    receive
    after T ->
        true
    end.
```

Receive with Timeout Value of Zero

A timeout value of 0 causes the body of the timeout to occur immediately, but before this happens, the system tries to match any patterns in the mailbox. We can use this to define a function flush_buffer, which entirely empties all messages in the mailbox of a process.

lib_misc.erl
```erlang
flush_buffer() ->
    receive
        _Any ->
            flush_buffer()
    after 0 ->
        true
    end.
```

Without the timeout clause, flush_buffer would suspend forever and not return when the mailbox was empty. We can also use a zero timeout to implement a form of "priority receive," as follows:

lib_misc.erl
```erlang
priority_receive() ->
    receive
        {alarm, X} ->
            {alarm, X}
    after 0 ->
        receive
            Any ->
                Any
        end
    end.
```

If there is *not* a message matching {alarm, X} in the mailbox, then priority_receive will receive the first message in the mailbox. If there is no message at all, it will suspend in the innermost receive and return the first message it receives. If there is a message matching {alarm, X}, then this message will be returned immediately. Remember that the after section is checked only after pattern matching has been performed on all the entries in the mailbox.

Without the after 0 statement, the alarm message would not be matched first.

Note: Using large mailboxes with priority receive is rather inefficient, so if you're going to use this technique, make sure your mailboxes are not too large.

receive with Timeout Value of Infinity

If the timeout value in a receive statement is the atom infinity, then the timeout will *never* trigger. This might be useful for programs where the timeout value

is calculated outside the receive statement. Sometimes the calculation might want to return an actual timeout value, and other times it might want to have the receive wait forever.

Implementing a Timer

We can implement a simple timer using receive timeouts.

The function stimer:start(Time, Fun) will evaluate Fun (a function of zero arguments) after Time ms. It returns a handle (which is a PID), which can be used to cancel the timer if required.

```
stimer.erl
-module(stimer).
-export([start/2, cancel/1]).

start(Time, Fun) -> spawn(fun() -> timer(Time, Fun) end).
cancel(Pid) -> Pid ! cancel.
timer(Time, Fun) ->
    receive
        cancel ->
            void
    after Time ->
            Fun()
    end.
```

We can test this as follows:

```
1> Pid = stimer:start(5000, fun() -> io:format("timer event~n") end).
<0.42.0>
timer event
```

Here I waited more than five seconds so that the timer would trigger. Now I'll start a timer and cancel it before the timer period has expired.

```
2> Pid1 = stimer:start(25000, fun() -> io:format("timer event~n") end).
<0.49.0>
3> stimer:cancel(Pid1).
cancel
```

Timeouts and timers are central to the implementation of many communication protocols. When we wait for a message, we don't want to wait forever, so we add a timeout as in the examples.

12.5 Selective Receive

The receive primitive is used to extract messages from the process mailbox, but it does more than simple pattern matching; it also queues unmatched messages for later processing and manages timeouts. The following statement:

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    ...
after
    Time ->
        ExpressionsTimeout
end
```

works as follows:

1. When we enter a receive statement, we start a timer (but only if an after section is present in the expression).

2. Take the first message in the mailbox and try to match it against Pattern1, Pattern2, and so on. If the match succeeds, the message is removed from the mailbox, and the expressions following the pattern are evaluated.

3. If none of the patterns in the receive statement matches the first message in the mailbox, then the first message is removed from the mailbox and put into a "save queue." The second message in the mailbox is then tried. This procedure is repeated until a matching message is found or until all the messages in the mailbox have been examined.

4. If none of the messages in the mailbox matches, then the process is suspended and will be rescheduled for execution the next time a new message is put in the mailbox. When a new message arrives, the messages in the save queue are not rematched; only the new message is matched.

5. As soon as a message has been matched, then all messages that have been put into the save queue are reentered into the mailbox in the order in which they arrived at the process. If a timer was set, it is cleared.

6. If the timer elapses when we are waiting for a message, then evaluate the expressions ExpressionsTimeout and put any saved messages back into the mailbox in the order in which they arrived at the process.

12.6 Registered Processes

If we want to send a message to a process, then we need to know its PID, but when a process is created, only the parent process knows the PID. No other process in the system knows about the process. This is often inconvenient since the PID has to be sent to all processes in the system that want to communicate with this process. On the other hand, it's very *secure*; if you don't reveal the PID of a process, other processes can't interact with it in any way.

Erlang has a method for *publishing* a process identifier so that any process in the system can communicate with this process. Such a process is called a *registered process*. There are four BIFs for managing registered processes.

register(AnAtom, Pid)

Register the process Pid with the name AnAtom. The registration fails if AnAtom has already been used to register a process.

unregister(AnAtom)

Remove any registrations associated with AnAtom.

Note: If a registered process dies, it will be automatically unregistered.

whereis(AnAtom) -> Pid | undefined

Find out whether AnAtom is registered. Return the process identifier Pid, or return the atom undefined if no process is associated with AnAtom.

registered() -> [AnAtom::atom()]

Return a list of all registered processes in the system.

Using register, we can revise the example in the code on page 183, and we can try to register the name of the process that we created.

```
1> Pid = spawn(area_server0, loop, []).
<0.51.0>
2> register(area, Pid).
true
```

Once the name has been registered, we can send it a message like this:

```
3> area ! {rectangle, 4, 5}.
Area of rectangle is 20
{rectangle,4,5}
```

We can use register to make a registered process that represents a clock.

clock.erl
```
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) ->
    register(clock, spawn(fun() -> tick(Time, Fun) end)).
stop() -> clock ! stop.
tick(Time, Fun) ->
    receive
        stop ->
            void
    after Time ->
            Fun(),
            tick(Time, Fun)
    end.
```

The clock will happily tick away until you stop it.

```
3> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK {1164,553538,392266}
TICK {1164,553543,393084}
TICK {1164,553548,394083}
TICK {1164,553553,395064}
4> clock:stop().
stop
```

12.7 A Word About Tail Recursion

Take a look at the receive loop in the area server that we wrote earlier:

area_server_final.erl
```
loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! {self(), Width * Ht},
            loop();
        {From, {circle, R}} ->
            From !  {self(), 3.14159 * R * R},
            loop();
        {From, Other} ->
            From ! {self(), {error,Other}},
            loop()
    end.
```

If you look carefully, you'll see that every time we receive a message, we process the message and then immediately call loop() again. Such a procedure is called *tail-recursive*. A tail-recursive function can be compiled so that the last function call in a sequence of statements can be replaced by a simple jump to the start of the function being called. This means that a tail-recursive function can loop forever without consuming stack space.

Suppose we wrote the following (incorrect) code:

```
Line 1  loop() ->
   -        receive
   -            {From, {rectangle, Width, Ht}} ->
   -                From ! {self(), Width * Ht},
   5                loop(),
   -                someOtherFunc();
   -            {From, {circle, R}} ->
   -                From !  {self(), 3.14159 * R * R},
   -                loop();
  10                ...
   -        end
   -  end
```

> ## A Concurrent Program Template
>
> When I write a concurrent program, I almost always start with something like this:
>
> ```erlang
> -module(ctemplate).
> -compile(export_all).
>
> start() ->
>     spawn(?MODULE, loop, []).
>
> rpc(Pid, Request) ->
>     Pid ! {self(), Request},
>     receive
>         {Pid, Response} ->
>             Response
>     end.
>
> loop(X) ->
>     receive
>         Any ->
>             io:format("Received:~p~n",[Any]),
>             loop(X)
>     end.
> ```
>
> The receive loop is just any empty loop that receives and prints any message that I send to it. As I develop the program, I'll start sending messages to the processes. Because I start with no patterns in the receive loop that match these messages, I'll get a printout from the code at the bottom of the receive statement. When this happens, I add a matching pattern to the receive loop and rerun the program. This technique largely determines the order in which I write the program: I start with a small program and slowly grow it, testing it as I go along.

In line 5, we call loop(), but the compiler must reason that "after I've called loop(), I have to return to here, since I have to call someOtherFunc() in line 6." So, it pushes the address of someOtherFunc onto the stack and jumps to the start of loop. The problem with this is that loop() never returns; instead, it just loops forever. So, each time we pass line 5, another return address gets pushed onto the control stack, and eventually the system runs out of space.

Avoiding this is easy; if you write a function F that never returns (such as loop()), make sure you never call anything *after* calling F, and don't use F in a list or tuple constructor.

12.8 Spawning with MFAs or Funs

Spawning a function with an explicit module, function name, and argument list (called an MFA) is the proper way to ensure that our running processes will be correctly updated with new versions of the module code if it is compiled while it is being used. The dynamic code upgrade mechanism does not work

with spawned funs. It works only with explicitly named MFAs. For more details, read Section 8.10, *Dynamic Code Loading,* on page 122.

If you don't care about dynamic code upgrade or you are certain that your program will never be changed in the future, use the spawn(Fun) form of spawn. If in doubt, use spawn(MFA).

That's it—you can now write concurrent programs!

Next we'll look at error recovery and see how we can write fault-tolerant concurrent programs using three more concepts: links, signals, and trapping process exits. That's what we'll find in the next chapter.

Exercises

1. Write a function start(AnAtom, Fun) to register AnAtom as spawn(Fun). Make sure your program works correctly in the case when two parallel processes simultaneously evaluate start/2. In this case, you must guarantee that one of these processes succeeds and the other fails.

2. Measure the process spawning time on your machine, using the program in Section 12.3, *Processes Are Cheap,* on page 189. Plot a graph of the number of processes against the process creation time. What can you deduce from the graph?

3. Write a ring benchmark. Create N processes in a ring. Send a message round the ring M times so that a total of N * M messages get sent. Time how long this takes for different values of N and M.

    Write a similar program in some other programming language you are familiar with. Compare the results. Write a blog, and publish the results on the Internet!

Errors in Concurrent Programs

Handling errors in concurrent programs involves a completely different way of thinking than handling errors in sequential programs. In this chapter, we'll build upon the principles you learned about in Chapter 6, *Error Handling in Sequential Programs*, on page 87, extending the ideas to concurrent programs.

We'll look at the underlying philosophy of error handling and at the details of how errors are propagated between processes and trapped by other processes. Finally we'll round off with some small examples that form a basis for programming fault-tolerant software.

Imagine a system with only one sequential process. If this process dies, we might be in deep trouble since no other process can help. For this reason, sequential languages have concentrated on the prevention of failure and an emphasis on *defensive programming*.

In Erlang we have a large number of processes at our disposal, so the failure of any individual process is not so important. We usually write only a small amount of defensive code and instead concentrate on writing *corrective code*. We take measures to detect the errors and then correct them after they have occurred.

13.1 Error Handling Philosophy

Error handling in concurrent Erlang programs is based on the idea of *remote detection and handling of errors*. Instead of handling an error in the process where the error occurs, we let the process die and correct the error in some other process.

When we design a fault-tolerant system, we assume that errors will occur, that processes will crash, and that machines will fail. Our job is to detect the errors after they have occurred and correct them if possible. Users of the

system should not notice any failures or suffer any loss of service while the error is being fixed.

Since we concentrate on cure rather than prevention, our systems have very little defensive code; instead, we have code to clean up the system after errors have occurred. This means we will concentrate on how to detect errors, how to identify what has gone wrong, and how to keep the system in a stable state.

Detecting errors and finding out why something failed is built into the Erlang VM at a very low level and is part of the Erlang programming language. Building groups of processes that observe each other and take corrective action when errors are detected is provided in the standard OTP libraries and is described in Section 23.5, *The Supervision Tree,* on page 396. This chapter is about the language aspects of error detection and recovery.

The Erlang philosophy for building fault-tolerant software can be summed up in two easy-to-remember phrases: "Let some other process fix the error" and "Let it crash."

Let Some Other Process Fix the Error

Processes are arranged to monitor each other for health. If a process dies, some other process can observe this and perform corrective actions.

For one process to observe another, we must create a *link* or *monitor* between the processes. If the linked or monitored processes dies, the observing process is informed.

Observing processes work transparently across machine boundaries, so a process running on one machine can monitor the behavior of a process running on a different machine. This is the basis for programming fault-tolerant systems. We cannot make fault-tolerant systems on one machine since the entire machine might crash, so we need at least two machines. One machine performs computations, and the other machines observe the first machine and take over if the first machine crashes.

This can be thought of as an extension to handling errors in sequential code. We can, after all, catch exceptions in sequential code and try to correct the error (this was the subject of Chapter 6, *Error Handling in Sequential Programs,* on page 87), but if this fails or if the entire machine fails, we let some other process fix the error.

Let It Crash

This will sound very strange to you if you come from a language like C. In C we are taught to write *defensive code*. Programs should check their arguments

and not crash. There is a very good reason for this in C: writing multiprocess code is extremely difficult and most applications have only one process, so if this process crashes the entire application, you're in big trouble. Unfortunately, this leads to large quantities of error checking code, which is intertwined with the non-error-checking code.

In Erlang we do exactly the opposite. We build our applications in two parts: a part that solves the problem and a part that corrects errors if they have occurred.

The part that solves the problem is written with as little defensive code as possible; we assume that all arguments to functions are correct and the programs will execute without errors.

The part that corrects errors is often *generic*, so the same error-correcting code can be used for many different applications. For example, in database transactions if something goes wrong in the middle of a transaction, we simply abort the transaction and let the system restore the database to the state it was in before the error occurred. In an operating system, if a process crashes, we let the operating system close any open files or sockets and restore the system to a stable state.

This leads to a clean separation of issues. We write code that solves problems and code that fixes problems, but the two are not intertwined. This can lead to a dramatic reduction in code volume.

Why Crash?

Crashing immediately when something goes wrong is often a very good idea; in fact, it has several advantages.

- We don't have to write defensive code to guard against errors; we just crash.

- We don't have to think about what to do; we just crash, and somebody else will fix the error.

- We don't make matters *worse* by performing additional computations after we know that things have gone wrong.

- We can get very good error diagnostics if we flag the first place where an error occurs. Often continuing after an error has occurred leads to even more errors and makes debugging even more difficult.

- When writing error recovery code, we don't need to bother about why something crashed; we just need to concentrate on cleaning up afterward.

- It simplifies the system architecture, so we can think about the application and error recovery as two separate problems, not as one interleaved problem.

That's enough of the philosophy. Now let's start drilling down into the details.

> ### Getting Some Other Guy to Fix It
>
> Letting somebody else fix an error rather than doing it yourself is a good idea and encourages specialization. If I need surgery, I go to a doctor and don't try to operate on myself.
>
> If something trivial in my car goes wrong, the car's control computer will try to fix it. If this fails and something big goes wrong, I have to take the car to the garage, and some other guy fixes it.
>
> If something trivial in an Erlang process goes wrong, I can try to fix it with a catch or try statement. But if this fails and something big goes wrong, I'd better just crash and let some other process fix the error.

13.2 Error Handling Semantics

In this section, you'll learn about the semantics of interprocess error handling. You'll see some new terms that you'll come across later in the chapter. The best way to understand error handing is to quickly read through the definitions and then skip to the next sections for a more intuitive understanding of the concepts involved. You can always refer to this section if you need to do so.

Processes

There are two types of processes: *normal processes* and *system processes*. spawn creates a normal process. A normal process can become a system process by evaluating the BIF process_flag(trap_exit, true).

Links

Processes can be linked. If the two processes A and B are linked and A terminates for any reason, an error signal will be sent to B and the other way around.

Link sets

The *link set* of a process P is the set of processes that are linked to P.

Monitors

Monitors are similar to links but are one-directional. If A monitors B and if B terminates for any reason, a "down" message will be sent to A but not the other way around.

Messages and error signals

Processes collaborate by exchanging *messages* or *error signals*. Messages are sent using the send primitive. Error signals are sent automatically when a process crashes or when a process terminates. The error signals are sent to the link set of the process that terminated.

Receipt of an error signal

When a system process receives an error signal, the signal is converted into a message of the form {'EXIT', Pid, Why}. Pid is the identity of the process that terminated, and Why is the reason for termination (sometimes called the *exit reason*). If the process terminates without an error, then Why will be the atom normal; otherwise, Why describes the error.

When a normal process receives an error signal, it will terminate if the exit reason is not normal. When it terminates, it also broadcasts an exit signal to its link set.

Explicit error signals

A process that evaluates exit(Why) will terminate (if this code is not executing within the scope of a catch or try primitive) and broadcast an exit signal with the reason Why to its link set.

A process can send a "fake" error signal by evaluating exit(Pid, Why). In this case, Pid will receive an exit signal with the reason Why. The process that called exit/2 does not die (this is deliberate).

Untrappable exit signals

When a system process receives a *kill signal*, it terminates. Kill signals are generated by calling exit(Pid, kill). This signal bypasses the normal error signal processing mechanism and is not converted into a message. The exit kill signal should be reserved for rogue processes that refuse to die using any of the other error handling mechanisms.

These definitions might look complicated, but a detailed understanding of how the mechanisms work is usually not necessary to write fault-tolerant code. The default behavior of the system tries to do "the right thing" as regard to error handling.

The next sections use a series of diagrams to illustrate how the error mechanisms work.

13.3 Creating Links

Imagine we have a set of unrelated processes; this is shown on the left side of the following figure. The links are represented by dashed lines.
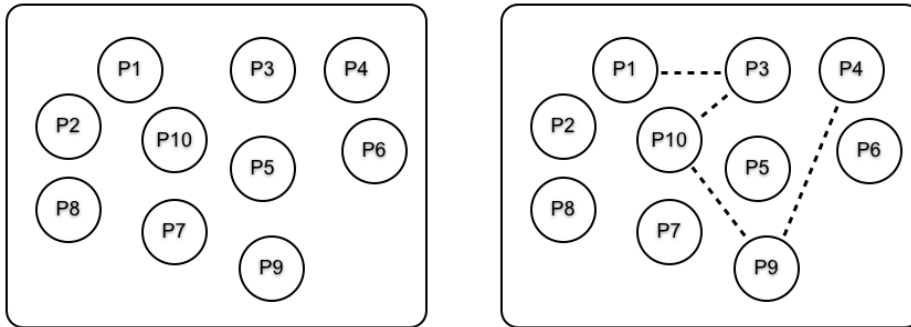
To create links, we call the primitive link(Pid), which creates a link between the calling process and Pid. So, if P1 calls link(P3), a link is created between P1 and P3.

After P1 calls link(P3), P3 calls link(P10), and so on, we arrive at the situation shown on the right side of the figure. Note that the link set of P1 has one element (P3), the link set of P3 has two elements (P1 and P10), and so on.

13.4 Groups of Processes That All Die Together

Often you'll want to create groups of processes that all die together. This is a very useful invariant for arguing about the behavior of a system. When processes collaborate to solve a problem and something goes wrong, we can sometimes recover, but if we can't recover, we just want to stop everything we were doing. This is rather like the notion of a transaction: either the processes do what they were supposed to do or they are all killed.

Assume we have some linked processes and that one of the linked processes dies. For example, see P9 in the following figure. The left side of the figure shows how the processes are linked before P9 dies. The right side shows which process are still alive after P9 has crashed and all error signals have been processed.
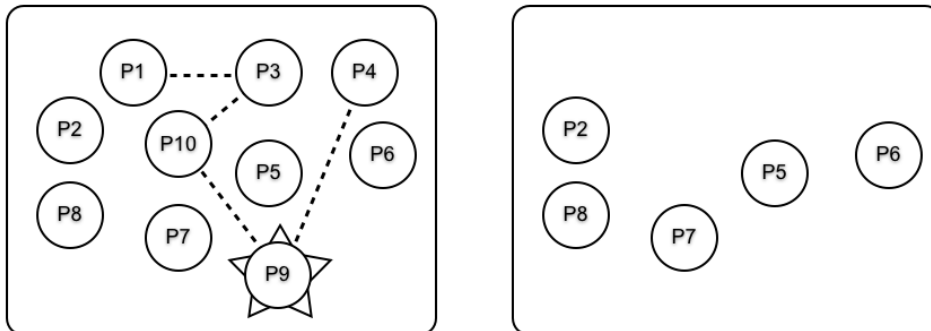
When P9 dies, an *error signal* is sent to processes P4 and P10. P4 and P10 also die because they are not system processes, and error signals are sent to any processes they are linked to. Ultimately, the error signals propagate to all the linked processes, and the entire group of linked processes dies.

Now if any of the processes P1, P3, P4, P9, or P10 die, they all die.

13.5 Setting Up a Firewall

Sometimes we don't want all our linked process to die, and we want to stop the propagation of errors through the system. The following figure illustrates this; here all linked process up to P3 die:
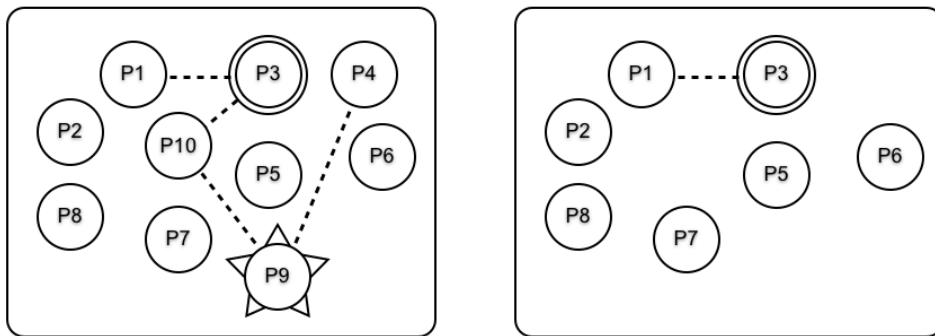
To achieve this, assume that P3 has evaluated process_flag(trap_exit, true) and become a system process (meaning that it can trap exit signals). This is shown with a double-circle border on the right side of the figure. After P9 crashed, the propagation of errors stopped at P3, so P1 and P3 did not die. This is shown on the right side of the figure.

P3 functions as a *firewall*, stopping errors from propagating to other processes in the system.

13.6 Monitors

Monitors are similar to links but with several significant differences.

- Monitors are unidirectional. If A monitors B and B dies, then A will be sent an exit message but not the other way around (recall that links were bidirectional, so if A and B were linked, the death of either process would result in the other process being informed).

- When a monitored process dies, a "down" message and not an exit signal is sent to the monitoring process. This means that the monitoring process does not have to become a system process in order to handle errors.

Monitors are used when you want asymmetry in error handling; links are used when you want symmetric error handling. Monitors are typically used by servers to monitor the behavior of clients.

The next section explains the semantics of the BIFs that manipulate links and monitors.

13.7 Error Handling Primitives

The primitives for manipulating links and monitors and for trapping and sending exit signals are as follows:

-spec spawn_link(Fun) -> Pid
-spec spawn_link(Mod, Fnc, Args) -> Pid

This behaves like spawn(Fun) or spawn(Mod,Func,Args) and also creates a link between the parent and child processes.

-spec spawn_monitor(Fun) -> {Pid, Ref}
-spec spawn_monitor(Mod, Func, Args) -> {Pid, Ref}

This is like spawn_link, but it creates a monitor rather than a link. Pid is the process identifier of the newly created process, and Ref is a reference to the process. If the process dies with the reason Why, then the message {'DOWN',Ref,process,Pid,Why} will be sent to the parent process.

-spec process_flag(trap_exit, true)

This turns the current process into a system process. A system process is a process that can receive and process error signals.

-spec link(Pid) -> true

This creates a link to the process Pid. Links are symmetric. If a process A evaluates link(B), then it will be linked to B. The net effect is the same as if B had evaluated link(A).

If the process Pid does not exist, then an exit noproc exception is raised.

If A is already linked to B and evaluates link(B) (or *vice versa*), the call is ignored.

-spec unlink(Pid) -> true

This removes any link between the current process and the process Pid.

-spec erlang:monitor(process, Item) -> Ref

This sets up a monitor. Item is a Pid or a registered name of a process.

-spec demonitor(Ref) -> true

This removes a monitor with reference Ref.

-spec exit(Why) -> none()

> This causes the current process to terminate with the reason Why. If the clause that executes this statement is not within the scope of a catch statement, then the current process will broadcast an exit signal, with argument Why to all processes to which it is currently linked. It will also broadcast a DOWN message to all processes that are monitoring it.

-spec exit(Pid, Why) -> true

> This sends an exit signal with the reason Why to the process Pid. The process executing this BIF does not itself die. This can be used to "fake" exit signals.

We can use these primitives to set up networks of processes that monitor each other, which then provide a basis for building fault-tolerant software.

13.8 Programming for Fault Tolerance

In this section, you'll learn a few simple techniques that can be used to make fault-tolerant code. This is not the whole story of how to make a fault-tolerant system, but it is the start of a story.

Performing an Action When a Process Dies

The function on_exit(Pid, Fun) watches the process Pid and evaluates Fun(Why) if the process exits with the reason Why.

```erlang
lib_misc.erl
on_exit(Pid, Fun) ->
    spawn(fun() ->
                  Ref = monitor(process, Pid),
                  receive
                      {'DOWN', Ref, process, Pid, Why} ->
                          Fun(Why)
                  end
          end).
```

monitor(process, Pid) (line 3) creates a monitor to Pid. When the process dies, a DOWN message is received (line 5) and calls Fun(Why) (line 6).

To test this, we'll define a function F that waits for a single message X and then computes list_to_atom(X).

```erlang
1> F = fun() ->
             receive
                 X -> list_to_atom(X)
             end
       end.
#Fun<erl_eval.20.69967518>
```

Why Spawning and Linking Must Be an Atomic Operation

Once upon a time Erlang had two primitives, spawn and link, and spawn_link(Mod, Func, Args) was defined like this:

```erlang
spawn_link(Mod, Func, Args) ->
    Pid = spawn(Mod, Fun, Args),
    link(Pid),
    Pid.
```

Then an obscure bug occurred. The spawned process died before the link statement was called, so the process died but no error signal was generated. This bug took a long time to find. To fix this, spawn_link was added as an atomic operation. Even simple-looking programs can be tricky when concurrency is involved.

We'll spawn this:

```erlang
2> Pid = spawn(F).
<0.61.0>
```

And we'll set up an on_exit handler to monitor it.

```erlang
3> lib_misc:on_exit(Pid,
                    fun(Why) ->
                        io:format(" ~p died with:~p~n",[Pid, Why])
                    end).
<0.63.0>
```

If we send an atom to Pid, the process will die (because it tries to evaluate list_to_atom of a nonlist), and the on_exit handler will be called.

```erlang
4> Pid ! hello.
hello
5>
=ERROR REPORT==== 14-May-2013::10:05:42 ===
Error in process <0.36.0> with exit value:
   {badarg,[{erlang,list_to_atom,[hello],[]}]}
```

The function that is invoked when the process dies can, of course, perform any computation it likes: it can ignore the error, log the error, or restart the application. The choice is up to the programmer.

Making a Set of Processes That All Die Together

Suppose we want to create several worker processes that are used to solve some problem. They evaluate the functions F1, F2, and so on. If any process dies, we want them all to die. We can do this by calling start([F1,F2, ...]).

```
start(Fs) ->
    spawn(fun() ->
                [spawn_link(F) || F <- Fs],
                receive
                    after
                        infinity -> true
                end
        end).
```

start(Fs) spawns a process, which spawns and links the worker processes and then waits for an infinite time. If any worker process dies, they all die.

If we want to know whether the processes have all died, we can add an on_exit handler to the start process.

```
Pid = start([F1, F2, ...]),
on_exit(Pid, fun(Why) ->
                    ... the code here runs if any worker
                    ... process dies
            end)
```

Making a Process That Never Dies

To wind up this chapter, we'll make a keep-alive process. The idea is to make a registered process that is always alive—if it dies for any reason, it will be immediately restarted.

We can use on_exit to program this.

lib_misc.erl
```
keep_alive(Name, Fun) ->
    register(Name, Pid = spawn(Fun)),
    on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).
```

This makes a registered process called Name that evaluates spawn(Fun). If the process dies for any reason, then it is restarted.

There is a rather subtle error in on_exit and keep_alive. If we stare hard at the following two lines of code:

```
Pid = register(...),
on_exit(Pid, fun(X) -> ..),
```

we see that there is a possibility that the process dies in the gap *between* these two statements. If the process dies before on_exit gets evaluated, then a link will be not be created, and the on_exit process will not work as you expected. This could happen if two programs try to evaluate keep_alive at the same time and with the same value of Name. This is called a *race condition*— two bits of code (this bit) and the code section that performs the link operation

inside on_exit are racing each other. If things go wrong here, your program might behave in an unexpected manner.

I'm not going to solve this problem here—I'll let you think about how to do this yourself. When you combine the Erlang primitives spawn, spawn_monitor, register, and so on, you must think carefully about possible race conditions and write your code in such a way that race conditions cannot happen.

Now you know all there is to know about error handling. Errors that cannot be trapped in sequential code flow out of the processes where they occurred, following links to other processes that can be programmed to take care of the errors. All the mechanisms we have described (the linking process and so on) work transparently across machine boundaries.

Crossing machine boundaries leads us to distributed programming. Erlang processes can spawn new processes *on other physical machines in the network*, making it easy to write distributed programs. Distributed programming is the subject of the next chapter.

Exercises

1. Write a function my_spawn(Mod, Func, Args) that behaves like spawn(Mod, Func, Args) but with one difference. If the spawned process dies, a message should be printed saying why the process died and how long the process lived for before it died.

2. Solve the previous exercise using the on_exit function shown earlier in this chapter.

3. Write a function my_spawn(Mod, Func, Args, Time) that behaves like spawn(Mod, Func, Args) but with one difference. If the spawned process lives for more than Time seconds, it should be killed.

4. Write a function that creates a registered process that writes out "I'm still running" every five seconds. Write a function that monitors this process and restarts it if it dies. Start the global process and the monitor process. Kill the global process and check that it has been restarted by the monitor process.

5. Write a function that starts and monitors several worker processes. If any of the worker processes dies abnormally, restart it.

6. Write a function that starts and monitors several worker processes. If any of the worker processes dies abnormally, kill all the worker processes and restart them all.

Distributed Programming

Writing distributed programs in Erlang is only a small step from writing concurrent programs. In distributed Erlang, we can spawn processes on remote nodes and machines. Having spawned a remote process, we'll see that all the other primitives, send, receive, link, and so on, work transparently over a network in the same way as they worked on a single node.

In this chapter, we'll introduce the libraries and Erlang primitives that we'll use to write distributed Erlang programs. *Distributed programs* are programs that are designed to run on networks of computers and that can coordinate their activities only by message passing.

Here are some reasons why we might want to write distributed applications:

Performance

We can make our programs go faster by arranging that different parts of the program are run in parallel on different machines.

Reliability

We can make fault-tolerant systems by structuring the system to run on several machines. If one machine fails, we can continue on another machine.

Scalability

As we scale up an application, sooner or later we will exhaust the capabilities of even the most powerful machine. At this stage, we have to add more machines to add capacity. Adding a new machine should be a simple operation that doesn't require large changes to the application architecture.

Intrinsically distributed application

Many applications are inherently distributed. If we write a multiuser game or chat system, different users will be scattered all over the globe. If we have a large number of users in a particular geographic location, we want to place the computation resources near the users.

Fun

> Most of the fun programs that I want to write are distributed. Many of these involve interaction with people and machines all over the world.

14.1 Two Models for Distribution

In this book we'll talk about two main models of distribution.

Distributed Erlang In distributed Erlang, programs are written to run on Erlang *nodes.* A node is a self-contained Erlang system containing a complete virtual machine with its own address space and own set of processes.

> We can spawn a process on any node, and all the message passing and error handling primitives we talked about in previous chapters work as in the single node case.

> Distributed Erlang applications run in a *trusted* environment—since any node can perform any operation on any other Erlang node, a high degree of trust is involved. Typically distributed Erlang applications will be run on clusters on the same LAN and behind a firewall, though they can run in an open network.

Socket-based distribution   Using TCP/IP sockets, we can write distributed applications that can run in an *untrusted* environment. The programming model is less powerful than that used in distributed Erlang but more secure. In Section 14.6, *Socket-Based Distribution*, on page 224, we'll see how to make applications using a simple socket-based distribution mechanism.

If you think back to the previous chapters, you'll recall that the basic unit that we construct programs from is the process. Writing a distributed Erlang program is easy; all we have to do is spawn our processes on the correct machines, and then everything works as before.

We are all used to writing sequential programs. Writing distributed programs is usually a lot more difficult. In this chapter, we'll look at a number of techniques for writing simple distributed programs. Even though the programs are simple, they are very useful.

We'll start with a number of small examples. To do this, we'll need to learn only two things; then we can make our first distributed program. We'll learn how to start an Erlang node and how to perform a remote procedure call on a remote Erlang node.

14.2 Writing a Distributed Program

When I develop a distributed application, I always work on the program in a specific order, which is as follows:

1. I write and test my program in a regular nondistributed Erlang session. This is what we've been doing up to now, so it presents no new challenges.

2. I test my program on two different Erlang nodes running *on the same computer.*

3. I test my program on two different Erlang nodes running *on two physically separated computers* either in the same local area network or anywhere on the Internet.

The final step can be problematic. If we run on machines within the same administrative domain, this is rarely a problem. But when the nodes involved belong to machines in different domains, we can run into problems with connectivity, and we have to ensure that our system firewalls and security settings are correctly configured.

To illustrate these steps, we'll make a simple name server. Specifically, we will do the following:

- Stage 1: Write and test the name server in a regular undistributed Erlang system.

- Stage 2: Test the name server on two nodes on the same machine.

- Stage 3: Test the name server on two different nodes on two different machines on the same local area network.

- Stage 4: Test the name server on two different machines belonging to two different domains in two different countries.

14.3 Building the Name Server

A *name server* is a program that, given a name, returns a value associated with that name. We can also change the value associated with a particular name.

Our first name server is extremely simple. It is not fault tolerant, so all the data it stores will be lost if it crashes. The point of this exercise is not to make a fault-tolerant name server but to get started with distributed programming techniques.

Stage 1: A Simple Name Server

Our name server kvs is a simple Key → Value, server. It has the following interface:

-spec kvs:start() -> true

    Start the server; this creates a server with the registered name kvs.

-spec kvs:store(Key, Value) -> true

    Associate Key with Value.

-spec kvs:lookup(Key) -> {ok, Value} | undefined

    Look up the value of Key, and return {ok, Value} if there is a value associated with Key; otherwise, return undefined.

The key-value server is implemented using the process dictionary get and put primitives, as follows:

```
socket_dist/kvs.erl
Line 1   -module(kvs).
    -    -export([start/0, store/2, lookup/1]).
    -
    -    start() -> register(kvs, spawn(fun() -> loop() end)).
    5
    -    store(Key, Value) -> rpc({store, Key, Value}).
    -
    -    lookup(Key) -> rpc({lookup, Key}).
    -
   10    rpc(Q) ->
    -        kvs ! {self(), Q},
    -        receive
    -            {kvs, Reply} ->
    -                Reply
   15        end.
    -
    -    loop() ->
    -        receive
    -            {From, {store, Key, Value}} ->
   20                put(Key, {ok, Value}),
    -                From ! {kvs, true},
    -                loop();
    -            {From, {lookup, Key}} ->
    -                From ! {kvs, get(Key)},
   25                loop()
    -        end.
```

Store messages are sent in line 6 and received in line 19. The main server starts in the loop function in line 17; it calls receive and waits for a store or lookup message and then just saves or retrieves the requested data from the

local process dictionary and sends a reply back to the client. We'll start by testing the server locally to see that it works correctly.

```
1> kvs:start().
true
2> kvs:store({location, joe}, "Stockholm").
true
3> kvs:store(weather, raining).
true
4> kvs:lookup(weather).
{ok,raining}
5> kvs:lookup({location, joe}).
{ok,"Stockholm"}
6> kvs:lookup({location, jane}).
undefined
```

So far, we get no unpleasant surprises. On to step 2. Let's distribute the application.

Stage 2: Client on One Node, Server on Second Node but Same Host

Now we'll start two Erlang nodes on the *same* computer. To do this, we need to open two terminal windows and start two Erlang systems.

First, we fire up a terminal shell and start a distributed Erlang node in this shell called gandalf; then, we start the server:

```
$ erl -sname gandalf
(gandalf@localhost) 1>  kvs:start().
true
```

The argument -sname gandalf means "start an Erlang node with name gandalf on the local host." Note how the Erlang shell prints the name of the Erlang node before the command prompt. The node name is of the form Name@Host. Name and Host are both atoms, so they will have to be quoted if they contain any nonatomic characters.

Important Note: If you run the previous command on your system, the node name might not be gandolf@localhost. It might be gandolf@H where H is your local hostname. This will depend upon how your system is configured. If this is the case, then you'll have to use the name H instead of localhost in all the examples that follow.

Next we start a *second* terminal session and start an Erlang node called bilbo. Then we can call the functions in kvs using the library module rpc. (Note that rpc is a standard Erlang library module, which is not the same as the rpc function we wrote earlier.)

```
$ erl -sname bilbo
(bilbo@localhost) 1> rpc:call(gandalf@localhost,
    kvs,store, [weather, fine]).
true
(bilbo@localhost) 2> rpc:call(gandalf@localhost,
    kvs,lookup,[weather]).
{ok,fine}
```

Now it may not look like it, but we've actually performed our first-ever distributed computation! The server ran on the first node that we started, and the client ran on the second node.

The call to set the value of weather was made on the bilbo node; we can swap back to gandalf and check the value of the weather.

```
(gandalf@localhost) 2> kvs:lookup(weather).
{ok,fine}
```

rpc:call(Node, Mod, Func, [Arg1, Arg2, ..., ArgN]) performs a *remote procedure call* on Node. The function to be called is Mod:Func(Arg1, Arg2, ..., ArgN).

As we can see, the program works as in the nondistributed Erlang case; now the only difference is that the client is running on one node and the server is running on a different node.

The next step is to run the client and the server on different machines.

Stage 3: Client and Server on Different Machines on the Same LAN

We're going to use two nodes. The first node is called gandalf on doris.myerl.example.com, and the second is called bilbo on george.myerl.example.com. Before we do this, we start two terminals using something like ssh or vnc on the two different machines. We'll call these two windows *doris* and *george*. Once we've done this, we can easily enter commands on both machines.

Step 1 is to start an Erlang node on doris.

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1> kvs:start().
true
```

Step 2 is to start an Erlang node on george and send some commands to gandalf.

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1> rpc:call(gandalf@doris.myerl.example.com,
    kvs,store,[weather,cold]).
true
(bilbo@george.myerl.example.com) 2> rpc:call(gandalf@doris.myerl.example.com,
    kvs,lookup,[weather]).
{ok,cold}
```

Things behave exactly as in the case with two different nodes on the same machine.

Now for this to work, things are slightly more complicated than in the case where we ran two nodes on the same computer. We have to take four steps.

1. Start Erlang with the -name parameter. When we have two nodes on the same machine, we use "short" names (as indicated by the -sname flag), but if they are on different networks, we use -name.

   We can also use -sname on two different machines when they are on the same subnet. Using -sname is also the only method that will work if no DNS service is available.

2. Ensure that both nodes have the same *cookie*. This is why both nodes were started with the command-line argument -setcookie abc. We'll talk more about cookies later in this chapter in Section 14.5, *The Cookie Protection System*, on page 222. *Note:* When we ran two nodes on the *same* machine, both nodes could access the same cookie file, $HOME/.erlang.cookie, which is why we didn't have to add the cookie to the Erlang command line.

3. Make sure the fully qualified hostnames of the nodes concerned are resolvable by DNS. In my case, the domain name myerl.example.com is purely local to my home network and is resolved locally by adding an entry to /etc/hosts.

4. Make sure that both systems have the same version of the code and the same version of Erlang. If you don't do this, you might get serious and mysterious errors. The easiest way to avoid problems is to have the same versions of Erlang running everywhere. Different versions of Erlang can run together, but there is no guarantee that this will work, so it's a good idea to check. In our case, the same version of the code for kvs has to be available on both systems. There are several ways of doing this.

   - In my setup at home, I have two physically separated computers with no shared file systems; here I physically copy kvs.erl to both machines and compile it before starting the programs.

   - On my work computer we use workstations with a shared NFS disk. Here I merely start Erlang in the shared directory from two different workstations.

   - Configure the code server to do this. I won't describe how to do this here. Take a look at the manual page for the module erl_prim_loader.

- Use the shell command nl(Mod). This loads the module Mod on all connected nodes.

    Note: For this to work, you have to make sure that all the nodes are connected. Nodes become connected when they first try to access each other. This happens the first time you evaluate any expression involving a remote node. The easiest way to do this is to evaluate net_adm:ping(Node) (see the manual page for net_adm for more details).

Success! We're running on two servers, on the same local area network. The next step is to move these onto two computers connected through the Internet.

Stage 4: Client and Server on Different Hosts in the Internet

In principle, this is the same as in stage 3, but now we have to be much more concerned with security. When we run two nodes on the same LAN, we probably don't have to worry too much about security. In most organizations, the LAN is isolated from the Internet by a firewall. Behind the firewall we are free to allocate IP addresses in a haphazard manner and generally misconfigure our machines.

When we connect several machines in an Erlang cluster on the Internet, we can expect to run into problems with firewalls that do not permit incoming connections. We will have to correctly configure our firewalls to accept incoming connections. There is no way to do this in a generic manner, since every firewall is different.

To prepare your system for distributed Erlang, you will have to take the following steps:

1. Make sure that port 4369 is open for both TCP and UDP traffic. This port is used by a program called *epmd* (short for the Erlang Port Mapper Daemon).

2. Choose a port or range of ports to be used for distributed Erlang, and make sure these ports are open. If these ports are Min and Max (use Min = Max if you want to use only one port), then start Erlang with the following command:

```
$ erl -name ... -setcookie ... -kernel inet_dist_listen_min Min \
                                inet_dist_listen_max Max
```

We've now seen how to run programs on sets of Erlang nodes and how to run them on the same local area network or over the Internet. Next we'll look at primitives that deal with nodes.

14.4 Libraries and BIFS for Distributed Programming

When we write distributed programs, we very rarely start from scratch. In the standard libraries, there are a number of modules that can be used to write distributed programs. These modules are written using the distribution BIFs, but they hide a lot of the complexity from the programmer.

Two modules in the standard distribution cover most needs.

- rpc provides a number of remote procedure call services.

- global has functions for the registration of names and locks in a distributed system and for the maintenance of a fully connected network.

The single most useful function in the module rpc is the following:

call(Node, Mod, Function, Args) -> Result | {badrpc, Reason}
>   This evaluates apply(Mod, Function, Args) on Node and returns the result Result or {badrpc, Reason} if the call fails.

The primitives that are used for writing distributed programs are as follows (for a fuller description of these BIFs, see the manual page for the erlang module[1]):

-spec spawn(Node, Fun) -> Pid
>   This works exactly like spawn(Fun), but the new process is spawned on Node.

-spec spawn(Node, Mod, Func, ArgList) -> Pid
>   This works exactly like spawn(Mod, Func, ArgList), but the new process is spawned on Node. spawn(Mod, Func, Args) creates a new process that evaluates apply(Mod, Func, Args). It returns the PID of the new process.
>
>   *Note*: This form of spawn is more robust than spawn(Node, Fun). spawn(Node, Fun) can break when the distributed nodes are not running exactly the same version of a particular module.

-spec spawn_link(Node, Fun) -> Pid
>   This works exactly like spawn_link(Fun), but the new process is spawned on Node.

-spec spawn_link(Node, Mod, Func, ArgList) -> Pid
>   This works like spawn(Node, Mod, Func, ArgList), but the new process is linked to the current process.

-spec disconnect_node(Node) -> bool() | ignored
>   This forcibly disconnects a node.

----

1.   http://www.erlang.org/doc/man/erlang.html

-spec monitor_node(Node, Flag) -> true

> If Flag is true, monitoring is turned on; if Flag is false, monitoring is turned off. If monitoring has been turned on, then the process that evaluated this BIF will be sent {nodeup, Node} and {nodedown, Node} messages if Node joins or leaves the set of connected Erlang nodes.

-spec node() -> Node

> This returns the name of the local node. nonode@nohost is returned if the node is not distributed.

-spec node(Arg) -> Node

> This returns the node where Arg is located. Arg can be a PID, a reference, or a port. If the local node is not distributed, nonode@nohost is returned.

-spec nodes() -> [Node]

> This returns a list of all other nodes in the network to which we are connected.

-spec is_alive() -> bool()

> This returns true if the local node is alive and can be part of a distributed system. Otherwise, it returns false.

In addition, send can be used to send messages to a locally registered process in a set of distributed Erlang nodes. The following syntax:

```
{RegName, Node} ! Msg
```

sends the message Msg to the registered process RegName on the node Node.

An Example of Remote Spawning

As a simple example, we can show how to spawn a process on a remote node. We'll start with the following program:

dist_demo.erl
```erlang
-module(dist_demo).

-export([rpc/4, start/1]).

start(Node) ->
    spawn(Node, fun() -> loop() end).

rpc(Pid, M, F, A) ->
    Pid ! {rpc, self(), M, F, A},
    receive
        {Pid, Response} ->
            Response
    end.
```

```
loop() ->
    receive
        {rpc, Pid, M, F, A} ->
            Pid ! {self(), (catch apply(M, F, A))},
            loop()
    end.
```

Then we start two nodes; both nodes have to be able to load this code. If both nodes are on the same host, then this is not a problem. We merely start two Erlang nodes from the same directory.

If the nodes are on two physically separated nodes with different file systems, then the program must be copied to all nodes and compiled before starting both the nodes (alternatively, the .beam file can be copied to all nodes). In the example, I'll assume we've done this.

On the host doris, we start a node named gandalf.

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.myerl.example.com) 1>
```

And on the host george, we start a node named bilbo, remembering to use the same cookie.

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.myerl.example.com) 1>
```

Now (on bilbo), we can spawn a process on the remote node (gandalf).

```
(bilbo@george.myerl.example.com) 1> Pid =
          dist_demo:start('gandalf@doris.myerl.example.com').
<5094.40.0>
```

Pid is now a process identifier of the process *on the remote node*, and we can call dist_demo:rpc/4 to perform a remote procedure call on the remote node.

```
(bilbo@george.myerl.example.com) 2> dist_demo:rpc(Pid, erlang, node, []).
'gandalf@doris.myerl.example.com'
```

This evaluates erlang:node() *on the remote node* and returns the value.

The File Server Revisited

In *The File Server Process,* on page 15, we built a simple file server with the promise that we would return to it later. Well, now is later. The previous section in this chapter showed how to set up a simple remote procedure call server, which we can use to transfer files between two Erlang nodes.

The following continues the example of the previous section:

```
(bilbo@george.myerl.example.com) 1> Pid =
    dist_demo:start('gandalf@doris.myerl.example.com').
<6790.42.0>
(bilbo@george.myerl.example.com) 2>
    dist_demo:rpc(Pid, file, get_cwd, []).
{ok,"/home/joe/projects/book/jaerlang2/Book/code"}
(bilbo@george.myerl.example.com) 3>
    dist_demo:rpc(Pid, file, list_dir, ["."]).
{ok,["adapter_db1.erl","processes.erl",
    "counter.beam","attrs.erl","lib_find.erl",...]}
(bilbo@george.myerl.example.com) 4>
    dist_demo:rpc(Pid, file, read_file, ["dist_demo.erl"]).
{ok,<<"-module(dist_demo).\n-export([rpc/4, start/1]).\n\n...>>}
```

On gandalf I started a distributed Erlang node in the code directory where I store the code examples for this book. On bilbo I'm making requests that result in remote procedure calls to the standard libraries on gandalf. I'm using three functions in the file module to access the file system on gandalf. file:get_cwd() returns the current working directory of the file server, file:list_dir(Dir) returns a list of the files in Dir, and file:read_file(File) reads the file File.

If you reflect for a moment, you'll realize that what we've just done is pretty amazing. We've made a file server without writing any code; we've just reused the library code in the module file and made it available through a simple remote procedure call interface.

Implementing a File Transfer Program

A few years ago I had to transfer a number of files between two networked machines with different operating systems. My first thought was to use FTP, but I needed an FTP server on one machine and an FTP client on the other. I couldn't find an FTP server for my server machine, and I didn't have root privileges on the server machine to install an FTP server. But I did have distributed Erlang running on both machines.

I then used exactly the technique I described here. It turned out to be quicker to write my own file server than to search for and install an FTP server.

If you're interested, I blogged[a] about it at the time.

_____

a.   http://armstrongonsoftware.blogspot.com/2006/09/why-i-often-implement-things-from.html

14.5 The Cookie Protection System

Access to a single node or set of nodes is secured by a cookie system. Each node has a single cookie, and this cookie must be the same as the cookies of any nodes to which the node talks. To ensure this, all nodes in a distributed

Erlang system must have been started with the same magic cookie or have their cookie changed to the same value by evaluating erlang:set_cookie.

The set of connected nodes having the same cookie defines an Erlang cluster.

For two distributed Erlang nodes to communicate, they must have the same *magic cookie.* We can set the cookie in three ways.

Important: The cookie protection system was designed for building distributed systems that run on a local area network (LAN) where the LAN itself was protected from the Internet by a firewall. Distributed Erlang applications running across the Internet should first set up secure connections between hosts and then use the cookie protection system.

- *Method 1*: Store the same cookie in the file $HOME/.erlang.cookie. This file contains a random string and is automatically created the first time Erlang is run on your machine.

  This file can be copied to all machines that we want to participate in a distributed Erlang session. Alternatively, we can explicitly set the value. For example, on a Linux system, we could give the following commands:

  ```
  $ cd
  $ cat > .erlang.cookie
  AFRTY12ESS3412735ASDF12378
  $ chmod 400 .erlang.cookie
  ```

  The chmod makes the .erlang.cookie file accessible only by the owner of the file.

- *Method 2*: When Erlang is started, we can use the command-line argument -setcookie C to set the magic cookie to C. Here's an example:

  ```
  $ erl -setcookie AFRTY12ESS3412735ASDF12378 ...
  ```

- *Method 3*: The BIF erlang:set_cookie(node(), C) sets the cookie of the local node to the atom C.

Note: If your environment is insecure, then method 1 or 3 is better than method 2 since on a Unix system anybody can discover your cookie using the ps command. Method 2 is useful only for testing.

In case you're wondering, cookies are never sent across the network in the clear. Cookies are used only for the initial authentication of a session. Distributed Erlang sessions are not encrypted but can be set up to run over encrypted channels. (Google the Erlang mailing list for up-to-date information on this.)

Up to now we have looked at how to write distributed programs using Erlang nodes and the distribution primitives. As an alternative, we can write distributed programs on top of a raw socket interface.

14.6 Socket-Based Distribution

In this section, we will write a simple program using socket-based distribution. As we have seen, distributed Erlang is fine for writing cluster applications where you can trust everybody involved but is less suitable in an open environment where not everyone can be trusted.

The main problem with distributed Erlang is that the client can decide to spawn *any* process on the server machine. So, to destroy your system, all you'd have to do is evaluate the following:

```
rpc:multicall(nodes(), os, cmd, ["cd /; rm -rf *"])
```

Distributed Erlang is useful in the situation where you own all the machines and want to control all of them from a single machine. But this model of computation is not suited to the situation where different people own the individual machines and want to control exactly which software can be executed on their machines.

In these circumstances, we will use a restricted form of spawn where the owner of a particular machine has explicit control over what gets run on their machines.

Controlling Processes with lib_chan

lib_chan is a module that allows a user to explicitly control which processes are spawned on their machines. The implementation of lib_chan is rather complex, so I've taken it out of the normal chapter flow; you can find it in Appendix 2, *A Socket Application*, on page 477. The interface is as follows:

-spec start_server() -> true

>   This starts a server on the local host. The behavior of the server is determined by the file $HOME/.erlang_config/lib_chan.conf.

-spec start_server(Conf) -> true

>   This starts a server on the local host. The behavior of the server is determined by the file Conf, which contains a list of tuples of the following form:

>   {port, NNNN}

>>   This starts listening to port number NNNN.

{service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgsS}

> This defines a service S protected by password P. If the service is started, then a process is created by spawning SomeMod:SomeFunc(MM, ArgsC, SomeArgsS) to handle messages from the client. Here MM is the PID of a proxy process that can be used to send messages to the client, and the argument ArgsC comes from the client connect call.

-spec connect(Host, Port, S, P, ArgsC) -> {ok, Pid} | {error, Why}

> Try to open the port Port on the host Host, and then try to activate the service S, which is protected with the password P. If the password is correct, {ok, Pid} will be returned, where Pid will be the process identifier of a proxy process that can be used to send messages to the server.

When a connection is established by the client calling connect/5, two proxy processes are spawned: one on the client side and the other on the server side. These proxy processes handle the conversion of Erlang messages to TCP packet data, trapping exits from the controlling processes, and socket closure.

This explanation might look complicated, but it will become a lot clearer when we use it. The following is a complete example of how to use lib_chan together with the kvs service that we described earlier.

The Server Code

First we write a configuration file.

```
{port, 1234}.
{service, nameServer, password, "ABXy45",
         mfa, mod_name_server, start_me_up, notUsed}.
```

This means we are going to offer a service called nameServer on port 1234 of our machine. The service is protected by the password ABXy45.

When a connection is created by the client calling the following:

```
connect(Host, 1234, nameServer, "ABXy45", nil)
```

the server will spawn mod_name_server:start_me_up(MM, nil, notUsed). MM is the PID of a proxy process that is used to talk to the client.

Important: At this stage, you should study the previous line of code and make sure you see where the arguments in the call come from.

- mod_name_server, start_me_up, and notUsed come from the configuration file.
- nil is the last argument in the connect call.

mod_name_server is as follows:

```
socket_dist/mod_name_server.erl
-module(mod_name_server).
-export([start_me_up/3]).

start_me_up(MM, _ArgsC, _ArgS) ->
    loop(MM).

loop(MM) ->
    receive
        {chan, MM, {store, K, V}} ->
            kvs:store(K, V),
            loop(MM);
        {chan, MM, {lookup, K}} ->
            MM ! {send, kvs:lookup(K)},
            loop(MM);
        {chan_closed, MM} ->
            true
    end.
```

mod_name_server follows this protocol:

- If the client sends the server a message {send, X}, it will appear in mod_name_server as a message of the form {chan, MM, X} (MM is the PID of the server proxy process).

- If the client terminates or the socket used in communication closes for any reason, then a message of the form {chan_closed, MM} will be received by the server.

- If the server wants to send a message X to the client, it does so by calling MM ! {send, X}.

- If the server wants to explicitly close the connection, it can do so by evaluating MM ! close.

This protocol is the middle-man protocol that is obeyed by both the client code and the server code. The socket middle-man code is explained in more detail in *lib_chan_mm: The Middle Man,* on page 480.

To test this code, we will first make sure that everything works on one machine.

Now we can start the name server (and the module kvs).

```
1> kvs:start().
true
2> lib_chan:start_server().
Starting a port server on 1234...
true
```

Now we can start a second Erlang session and test this from any client.

```
1> {ok, Pid} = lib_chan:connect("localhost",1234,nameServer,"ABXy45","").
{ok, <0.43.0>}
2> lib_chan:cast(Pid, {store, joe, "writing a book"}).
{send,{store,joe,"writing a book"}}
3> lib_chan:rpc(Pid, {lookup, joe}).
{ok,"writing a book"}
4> lib_chan:rpc(Pid, {lookup, jim}).
undefined
```

Having tested that this works on one machine, we go through the same steps we described earlier and perform similar tests on two physically separated machines.

Note that in this case, it is the owner of the remote machine who decides the contents of the configuration file. The configuration file specifies which applications are permitted on this machine and which port is to be used to communicate with these applications.

We're now at the point where we can write distributed programs. A whole new world opens up. If writing sequential programs is fun, then writing distributed program is fun squared or fun cubed. I really recommend you do the following YAFS exercise; this basic code structure is central to many applications.

We have now covered sequential, concurrent, and distributed programming. In the next part of the book, we'll look at how to interface foreign language code, and we'll look at some of the major Erlang libraries and how to debug code. Then we'll see how complex Erlang systems can be built using the OTP structuring principles and libraries.

Exercises

1. Start two nodes on the same host. Look up the manual page for the rpc module. Perform some remote procedure calls on the two nodes.

2. Repeat the previous exercise, only with the two nodes on the same LAN.

3. Repeat the previous exercise, only with the two nodes on different networks.

4. Write YAFS (Yet Another File Server) using the libraries in lib_chan. You will learn a lot by doing this. Add "bells and whistles" to your file server.

Part IV

Programming Libraries and Frameworks

This part of the book covers the major libraries for programming with files, sockets, and databases. We also cover debugging techniques and the OTP framework.

Interfacing Techniques

Building systems often involves interfacing applications written in different programming languages with our system. We might use C for efficiency or writing low-level hardware drivers, or we might want to integrate a library written in Java or Ruby or some other programming language. We can interface foreign language programs to Erlang in a number of ways.

- By running the programs *outside* the Erlang virtual machine in an external operating system process. This is the *safe* way of doing things. If the foreign language code is incorrect, it will not crash the Erlang system. Erlang controls the external process through a device called a *port* and communicates with the external process through a byte-oriented communication channel. Erlang is responsible for starting and stopping the external program and can monitor and restart it if it crashes. The external process is called a *port process* since it is controlled through an Erlang port.

- By running an OS command from within Erlang and capturing the result.

- By running the foreign language code *inside* the Erlang virtual machine. This involves linking the code with the code for the Erlang virtual machine. This is the *unsafe* way of doing things. Errors in the foreign language code might crash the Erlang system. Although it is unsafe, it is useful since it is more efficient than using an external process.

  Linking code into the Erlang kernel can be used only for languages like C that produce native object code and can't be used with languages like Java that have their own virtual machines.

In this chapter we'll look at interfacing Erlang using ports and OS commands. In addition, there are a number of advanced interfacing techniques using linked-in drivers, natively implemented functions (NIFs), and C-nodes. The advanced techniques are not covered in the book, but at the end of the

chapter, there is a short overview of these techniques and some pointers to reference material.

15.1 How Erlang Communicates with External Programs

Erlang communicates with external programs through objects called *ports*. If we send a message to a port, the message will be sent to the external program connected to the port. Messages from the external program will appear as Erlang messages that come from the ports.

As far as the programmer is concerned, the port behaves just like an Erlang process. You can send messages to it, you can register it (just like a process), and so on. If the external program crashes, then an exit signal will be sent to the connected process, and if the connected process dies, then the external program will be killed.

Note the difference between using a port to communicate with an external process and a socket. If you use a port, the port will behave like an Erlang process, so you can link to it, send messages to it from a remote distributed Erlang node, and so on. If you use a socket, it will not behave like a process.

The process that creates a port is called the *connected process* for that port. The connected process has a special significance: all messages to the port must be tagged with the PID of the connected process, and all messages from the external program are sent to the connected processes.

We can see the relationship between a connected process (C), a port (P), and an external operating system process in the following figure:
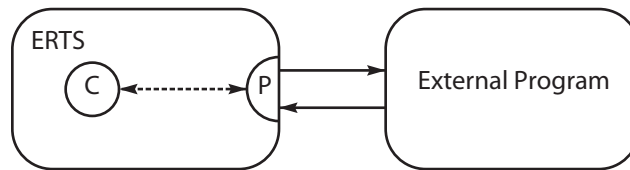
ERTS = Erlang runtime system
C = An Erlang process that is connected to the port
P = A port

To create a port, we call open_port, which is specified as follows:

```
-spec open_port(PortName, [Opt]) -> Port
```
   PortName is one of the following:

{spawn, Command}
: Start an external program. Command is the name of an external program. Command runs outside the Erlang workspace unless a linked-in driver with the name Command is found.

{fd, In, Out}
: Allow an Erlang process to access any currently opened file descriptors used by Erlang. The file descriptor In can be used for standard input, and the file descriptor Out can be used for standard output.

Opt is one of the following:

{packet, N}
: Packets are preceded by an N (1, 2, or 4) byte length count.

stream
: Messages are sent without packet lengths. The application must know how to handle these packets.

{line, Max}
: Deliver messages on a one-per line basis. If the line is more than Max bytes, then it is split at Max bytes.

{cd, Dir}
: Valid only for the {spawn, Command} option. The external program starts in Dir.

{env, Env}
: Valid only for the {spawn, Command} option. The environment of the external program is extended with the environment variables in the list Env. Env is a list of {VarName, Value} pairs, where VarName and Value are strings.

This is not a complete list of the arguments to open_port. You can find the precise details of the arguments in the manual page for the module erlang.

The following messages can be sent to a port; note that in all of these messages, PidC is the PID of the connected process.

Port ! {PidC, {command, Data}}
: Send Data (an I/O list) to the port.

Port ! {PidC, {connect, Pid1}}
: Change the PID of the connected process from PidC to Pid1.

Port ! {PidC, close}
: Close the port.

The connected process can receive a message from the external program by writing this:

```
receive
    {Port, {data, Data}} ->
        ... Data comes from the external process ...
```

In the following sections, we'll interface Erlang to a simple C program. The C program is deliberately short so as not to distract from the details of how we do the interfacing.

15.2 Interfacing an External C Program with a Port

We'll start with some simple C code. example1.c contains two functions. The first function computes the sum of two integers, and the second computes twice its argument.

```
ports/example1.c
int sum(int x, int y){
  return x+y;
}

int twice(int x){
  return 2*x;
}
```

Our final goal is to call these routines from Erlang. We'd like to be able to call them as follows:

```
X1 = example1:sum(12,23),
Y1 = example1:twice(10),
```

As far as the user is concerned, example1 is an Erlang module, and therefore all details of the interface to the C program should be hidden inside the module example1.

To implement this, we need to turn function calls such as sum(12,23) and twice(10) into sequences of bytes that we send to the external program by means of the port. The port adds a length count to the byte sequence and sends the result to the external program. When the external program replies, the port receives the reply and sends the result to the connected process for the port.

The protocol we use is very simple.

- All packets start with a 2-byte length code (Len) followed by Len bytes of data. This header is automatically added by the port when we open it with argument {packet,2}.

- We encode the call sum(N, M) as the byte sequence [1,N,M].

- We encode the call twice(N) as the byte sequence [2,N].

- Arguments and return values are assumed to be a single byte long.

Both the external C program and the Erlang program must follow this protocol. The following figure illustrates what happens after we have called example1:sum(12,23). It shows how the port is wired up to the external C program.
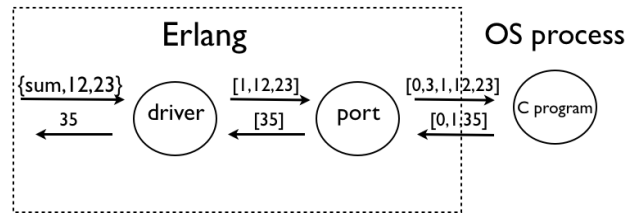
What happens is as follows:

1. The driver encodes the sum(12,23) function call into the byte sequence [1,12,23] and sends the {self(), {command, [1,12,23]}} message to the port.

2. The port driver adds a 2-byte length header to the message and sends the byte sequence 0,3,1,12,23 to the external program.

3. The external program reads these five bytes from standard input, calls the sum function, and then writes the byte sequence 0,1,35 to standard output.

    The first two bytes contains the packet length. This is followed by the result, 35, which is 1-byte long.

4. The port driver removes the length header and sends a {Port, {data, [35]}} message to the connected process.

5. The connected process decodes this message and returns the result to the calling program.

We now have to write programs on both sides of the interface that follow this protocol.

The C Program

The C program has three files.

- example1.c: Contains the functions that we want to call (we saw this earlier)

- example1_driver.c: Manages the byte stream protocol and calls the routines in example1.c

- erl_comm.c: Has routines for reading and writing memory buffers

example1_driver.c

This code has a loop that reads commands from standard input, calls the application routines, and writes the results to standard output. Note that if you want to debug this program, you can write to stderr; there is a commented-out fprintf statement in the code that shows how to do this.

```
ports/example1_driver.c
#include <stdio.h>
#include <stdlib.h>
typedef unsigned char byte;

int read_cmd(byte *buff);
int write_cmd(byte *buff, int len);
int sum(int x, int y);
int twice(int x);

int main() {
  int fn, arg1, arg2, result;
  byte buff[100];

  while (read_cmd(buff) > 0) {
    fn = buff[0];

    if (fn == 1) {
      arg1 = buff[1];
      arg2 = buff[2];

      /* debug -- you can print to stderr to debug
         fprintf(stderr,"calling sum %i %i\n",arg1,arg2); */
      result = sum(arg1, arg2);
    } else if (fn == 2) {
      arg1 = buff[1];
      result = twice(arg1);
    } else {
      /* just exit on unknown function */
      exit(EXIT_FAILURE);
    }
    buff[0] = result;
    write_cmd(buff, 1);
  }
}
```

erl_comm.c

Finally, here's the code to read and write data to and from standard input and output. The code is written to allow for possible fragmentation of the data.

ports/erl_comm.c

```c
/* erl_comm.c */
#include <unistd.h>
typedef unsigned char byte;

int read_cmd(byte *buf);
int write_cmd(byte *buf, int len);
int read_exact(byte *buf, int len);
int write_exact(byte *buf, int len);

int read_cmd(byte *buf)
{
  int len;
  if (read_exact(buf, 2) != 2)
    return(-1);
  len = (buf[0] << 8) | buf[1];
  return read_exact(buf, len);
}
int write_cmd(byte *buf, int len)
{
  byte li;
  li = (len >> 8) & 0xff;
  write_exact(&li, 1);
  li = len & 0xff;
  write_exact(&li, 1);
  return write_exact(buf, len);
}
int read_exact(byte *buf, int len)
{
  int i, got=0;
  do {
    if ((i = read(0, buf+got, len-got)) <= 0)
      return(i);
    got += i;
  } while (got<len);
  return(len);
}
int write_exact(byte *buf, int len)
{
  int i, wrote = 0;
  do {
    if ((i = write(1, buf+wrote, len-wrote)) <= 0)
      return (i);
    wrote += i;
  } while (wrote<len);
  return (len);
}
```

This code is specialized for handling data with a 2-byte length header, so it matches up with the {packet, 2} option given to the port driver program.

The Erlang Program

The Erlang side of the port is driven by the following program:

```erlang
ports/example1.erl
-module(example1).
-export([start/0, stop/0]).
-export([twice/1, sum/2]).

start() ->
    register(example1,
             spawn(fun() ->
                       process_flag(trap_exit, true),
                       Port = open_port({spawn, "./example1"}, [{packet, 2}]),
                       loop(Port)
                   end)).

stop() ->
    ?MODULE ! stop.
twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}).
call_port(Msg) ->
    ?MODULE ! {call, self(), Msg},
    receive
        {?MODULE, Result} ->
            Result
    end.

loop(Port) ->
    receive
        {call, Caller, Msg} ->
            Port ! {self(), {command, encode(Msg)}},
            receive
                {Port, {data, Data}} ->
                    Caller ! {?MODULE, decode(Data)}
            end,
            loop(Port);
        stop ->
            Port ! {self(), close},
            receive
                {Port, closed} ->
                    exit(normal)
            end;
        {'EXIT', Port, Reason} ->
            exit({port_terminated, Reason})
    end.

encode({sum, X, Y}) -> [1, X, Y];
encode({twice, X})  -> [2, X].

decode([Int]) -> Int.
```

This code follows a fairly standard pattern. In start/0, we create a registered process (server) called example1. call_port/1 implements a remote procedure call toward the server. twice/1 and sum/2 are interface routines that must be exported and that make remote procedure calls to the server. In loop/1, we encode the requests to the external program and take care of the return values from the external program.

That completes the programs. All we now need is a makefile to build the programs.

Compiling and Linking the Port Program

This makefile compiles and links the port driver and linked-in driver programs described in this chapter together with all associated Erlang code. The makefile has been tested only on Mac OS X Mountain Lion and will need modifying for other operating systems. It also includes a small test program, which is run each time the code is rebuilt.

ports/Makefile.mac
```
.SUFFIXES: .erl .beam .yrl

.erl.beam:
        erlc -W $<

MODS = example1 example1_lid unit_test

all:    ${MODS:%=%.beam} example1 example1_drv.so
        @erl -noshell -s unit_test start
example1:  example1.c erl_comm.c example1_driver.c
        gcc -o example1 example1.c erl_comm.c example1_driver.c
example1_drv.so: example1_lid.c example1.c
        gcc -arch i386 -I /usr/local/lib/erlang/usr/include\
            -o example1_drv.so -fPIC -bundle -flat_namespace -undefined suppress\
            example1.c example1_lid.c
clean:
        rm example1 example1_drv.so *.beam
```

Running the Program

Now we can run the program.

```
1> example1:start().
true
2> example1:sum(45, 32).
77
4> example1:twice(10).
20
...
```

This completes our first example port program. The port protocol that the program implements is the principal way in which Erlang communicates with the external world.

Before passing to the next topic, note the following:

- The example program made no attempt to unify Erlang's and C's ideas of what an integer is. We just assumed that an integer in Erlang and C was a single byte and ignored all problems of precision and signedness. In a realistic application, we would have to think rather carefully about the exact types and precisions of the arguments concerned. This can in fact be rather difficult, because Erlang happily manages integers of an arbitrary size, whereas languages such as C have fixed ideas about the precision of integers and so on.

- We couldn't just run the Erlang functions without first having started the driver that was responsible for the interface (that is, some program had to evaluate example1:start() *before* we were able to run the program). We would like to be able to do this automatically when the system is started. This is perfectly possible but needs some knowledge of how the system starts and stops. We'll deal with this later in Section 23.7, *The Application, on page 403*.

15.3 Calling a Shell Script from Erlang

Suppose we want to call a shell script from Erlang. To do this, we can use the library function os:cmd(Str). This runs the command in the string Str and captures the result. Here's an example using the ifconfig command:

```
1> os:cmd("ifconfig").
"lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384\n\t...
```

The result will need parsing to extract the information we are interested in.

15.4 Advanced Interfacing Techniques

In addition to the techniques discussed earlier, there are a few additional techniques available for interfacing Erlang to external programs.

The techniques described next are being continually improved and tend to change with time more rapidly than Erlang itself. For this reason, they are not described here in detail. The descriptions have been moved into online archives so that they can be updated more quickly.

Linked-in Drivers

These programs obey the same protocol as the port drivers discussed earlier. The only difference is that the driver code is linked into the Erlang kernel and thus runs inside the Erlang OS main process. To make a linked-in driver, a small amount of code must be added to initialize the driver, and the driver must be compiled and linked with the Erlang VM.

git://github.com/erlang/linked_in_drivers.git has up-to-date examples of linked-in drivers and how to compile them for various operating systems.

NIFS

NIFs are *natively implemented functions*. These are functions that are written in C (or some language that compiles to native code) and that are linked into the Erlang VM. NIFs pass arguments directly onto the Erlang processes' stacks and heaps and have direct access to all the Erlang internal data structure.

Examples and up-to-date information about NIFS are available from git://github.com/erlang/nifs.git.

C-Nodes

C nodes are nodes implemented in C that obey the Erlang distribution protocol. A "real" distributed Erlang node can talk to a C-node and will think that the C-node is an Erlang node (provided it doesn't try to do anything fancy on the C-node like sending it Erlang code to execute).

C-nodes are described in the Interoperability tutorial at http://www.erlang.org/doc/tutorial/introduction.html.

So, now we know how to interface Erlang to the external world. In the next couple of chapters, we'll see how to access files and sockets from within Erlang.

Exercises

1. Download the code for a port driver given earlier and test it on your system.

2. Go to git://github.com/erlang/linked_in_drivers.git. Download the code for a linked-in driver and test it on your system. The tricky part of this is finding the correct commands to compile and link the code. If you fail this exercise, ask for help on the Erlang mailing list.

3. See whether you can find an operating system command to discover which CPU your computer has. If you can find such a command, write a function that returns your CPU type, using the function os:cmd to call the OS command.

Programming with Files

In this chapter, we'll look at some of the most commonly used functions for manipulating files. The standard Erlang release has a large number of functions for working with files. We're going to concentrate on the small fraction of these that I use to write most of my programs and that you'll use the most frequently as well. We'll also see a few examples of techniques for writing efficient file handling code. In addition, I'll briefly mention some of the more rarely used file operations so you'll know they exist. If you want more details of the rarely used techniques, consult the manual pages.

We'll concentrate on the following areas:

- Overview of the main modules used for manipulating files
- Different ways of reading a file
- Different ways of writing to a file
- Directory operations
- Finding information about a file

16.1 Modules for Manipulating Files

The functions for file manipulation are organized into four modules.

file   This has routines for opening, closing, reading, and writing files; listing directories; and so on. A short summary of some of the more frequently used functions in file is shown in Table 7, *Summary of file operations (in module file), on page 245*. For full details, consult the manual page for the file module.

filename This module has routines that manipulate filenames in a platform-independent manner, so you can run the same code on a number of different operating systems.

filelib This module is an extension to file, which contains a number of utilities for listing files, checking file types, and so on. Most of these are written using the functions in file.

io This module has routines that work on opened files. It contains routines for parsing data in a file and writing formatted data to a file.

16.2 Ways to Read a File

Let's look at some options when it comes to reading files. We'll start by writing five little programs that open a file and input the data in a number of ways.

The contents of a file are just a sequence of bytes. Whether they mean anything depends upon the interpretation of these bytes.

To demonstrate this, we'll use the same input file for all our examples. It actually contains a sequence of Erlang terms. Depending upon how we open and read the file, we can interpret the contents as a sequence of Erlang terms, as a sequence of text lines, or as raw chunks of binary data with no particular interpretation.

Here's the raw data in the file:

```
data1.dat
{person, "joe", "armstrong",
        [{occupation, programmer},
         {favoriteLanguage, erlang}]}.

{cat, {name, "zorro"},
      {owner, "joe"}}.
```

Now we'll read parts of this file in a number of ways.

Reading All the Terms in the File

data1.dat contains a sequence of Erlang terms; we can read all of these by calling file:consult as follows:

```
1> file:consult("data1.dat").
{ok,[{person,"joe",
             "armstrong",
      [{occupation,programmer},{favoriteLanguage,erlang}]},
     {cat,{name,"zorro"},{owner,"joe"}}]}
```

file:consult(File) assumes that File contains a sequence of Erlang terms. It returns {ok, [Term]} if it can read all the terms in the file; otherwise, it returns {error, Reason}.

Function	Description
change_group	Change group of a file.
change_owner	Change owner of a file.
change_time	Change the modification or last access time of a file.
close	Close a file.
consult	Read Erlang terms from a file.
copy	Copy file contents.
del_dir	Delete a directory.
delete	Delete a file.
eval	Evaluate Erlang expressions in a file.
format_error	Return a descriptive string for an error reason.
get_cwd	Get the current working directory.
list_dir	List files in a directory.
make_dir	Make a directory.
make_link	Make a hard link to a file.
make_symlink	Make a symbolic link to a file or directory.
open	Open a file.
position	Set the position in a file.
pread	Read from a file at a certain position.
pwrite	Write to a file at a certain position.
read	Read from a file.
read_file	Read an entire file.
read_file_info	Get information about a file.
read_link	See what a link is pointing to.
read_link_info	Get information about a link or file.
rename	Rename a file.
script	Evaluate and return the value of Erlang expressions in a file.
set_cwd	Set the current working directory.
sync	Synchronize the in-memory state of a file with that on the physical medium.
truncate	Truncate a file.
write	Write to a file.
write_file	Write an entire file.
write_file_info	Change information about a file.

Table 7—Summary of file operations (in module file)

Reading the Terms in the File One at a Time

If we want to read the terms in a file one at a time, we first open the file with file:open, then we read the individual terms with io:read until we reach the end of file, and finally we close the file with file:close.

Here's a shell session that shows what happens when we read the terms in a file one at a time:

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.36.0>}
2> io:read(S, '').
{ok,{person,"joe",
     "armstrong",
     [{occupation,programmer},{favoriteLanguage,erlang}]}}
3> io:read(S, '').
{ok,{cat,{name,"zorro"},{owner,"joe"}}}
4> io:read(S, '').
eof
5> file:close(S).
ok
```

The functions we've used here are as follows:

-spec file:open(File, read) -> {ok, IoDevice} | {error, Why}

> Tries to open File for reading. It returns {ok, IoDevice} if it can open the file; otherwise, it returns {error, Reason}. IoDevice is an I/O device that is used to access the file.

-spec io:read(IoDevice, Prompt) -> {ok, Term} | {error,Why} | eof

> Reads an Erlang term Term from IoDevice. Prompt is ignored if IoDevice represents an opened file. Prompt is used only to provide a prompt if we use io:read to read from standard input.

-spec file:close(IoDevice) -> ok | {error, Why}

> Closes IoDevice.

Using these routines we could have implemented file:consult, which we used in the previous section. Here's how file:consult might have been defined:

lib_misc.erl
```
consult(File) ->
    case file:open(File, read) of
        {ok, S} ->
            Val = consult1(S),
            file:close(S),
            {ok, Val};
        {error, Why} ->
            {error, Why}
    end.
```

```
consult1(S) ->
    case io:read(S, '') of
        {ok, Term} -> [Term|consult1(S)];
        eof        -> [];
        Error      -> Error
    end.
```

This is *not* how file:consult is actually defined. The standard libraries use an improved version with better error reporting.

Now is a good time to look at the version included in the standard libraries. If you've understood the earlier version, then you should find it easy to follow the code in the libraries. There's only one problem: we need to find the source of the file.erl code. To find this, we use the function code:which, which can locate the object code for any module that has been loaded.

```
1> code:which(file).
"/usr/local/lib/erlang/lib/kernel-2.16.1/ebin/file.beam"
```

In the standard release, each library has two subdirectories. One, called src, contains the source code. The other, called ebin, contains compiled Erlang code. So, the source code for file.erl should be in the following directory:

/usr/local/lib/erlang/lib/kernel-2.16.1/src/file.erl

When all else fails and the manual pages don't provide the answers to all your questions about the code, then a quick peek at the source code can often reveal the answer. Now I know this shouldn't happen, but we're all human, and sometimes the documentation doesn't answer all your questions.

Reading the Lines in a File One at a Time

If we change io:read to io:get_line, we can read the lines in the file one at a time. io:get_line reads characters until it encounters a line-feed character or end-of-file. Here's an example:

```
1> {ok, S} = file:open("data1.dat", read).
{ok,<0.43.0>}
2> io:get_line(S, '').
"{person, \"joe\", \"armstrong\",\n"
3> io:get_line(S, '').
"\t[{occupation, programmer},\n"
4> io:get_line(S, '').
"\t {favoriteLanguage, erlang}]}.\n"
5> io:get_line(S, '').
"\n"
6> io:get_line(S, '').
"{cat, {name, \"zorro\"},\n"
7> io:get_line(S, '').
```

```
"      {owner, \"joe\"}}.\n"
8> io:get_line(S, '').
eof
9> file:close(S).
ok
```

Reading the Entire File into a Binary

You can use file:read_file(File) to read an entire file into a binary using a single atomic operation.

```
1> file:read_file("data1.dat").
{ok,<<"{person, \"joe\", \"armstrong\""...>>}
```

file:read_file(File) returns {ok, Bin} if it succeeds and returns {error, Why} otherwise. This is by far the most efficient way of reading files, and it's a method that I use a lot. For most operations, I read the entire file into memory in one operation, manipulate the contents, and store the file in a single operation (using file:write_file). We'll give an example of this later.

Reading a File with Random Access

If the file we want to read is very large or if it contains binary data in some externally defined format, then we can open the file in raw mode and read any portion of it using file:pread.

Here's an example:

```
1> {ok, S} = file:open("data1.dat", [read,binary,raw]).
{ok,{file_descriptor,prim_file,{#Port<0.106>,5}}}
2> file:pread(S, 22, 46).
{ok,<<"rong\",\n\t[{occupation, programmer},\n\t {favorite">>}
3> file:pread(S, 1, 10).
{ok,<<"person, \"j">>}
4> file:pread(S, 2, 10).
{ok,<<"erson, \"jo">>}
5> file:close(S).
ok
```

file:pread(IoDevice, Start, Len) reads exactly Len bytes from IoDevice starting at byte Start (the bytes in the file are numbered so that the first byte in the file is at position 0). It returns {ok, Bin} or {error, Why}.

Finally, we'll use the routines for random file access to write a utility routine that we'll need in the next chapter. In Section 17.6, *A SHOUTcast Server, on page 281*, we'll develop a simple SHOUTcast server (this is a server for so-called streaming media, in this case for streaming MP3). Part of this server needs to be able to find the artist and track names that are embedded in an MP3 file. We will do this in the next section.

Reading MP3 Metadata

MP3 is a binary format used for storing compressed audio data. MP3 files do not themselves contain information about the content of the file, so, for example, in an MP3 file that contains music, the name of the artist who recorded the music is not contained in the audio data. This data (the track name, artist name, and so on) is stored inside the MP3 files in a tagged block format known as ID3. ID3 tags were invented by a programmer called Eric Kemp to store metadata describing the content of an audio file. There are actually a number of ID3 formats, but for our purposes, we'll write code to access only the two simplest forms of ID3 tag, namely, the ID3v1 and ID3v1.1 tags.

The ID3v1 tag has a simple structure—the last 128 bytes of the file contain a fixed-length tag. The first 3 bytes contain the ASCII characters TAG, followed by a number of fixed-length fields. The entire 128 bytes is packed as follows:

Length	Contents
3	Header containing the characters *TAG*
30	Title
30	Artist
30	Album
4	Year
30	Comment
1	Genre

In the ID3v1 tag there was no place to add a track number. A method for doing this was suggested by Michael Mutschler, in the ID3v1.1 format. The idea was to change the 30-byte comment field to the following:

Length	Contents
28	Comment
1	0 (a zero)
1	Track number

It's easy to write a program that tries to read the ID3v1 tags in an MP3 file and matches the fields using the binary bit-matching syntax. Here's the program:

```
id3_v1.erl
-module(id3_v1).
-import(lists, [filter/2, map/2, reverse/1]).
-export([test/0, dir/1, read_id3_tag/1]).
```

```erlang
test() -> dir("/home/joe/music_keep").

dir(Dir) ->
    Files = lib_find:files(Dir, "*.mp3", true),
    L1 = map(fun(I) ->
                     {I, (catch read_id3_tag(I))}
             end,  Files),
    %% L1 = [{File, Parse}] where Parse = error | [{Tag,Val}]
    %% we now have to remove all the entries from L where
    %% Parse = error. We can do this with a filter operation
    L2 = filter(fun({_,error}) -> false;
                   (_) -> true
                end, L1),
    lib_misc:dump("mp3data", L2).

read_id3_tag(File) ->
    case file:open(File, [read,binary,raw]) of
        {ok, S} ->
            Size = filelib:file_size(File),
            {ok, B2} = file:pread(S, Size-128, 128),
            Result = parse_v1_tag(B2),
            file:close(S),
            Result;
        _Error ->
            error
    end.
parse_v1_tag(<<$T,$A,$G,
              Title:30/binary, Artist:30/binary,
              Album:30/binary, _Year:4/binary,
              _Comment:28/binary, 0:8,Track:8,_Genre:8>>) ->
    {"ID3v1.1",
     [{track,Track}, {title,trim(Title)},
      {artist,trim(Artist)}, {album, trim(Album)}]};
parse_v1_tag(<<$T,$A,$G,
              Title:30/binary, Artist:30/binary,
              Album:30/binary, _Year:4/binary,
              _Comment:30/binary,_Genre:8>>) ->
    {"ID3v1",
     [{title,trim(Title)},
      {artist,trim(Artist)}, {album, trim(Album)}]};
parse_v1_tag(_) ->
    error.

trim(Bin) ->
    list_to_binary(trim_blanks(binary_to_list(Bin))).
trim_blanks(X) -> reverse(skip_blanks_and_zero(reverse(X))).

skip_blanks_and_zero([$\s|T]) -> skip_blanks_and_zero(T);
skip_blanks_and_zero([0|T])   -> skip_blanks_and_zero(T);
skip_blanks_and_zero(X)       -> X.
```

The main entry point to our program is id3_v1:dir(Dir). The first thing we do is find all our MP3 files by calling lib_find:find(Dir, "*.mp3", true) (shown later in Section 16.6, *A Find Utility*, on page 258), which recursively scans the directories under Dir looking for MP3 files.

Having found the file, we parse the tags by calling read_id3_tag. Parsing is greatly simplified because we can merely use the bit-matching syntax to do the parsing for us, and then we can trim the artist and track names by removing trailing whitespace and zero-padding characters, which delimit the character strings. Finally, we dump the results in a file for later use (lib_misc:dump is described in *Dumping to a File*, on page 349).

Most music files are tagged with ID3v1 tags, even if they also have ID3v2, v3, and v4 tags—the later tagging standards added a differently formatted tag to the beginning of the file (or more rarely in the middle of the file). Tagging programs often appear to add both ID3v1 tags and additional (and more difficult to read) tags at the start of the file. For our purposes, we'll be concerned only with files containing valid ID3v1 and ID3v1.1 tags.

Now that we know how to read a file, we can move to the different ways of writing to a file.

16.3 Ways to Write a File

Writing to a file involves pretty much the same operations as reading a file. Let's look at them.

Writing a List of Terms to a File

Suppose we want to create a file that we can read with file:consult. The standard libraries don't actually contain a function for this, so we'll write our own. Let's call this function unconsult.

lib_misc.erl
```
unconsult(File, L) ->
    {ok, S} = file:open(File, write),
    lists:foreach(fun(X) -> io:format(S, "~p.~n",[X]) end, L),
    file:close(S).
```

We can run this in the shell to create a file called test1.dat.

```
1> lib_misc:unconsult("test1.dat",
                      [{cats,["zorrow","daisy"]},
                       {weather,snowing}]).
ok
```

Let's check that it's OK.

```
2> file:consult("test1.dat").
{ok,[{cats,["zorrow","daisy"]},{weather,snowing}]}
```

unconsult opens the file in write mode and calls io:format(S, "~p.~n", [X]) to write terms to the file.

io:format is the workhorse for creating formatted output. To produce formatted output, we call the following:

-spec io:format(IoDevice, Format, Args) -> ok

> ioDevice is an I/O device (which must have been opened in write mode), Format is a string containing formatting codes, and Args is a list of items to be output.

For each item in Args, there must be a formatting command in the format string. Formatting commands begin with a tilde (~) character. Here are some of the most commonly used formatting commands:

~n Write a line feed. ~n is smart and writes a line feed in a platform-dependent way. So, on a Unix machine, ~n will write ASCII (10) to the output stream, and on a Windows machine it will write carriage-return line-feed ASCII (13, 10) to the output stream.

~p Pretty-print the argument.

~s The argument is a string or I/O list, or an atom and will be printed without any surrounding quotation marks.

~w Write data with the standard syntax. This is used to output Erlang terms.

The format string has about ten quadzillion arguments that nobody in their right mind could remember. You'll find a complete list in the man page for the module io.

I remember only ~p, ~s, and ~n. If you start with these, you won't have many problems.

Aside

I lied—you'll probably need more than just ~p, ~s, and ~n. Here are a few examples:

```
Format                           Result
======                           ======
io:format("|~10s|",["abc"])      |       abc|
io:format("|~-10s|",["abc"])     |abc       |
io:format("|~10.3.+s|",["abc"])  |+++++++abc|
io:format("|~-10.10.+s|",["abc"]) |abc+++++++|
io:format("|~10.7.+s|",["abc"])  |+++abc++++|
```

Writing Lines to a File

The following is similar to the previous example—we just use a different formatting command:

```
1> {ok, S} = file:open("test2.dat", write).
{ok,<0.62.0>}
2> io:format(S, "~s~n", ["Hello readers"]).
ok
3> io:format(S, "~w~n", [123]).
ok
4> io:format(S, "~s~n", ["that's it"]).
ok
5> file:close(S).
ok
```

This created a file called test2.dat with the following contents:

```
Hello readers
123
that's it
```

Writing an Entire File in One Operation

This is the most efficient way of writing to a file. file:write_file(File, IO) writes the data in IO, which is an I/O list, to File. (An I/O list is a list whose elements are I/O lists, binaries, or integers from 0 to 255. When an I/O list is output, it is automatically "flattened," which means that all the list brackets are removed.) This method is extremely efficient and is one that I often use. The program in the next section illustrates this.

Listing URLs from a File

Let's write a simple function called urls2htmlFile(L, File) that takes a list of URLs, L, and creates an HTML file, where the URLs are presented as clickable links. This lets us play with the technique of creating an entire file in a single I/O operation. We'll write our program in the module scavenge_urls. First here's the program header:

```
scavenge_urls.erl
-module(scavenge_urls).
-export([urls2htmlFile/2, bin2urls/1]).
-import(lists, [reverse/1, reverse/2, map/2]).
```

The program has two entry points. urls2htmlFile(Urls, File) takes a list of URLs and creates an HTML file containing clickable links for each URL, and bin2urls(Bin) searches through a binary and returns a list of all the URLs contained in the binary. urls2htmlFile is as follows:

```
scavenge_urls.erl
urls2htmlFile(Urls, File) ->
    file:write_file(File, urls2html(Urls)).

bin2urls(Bin) ->  gather_urls(binary_to_list(Bin), []).

urls2html(Urls) -> [h1("Urls"),make_list(Urls)].

h1(Title) -> ["<h1>", Title, "</h1>\n"].

make_list(L) ->
    ["<ul>\n",
     map(fun(I) -> ["<li>",I,"</li>\n"] end, L),
     "</ul>\n"].
```

This code returns an I/O list of characters. Note we make no attempt to flatten the list (which would be rather inefficient); we make a deep list of characters and just throw it at the output routines. When we write an I/O list to a file with file:write_file, the I/O system automatically flattens the list (that is, it outputs only the characters embedded in the lists and not the list brackets themselves). Finally, here's the code to extract the URLs from a binary:

```
scavenge_urls.erl
gather_urls("<a href" ++ T, L) ->
    {Url, T1} = collect_url_body(T, reverse("<a href")),
    gather_urls(T1, [Url|L]);
gather_urls([_|T], L) ->
    gather_urls(T, L);
gather_urls([], L) ->
    L.

collect_url_body("</a>" ++ T, L) -> {reverse(L, "</a>"), T};
collect_url_body([H|T], L)       -> collect_url_body(T, [H|L]);
collect_url_body([], _)          -> {[],[]}.
```

To run this, we need to get some data to parse. The input data (a binary) is the content of an HTML page, so we need an HTML page to scavenge. For this we'll use socket_examples:nano_get_url (see *Fetching Data from a Server*, on page 264).

We'll do this step by step in the shell.

```
1> B = socket_examples:nano_get_url("www.erlang.org"),
   L = scavenge_urls:bin2urls(B),
   scavenge_urls:urls2htmlFile(L, "gathered.html").
ok
```

This produces the file gathered.html.

```
gathered.html
<h1>Urls</h1>
<ul>
<li><a href="old_news.html">Older news.....</a></li>
<li><a href="http://www.erlang-consulting.com/training_fs.html">here</a></li>
<li><a href="project/megaco/">Megaco home</a></li>
<li><a href="EPLICENSE">Erlang Public License (EPL)</a></li>
<li><a href="user.html#smtp_client-1.0">smtp_client-1.0</a></li>
<li><a href="download-stats/">download statistics graphs</a></li>
<li><a href="project/test_server">Erlang/OTP Test
Server</a></li>
<li><a href="http://www.erlang.se/euc/06/">proceedings</a></li>
<li><a href="/doc/doc-5.5.2/doc/highlights.html">
      Read more in the release highlights. </a></li>
<li><a href="index.html"><img src="images/erlang.gif"
    border="0" alt="Home"></a></li>
</ul>
```

Writing to a Random-Access File

Writing to a file in random-access mode is similar to reading. First, we have to open the file in write mode. Next, we use file:pwrite(IoDev, Position, Bin) to write to the file.

Here's an example:

```
1> {ok, S} = file:open("some_filename_here", [raw,write,binary]).
{ok, ...}
2> file:pwrite(S, 10, <<"new">>).
ok
3> file:close(S).
ok
```

This writes the characters *new* starting at an offset of 10 in the file, overwriting the original content.

16.4 Directory and File Operations

Three functions in file are used for directory operations. list_dir(Dir) is used to produce a list of the files in Dir, make_dir(Dir) creates a new directory, and del_dir(Dir) deletes a directory.

If we run list_dir on the code directory that I'm using to write this book, we'll see something like the following:

```
1> cd("/home/joe/book/erlang/Book/code").
/home/joe/book/erlang/Book/code
ok
2> file:list_dir(".").
{ok,["id3_v1.erl~",
```

```
    "update_binary_file.beam",
    "benchmark_assoc.beam",
    "id3_v1.erl",
    "scavenge_urls.beam",
    "benchmark_mk_assoc.beam",
    "benchmark_mk_assoc.erl",
    "id3_v1.beam",
    "assoc_bench.beam",
    "lib_misc.beam",
    "benchmark_assoc.erl",
    "update_binary_file.erl",
    "foo.dets",
    "big.tmp",
    ...
```

Observe that there is no particular order to the files and no indication as to whether the files in the directory are files or directories, what the sizes are, and so on.

To find out about the individual files in the directory listing, we'll use file:read_file_info, which is the subject of the next section.

Finding Information About a File

To find out about a file F, we call file:read_file_info(F). This returns {ok, Info} if F is a valid file or directory name. Info is a record of type #file_info, which is defined as follows:

```erlang
-record(file_info,
  {size,            % Size of file in bytes.
   type,            % Atom: device, directory, regular,
                    % or other.
   access,          % Atom: read, write, read_write, or none.
   atime,           % The local time the file was last read:
                    % {{Year, Mon, Day}, {Hour, Min, Sec}}.
   mtime,           % The local time the file was last written.
   ctime,           % The interpretation of this time field
                    % is dependent on operating system.
                    % On Unix it is the last time the file or
                    % or the inode was changed.  On Windows,
                    % it is the creation time.
   mode,            % Integer: File permissions.  On Windows,
                    % the owner permissions will be duplicated
                    % for group and user.
   links,           % Number of links to the file (1 if the
                    % filesystem doesn't support links).
   ...
  }).
```

Note: The mode and access fields overlap. You can use mode to set several file attributes in a single operation, whereas you can use access for simpler operations.

To find the size and type of a file, we call read_file_info as in the following example (note we have to include file.hrl, which contains the definition of the #file_info record):

```
lib_misc.erl
-include_lib("kernel/include/file.hrl").
file_size_and_type(File) ->
    case file:read_file_info(File) of
        {ok, Facts} ->
            {Facts#file_info.type, Facts#file_info.size};
        _ ->
            error
    end.
```

Now we can augment the directory listing returned by list_file by adding information about the files in the function ls() as follows:

```
lib_misc.erl
ls(Dir) ->
    {ok, L} = file:list_dir(Dir),
    lists:map(fun(I) -> {I, file_size_and_type(I)} end, lists:sort(L)).
```

Now when we list the files, they are ordered and contain additional useful information.

```
1> lib_misc:ls(".").
[{"Makefile",{regular,1244}},
 {"README",{regular,1583}},
 {"abc.erl",{regular,105}},
 {"alloc_test.erl",{regular,303}},
 ...
 {"socket_dist",{directory,4096}},
 ...
```

As a convenience, the module filelib exports a few small routines such as file_size(File) and is_dir(X). These are merely interfaces to file:read_file_info. If we just want to get the size of a file, it is more convenient to call filelib:file_size than to call file:read_file_info and unpack the elements of the #file_info record.

Copying and Deleting Files

file:copy(Source, Destination) copies the file Source to Destination.

file:delete(File) deletes File.

16.5 Bits and Pieces

So far we've mentioned most of the functions that I use on a day-to-day basis for manipulating files. The topics listed next are not discussed here; details can be found in the manual pages.

File modes

When we open a file with file:open, we open the file in a particular mode or a combination of modes. There are actually many more modes than we might think; for example, it's possible to read and write gzip-compressed files with the compressed mode flag, and so on. The full list is in the manual pages.

Modification times, groups, symlinks

We can set all of these with routines in file.

Error codes

I've rather blandly said that all errors are of the form {error, Why}; in fact, Why is an atom (for example, enoent means a file does not exist, and so on) —there are a large number of error codes, and they are all described in the manual pages.

filename

The filename module has some useful routines for ripping apart full file-names in directories and finding the file extensions, and so on, as well as for rebuilding filenames from the component parts. All this is done in a platform-independent manner.

filelib

The filelib module has a small number of routines that can save us some work. For example, filelib:ensure_dir(Name) ensures that all parent directories for the given file or directory name Name exist, trying to create them if necessary.

16.6 A Find Utility

As a final example, we'll use file:list_dir and file:read_file_info to make a general-purpose "find" utility.

The main entry point to this module is as follows:

```
lib_find:files(Dir, RegExp, Recursive, Fun, Acc0)
```

The arguments are as follows:

Dir

The directory name to start the file search in.

RegExp

> A shell-style regular expression to test the files we have found. If the files we encounter match this regular expression, then Fun(File, Acc) will be called, where File is the name of the file that matches the regular expression.

Recursive = true | false

> A flag that determines whether search should recurse into the subdirectories of the current directory in the search path.

Fun(File, AccIn) -> AccOut

> A function that is applied to the File if regExp matches File. Acc is an accumulator whose initial value is Acc0. Each time Fun is called, it must return a new value of the accumulator that is passed into Fun the next time it is called. The final value of the accumulator is the return value of lib_find:files/5

We can pass any function we want into lib_find:files/5. For example, we can build a list of files using the following function, passing it an empty list initially:

```erlang
fun(File, Acc) -> [File|Acc] end
```

The module entry point lib_find:files(Dir, ShellRegExp, Flag) provides a simplified entry point for a more common usage of the program. Here ShellRegExp is a shell-style wildcard pattern that is easier to write than the full form of a regular expression.

As an example of the short form of calling sequence, the following call:

```erlang
lib_find:files(Dir, "*.erl", true)
```

recursively finds all Erlang files under Dir. If the last argument had been false, then only the Erlang files in the directory Dir would have been found—it would not look in subdirectories.

Finally, here's the code:

```erlang
lib_find.erl
-module(lib_find).
-export([files/3, files/5]).
-import(lists, [reverse/1]).

-include_lib("kernel/include/file.hrl").

files(Dir, Re, Flag) ->
    Re1 = xmerl_regexp:sh_to_awk(Re),
    reverse(files(Dir, Re1, Flag, fun(File, Acc) ->[File|Acc] end, [])).

files(Dir, Reg, Recursive, Fun, Acc) ->
    case file:list_dir(Dir) of
        {ok, Files} -> find_files(Files, Dir, Reg, Recursive, Fun, Acc);
```

```erlang
        {error, _}  -> Acc
    end.

find_files([File|T], Dir, Reg, Recursive, Fun, Acc0) ->
    FullName = filename:join([Dir,File]),
    case file_type(FullName) of
        regular ->
            case re:run(FullName, Reg, [{capture,none}]) of
                match  ->
                    Acc = Fun(FullName, Acc0),
                    find_files(T, Dir, Reg, Recursive, Fun, Acc);
                nomatch ->
                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)
            end;
        directory ->
            case Recursive of
                true ->
                    Acc1 = files(FullName, Reg, Recursive, Fun, Acc0),
                    find_files(T, Dir, Reg, Recursive, Fun, Acc1);
                false ->
                    find_files(T, Dir, Reg, Recursive, Fun, Acc0)
            end;
        error ->
            find_files(T, Dir, Reg, Recursive, Fun, Acc0)
    end;
find_files([], _, _, _, _, A) ->
    A.

file_type(File) ->
    case file:read_file_info(File) of
        {ok, Facts} ->
            case Facts#file_info.type of
                regular   -> regular;
                directory -> directory;
                _         -> error
            end;
        _ ->
            error
    end.
```

That was how to find files. You'll note that the program was purely sequential. To speed it up, you might like to parallelize it using several parallel processes. I'm not going to do this here. I'll let you think about it.

File access in Erlang when combined with concurrency gives us a powerful tool for solving complex problems. If we want to analyze a large number of files in parallel, we spawn many processes where each process analyzes an individual file. The only thing we have to take care of is making sure that two processes don't try to read and write the same file at the same time.

Efficient file operations are best achieved by reading and writing files in a single operation and by creating I/O lists prior to writing a file.

Exercises

1. When an Erlang file X.erl is compiled, a file X.beam is created (if the compilation succeeded). Write a program that checks whether an Erlang module needs recompiling. Do this by comparing the last modified time stamps of the Erlang and beam files concerned.

2. Write a program to compute the MD5 checksum of a small file, and use the BIF erlang:md5/1 to compute the MD5 checksum of the data (see the Erlang manual page for details of this BIF).

3. Repeat the previous exercise for a large file (say a few hundred megabytes). This time read the file in small chunks, using erlang:md5_init, erlang:md5_update, and erlang:md5_final to compute the MD5 sum of the file.

4. Use the lib_find module to find all .jpg files on your computer. Compute the MD5 checksum of each file by comparing MD5 checksums to see whether any two images are identical.

5. Write a cache mechanism that computes the MD5 checksum of a file and remembers it in a cache, together with the last modified time of the file. When we want the MD5 sum of a file, check the cache to see whether the value has been computed, and recompute it if the last modified time of the file has been changed.

6. Twits are exactly 140 bytes long. Write a random-access twit storage module called twit_store.erl that exports the following: init(K) allocates space for K twits. store(N, Buf) stores twit number N (1..K) with data Buf (a 140-byte binary) in the store. fetch(N) fetches the data for twit number N.

Programming with Sockets

Most of the more interesting programs that I write involve sockets one way or another. Programming with sockets is fun because it allows applications to interact with other machines on the Internet, which has far more potential than just performing local operations.

A socket is a communication channel that allows machines to communicate over the Internet using the Internet Protocol (IP). In this chapter, we'll concentrate on the two core protocols of the Internet: *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP).

UDP lets applications send short messages (called *datagrams*) to each other, but there is no guarantee of delivery for these messages. They can also arrive out of order. TCP, on the other hand, provides a reliable stream of bytes that are delivered in order as long as the connection is established. Sending data by TCP incurs a larger overhead than sending data by UDP. You can choose between a reliable and slower channel (TCP) or a faster and unreliable channel (UDP).

There are two main libraries for programming with sockets: gen_tcp for programming TCP applications and gen_udp for programming UDP applications.

In this chapter, we'll see how to program clients and servers using TCP and UDP sockets. We'll go through the different forms of servers that are possible (parallel, sequential, blocking, and nonblocking) and see how to program traffic-shaping applications that can control the flow of data to the application.

17.1 Using TCP

We'll start our adventures in socket programming by looking at a simple TCP program that fetches data from a server. After this, we'll write a simple

sequential TCP server and show how it can be parallelized to handle multiple parallel sessions.

Fetching Data from a Server

Let's start by writing a little function called nano_get_url/0 that uses a TCP socket to fetch an HTML page from http://www.google.com.

```
socket_examples.erl
nano_get_url() ->
    nano_get_url("www.google.com").

nano_get_url(Host) ->
❶   {ok,Socket} = gen_tcp:connect(Host,80,[binary, {packet, 0}]),
❷   ok = gen_tcp:send(Socket, "GET / HTTP/1.0\r\n\r\n"),
    receive_data(Socket, []).

receive_data(Socket, SoFar) ->
    receive
❸       {tcp,Socket,Bin} ->
            receive_data(Socket, [Bin|SoFar]);
❹       {tcp_closed,Socket} ->
❺           list_to_binary(reverse(SoFar))
    end.
```

This works as follows:

❶ We open a TCP socket to port 80 of http://www.google.com by calling gen_tcp:connect. The argument binary in the connect call tells the system to open the socket in "binary" mode and deliver all data to the application as binaries. {packet,0} means the TCP data is delivered directly to the application in an unmodified form.

❷ We call gen_tcp:send and send the message GET / HTTP/1.0\r\n\r\n to the socket. Then we wait for a reply. The reply doesn't come all in one packet but comes fragmented, a bit at a time. These fragments will be received as a sequence of messages that are sent to the process that opened (or controls) the socket.

❸ We receive a {tcp,Socket,Bin} message. The third argument in this tuple is a binary. This is because we opened the socket in binary mode. This message is one of the data fragments sent to us from the web server. We add it to the list of fragments we have received so far and wait for the next fragment.

❹ We receive a {tcp_closed, Socket} message. This happens when the server has finished sending us data.

❺ When all the fragments have come, we've stored them in the wrong order, so we reverse the order and concatenate all the fragments.

The code that reassembled the fragments looked like this:

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, [Bin|SoFar]);
        {tcp_closed,Socket} ->
            list_to_binary(reverse(SoFar))
    end.
```

So, as the fragments arrive, we just add them to the head of the list SoFar. When all the fragments have arrived and the socket is closed, we reverse the list and concatenate all the fragments.

You might think that it would be better to write the code to accumulate the fragments like this:

```
receive_data(Socket, SoFar) ->
    receive
        {tcp,Socket,Bin} ->
            receive_data(Socket, list_to_binary([SoFar,Bin]));
        {tcp_closed,Socket} ->
            SoFar
    end.
```

This code is correct but less efficient than the original version. The reason is that in the latter version we are continually appending a new binary to the end of the buffer, which involves a lot of copying of data. It's much better to accumulate all the fragments in a list (which will end up in the wrong order) and then reverse the entire list and concatenate all the fragments in one operation.

Let's just test that our little HTTP client works.

```
1> B = socket_examples:nano_get_url().
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"...>>
```

Note: When you run nano_get_url, the result is a binary, so you'll see what a binary looks like when pretty printed in the Erlang shell. When binaries are pretty printed, all control characters are displayed in an escaped format. And the binary is truncated, which is indicated by the three dots (...>>) at the end of the printout. If you want to see all of the binary, you can print it with io:format or break it into pieces with string:tokens.

Writing a Web Server

Writing something like a web client or a server is great fun. Sure, other people have already written these things, but if we really want to understand how they work, digging under the surface and finding out exactly how they work is very instructive. Who knows—maybe our web server will be better than the best.

To build a web server, or for that matter any software that implements a standard Internet protocol, we need to use the right tools and need to know exactly which protocols to implement.

In our example code that fetched a web page, we opened port 80 and sent it a GET / HTTP/1.0\r\n\r\n command. We used the HTTP protocol defined in RFC 1945. All the major protocols for Internet services are defined in *requests for comments* (RFCs). The official website for all RFCs is http://www.ietf.org (home of the Internet Engineering Task Force).

The other invaluable source of information is a *packet sniffer*. With a packet sniffer we can capture and analyze all the IP packets coming from and going to our application. Most packet sniffers include software that can decode and analyze the data in the packets and present the data in a meaningful manner. One of the most well-known and possibly the best is Wireshark (previously known as Ethereal), available from http://www.wireshark.org.

Armed with a packet sniffer dump and the appropriate RFCs, we're ready to write our next killer application.

```
2> io:format("~p~n",[B]).
<<"HTTP/1.0 302 Found\r\nLocation: http://www.google.se/\r\n
  Cache-Control: private\r\nSet-Cookie: PREF=ID=b57a2c:TM"
 TM=176575171639526:LM=1175441639526:S=gkfTrK6AFkybT3;
  expires=Sun, 17-Jan-2038 19:14:07
  ... several lines omitted ...
>>
3> string:tokens(binary_to_list(B),"\r\n").
["HTTP/1.0 302 Found",
 "Location: http://www.google.se/",
 "Cache-Control: private",
 "Set-Cookie: PREF=ID=ec7f0c7234b852dece4:TM=11713424639526:
LM=1171234639526:S=gsdertTrK6AEybT3;
expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/; domain=.google.com",
 "Content-Type: text/html",
 "Server: GWS/2.1",
 "Content-Length: 218",
 "Date: Fri, 16 Jan 2009 15:25:26 GMT",
 "Connection: Keep-Alive",
... lines omitted ...
```

Note that the 302 response code is not an error; it's the expected response of this command, which is to redirect to a new address. Also note that this

example shows how socket communication works and does not strictly follow
the HTTP protocol.

This is more or less how a web client works (with the emphasis on *less*—we
would have to do a lot of work to correctly render the resulting data in a web
browser). The previous code is, however, a good starting point for your own
experiments. You might like to try modifying this code to fetch and store an
entire website or automatically go and read your email. The possibilities are
boundless.

A Simple TCP Server

In the previous section, we wrote a simple client. Now let's write a server.

This server opens port 2345 and then waits for a single message. This message
is a binary that contains an Erlang term. The term is an Erlang string that
contains an expression. The server evaluates the expression and sends the
result to the client by writing the result to the socket.

To write this program (and indeed any program that runs over TCP/IP), we
have to answer a few simple questions.

- How is the data organized? How do we know how much data makes up
  a single request or response?

- How is the data within a request or the response encoded and decoded?
  (Encoding the data is sometimes called *marshaling*, and decoding the data
  is sometimes called *demarshaling*.)

TCP socket data is just an undifferentiated stream of bytes. During transmission,
this data can be broken into arbitrary-sized fragments, so we need some conven-
tion so that we know how much data represents a single request or response.

In the Erlang case we use the simple convention that every logical request or
response will be preceded by an N (1, 2, or 4) byte length count. This is the
meaning of the {packet, N} argument in the gen_tcp:connect and gen_tcp:listen
functions. The word *packet* here refers to the length of an application request
or response message, not to the physical packet seen on the wire. Note that
the arguments to packet used by the client and the server *must* agree. If the
server was opened with {packet,2} and the client with {packet,4}, then nothing
would work.

Having opened a socket with the {packet,N} option, we don't need to worry
about data fragmentation. The Erlang drivers will make sure that all fragment-
ed data messages are reassembled to the correct lengths before delivering
them to the application.

The next concern is data *encoding* and *decoding*. We'll use the simplest possible way of encoding and decoding messages using term_to_binary to encode Erlang terms and using its inverse, binary_to_term, to decode the data.

Note that the packaging convention and encoding rules needed for the client to talk to the server is achieved in two lines of code, by using the {packet,4} option when we open the socket and by using term_to_binary and its inverse to encode and decode the data.

The ease with which we can package and encode Erlang terms gives us a significant advantage over text-based methods such as HTTP or XML. Using the Erlang BIF term_to_binary and its inverse binary_to_term is typically more than an order of magnitude faster than performing an equivalent operation using XML terms and involves sending far less data. Now to the programs. First, here's a very simple server:

socket_examples.erl
```
   start_nano_server() ->
❶      {ok, Listen} = gen_tcp:listen(2345, [binary, {packet, 4},
                                            {reuseaddr, true},
                                            {active, true}]),
❷      {ok, Socket} = gen_tcp:accept(Listen),
❸      gen_tcp:close(Listen),
       loop(Socket).
   loop(Socket) ->
       receive
           {tcp, Socket, Bin} ->
               io:format("Server received binary = ~p~n",[Bin]),
❹              Str = binary_to_term(Bin),
               io:format("Server (unpacked)  ~p~n",[Str]),
❺              Reply = lib_misc:string2value(Str),
               io:format("Server replying = ~p~n",[Reply]),
❻              gen_tcp:send(Socket, term_to_binary(Reply)),
               loop(Socket);
           {tcp_closed, Socket} ->
               io:format("Server socket closed~n")
       end.
```

This works as follows:

❶ First we call gen_tcp:listen to listen for a connection on port 2345 and set up the message packaging conventions. {packet, 4} means that each application message will be preceded by a 4-byte length header. Then gen_tcp:listen(..) returns {ok, Listen} or {error, Why}, but we're interested only in the return case where we were able to open a socket. Therefore, we write the following code:

{ok, Listen} = gen_tcp:listen(...),

This causes the program to raise a pattern matching exception if gen_tcp:listen returns {error, …}. In the successful case, this statement binds Listen to the new listening socket. There's only one thing we can do with a listening socket, and that's to use it as an argument to gen_tcp:accept.

❷ Now we call gen_tcp:accept(Listen). At this point, the program will suspend and wait for a connection. When we get a connection, this function returns with the variable Socket bound to a socket that can be used to talk to the client that performed the connection.

❸ When accept returns, we immediately call gen_tcp:close(Listen). This closes down the listening socket, so the server will not accept any new connections. This does not affect the existing connection; it just prevents new connections.

❹ We decode the input data (unmarshaling).

❺ Then we evaluate the string.

❻ Then we encode the reply data (marshaling) and send it back to the socket.

Note that this program accepts only a single request; once the program has run to completion, then no more connections will be accepted.

This is the simplest of servers that illustrates how to package and encode the application data. It accepts a request, computes a reply, sends the reply, and terminates.

To test the server, we need a corresponding client.

```
socket_examples.erl
nano_client_eval(Str) ->
    {ok, Socket} =
        gen_tcp:connect("localhost", 2345,
                        [binary, {packet, 4}]),
    ok = gen_tcp:send(Socket, term_to_binary(Str)),
    receive
        {tcp,Socket,Bin} ->
            io:format("Client received binary = ~p~n",[Bin]),
            Val = binary_to_term(Bin),
            io:format("Client result = ~p~n",[Val]),
            gen_tcp:close(Socket)
    end.
```

To test our code, we'll run both the client and the server on the same machine, so the hostname in the gen_tcp:connect function is hardwired to localhost.

Note how term_to_binary is called in the client to encode the message and how binary_to_term is called in the server to reconstruct the message.

To run this, we need to open two terminal windows and start an Erlang shell in each of the windows.

First we start the server.

```
1> socket_examples:start_nano_server().
```

We won't see any output in the server window, since nothing has happened yet. Then we move to the client window and give the following command:

```
1> socket_examples:nano_client_eval("list_to_tuple([2+3*4,10+20])").
```

In the server window, we should see the following:

```
Server received binary = <<131,107,0,28,108,105,115,116,95,116,
                           111,95,116,117,112,108,101,40,91,50,
                           43,51,42,52,44,49,48,43,50,48,93,41>>
Server (unpacked)  "list_to_tuple([2+3*4,10+20])"
Server replying = {14,30}
```

In the client window, we'll see this:

```
Client received binary = <<131,104,2,97,14,97,30>>
Client result = {14,30}
ok
```

Finally, in the server window, we'll see this:

```
Server socket closed
```

Sequential and Parallel Servers

In the previous section, we made a server that accepted only one connection and then terminated. By changing this code slightly, we can make two different types of server.

- A sequential server—one that accepts one connection at a time

- A parallel server—one that accepts multiple parallel connections at the same time

The original code started like this:

```
start_nano_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket).
 ...
```

We'll be changing this to make our two server variants.

A Sequential Server

To make a sequential server, we change this code to the following:

```
start_seq_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    seq_loop(Listen).

seq_loop(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket),
    seq_loop(Listen).

loop(..) -> %% as before
```

This works pretty much as in the previous example, but since we want to serve more than one request, we leave the listening socket open and don't call gen_tcp:close(Listen). The other difference is that after loop(Socket) has finished, we call seq_loop(Listen) again, which waits for the next connection.

If a client tries to connect to the server while the server is busy with an existing connection, then the connection will be queued until the server has finished with the existing connection. If the number of queued connections exceeds the listen backlog, then the connection will be rejected.

We've shown only the code that starts the server. Stopping the server is easy (as is stopping a parallel server); just kill the process that started the server or servers. gen_tcp links itself to the controlling process, and if the controlling process dies, it closes the socket.

A Parallel Server

The trick to making a parallel server is to immediately spawn a new process each time gen_tcp:accept gets a new connection.

```
start_parallel_server() ->
    {ok, Listen} = gen_tcp:listen(...),
    spawn(fun() -> par_connect(Listen) end).

par_connect(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_connect(Listen) end),
    loop(Socket).

loop(..) -> %% as before
```

This code is similar to the sequential server that we saw earlier. The crucial difference is the addition of a spawn, which makes sure that we create a parallel process for each new socket connection. Now is a good chance to compare

the two. You should look at the placement of the spawn statements and see how these turned a sequential server into a parallel server.

All three servers call gen_tcp:listen and gen_tcp:accept; the only difference is whether we call these functions in a parallel program or a sequential program.

Notes

Be aware of the following:

- The process that creates a socket (by calling gen_tcp:accept or gen_tcp:connect) is said to be the *controlling process* for that socket. All messages from the socket will be sent to the controlling process; if the controlling process dies, then the socket will be closed. The controlling process for a socket can be changed to NewPid by calling gen_tcp:controlling_process(Socket, NewPid).

- Our parallel server can potentially create many thousands of connections. We might want to limit the maximum number of simultaneous connections. This can be done by maintaining a counter of how many connections are alive at any one time. We increment this counter every time we get a new connection, and we decrement the counter each time a connection finishes. We can use this to limit the total number of simultaneous connections in the system.

- After we have accepted a connection, it's a good idea to explicitly set the required socket options, like this:

```
{ok, Socket} = gen_tcp:accept(Listen),
inet:setopts(Socket, [{packet,4},binary,
                      {nodelay,true},{active, true}]),
loop(Socket)
```

- As of Erlang version R11B-3, several Erlang processes are allowed to call gen_tcp:accept/1 on the same listen socket. This simplifies making a parallel server, because you can have a pool of prespawned processes, all waiting in gen_tcp:accept/1.

17.2 Active and Passive Sockets

Erlang sockets can be opened in one of three modes: *active*, *active once*, or *passive*. This is done by including an option {active, true | false | once} in the Options argument to either gen_tcp:connect(Address, Port, Options or gen_tcp:listen(Port, Options).

If {active, true} is specified, then an active socket will be created; {active, false} specifies a passive socket. {active, once} creates a socket that is active but only

for the reception of one message; after it has received this message, it must be reenabled before it can receive the next message.

We'll go through how these different types of sockets are used in the following sections.

The difference between active and passive sockets has to do with what happens when messages are received by the socket.

- Once an active socket has been created, the controlling process will be sent {tcp, Socket, Data} messages as data is received. There is no way the controlling process can control the flow of these messages. A rogue client could send thousands of messages to the system, and these would all be sent to the controlling process. The controlling process cannot stop this flow of messages.

- If the socket was opened in passive mode, then the controlling process has to call gen_tcp:recv(Socket, N) to receive data from the socket. It will then try to receive exactly N bytes from the socket. If N = 0, then all available bytes are returned. In this case, the server can control the flow of messages from the client by choosing when to call gen_tcp:recv.

Passive sockets are used to control the flow of data to a server. To illustrate this, we can write the message reception loop of a server in three ways.

- Active message reception (nonblocking)
- Passive message reception (blocking)
- Hybrid message reception (partial blocking)

Active Message Reception (Nonblocking)

Our first example opens a socket in active mode and then receives messages from the socket.

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, true}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            ... do something with the data ...
        {tcp_closed, Socket} ->
            ...
    end.
```

This process cannot control the flow of messages to the server loop. If the client produces data faster than the server can consume this data, then the system can be *flooded* with messages—the message buffers will fill up, and the system might crash or behave strangely.

This type of server is called a *nonblocking* server because it cannot block the client. We should write a nonblocking server only if we can convince ourselves that it can keep up with the demands of the clients.

Passive Message Reception (Blocking)

In this section, we'll write a blocking server. The server opens the socket in passive mode by setting the {active, false} option. This server cannot be crashed by an overactive client that tries to flood it with too much data.

The code in the server loop calls gen_tcp:recv every time it wants to receive data. The client will block until the server has called recv. Note that the OS does some buffering that allows the client to send a small amount of data before it blocks even if recv has not been called.

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, false}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    case gen_tcp:recv(Socket, N) of
        {ok, B} ->
            ... do something with the data ...
            loop(Socket);
        {error, closed}
            ...
    end.
```

The Hybrid Approach (Partial Blocking)

You might think that using passive mode for all servers is the correct approach. Unfortunately, when we're in passive mode, we can wait for the data from only one socket. This is useless for writing servers that must wait for data from multiple sockets.

Fortunately, we can adopt a hybrid approach, neither blocking nor nonblocking. We open the socket with the option {active, once}. In this mode, the socket is active *but for only one message*. After the controlling processes has been sent a message, it must explicitly call inet:setopts to reenable reception of the next message. The system will block until this happens. This is the best of both worlds. Here's what the code looks like:

```
{ok, Listen} = gen_tcp:listen(Port, [..,{active, once}...]),
{ok, Socket} = gen_tcp:accept(Listen),
loop(Socket).

loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            ... do something with the data ...
            %% when you're ready enable the next message
            inet:setopts(Sock, [{active, once}]),
            loop(Socket);
        {tcp_closed, Socket} ->
            ...
    end.
```

Using the {active, once} option, the user can implement advanced forms of flow control (sometimes called *traffic shaping*) and thus prevent a server from being flooded by excessive messages.

Finding Out Where Connections Come From

Suppose we write some kind of online server and find that somebody keeps spamming our site. To try to prevent this, we need to know where the connection came from. To discover this, we can call inet:peername(Socket).

@spec inet:peername(Socket) -> {ok, {IP_Address, Port}} | {error, Why}
This returns the IP address and port of the other end of the connection so the server can discover who initiated the connection. IP_Address is a tuple of integers, with {N1,N2,N3,N4} representing the IP address for IPv4 and {K1,K2,K3,K4,K5,K6,K7,K8} representing it for IPv6. Here Ni are integers in the range 0 to 255, and Ki are integers in the range 0 to 65535.

17.3 Error Handling with Sockets

Error handling with sockets is extremely easy—basically you don't have to do anything. As we said earlier, each socket has a controlling process (that is, the process that created the socket). If the controlling process dies, then the socket will be automatically closed.

This means that if we have, for example, a client and a server and the server dies because of a programming error, the socket owned by the server will be automatically closed, and the client will be sent a {tcp_closed, Socket} message.

We can test this mechanism with the following small program:

```
socket_examples.erl
error_test() ->
    spawn(fun() -> error_test_server() end),
    lib_misc:sleep(2000),
    {ok,Socket} = gen_tcp:connect("localhost",4321,[binary, {packet, 2}]),
    io:format("connected to:~p~n",[Socket]),
    gen_tcp:send(Socket, <<"123">>),
    receive
        Any ->
            io:format("Any=~p~n",[Any])
    end.
error_test_server() ->
    {ok, Listen} = gen_tcp:listen(4321, [binary,{packet,2}]),
    {ok, Socket} = gen_tcp:accept(Listen),
    error_test_server_loop(Socket).
error_test_server_loop(Socket) ->
    receive
        {tcp, Socket, Data} ->
            io:format("received:~p~n",[Data]),
            _ = atom_to_list(Data),
            error_test_server_loop(Socket)
    end.
```

When we run it, we see the following:

```
1> socket_examples:error_test().
connected to:#Port<0.152>
received:<<"123">>
=ERROR REPORT==== 30-Jan-2009::16:57:45 ===
Error in process <0.77.0> with exit value:
 {badarg,[{erlang,atom_to_list,[<<3 bytes>>]},
 {socket_examples,error_test_server_loop,1}]}
Any={tcp_closed,#Port<0.152>}
ok
```

We spawn a server, sleep for two seconds to give it a chance to start, and then send it a message containing the binary <<"123">>. When this message arrives at the server, the server tries to compute atom_to_list(Data) where Data is a binary and immediately crashes. The system monitor prints the diagnostic that you can see in the shell. Now that the controlling process for the server side of the socket has crashed, the (server-side) socket is automatically closed. The client is then sent a {tcp_closed, Socket} message.

17.4 UDP

Now let's look at the User Datagram Protocol (UDP). Using UDP, machines on the Internet can send each other short messages called *datagrams*. UDP datagrams are unreliable. This means if a client sends a sequence of UDP

datagrams to a server, then the datagrams might arrive out of order, not at all, or even more than once, but the individual datagrams, if they arrive, will be undamaged. Large datagrams can get split into smaller fragments, but the IP protocol will reassemble the fragments before delivering them to the application.

UDP is a *connectionless* protocol, which means the client does not have to establish a connection to the server before sending it a message. This means that UDP is well suited for applications where large numbers of clients send small messages to a server.

Writing a UDP client and server in Erlang is much easier than writing in the TCP case since we don't have to worry about maintaining connections to the server.

The Simplest UDP Server and Client

First let's discuss the server. The general form of a UDP server is as follows:

```
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    loop(Socket).
loop(Socket) ->
    receive
       {udp, Socket, Host, Port, Bin} ->
            BinReply = ... ,
            gen_udp:send(Socket, Host, Port, BinReply),
            loop(Socket)
    end.
```

This is somewhat easier than the TCP case since we don't need to worry about our process receiving "socket closed" messages. Note that we opened the socket in a *binary* mode, which tells the driver to send all messages to the controlling process as binary data.

Now the client. Here's a very simple client. It merely opens a UDP socket, sends a message to the server, waits for a reply (or timeout), and then closes the socket and returns the value returned by the server.

```
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    ok = gen_udp:send(Socket, "localhost", 4000, Request),
    Value = receive
                {udp, Socket, _, _, Bin} ->
                    {ok, Bin}
            after 2000 ->
                error
            end,
    gen_udp:close(Socket),
    Value
```

We must have a timeout since UDP is unreliable and we might not actually get a reply.

A UDP Factorial Server

We can easily build a UDP server that computes the good ol' factorial of any number that is sent to it. The code is modeled on that in the previous section.

udp_test.erl
```erlang
-module(udp_test).
-export([start_server/0, client/1]).

start_server() ->
    spawn(fun() -> server(4000) end).

%% The server
server(Port) ->
    {ok, Socket} = gen_udp:open(Port, [binary]),
    io:format("server opened socket:~p~n",[Socket]),
    loop(Socket).

loop(Socket) ->
    receive
        {udp, Socket, Host, Port, Bin} = Msg ->
            io:format("server received:~p~n",[Msg]),
            N = binary_to_term(Bin),
            Fac = fac(N),
            gen_udp:send(Socket, Host, Port, term_to_binary(Fac)),
            loop(Socket)
    end.

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

%% The client

client(N) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    io:format("client opened socket=~p~n",[Socket]),
    ok = gen_udp:send(Socket, "localhost", 4000,
                      term_to_binary(N)),
    Value = receive
                {udp, Socket, _, _, Bin} = Msg ->
                    io:format("client received:~p~n",[Msg]),
                    binary_to_term(Bin)
            after 2000 ->
                    0
            end,
    gen_udp:close(Socket),
    Value.
```

Note that I have added a few print statements so we can see what's happening when we run the program. I always add a few print statements when I develop a program and then edit or comment them out when the program works.

Now let's run this example. First we start the server.

```
1> udp_test:start_server().
server opened socket:#Port<0.106>
<0.34.0>
```

This runs in the background, so we can make a client request to request the value of factorial 40.

```
2> udp_test:client(40).
client opened socket=#Port<0.105>
server received:{udp,#Port<0.106>,{127,0,0,1},32785,<<131,97,40>>}
client received:{udp,#Port<0.105>,
                 {127,0,0,1}, 4000,
                 <<131,110,20,0,0,0,0,0,64,37,5,255,
                   100,222,15,8,126,242,199,132,27,
                   232,234,142>>}
815915283247897734345611269596115894272000000000
```

And now we have a little UDP factorial server. Just for fun you might like to try writing the TCP equivalent of this program and benchmarking the two against each other.

UDP Packet Gotchas

We should note that because UDP is a connectionless protocol, the server has no way to block the client by refusing to read data from it—the server has no idea who the clients are.

Large UDP packets might become fragmented as they pass through the network. Fragmentation occurs when the UDP data size is greater than the maximum transfer unit (MTU) size allowed by the routers that the packet passes through when it travels over the network. The usual advice given in tuning a UDP network is to start with a small packet size (say, about 500 bytes) and then gradually increase it while measuring throughput. If at some point the throughput drops dramatically, then you know the packets are too large.

A UDP packet can be delivered twice (which surprises some people), so you have to be careful writing code for remote procedure calls. It might happen that the reply to a second query was in fact a duplicated answer to the first query. To avoid this, we could modify the client code to include a unique reference and check that this reference is returned by the server. To generate a

unique reference, we call the Erlang BIF make_ref, which is guaranteed to return a globally unique reference. The code for a remote procedure call now looks like this:

```erlang
client(Request) ->
    {ok, Socket} = gen_udp:open(0, [binary]),
    Ref = make_ref(), %% make a unique reference
    B1 = term_to_binary({Ref, Request}),
    ok = gen_udp:send(Socket, "localhost", 4000, B1),
    wait_for_ref(Socket, Ref).

wait_for_ref(Socket, Ref) ->
    receive
        {udp, Socket, _, _, Bin} ->
            case binary_to_term(Bin) of
                {Ref, Val} ->
                    %% got the correct value
                    Val;
                {_SomeOtherRef, _} ->
                    %% some other value throw it away
                    wait_for_ref(Socket, Ref)
            end;
    after 1000 ->
        ...
    end.
```

Now we're done with UDP. UDP is often used for online gaming where low latency is required, and it doesn't matter if the odd packet is lost.

17.5 Broadcasting to Multiple Machines

Finally we'll see how to set up a broadcast channel. This code is simple.

broadcast.erl
```erlang
-module(broadcast).
-compile(export_all).

send(IoList) ->
    case inet:ifget("eth0", [broadaddr]) of
        {ok, [{broadaddr, Ip}]} ->
            {ok, S} =  gen_udp:open(5010, [{broadcast, true}]),
            gen_udp:send(S, Ip, 6000, IoList),
            gen_udp:close(S);
        _ ->
            io:format("Bad interface name, or\n"
                      "broadcasting not supported\n")
    end.

listen() ->
    {ok, _} = gen_udp:open(6000),
    loop().
```

```
loop() ->
    receive
        Any ->
            io:format("received:~p~n", [Any]),
            loop()
    end.
```

Here we need two ports, one to send the broadcast and the other to listen for answers. We've chosen port 5010 to send the broadcast request and 6000 to listen for broadcasts (these two numbers have no significance; I just choose two free ports on my system).

Only the process performing a broadcast opens port 5010, but all machines in the network call broadcast:listen(), which opens port 6000 and listens for broadcast messages.

broadcast:send(IoList) broadcasts IoList to all machines on the local area network.

Note: For this to work, the name of the interface must be correct, and broadcasting must be supported. On my iMac, for example, I use the name "en0" instead of "eth0." Note also that if hosts running UDP listeners are on different network subnets, the UDP broadcasts are unlikely to reach them, because by default routers drop such UDP broadcasts.

17.6 A SHOUTcast Server

To finish off this chapter, we'll use our newly acquired skills in socket programming to write a SHOUTcast server. SHOUTcast is a protocol developed by the folks at Nullsoft for streaming audio data.[1] SHOUTcast sends MP3- or AAC-encoded audio data using HTTP as the transport protocol.

To see how things work, we'll first look at the SHOUTcast protocol. Then we'll look at the overall structure of the server. We'll finish with the code.

The SHOUTcast Protocol

The SHOUTcast protocol is simple.

1. First the client (which can be something like XMMS, Winamp, or iTunes) sends an HTTP request to the SHOUTcast server. Here's the request that XMMS generates when I run my SHOUTcast server at home:

    ```
    GET / HTTP/1.1
    Host: localhost
    User-Agent: xmms/1.2.10
    Icy-MetaData:1
    ```

———————————

1. http://www.shoutcast.com/

2. My SHOUTcast server replies with this:

    ```
    ICY 200 OK
    icy-notice1: <BR>This stream requires
      <a href=http://www.winamp.com/>;Winamp</a><BR>
    icy-notice2: Erlang Shoutcast server<BR>
    icy-name: Erlang mix
    icy-genre: Pop Top 40 Dance Rock
    icy-url: http://localhost:3000
    content-type: audio/mpeg
    icy-pub: 1
    icy-metaint: 24576
    icy-br: 96
    ... data ...
    ```

3. Now the SHOUTcast server sends a continuous stream of data. The data
    has the following structure:

    F H F H F H F ...

    F is a block of MP3 audio data that must be exactly 24,576 bytes long (the
    value given in the icy-metaint parameter). H is a header block. The header
    block consists of a single-byte K followed by exactly 16*K bytes of data.
    Thus, the smallest header block that can be represented in the binary is
    <<0>>. The next header block can be represented as follows:

    <<1,B1,B2, ..., B16>>

    The content of the data part of the header is a string of the form StreamTitle='
    ... ';StreamUrl='http:// ...';, which is zero padded to the right to fill up the block.

How the SHOUTcast Server Works

To make a server, we have to attend to the following details:

1. Make a *playlist.* Our server uses a file containing a list of song titles we
    created in *Reading MP3 Metadata,* on page 249. Audio files are chosen at
    random from this list.

2. Make a parallel server so we can serve several streams in parallel. We do
    this using the techniques described in *A Parallel Server,* on page 271.

3. For each audio file, we want to send only the audio data and *not* the
    embedded ID3 tags to the client. Audio encoders are supposed to skip
    over bad data, so in principle we could send the ID3 tags along with the
    data. In practice, the program seems to work better if we remove the ID3
    tags.

To remove the tags, we use the code in code/id3_tag_lengths.erl, which is in the book's downloadable source code.[2]

Pseudocode for the SHOUTcast Server

Before we look at the final program, let's look at the overall flow of the code with the details omitted.

```
start_parallel_server(Port) ->
    {ok, Listen} = gen_tcp:listen(Port, ..),
    %% create a song server -- this just knows about all our music
    PidSongServer = spawn(fun() -> songs() end),
    spawn(fun() -> par_connect(Listen, PidSongServer) end).

%% spawn one of these processes per connection
par_connect(Listen, PidSongServer) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    %% when accept returns spawn a new process to
    %% wait for the next connection
    spawn(fun() -> par_connect(Listen, PidSongServer) end),
    inet:setopts(Socket, [{packet,0},binary, {nodelay,true},
                          {active, true}]),
    %% deal with the request
    get_request(Socket, PidSongServer, []).

%% wait for the TCP request
get_request(Socket, PidSongServer, L) ->
    receive
        {tcp, Socket, Bin} ->
            ... Bin contains the request from the client
            ... if the request is fragmented we call loop again ...
            ... otherwise we call
            .... got_request(Data, Socket, PidSongServer)
        {tcp_closed, Socket} ->
            ... this happens if the client aborts
            ... before it has sent a request (very unlikely)
    end.

%% we got the request -- send a reply
got_request(Data, Socket, PidSongServer) ->
    .. data is the request from the client ...
    .. analyze it ...
    .. we'll always allow the request ..
    gen_tcp:send(Socket, [response()]),
    play_songs(Socket, PidSongServer).

%% play songs forever or until the client quits
```

---

2. http://pragprog.com/titles/jaerlang2/source_code

```
play_songs(Socket, PidSongServer) ->
     ... PidSongServer keeps a list of all our MP3 files
     Song = rpc(PidSongServer, random_song),
     ... Song is a random song ...
     Header = make_header(Song),
     ... make the header ...
     {ok, S} = file:open(File, [read,binary,raw]),
     send_file(1, S, Header, 1, Socket),
     file:close(S),
     play_songs(Socket, PidSongServer).
send_file(K, S, Header, OffSet, Socket) ->
  ... send the file in chunks to the client ...
  ... returns when the entire file is sent ...
  ... but exits if we get an error when writing to
  ... the socket -- this happens if the client quits
```

If you look at the real code, you'll see the details differ slightly, but the principles are the same. The full code listing is not shown here but is in the file code/shout.erl.

Running the SHOUTcast Server

To run the server and test that it works, we need to perform these three steps:

1. Make a playlist.
2. Start the server.
3. Point a client at the server.

To make the playlist, follow these steps:

1. Change to the code directory.

2. Edit the path in the function start1 in the file mp3_manager.erl to point to the root of the directories that contain the audio files you want to serve.

3. Compile mp3_manager, and give the command mp3_manager:start1(). You should see something like the following:

    ```
    1> c(mp3_manager).
    {ok,mp3_manager}
    2> mp3_manager:start1().
    Dumping term to mp3data
    ok
    ```

    If you're interested, you can now look in the file mp3data to see the results of the analysis.

Now we can start the SHOUTcast server.

```
1> shout:start().
...
```

To test the server, follow these steps:

1. Go to another window to start an audio player, and point it to the stream called http://localhost:3000.

    On my system I use XMMS and give the following command:

    xmms http://localhost:3000

    Note: If you want to access the server from another computer, you'll have to give the IP address of the machine where the server is running. So, for example, to access the server from my Windows machine using Winamp, I use the Play > URL menu in Winamp and enter the address http://192.168.1.168:3000 in the Open URL dialog box.

    On my iMac using iTunes I use the Advanced > Open Stream menu and give the previous URL to access the server.

2. You'll see some diagnostic output in the window where you started the server.

3. Enjoy!

In this chapter, we looked at only the most commonly used functions for manipulating sockets. You can find more information about the socket APIs in the manual pages for gen_tcp, gen_udp, and inet.

The combination of a simple socket interface, together with the BIFs term_to_binary/1 and its inverse binary_to_term, makes networking really easy. I recommend that you work through the following exercises to experience this firsthand.

In the next chapter, we'll look at websockets. Using websockets, an Erlang process can communicate directly with a web browser without following the HTTP protocol. This is ideal for implementing low-latency web applications and provides an easy way to program web applications.

Exercises

1. Modify the code for nano_get_url/0 (in section *Fetching Data from a Server, on page 264*), adding appropriate headers where necessary and performing redirects if needed, in order to fetch any web page. Test this on several sites.

2. Enter the code for *A Simple TCP Server, on page 267*. Then modify the code to receive a {Mod, Func, Args} tuple instead of a string. Finally compute Reply = apply(Mod, Func, Args) and send the value back to the socket.

Write a function `nano_client_eval(Mod, Func, Args)` that is similar to the version shown earlier in this chapter, which encodes `Mod`, `Func`, and `Arity` in a form understood by the modified server code.

Test that the client and server code function correctly, first on the same machine, then on two machines in the same LAN, and then on two machines on the Internet.

3. Repeat the previous exercise using UDP instead of TCP.

4. Add a layer of cryptography by encoding the binary immediately before sending it to the outgoing socket and decoding it immediately after it is received on the incoming socket.

5. Make a simple "email-like" system. Use Erlang terms as messages and store them in a directory `${HOME}/mbox`.

Browsing with Websockets and Erlang

In this chapter, we will see how to build applications in the browser and extend the idea of using message passing to outside Erlang. This way, we can easily build distributed applications and integrate them with a web browser. Erlang thinks that the web browser is just another Erlang process, which simplifies our programming model, putting everything into the same conceptual framework.

We're going to pretend that a web browser is an Erlang process. If we want the browser to do something, we'll send it a message; if something happens within the browser that we need to attend to, the browser will send us a message. All of this is possible thanks to *websockets*. Websockets are part of the HTML5 standard and are bidirectional asynchronous sockets that can be used to pass messages between a browser and an external program. In our case, the external program is the Erlang runtime system.

To interface the Erlang runtime system to websockets, we run a simple Erlang web server, called *cowboy*, to manage the socket and the websocket protocol. Details of how to install cowboy are covered in Chapter 25, *Third-Party Programs*, on page 425. To simplify things, we assume that all messages between Erlang and the browser are JSON messages.

On the Erlang side of the application these messages appear as Erlang maps (see Section 5.3, *Maps: Associative Key-Value Stores*, on page 79), and in the browser these messages appear as JavaScript objects.

In the rest of this chapter, we'll look at six example programs, including the code that runs in the browser and the code that runs in the server. Finally, we'll look at the client-server protocol and see how messages from Erlang to the browser are processed.

To run these examples, we need three things: some code that runs in the browser, some code that runs in an Erlang server, and an Erlang server that understands the websockets protocol. We're not going to look at all the code here; we'll look at the code that runs in the browser and in the server but not the code for the server itself. All the examples can be found at https://github.com/joearms/ezwebframe. The browser code in the examples has been tested only in the Chrome browser.

Note: The code shown here is a simplified version of the code in the ezwebframe repository. The code here is written using maps. The code in the repository is kept in sync with the Erlang distribution and will reflect any changes to Erlang when maps are introduced in version R17 of Erlang (expected in late 2013, but maps will appear in branches on GitHub before the official release).

To run the code yourself, you'll need to download the code and follow the installation interactions. As far as we are concerned, the interesting parts of the code are the part that runs in the browser and the part that runs in the server.

All the examples use a simple technique for controlling the browser from Erlang. If Erlang wants the browser to do something, it just sends the browser a message telling it what to do. If the user wants to do something, they click a button or some other control in the browser and a message is sent to Erlang. The first example shows in detail how this works.

18.1 Creating a Digital Clock

The following image shows the clock running in a browser. All the irrelevant details of the browser window, such as the menus, toolbars, and scrollbars, are not shown so that we can concentrate on the code.

The essential part of this application is the display. This contains a time, which is updated every second. From the Erlang point of view, the entire browser is a process; so, to update the clock to the value shown earlier, Erlang sent the browser the following message:

```
Browser ! #{ cmd => fill_div, id => clock, txt => <<"16:30:52">> }
```

Inside the browser, we have loaded an HTML page with a small fragment of HTML like this:

```
<div id='clock'>
  ...
</div>
```

When the browser receives a fill_div, it converts this into the JavaScript command fill_div({cmd:'fill_div', id:'clock', txt:'16:30:52'}), which then fills the content of the div with the required string.

Note how the Erlang message containing a frame gets converted to an equivalent JavaScript function call, which is evaluated in the browser. Extending the system is extremely easy. All you have to do is write a small JavaScript function corresponding to the Erlang message that you need to process.

To complete the picture, we need to add the code that starts and stops the clock. Putting everything together, the HTML code looks like this:

```
websockets/clock1.html
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./clock1.css" type="text/css">
<body>
  <div id="clock"></div>
  <button id="start" class="live_button">start</button>
  <button id="stop"  class="live_button">stop</button>
</body>
<script>
$(document).ready(function(){
   connect("localhost", 2233, "clock1");
   });
</script>
```

First, we load two JavaScript libraries and a style sheet. clock1.css is used to style the display of the clock.

Second, there is some HTML that creates the display. Finally, we have a small fragment of JavaScript that is run when the page is loaded.

Note: In all our examples we assume some familiarity with jQuery. jQuery (http://jquery.com) is an extremely popular JavaScript library that simplifies manipulating objects in the browser.

websock.js has all the code necessary for opening a websocket and connecting the objects in the browser DOM to Erlang. It does the following:

1. Adds click handlers to all buttons with class live_button in the page. The click handlers send messages to Erlang when the buttons are clicked.

2. Tries to start a websocket connection to http://localhost:2233. On the server side, the function clock1:start(Browser) will be called in a freshly spawned process. All this is achieved by calling the JavaScript function connect("localhost", 2233, "clock1"). The number 2233 has no particular significance; any unused port number over 1023 would do.

Now here's the Erlang code:

```
websockets/clock1.erl
-module(clock1).
-export([start/1, current_time/0]).

start(Browser) ->
    Browser ! #{ cmd => fill_div, id => clock, txt => current_time() },
    running(Browser).

running(Browser) ->
    receive
        {Browser, #{ clicked => <<"stop">>} } ->
            idle(Browser)
    after 1000 ->
            Browser ! #{ cmd => fill_div, id => clock, txt => current_time() },
            running(Browser)
    end.

idle(Browser) ->
    receive
        {Browser, #{clicked => <<"start">>} } ->
            running(Browser)
    end.

current_time() ->
    {Hour,Min,Sec} = time(),
    list_to_binary(io_lib:format("~2.2.0w:~2.2.0w:~2.2.0w",
                                 [Hour,Min,Sec])).
```

The Erlang code begins execution in start(Browser); Browser is a process representing the browser. This is the first interesting line of code:

```
Browser ! #{ cmd => fill_div, id => clock, txt => current_time() }
```

This updates the display. I've repeated this line for emphasis. My editor told me to remove it. But no. To me this is very beautiful code. To get the browser to do something, we send it a message. *Just like Erlang.* We've tamed the browser. It looks like an Erlang processes. Whoopeee.

After the initializing, clock1 calls running/1. If a {clicked => <<"stop">>} message is received, then we call idle(Browser). Otherwise, after a timeout of one second, we send a command to the browser telling it to update the clock and call ourselves.

idle/1 waits for a start message and then calls running/1.

18.2 Basic Interaction

Our next example has a scrollable text area for displaying data and an entry. When you enter text in the entry and press the carriage return, a message is sent to the browser. The browser responds with a message that updates the display.
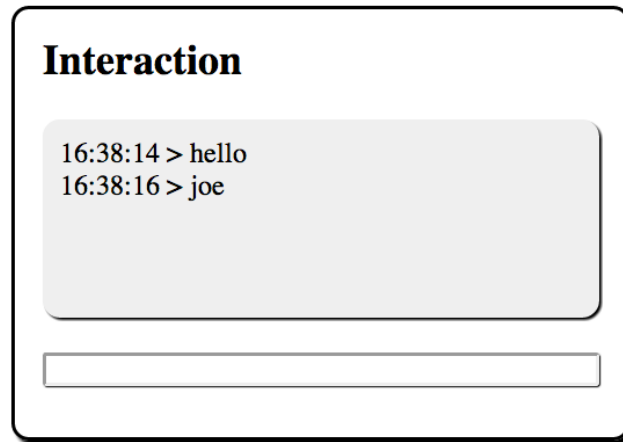
The HTML code for this is as follows:

```
websockets/interact1.html
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./interact1.css" type="text/css">
<body>
  <h2>Interaction</h2>
  <div id="scroll"></div>
  <br>
  <input id="input" class="live_input"></input>
</body>
<script>
$(document).ready(function(){
    connect("localhost", 2233, "interact1");
    });
</script>
```

And here is the Erlang:

```
websockets/interact1.erl
-module(interact1).
-export([start/1]).

start(Browser) -> running(Browser).

running(Browser) ->
    receive
        {Browser, #{entry => <<"input">>, txt => Bin} }
            Time = clock1:current_time(),
            Browser ! #{cmd => append_div, id => scroll,
                        txt => list_to_binary([Time, " > ", Bin, "<br>"])}
    end,
    running(Browser).
```

This works in a similar manner to the clock example. The entry sends a message containing the text in the entry to the browser each time the user hits Enter in the entry. The Erlang process that manages the window receives this message and sends a message back to the browser that causes the display to update.

18.3 An Erlang Shell in the Browser

We can use the code in the interface pattern to make an Erlang shell that runs in the browser.
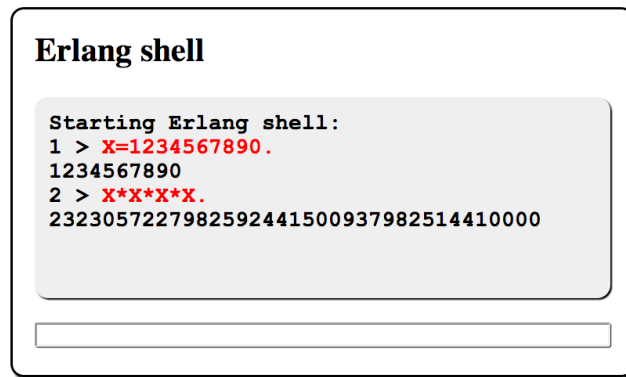
We won't show all the code since it is similar to that in the interaction example. These are the relevant parts of the code:

```
websockets/shell1.erl
start(Browser) ->
    Browser ! #{cmd => append_div, id => scroll,
                txt => <<"Starting Erlang shell:<br>">>},
    B0 = erl_eval:new_bindings(),
    running(Browser, B0, 1).
running(Browser, B0, N) ->
```

```
receive
    {Browser, #{entry => <<"input">>}, txt => Bin}} ->
        {Value, B1} = string2value(binary_to_list(Bin), B0),
        BV = bf("~w > <font color='red'>~s</font><br>~p<br>",
                [N, Bin, Value]),
        Browser ! #{cmd => append_div, id => scroll, txt => BV},
        running(Browser, B1, N+1)
end.
```

The tricky bit is done in the code that parses and evaluates the input string.

```
websockets/shell1.erl
string2value(Str, Bindings0) ->
    case erl_scan:string(Str, 0) of
      {ok, Tokens, _} ->
        case erl_parse:parse_exprs(Tokens) of
          {ok, Exprs} ->
            {value, Val, Bindings1} = erl_eval:exprs(Exprs, Bindings0),
              {Val, Bindings1};
          Other ->
            io:format("cannot parse:~p Reason=~p~n",[Tokens,Other]),
                      {parse_error, Bindings0}
        end;
      Other ->
        io:format("cannot tokenise:~p Reason=~p~n",[Str,Other])
    end.
```

And now we have an Erlang shell running in the browser. Admittedly it's a very basic shell, but it illustrates all the techniques necessary to build a much more sophisticated shell.

18.4 Creating a Chat Widget

In the next section in this chapter, we will develop an IRC control program. This program needs a chat widget:
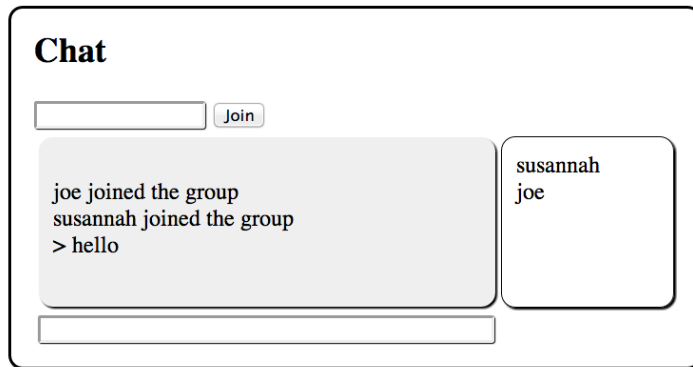
The code that creates the widget is as follows:

```
websockets/chat1.html
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./chat1.css" type="text/css">

<body>
  <h2>Chat</h2>
  <input id="nick_input"/>
  <button id="join">Join</button>
  <br/>
  <table>
    <tr>
      <td><div id="scroll"></div></td>
      <td><div id="users"></div></td>
    </tr>
    <tr>
      <td colspan="2">
        <input id="tell" class="live_input"/>
      </td>
    </tr>
  </table>
</body>

<script>
$(document).ready(function(){
  $("#join").click(function(){
      var val = $("#nick_input").val();
      send_json({'join':val});
      $("#nick_input").val("");
      });
    connect("localhost", 2233, "chat1");
});
</script>
```

This code is broadly similar to the code in the previous examples. The only difference is in the use of the Join button. When we click the Join button, we want to perform a local action in the browser and not send a message to the controller. Using JQuery, the code for this is as follows:

```
$("#join").click(function(){
    var val = $("#nick_input").val();
    send_json({'join':val});
    $("#nick_input").val("");
  })
```

This code hooks an event handler onto the Join button. When the Join button is clicked, we read the nickname entry field, send a join message to Erlang, and clear the entry.

The Erlang side of things is equally simple. We have to respond to two messages: a join message sent when the user clicks Join and a tell message when the user hits the carriage return in the entry field at the bottom of the widget.

To test the widget, we can use the following code:

```erlang
websockets/chat1.erl
-module(chat1).
-export([start/1]).

start(Browser) ->
    running(Browser, []).

running(Browser, L) ->
    receive
        {Browser, #{join => Who}} ->
            Browser ! #{cmd => append_div ,id => scroll,
                        txt => list_to_binary([Who, " joined the group\n"])},
            L1 = [Who,"<br>"|L],
            Browser ! #{cmd => fill_div, id => users,
                        txt => list_to_binary(L1)},
            running(Browser, L1);
        {Browser,#{entry => <<"tell">>, txt => Txt}} ->
            Browser ! #{cmd => append_div, id => scroll,
                        txt => list_to_binary([" > ", Txt, "<br>"])},
            running(Browser, L);
        X ->
            io:format("chat received:~p~n",[X])
    end,
    running(Browser, L).
```

This is not the real code that will control the IRC application but just a test stub. When the join message is received, the scroll region is updated, and the list of users in the user's div is changed. When the tell message is received, only the scroll area is changed.

18.5 IRC Lite

The chat widget of the previous section can be easily extended to make a more realistic chat program. To do so, we'll change the code for the chat widget to the following:

```html
websockets/chat2.html
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./chat1.css" type="text/css">

<body>
  <h2>Chat</h2>
  <div id="idle">
```

```
      <input id="nick_input"/>
      <button id="join">Join</button>
      <br/>
  </div>

  <div id="running">
    <table>
      <tr>
        <td><div id="scroll"></div></td>
        <td><div id="users"></div></td>
      </tr>
      <tr>
        <td colspan="2">
          <input id="tell" class="live_input"/><br/>
          <button class="live_button" id="leave">Leave</button>
        </td>
      </tr>
    </table>
  </div>
</body>

<script>
$(document).ready(function(){
  $("#running").hide();
  $("#join").click(function(){
      var val = $("#nick_input").val();
      send_json({'join':val});
      $("#nick_input").val("");
      });

  connect("localhost", 2233, "chat2");
  });

function hide_div(o){
  $("#" + o.id).hide();
}

function show_div(o){
  $("#" + o.id).show();
}
</script>
```

This code has two main divs called idle and running. One of these is hidden,
and the other is displayed. When the user clicks the Join button, a request
is made to the IRC server to join the chat. If the username is not used, the
server replies with a welcome message, and the chat handler program hides
the idle div and displays the running div. The corresponding Erlang code is as
follows:

```erlang
websockets/chat2.erl
-module(chat2).
-export([start/1]).

start(Browser) ->
    idle(Browser).

idle(Browser) ->
    receive
        {Browser, #{join => Who}} ->
            irc ! {join, self(), Who},
            idle(Browser);
        {irc, welcome, Who} ->
            Browser ! #{cmd => hide_div, id => idle},
            Browser ! #{cmd => show_div, id => running},
            running(Browser, Who);
        X ->
            io:format("chat idle received:~p~n",[X]),
            idle(Browser)
    end.

running(Browser, Who) ->
    receive
        {Browser,#{entry => <<"tell">>, txt => Txt}} ->
            irc ! {broadcast, Who, Txt},
            running(Browser, Who);
        {Browser,#{clicked => <<"Leave">>}} ->
            irc ! {leave, Who},
            Browser ! #{cmd => hide_div, id => running},
            Browser ! #{cmd => show_div, id => idle},
            idle(Browser);
        {irc, scroll, Bin} ->
            Browser ! #{cmd => append_div, id => scroll, txt => Bin},
            running(Browser, Who);
        {irc, groups, Bin} ->
            Browser ! #{cmd => fill_div, id => users, txt => Bin},
            running(Browser, Who);
        X ->
            io:format("chat running received:~p~n",[X]),
            running(Browser, Who)
    end.
```

In the running state, the chat controller can receive one of four messages. Two messages come from the browser; a tell message is received when the user types a message into the chat entry field, and a leave message is received if the user clicks the Leave button. These messages are relayed to the IRC server, and in the case of the leave message, further messages are sent to the browser to hide the running div and show the idle div, so we are back to where we started.

The other two messages come from the IRC server and tell the controller to update either the scroll region or the list of users.

The code for the IRC controller is pretty simple.

```
websockets/irc.erl
-module(irc).
-export([start/0]).

start() ->
    register(irc, spawn(fun() -> start1() end)).

start1() ->
    process_flag(trap_exit, true),
    loop([]).

loop(L) ->
    receive
        {join, Pid, Who} ->
            case lists:keysearch(Who,1,L) of
                false ->
                    L1 = L ++ [{Who,Pid}],
                    Pid ! {irc, welcome, Who},
                    Msg = [Who, <<" joined the chat<br>">>],
                    broadcast(L1, scroll, list_to_binary(Msg)),
                    broadcast(L1, groups, list_users(L1)),
                    loop(L1);
                {value,_} ->
                    Pid ! {irc, error, <<"Name taken">>},
                    loop(L)
            end;
        {leave, Who} ->
            case lists:keysearch(Who,1,L) of
                false ->
                    loop(L);
                {value,{Who,Pid}} ->
                    L1 = L -- [{Who,Pid}],
                    Msg = [Who, <<" left the chat<br>">>],
                    broadcast(L1, scroll, list_to_binary(Msg)),
                    broadcast(L1, groups, list_users(L1)),
                    loop(L1)
            end;
        {broadcast, Who, Txt} ->
            broadcast(L, scroll,
                    list_to_binary([" > ", Who, " >> ", Txt, "<br>"])),
            loop(L);
        X ->
            io:format("irc:received:~p~n",[X]),
            loop(L)
    end.
```

```
broadcast(L, Tag, B) ->
    [Pid ! {irc, Tag, B} || {_,Pid} <- L].

list_users(L) ->
    L1 = [[Who,"<br>"] || {Who,_}<- L],
    list_to_binary(L1).
```

This code needs little explanation. If the IRC server receives a join message and the username is unused, it broadcasts a new list of users to all connected users and a join message to the scroll region of all connected users. If a leave message is received, it removes the user from the list of current users and broadcasts information about this to all connected users.

To run this in a distributed system, one machine needs to host the IRC server. All other machines need to know the IP address or the hostname of the machine running the server. So, for example, if the IP address of the machine running the IRC server is AAA.BBB.CCC.DDD, then all machines should request the page with URL http://AAA.BBB.CCC.DDD:2233/chat2.html (port 2233 is the default port number).

18.6 Graphics in the Browser

So far, we've seen only text in the browser. As soon as we can push graphic objects to the browser, a whole new world opens. This is really easy, since we can use the Scalable Vector Graphics (SVG) format that is built into modern browsers.
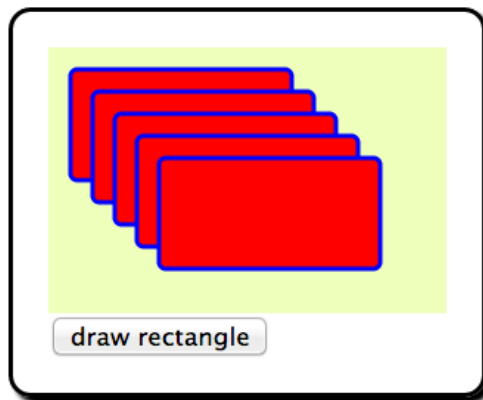
Here's what we are going to do. We'll create an SVG canvas in the browser with a button below it. When the button is clicked, a message is sent to Erlang. Erlang replies with a command telling the browser to add a rectangle to the SVG canvas. The previous screenshot shows the widget after clicking the draw rectangle button five times. The Erlang code to do this is as follows:

```
websockets/svg1.erl
-module(svg1).
-export([start/1]).

start(Browser) ->
    Browser ! #{cmd => add_canvas, tag => svg, width => 180, height => 120},
    running(Browser, 10, 10).

running(Browser, X, Y) ->
    receive
        {Browser,#{clicked => <<"draw rectangle">>}} ->
            Browser ! #{cmd => add_svg_thing, type => rect,
                        rx => 3, ry => 3, x => X, y => Y,
                        width => 100,height => 50,
                        stroke => blue,'stroke-width' => 2,
                        fill =>  red},
            running(Browser, X+10, Y+10)
    end.
```

The Erlang code sends two messages to the browser: [{cmd,add_canvas}, ... and
[{cmd,add_svg_thing}, ...]. So, in the JavaScript that is loaded into the browser,
we have to provide definitions of these commands.

```
websockets/svg1.html
<script type="text/javascript" src="./jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="./websock.js"></script>
<link rel="stylesheet" href="./svg1.css" type="text/css">
<body>
  <div id="svg"></div>
  <button id="start" class="live_button">draw rectangle</button>
</body>

<script>
$(document).ready(function(){
   connect("localhost", 2233, "svg1");
   });

var canvas;
var svg_ns='http://www.w3.org/2000/svg';

function add_canvas(o){
    canvas = document.createElementNS(svg_ns, 'svg');
    canvas.setAttribute("width",  o.width);
    canvas.setAttribute("height", o.height );
    canvas.setAttribute("style",  "background-color:#eeffbb");
    $('#'+o.tag).append(canvas);
}

function add_svg_thing(o){
    var obj = document.createElementNS(svg_ns, o.type);
```

```
    for(key in o){
        var val = o[key];
        obj.setAttributeNS(null, key, val);
    };
    canvas.appendChild(obj);
}
</script>
```

These two functions are all you need to get started with SVG graphics. Note that when we sent a command to draw a rectangle to the browser, the command contained a lot of parameters. Attributes like rx and ry are used to make rounded corners on the rectangle. A full list of these attributes can be found in the W3C specification.[1]

18.7 The Browser Server Protocol

The Browser server protocol is extremely simple. It makes use of JSON messages sent over a websocket. This fits nicely with both Erlang and JavaScript and makes interfacing the browser to Erlang really easy.

Sending a Message from Erlang to the Browser

To change something in the browser, we send a message from Erlang to the browser. Suppose we have a div in the browser, declared like this:

```
<div id="id123"></div>
```

To change the content of the div to the string abc, Erlang sends the following JSON message to the websocket that is connected to the browser:

```
[{cmd:'fill_div', id:'id123', txt:'abc'}]
```

The code in webserver.js, which is activated when the browser receives a message from a websocket, is a callback routine called onMessage. It is set up with the following code:

```
websocket = new WebSocket(wsUri);
...
websocket.onmessage = onMessage;
```

The callback is defined like this:

```
function onMessage(evt) {
    var json = JSON.parse(evt.data);
    do_cmds(json);
}

function do_cmds(objs){
```

---

1. http://www.w3.org/TR/SVG/

```
  for(var i = 0; i < objs.length; i++){
     var o = objs[i];
     if(eval("typeof("+o.cmd+")") == "function"){
       eval(o.cmd + "(o)");
     } else {
       alert("bad_command:"+o.cmd);
     };
  };
}
```

do_cmds(objs) expects to receive a list of commands of the following form:

```
[{cmd:command1, ...:..., ...:...},
 {cmd:command2, ...:..., ...:...},
 ...
 {cmd:commandN, ...:..., ...:...}]
```

Each command in the list is a JavaScript object, which *must* contain a key called cmd. For each object x in the list, the system checks whether there is a function called x.cmd, and if there is, it calls x.cmd(x). So, {cmd:'fill_div', id:'id123', txt:'abc'} causes the following:

```
fill_div({cmd:'fill_div', id:'id123', txt:'abc'})
```

to be called. This method of encoding and evaluating commands is easily extensible so we can add more commands to the interface as necessary.

Messages from the Browser to Erlang

In the browser when we click a button, we evaluate commands like this:

```
send_json({'clicked':txt}));
```

send_json(x) encodes the argument x as a JSON term and writes it to the websocket. This message is received in websocket.erl where it is converted to a frame and sent to the controlling process that manages the websocket.

We have seen how to extend the notion of message passing to outside Erlang and use message passing to directly control a browser. From the point of view of an Erlang programmer, the world now consists of a well-ordered space where everything responds to Erlang messages. We don't have one way of doing things inside Erlang and another way of doing things outside Erlang. This adds a sense of order and unity to our programs, making a complex world appear simple. A web browser is, of course, a massively complicated object. But by making it respond in a predictable manner to a small number of messages, we can easily contain this complexity and harness it to build powerful applications.

In this chapter, we made the web browser look like an Erlang process. We can send messages to the browser to get it to do things, and when things happen in the browser, we get sent messages. In Erlang we use maps to represent messages, which get encoded as JSON messages and appear in the browser as JavaScript objects. The small conceptual gap between the Erlang and JavaScript representations of the messages simplifies programming since we don't have to be bothered with the details of changing representations.

Now we'll change the subject. The next two chapters have to do with storing large volumes of data. The first of these details the low-level storage modules ets and dets. The second details the Erlang database mnesia, which is implemented using ets and dets.

Exercises

1. The process started in shell1.erl is very primitive. If it crashes, the web application locks and cannot proceed. Add error recovery to the application. Add features to replay old commands.

2. Read the code in websockets.js and trace exactly what happens when you click a live button in the browser. Follow the code through the JavaScript to the websocket and then from the websocket to Erlang. How does the message you get when you click a button find the Erlang controlling processes involved?

3. The IRC lite program is a fully functioning chat program. Try running it and check that it works. You might find that it does not work because firewalls, and so on, are blocking access to the server. If so, investigate this and see whether you can open up the firewalls. Try to find the specification of the real IRC protocol, and you'll find it's a lot longer than the version here. Why is this? Extend the irc system with an authentication system for users.

4. The IRC program uses a centralized server. Could you change it so as to eliminate the central server and use a network of peers? How about adding SVG graphics to the chat client or using the audio interface in HTML5 to send and receive sounds?

Storing Data with ETS and DETS

ets and dets are two system modules that you can use for the efficient storage of large numbers of Erlang terms. ETS is short for *Erlang term storage*, and DETS is short for *disk ETS*.

ETS and DETS perform basically the same task: they provide large key-value lookup tables. ETS is memory resident, while DETS is disk resident. ETS is highly efficient—using ETS, you can store colossal amounts of data (if you have enough memory) and perform lookups in constant (or in some cases logarithmic) time. DETS provides almost the same interface as ETS but stores the tables on disk. Because DETS uses disk storage, it is far slower than ETS but will have a much smaller memory footprint when running. In addition, ETS and DETS tables can be shared by several processes, making interprocess access to common data highly efficient.

ETS and DETS tables are data structures for associating *keys* with *values*. The most commonly performed operations on tables are *insertions* and *lookups*. An ETS or DETS table is just a collection of Erlang tuples.

Data stored in an ETS table is stored in RAM and is *transient*. The data will be deleted when the ETS table is disposed of or the owning Erlang process terminates. Data stored in DETS tables is *persistent* and should survive an entire system crash. When a DETS table is opened, it is checked for consistency. If it is found to be corrupt, then an attempt is made to repair the table (which can take a long time since all the data in the table is checked).

This should recover all data in the table, though the last entry in the table might be lost if it was being made at the time of the system crash.

ETS tables are widely used in applications that have to manipulate large amounts of data in an efficient manner and where it is too costly to program with nondestructive assignment and "pure" Erlang data structures.

ETS tables look as if they were implemented in Erlang, but in fact they are implemented in the underlying runtime system and have different performance characteristics than ordinary Erlang objects. In particular, ETS tables are not garbage collected; this means there are no garbage collection penalties involved in using extremely large ETS tables, though slight penalties are incurred when we create or access ETS objects.

19.1 Types of Table

ETS and DETS tables store tuples. One of the elements in the tuple (by default, the first) is called the *key* of the table. We insert tuples into the table and extract tuples from the table based on the key. What happens when we insert a tuple into a table depends upon the type of the table and the value of the key. Some tables, called *sets*, require that all the keys in the table are unique. Others, called *bags*, allow several tuples to have the same key.

Choosing the correct type of table has important consequences for the performance of your applications.

Each of the basic set and bag table types has two variants, making for a total of four types of table: sets, ordered sets, bags, and duplicate bags. In a set, all the keys in the different tuples in the table must be unique. In an ordered set, the tuples are sorted. In a bag there can be more than one tuple with the same key, but no two tuples in the bag can be identical. In a duplicate bag several tuples can have the same key, and the same tuple can occur many times in the same table.

There are four basic operations on ETS and DETS tables.

Create a new table or open an existing table.
    This we do with ets:new or dets:open_file.

Insert a tuple or several tuples into a table.
    Here we call insert(TableId, X), where X is a tuple or a list of tuples. insert has the same arguments and works the same way in ETS and DETS.

Look up a tuple in a table.
    Here we call lookup(TableID, Key). The result is a list of tuples matching Key. lookup is defined for both ETS and DETS.

    The return value of lookup is always a list of tuples. This is so we can use the same lookup function on bags and sets. If the table type is a bag, then several tuples can have the same key, but if the table type is a set, then there will be only one element in the list if the lookup succeeds. We'll look at the table types in the next section.

If no tuples in the table have the required key, then an empty list is returned.

Dispose of a table.

When we've finished with a table, we can tell the system by calling dets:close(TableId) or ets:delete(TableId).

We can illustrate how these work with the following little test program:

```
ets_test.erl
-module(ets_test).
-export([start/0]).

start() ->
    lists:foreach(fun test_ets/1,
                  [set, ordered_set, bag, duplicate_bag]).

test_ets(Mode) ->
    TableId = ets:new(test, [Mode]),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {b,2}),
    ets:insert(TableId, {a,1}),
    ets:insert(TableId, {a,3}),
    List = ets:tab2list(TableId),
    io:format("~-13w => ~p~n", [Mode, List]),
    ets:delete(TableId).
```

This program creates an ETS table in one of four modes and inserts the tuples {a,1}, {b,2}, {a,1}, and finally {a,3} into the table. Then we call tab2list, which converts the entire table into a list and prints it.

When we run this, we get the following output:

```
1> ets_test:start().
set           => [{b,2},{a,3}]
ordered_set   => [{a,3},{b,2}]
bag           => [{b,2},{a,1},{a,3}]
duplicate_bag => [{b,2},{a,1},{a,1},{a,3}]
```

For the set table type, each key occurs only once. If we insert the tuple {a,1} in the table followed by {a,3}, then the final value will be {a,3}. The only difference between a set and an ordered set is that the elements in an ordered set are ordered by the key. We can see the order when we convert the table to a list by calling tab2list.

The bag table types can have multiple occurrences of the key. So, for example, when we insert {a,1} followed by {a,3}, then the bag will contain both tuples, not just the last. In a duplicate bag, multiple identical tuples are allowed in the bag, so when we insert {a,1} followed by {a,1} into the bag, then the

resulting table contains two copies of the {a,1} tuple; however, in a regular bag, there would be only one copy of the tuple.

19.2 ETS Table Efficiency Considerations

Internally, ETS tables are represented by hash tables (except ordered sets, which are represented by balanced binary trees). This means there is a slight space penalty for using sets and a time penalty for using ordered sets. Inserting into sets takes place in constant time, but inserting into an ordered set takes place in a time proportional to the log of the number of entries in the table.

When you choose between a set and an ordered set, you should think about what you want to do with the table after it has been constructed—if you want a sorted table, then use an ordered set.

Bags are more expensive to use than duplicate bags, since on each insertion all elements with the same key have to be compared for equality. If there are large numbers of tuples with the same key, this can be rather inefficient.

ETS tables are stored in a separate storage area that is not associated with normal process memory. An ETS table is said to be owned by the process that created it—when that process dies or when ets:delete is called, then the table is deleted. ETS tables are not garbage collected, which means that large amounts of data can be stored in the table without incurring garbage collection penalties.

When a tuple is inserted into an ETS table, all the data structures representing the tuple are copied from the process stack and heap into the ETS table. When a lookup operation is performed on a table, the resultant tuples are copied from the ETS table to the stack and heap of the process.

This is true for all data structures except large binaries. Large binaries are stored in their own off-heap storage area. This area can be shared by several processes and ETS tables, and the individual binaries are managed with a reference-counting garbage collector that keeps track of how many different processes and ETS tables use the binary. When the use count for the number of processes and tables that use a particular binary goes down to zero, then the storage area for the binary can be reclaimed.

All of this might sound rather complicated, but the upshot is that sending messages between processes that contain large binaries is very cheap, and inserting tuples into ETS tables that contain binaries is also very cheap. A good rule is to use binaries as much as possible for representing strings and large blocks of untyped memory.

19.3 Creating an ETS Table

You create ETS tables by calling ets:new. The process that creates the table is called the *owner* of the table. When you create the table, it has a set of options that cannot be changed. If the owner process dies, space for the table is automatically deallocated. You can delete the table by calling ets:delete.

The arguments to ets:new are as follows:

-spec ets:new(Name, [Opt]) -> TableId

> Name is an atom. [Opt] is a list of options, taken from the following:

> set | ordered_set | bag | duplicate_bag
>> This creates an ETS table of the given type (we talked about these earlier).

> private
>> This creates a private table. Only the owner process can read and write this table.

> public
>> This creates a public table. Any process that knows the table identifier can read and write this table.

> protected
>> This creates a protected table. Any process that knows the table identifier can read this table, but only the owner process can write to the table.

> named_table
>> If this is present, then Name can be used for subsequent table operations.

> {keypos, K}
>> Use K as the key position. Normally position 1 is used for the key. Probably the only time when we would use this option is if we store an Erlang record (which is actually a disguised tuple), where the first element of the record contains the record name.

Note: Opening an ETS table with zero options is the same as opening it with the options [set,protected,{keypos,1}].

All the code in this chapter uses protected ETS tables. Protected tables are particularly useful since they allow data sharing at virtually zero cost. All local processes that know the table identifier can read the data, but only one process can change the data in the table.

> ### ETS Tables As Blackboards
>
> Protected tables provide a type of "blackboard system." You can think of a protected ETS table as a kind of named blackboard. Anybody who knows the name of the blackboard can read the blackboard, but only the owner can write on the blackboard.
>
> *Note*: An ETS table that has been opened in `public` mode can be written and read by any process that knows the table name. In this case, the user must ensure that reads and writes to the table are performed in a consistent manner.

19.4 Example Programs with ETS

The examples in this section have to do with trigram generation. This is a nice "show-off" program that demonstrates the power of the ETS tables.

Our goal is to write a heuristic program that tries to predict whether a given string is an English word.

To predict whether a random sequence of letters is an English word, we'll analyze which *trigrams* occur in the word. A trigram is a sequence of three letters. Now, not all sequences of three letters can occur in a valid English word. For example, there are no English words where the three-letter combinations *akj* and *rwb* occur. So, to test whether a string might be an English word, all we have to do is test all sequences of three consecutive letters in the string against the set of trigrams generated from a large set of English words.

The first thing our program does is to compute all trigrams in the English language from a very large set of words. To do this, we use ETS sets. The decision to use an ETS set is based on a set of measurements of the relative performances of ETS sets and ordered sets and of using "pure" Erlang sets as provided by the sets module.

This is what we're going to do in the next few sections:

1. Make an *iterator* that runs through all the trigrams in the English language. This will greatly simplify writing code to insert the trigrams into different table types.

2. Create ETS tables of type `set` and `ordered_set` to represent all these trigrams. Also, build a set containing all these trigrams.

3. Measure the time to *build* these different tables.

4. Measure the time to *access* these different tables.

5. Based on the *measurements*, choose the best method and write access routines for the best method.

All the code is in lib_trigrams. We're going to present this in sections, leaving out some of the details. But don't worry, the complete code is in the file code/lib_trigrams.erl available from the book's home page.[1]

The Trigram Iterator

We'll define a function called for_each_trigram_in_the_english_language(F, A). This function applies the fun F to every trigram in the English language. F is a fun of type fun(Str, A) -> A, Str ranges over all trigrams in the language, and A is an accumulator.

To write our iterator, we need a massive word list. (Note: I've called this an iterator here; to be more strict, it's actually a fold operator very much like lists:foldl.) I've used a collection of 354,984 English words[2] to generate the trigrams. Using this word list, we can define the trigram iterator as follows:

lib_trigrams.erl
```erlang
for_each_trigram_in_the_english_language(F, A0) ->
    {ok, Bin0} = file:read_file("354984si.ngl.gz"),
    Bin = zlib:gunzip(Bin0),
    scan_word_list(binary_to_list(Bin), F, A0).

scan_word_list([], _, A) ->
    A;
scan_word_list(L, F, A) ->
    {Word, L1} = get_next_word(L, []),
    A1 = scan_trigrams([$\s|Word], F, A),
    scan_word_list(L1, F, A1).

%% scan the word looking for \r\n
%% the second argument is the word (reversed) so it
%% has to be reversed when we find \r\n or run out of characters

get_next_word([$\r,$\n|T], L) -> {reverse([$\s|L]), T};
get_next_word([H|T], L)       -> get_next_word(T, [H|L]);
get_next_word([], L)          -> {reverse([$\s|L]), []}.

scan_trigrams([X,Y,Z], F, A) ->
    F([X,Y,Z], A);
scan_trigrams([X,Y,Z|T], F, A) ->
    A1 = F([X,Y,Z], A),
    scan_trigrams([Y,Z|T], F, A1);
scan_trigrams(_, _, A) ->
    A.
```

---

1. http://pragprog.com/titles/jaerlang2/source_code
2. http://www.dcs.shef.ac.uk/research/ilash/Moby/.

Note two points here. First, we used zlib:gunzip(Bin) to unzip the binary in the source file. The word list is rather long, so we prefer to save it on disk as a compressed file rather than as a raw ASCII file. Second, we add a space before and after each word; in our trigram analysis, we want to treat space as if it were a regular letter.

Build the Tables

We build our ETS tables like this:

lib_trigrams.erl
```
make_ets_ordered_set() -> make_a_set(ordered_set, "trigramsOS.tab").
make_ets_set()         -> make_a_set(set, "trigramsS.tab").

make_a_set(Type, FileName) ->
    Tab = ets:new(table, [Type]),
    F = fun(Str, _) -> ets:insert(Tab, {list_to_binary(Str)}) end,
    for_each_trigram_in_the_english_language(F, 0),
    ets:tab2file(Tab, FileName),
    Size = ets:info(Tab, size),
    ets:delete(Tab),
    Size.
```

Note how when we have isolated a trigram of three letters, ABC, we actually insert the tuple {<<"ABC">>} into the ETS table representing the trigrams. This looks funny—a tuple with only *one* element. Normally a tuple is a container for several elements, so it doesn't make sense to have a tuple with only one element. But remember all the entries in an ETS table are tuples, and by default the key in a tuple is the first element in the tuple. So, in our case, the tuple {Key} represents a key with no value.

Now for the code that builds a set of all trigrams (this time with the Erlang module sets and not ETS):

lib_trigrams.erl
```
make_mod_set() ->
    D = sets:new(),
    F = fun(Str, Set) -> sets:add_element(list_to_binary(Str),Set) end,
    D1 = for_each_trigram_in_the_english_language(F, D),
    file:write_file("trigrams.set", [term_to_binary(D1)]).
```

Table-Building Time

The function lib_trigrams:make_tables(), shown in the listing at the end of the chapter, builds all the tables. It includes some instrumentation so we can measure the size of our tables and the time taken to build the tables.

```
1> lib_trigrams:make_tables().
Counting - No of trigrams=3357707 time/trigram=0.577938
Ets ordered Set size=19.0200 time/trigram=2.98026
Ets set size=19.0193 time/trigram=1.53711
Module Set size=9.43407 time/trigram=9.32234
ok
```

This tells us that there were 3.3 million trigrams, and it took half a microsecond to process each trigram in the word list.

The insertion time per trigram was 2.9 microseconds in an ETS ordered set, 1.5 microseconds in an ETS set, and 9.3 microseconds in an Erlang set. As for storage, ETS sets and ordered sets took 19 bytes per trigram, while the module sets took 9 bytes per trigram.

Table Access Times

OK, so the tables took some time to build, but in this case *it doesn't matter*. Now we'll write some code to measure the access times. We'll look up every trigram in our table exactly once and then take the average time per lookup. Here's the code that performs the timings:

lib_trigrams.erl
```erlang
timer_tests() ->
    time_lookup_ets_set("Ets ordered Set", "trigramsOS.tab"),
    time_lookup_ets_set("Ets set", "trigramsS.tab"),
    time_lookup_module_sets().

time_lookup_ets_set(Type, File) ->
    {ok, Tab} = ets:file2tab(File),
    L = ets:tab2list(Tab),
    Size = length(L),
    {M, _} = timer:tc(?MODULE, lookup_all_ets, [Tab, L]),
    io:format("~s lookup=~p micro seconds~n",[Type, M/Size]),
    ets:delete(Tab).

lookup_all_ets(Tab, L) ->
    lists:foreach(fun({K}) -> ets:lookup(Tab, K) end, L).

time_lookup_module_sets() ->
    {ok, Bin} = file:read_file("trigrams.set"),
    Set = binary_to_term(Bin),
    Keys = sets:to_list(Set),
    Size = length(Keys),
    {M, _} = timer:tc(?MODULE, lookup_all_set, [Set, Keys]),
    io:format("Module set lookup=~p micro seconds~n",[M/Size]).

lookup_all_set(Set, L) ->
    lists:foreach(fun(Key) -> sets:is_element(Key, Set) end, L).
```

Here we go:

```
1> lib_trigrams:timer_tests().
Ets ordered Set lookup=1.79964 micro seconds
Ets set lookup=0.719279 micro seconds
Module sets lookup=1.35268 micro seconds
ok
```

These timings are in average microseconds per lookup.

And the Winner Is…

Well, it was a walkover. The ETS set won by a large margin. On my machine, sets took about half a microsecond per lookup—that's pretty good!

Note: Performing tests like the previous one and actually measuring how long a particular operation takes is considered good programming practice. We don't need to take this to extremes and time everything, only the most time-consuming operations in our program. The non-time-consuming operations should be programmed in the most *beautiful* way possible. If we are forced to write nonobvious ugly code for efficiency reasons, then it should be well documented.

Now we can write the routines that try to predict whether a string is a proper English word.

To test whether a string might be an English language word, we scan through all the trigrams in the string and check that each trigram occurs in the trigram table that we computed earlier. The function is_word does this.

lib_trigrams.erl
```erlang
is_word(Tab, Str) -> is_word1(Tab, "\s" ++ Str ++ "\s").
is_word1(Tab, [_,_,_]=X) -> is_this_a_trigram(Tab, X);
is_word1(Tab, [A,B,C|D]) ->
    case is_this_a_trigram(Tab, [A,B,C]) of
        true  -> is_word1(Tab, [B,C|D]);
        false -> false
    end;
is_word1(_, _) ->
    false.
is_this_a_trigram(Tab, X) ->
    case ets:lookup(Tab, list_to_binary(X)) of
        [] -> false;
        _  -> true
    end.
open() ->
    File = filename:join(filename:dirname(code:which(?MODULE)),
                         "/trigramsS.tab"),
    {ok, Tab} = ets:file2tab(File),
    Tab.
close(Tab) -> ets:delete(Tab).
```

The functions open and close open the ETS table we computed earlier and must bracket any call to is_word.

The other trick I used here was the way in which I located the external file containing the trigram table. I store this in the same directory as the directory where the code for the current module is loaded from. code:which(?MODULE) returns the filename where the object code for ?MODULE was located.

19.5 Storing Tuples on Disk

ETS tables store tuples in memory. DETS (short for Disk ETS) provides Erlang tuple storage on disk. DETS files have a maximum size of 2GB. DETS files must be opened before they can be used, and they should be properly closed when finished with. If they are not properly closed, then they will be automatically repaired the next time they are opened. Since the repair can take a long time, it's important to close them properly before finishing your application.

DETS tables have different sharing properties from ETS tables. When a DETS table is opened, it must be given a global name. If two or more local processes open a DETS table with the same name and options, then they will share the table. The table will remain open until all processes have closed the table (or crashed).

Example: A Filename Index

We want to create a disk-based table that maps filenames onto integers, and vice versa. We'll define the function filename2index and its inverse function index2filename.

To implement this, we'll create a DETS table and populate it with three different types of tuple.

{free, N}
> N is the first free index in the table. When we enter a new filename in the table, it will be assigned the index N.

{FileNameBin, K}
> FileNameBin (a binary) has been assigned index K.

{K, FileNameBin}
> K (an integer) represents the file FileNameBin.

Note how the addition of every new file adds two entries to the table: a File → Index entry and an inverse Index → Filename. This is for efficiency reasons. When ETS or DETS tables are built, only one item in the tuple acts as a key. Matching on a tuple element that is not the key can be done, but it is very

inefficient because it involves searching through the entire table. This is a particularly expensive operation when the entire table resides on disk.

Now let's write the program. We'll start with routines to open and close the DETS table that will store all our filenames.

```erlang
lib_filenames_dets.erl
-module(lib_filenames_dets).
-export([open/1, close/0, test/0, filename2index/1, index2filename/1]).

open(File) ->
    io:format("dets opened:~p~n", [File]),
    Bool = filelib:is_file(File),
    case dets:open_file(?MODULE, [{file, File}]) of
        {ok, ?MODULE} ->
            case Bool of
                true  -> void;
                false -> ok = dets:insert(?MODULE, {free,1})
            end,
            true;
        {error,Reason} ->
            io:format("cannot open dets table~n"),
            exit({eDetsOpen, File, Reason})
    end.

close() -> dets:close(?MODULE).
```

The code for open automatically initializes the DETS table by inserting the tuple {free, 1} if a new table is created. filelib:is_file(File) returns true if File exists; otherwise, it returns false. Note that dets:open_file either creates a new file or opens an existing file, which is why we have to check whether the file exists before calling dets:open_file.

In this code we've used the macro ?MODULE a lot of times; ?MODULE expands to the current module name (which is lib_filenames_dets). Many of the calls to DETS need a unique atom argument for the table name. To generate a unique table name, we just use the module name. Since there can't be two Erlang modules in the system with the same name, then if we follow this convention every-where, we'll be reasonably sure that we have a unique name to use for the table name.

I used the macro ?MODULE instead of explicitly writing the module name every time because I have a habit of changing module names as I write my code. Using macros, if I change the module name, the code will still be correct.

Once we've opened the file, injecting a new filename into the table is easy. This is done as a side effect of calling filename2index. If the filename is in the

table, then its index is returned; otherwise, a new index is generated, and the table is updated, this time with three tuples.

```
lib_filenames_dets.erl
filename2index(FileName) when is_binary(FileName) ->
    case dets:lookup(?MODULE, FileName) of
        [] ->
            [{_,Free}] = dets:lookup(?MODULE, free),
            ok = dets:insert(?MODULE,
                    [{Free,FileName},{FileName,Free},{free,Free+1}]),
            Free;
        [{_,N}] ->
            N
    end.
```

Note how we store three tuples in the table. The second argument to dets:insert is either a tuple or *a list of tuples*. Note also that the filename is represented by a binary. This is for efficiency reasons. It's a good idea to get into the habit of using binaries to represent strings in ETS and DETS tables.

The observant reader might have noticed that there is a potential race condition in filename2index. If two parallel processes call dets:lookup before dets:insert gets called, then filename2index will return an incorrect value. For this routine to work, we must ensure that it is only ever called by one process at a time.

Converting an index to a filename is easy.

```
lib_filenames_dets.erl
index2filename(Index) when is_integer(Index) ->
    case dets:lookup(?MODULE, Index) of
        []       -> error;
        [{_,Bin}] -> Bin
    end.
```

There's a small design decision here. We have to decide what should happen if we call index2filename(Index) and there is no filename associated with this index. We could crash the caller by calling exit(ebadIndex), but in this case we've chosen a gentler alternative: we just return the atom error. The caller can distinguish between a valid filename and incorrect value since all valid returned filenames are of type binary.

Note also the guard tests in filename2index and index2filename. These check that the arguments have the required type. It's a good idea to test these, because entering data of the wrong type into a DETS table can cause situations that are very difficult to debug. We can imagine storing data in a table with the wrong type and reading the table months later, by which time it's too late to

do anything about it. It's best to check that all the data is correct before adding it to the table.

19.6 What Haven't We Talked About?

ETS and DETS tables support a number of operations that we haven't talked about in this chapter. These operations fall into the following categories:

- Fetching and deleting objects based on a pattern

- Converting between ETS and DETS tables and between ETS tables and disk files

- Finding resource usage for a table

- Traversing all elements in a table

- Repairing a broken DETS table

- Visualizing a table

More information about ETS[3] and DETS[4] is available online.

ETS and DETS tables were designed for efficient low-level in-memory and disk storage of Erlang terms, but this is not the end of the story. For more sophisticated data storage, we need a database.

In the next chapter, we'll introduce Mnesia, which is a real-time database written in Erlang and which is part of the standard Erlang distribution. Mnesia uses ETS and DETS tables internally, and a lot of the routines exported from the ets and dets modules are intended for internal use from Mnesia. Mnesia can do all kinds of operations that are not possible using single ETS and DETS tables. For example, we can index on more than the primary key, so the kind of double insertion trick that we used in the filename2index example is not necessary. Mnesia will actually create several ETS or DETS tables to do this, but this is hidden from the user.

Exercises

1. Mod:module_info(exports) returns a list of all the exported functions in the module Mod. Use this function to find all the exported functions from the Erlang system libraries. Make a key-value lookup table where the key is a {Function,Arity} pair and the value is a module name. Store this data in ETS and DETS tables.

---

3. http://www.erlang.org/doc/man/ets.html
4. http://www.erlang.org/doc/man/dets.html

Hint: Use code:lib_dir() and code:lib_dir(LibName) to find the names of all the modules in the system.

2. Make a shared ETS counter table. Implement a function called count:me(Mod,Line) that you can add to your code. You'll call the function by adding count:me(?MODULE, ?LINE) lines to your code. Every time this function is called, it should increment a counter that counts how many times this line has been executed. Implement routines to initialize and read the counters.

3. Write a program to detect plagiarisms in text. To do this, use a two-pass algorithm. In pass 1, break the text into 40-character blocks and compute a checksum for each 40-character block. Store the checksum and filename in an ETS table. In pass 2, compute the checksums of each 40-character block in the data and compare with the checksums in the ETS table.

Hint: You will need to compute a "rolling checkum"[5] to do this. For example, if C1 = B1 + B2 + … B40 and C2 = B2 + B3 + … B41, then C2 can be quickly computed by observing that C2 = C1 + B41 - B1.

_____

5. http://en.wikipedia.org/wiki/Rolling_hash

Mnesia: The Erlang Database

Suppose you want to write a multiuser game, make a new website, or create an online payment system. You'll probably need a database management system (DBMS).

Mnesia is a database written in Erlang for demanding telecommunications applications, and it is part of the standard Erlang distribution. It can be configured with RAM replicates on two physically separated nodes to provide a fast fault-tolerant data store. It provides transactions and comes with its own query language.

Mnesia is extremely fast, and it can store any type of Erlang data structure. It's also highly configurable. Database tables can be stored in RAM (for speed) or on disk (for persistence), and the tables can be replicated on different machines to provide fault-tolerant behavior.

20.1 Creating the Initial Database

Before we can do anything, we have to create an Mnesia database. You need to do this only once.

```
$ erl
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
$ ls
Mnesia.nonode@nohost
```

mnesia:create_schema(NodeList) initiates a new Mnesia database on all the nodes in NodeList (which must be a list of valid Erlang nodes). In our case, we gave the node list as [node()], that is, the current node. Mnesia is initialized and creates a directory structure called Mnesia.nonode@nohost to store the database.

> ### Why Is the DBMS Called Mnesia?
>
> The original name was Amnesia. One of our bosses didn't like the name. He said, "You can't possibly call it Amnesia—you can't have a database that forgets things!" So, we dropped the *A*, and the name stuck.

Then we exit from the Erlang shell and issue the operating system's ls command to verify this.

If we repeat the exercise with a distributed node called joe, we get the following:

```
$ erl -name joe
(joe@doris.myerl.example.com) 1> mnesia:create_schema([node()]).
ok
(joe@doris.myerl.example.com) 2> init:stop().
ok
$ ls
Mnesia.joe@doris.myerl.example.com
```

Or we can point to a specific database when we start Erlang.

```
$ erl  -mnesia dir '"/home/joe/some/path/to/Mnesia.company"'
1> mnesia:create_schema([node()]).
ok
2> init:stop().
ok
```

/home/joe/some/path/to/Mnesia.company is the name of the directory in which the database will be stored.

20.2 Database Queries

Once we've created our database, we can start playing with it. We'll start by looking at Mnesia queries. As you look through this, you might be surprised to see that Mnesia queries look a lot like both SQL and list comprehensions, so there's actually very little you need to learn to get started. In fact, it really isn't that surprising that list comprehensions and SQL look a lot alike. Both are based on mathematical set theory.

In all our examples, I'll assume that you have created a database with two tables called shop and cost. These tables contain the data shown in Table 8, *The shop Table*, on page 323 and Table 9, *The cost Table*, on page 323.

A table in Mnesia is a set or bag of rows, where each row is an Erlang record. To represent these tables in Mnesia, we need record definitions that define the columns in the tables. These are as follows:

Item	Quantity	Cost
apple	20	2.3
orange	100	3.8
pear	200	3.6
banana	420	4.5
potato	2456	1.2

Table 8—The shop Table

Name	Price
apple	1.5
orange	2.4
pear	2.2
banana	1.5
potato	0.6

Table 9—The cost Table

test_mnesia.erl
```erlang
-record(shop, {item, quantity, cost}).
-record(cost, {name, price}).
```

Before we can manipulate the database, we need to create a database schema, start the database, add some table definitions and stop the database, and restart it. We need to do this only once. Here's the code:

test_mnesia.erl
```erlang
do_this_once() ->
    mnesia:create_schema([node()]),
    mnesia:start(),
    mnesia:create_table(shop,   [{attributes, record_info(fields, shop)}]),
    mnesia:create_table(cost,   [{attributes, record_info(fields, cost)}]),
    mnesia:create_table(design, [{attributes, record_info(fields, design)}]),
    mnesia:stop().
```

```erlang
1> test_mnesia:do_this_once().
stopped
=INFO REPORT==== 24-May-2013::15:27:56 ===
application: mnesia
exited: stopped
type: temporary
```

Now we can move on to our examples.

Selecting All Data in a Table

Here's the code to select all the data in the shop table. (For those of you who know SQL, each code fragment starts with a comment showing the equivalent SQL to perform the corresponding operation.)

test_mnesia.erl
```erlang
%% SQL equivalent
%%   SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)]));
```

The heart of the matter is the call to qlc:q, which compiles the query (its parameter) into an internal form that is used to query the database. We pass the resulting query to a function called do(), which is defined toward the bottom of test_mnesia. It is responsible for running the query and returning the result. To make all this easily callable from erl, we map it to the function demo(select_shop). (The entire module is in the file code/test_mnesia.erl available from the book's home page.[1])

Before we can use the database, we need a routine to start it and load the table definitions. This has to be run before using the database, but it has to be run only once per Erlang session.

```
test_mnesia.erl
%% SQL equivalent
%%  SELECT * FROM shop;

demo(select_shop) ->
    do(qlc:q([X || X <- mnesia:table(shop)]));
```

Now we can start the database and make a query.

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
3> test_mnesia:demo(select_shop).
[{shop,potato,2456,1.2},
{shop,orange,100,3.8},
{shop,apple,20,2.3},
{shop,pear,200,3.6},
{shop,banana,420,4.5}]
```

Note: The rows in the table can come out in any order.

The line that sets up the query in this example is as follows:

```
qlc:q([X || X <- mnesia:table(shop)])
```

This looks very much like a list comprehension (see Section 4.5, *List Comprehensions, on page 59*). In fact, qlc stands for *query list comprehensions*. It is one of the modules we can use to access data in an Mnesia database.

[X || X <- mnesia:table(shop)] means "the list of X such that X is taken from the shop Mnesia table." The values of X are Erlang shop records.

---

1. http://pragprog.com/titles/jaerlang2/source_code

Note: The argument of qlc:q/1 must be a list comprehension literal and not something that evaluates to such an expression. So, for example, the following code is *not* equivalent to the code in the example:

```
Var = [X || X <- mnesia:table(shop)],
qlc:q(Var)
```

Choosing Data from a Table

Here's a query that selects the item and quantity columns from the shop table:

test_mnesia.erl
```
%% SQL equivalent
%%  SELECT item, quantity FROM shop;

demo(select_some) ->
    do(qlc:q([{X#shop.item, X#shop.quantity} || X <- mnesia:table(shop)]));
```

4> test_mnesia:demo(select_some).
```
[{orange,100},{pear,200},{banana,420},{potato,2456},{apple,20}]
```

In the previous query, the values of X are records of type shop. If you recall the record syntax described in Section 5.2, *Naming Tuple Items with Records*, on page 76, you'll remember that X#shop.item refers to the item field of the shop record. So, the tuple {X#shop.item, X#shop.quantity} is a tuple of the item and quantity fields of X.

Conditionally Selecting Data from a Table

Here's a query that lists all items in the shop table where the number of items in stock is less than 250. Maybe we'll use this query to decide which items to reorder. Notice how the condition is described naturally as part of the list comprehension.

test_mnesia.erl
```
%% SQL equivalent
%%   SELECT shop.item FROM shop
%%   WHERE  shop.quantity < 250;

demo(reorder) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250
                                 ]));
```

5> test_mnesia:demo(reorder).
```
[orange,pear,apple]
```

Selecting Data from Two Tables (Joins)

Now let's suppose that we want to reorder an item only if there are fewer than 250 items in stock and the item costs less than 2.0 currency units. To do this, we need to access two tables. Here's the query:

test_mnesia.erl
```
%% SQL equivalent
%%   SELECT shop.item
%%   FROM shop, cost
%%   WHERE shop.item = cost.name
%%     AND cost.price < 2
%%     AND shop.quantity < 250

demo(join) ->
    do(qlc:q([X#shop.item || X <- mnesia:table(shop),
                             X#shop.quantity < 250,
                             Y <- mnesia:table(cost),
                             X#shop.item =:= Y#cost.name,
                             Y#cost.price < 2
                                ])).
```

```
6> test_mnesia:demo(join).
[apple]
```

The key here is the join between the name of the item in the shop table and the name in the cost table.

```
X#shop.item =:= Y#cost.name
```

20.3 Adding and Removing Data in the Database

Again, we'll assume we have created our database and defined a shop table. Now we want to add or remove a row from the table.

Adding a Row

We can add a row to the shop table as follows:

test_mnesia.erl
```
add_shop_item(Name, Quantity, Cost) ->
    Row = #shop{item=Name, quantity=Quantity, cost=Cost},
    F = fun() ->
                mnesia:write(Row)
        end,
    mnesia:transaction(F).
```

This creates a shop record and inserts it into the table.

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
```

```
{atomic, ok}
%% list the shop table
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% add a new row
4> test_mnesia:add_shop_item(orange, 236, 2.8).
 {atomic,ok}
%% list the shop table again so we can see the change
5> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
```

Note: The *primary key* of the shop table is the first column in the table, that is, the item field in the shop record. The table is of type "set" (see a discussion of the set and bag types in Section 19.1, *Types of Table*, on page 306). If the newly created record has the same primary key as an existing row in the database table, it will overwrite that row; otherwise, a new row will be created.

Removing a Row

To remove a row, we need to know the object ID (OID) of the row. This is formed from the table name and the value of the primary key.

test_mnesia.erl

```
remove_shop_item(Item) ->
    Oid = {shop, Item},
    F = fun() ->
                mnesia:delete(Oid)
        end,
    mnesia:transaction(F).
```

```
6> test_mnesia:remove_shop_item(pear).
{atomic,ok}
%% list the table -- the pear has gone
7> test_mnesia:demo(select_shop).
[{shop,orange,236,2.80000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
8> mnesia:stop().
ok
```

20.4 Mnesia Transactions

When we added or removed data from the database or performed a query, we wrote the code something like this:

```erlang
do_something(...) ->
    F = fun() ->
            % ...
            mnesia:write(Row)
            % ... or ...
            mnesia:delete(Oid)
            % ... or ...
            qlc:e(Q)
        end,
    mnesia:transaction(F)
```

F is a fun with zero arguments. Inside F we called some combination of mnesia:write/1, mnesia:delete/1, or qlc:e(Q) (where Q is a query compiled with qlc:q/1). Having built the fun, we call mnesia:transaction(F), which evaluates the expression sequence in the fun.

Transactions guard against faulty program code but more importantly against concurrent access of the database. Suppose we have two processes that try to simultaneously access the same data. For example, suppose I have $10 in my bank account. Now suppose two people try to simultaneously withdraw $8 from that account. I would like one of these transactions to succeed and the other to fail.

This is exactly the guarantee that mnesia:transaction/1 provides. Either all the reads and writes to the tables in the database within a particular transaction succeed, or none of them does. If none of them does, the transaction is said to fail. If the transaction fails, no changes will be made to the database.

The strategy that Mnesia uses for this is a form of *pessimistic locking*. Whenever the Mnesia transaction manager accesses a table, it tries to lock the record or the entire table depending upon the context. If it detects that this might lead to deadlock, it immediately aborts the transaction and undoes any changes it has made.

If the transaction initially fails because some other process is accessing the data, the system waits for a short time and retries the transaction. One consequence of this is that the code inside the transaction fun might be evaluated a large number of times.

For this reason, the code inside a transaction fun should not do anything that has any side effects. For example, if we were to write the following:

```
F = fun() ->
        ...
        io:format("reading ..."), %% don't do this
        ...
    end,
mnesia:transaction(F),
```

we might get a lot of output, since the fun might be retried many times.

Note 1: mnesia:write/1 and mnesia:delete/1 should be called only inside a fun that is processed by mnesia:transaction/1.

Note 2: You should never write code to explicitly catch exceptions in the Mnesia access functions (mnesia:write/1, mnesia:delete/1, and so on) since the Mnesia transaction mechanism itself relies upon these functions throwing exceptions on failure. If you catch these exceptions and try to process them yourself, you will break the transaction mechanism.

Aborting a Transaction

Near to our shop, there's a farm. And the farmer grows apples. The farmer loves oranges, and he pays for the oranges with apples. The going rate is two apples for each orange. So, to buy N oranges, the farmer pays 2*N apples. Here's a function that updates the database when the farmer buys some oranges:

test_mnesia.erl
```
farmer(Nwant) ->
    %% Nwant = Number of oranges the farmer wants to buy
    F = fun() ->
                %% find the number of apples
                [Apple] = mnesia:read({shop,apple}),
                Napples = Apple#shop.quantity,
                Apple1  = Apple#shop{quantity = Napples + 2*Nwant},
                %% update the database
                mnesia:write(Apple1),
                %% find the number of oranges
                [Orange] = mnesia:read({shop,orange}),
                NOranges = Orange#shop.quantity,
                if
                    NOranges >= Nwant ->
                        N1 =  NOranges - Nwant,
                        Orange1 = Orange#shop{quantity=N1},
                        %% update the database
                        mnesia:write(Orange1);
                    true ->
                        %% Oops -- not enough oranges
                        mnesia:abort(oranges)
                end
        end,
    mnesia:transaction(F).
```

This code is written in a pretty stupid way because I want to show how the transaction mechanism works. First, I update the number of apples in the database. This is done *before* I check the number of oranges. The reason I do this is to show that this change gets "undone" if the transaction fails. Normally, I'd delay writing the orange and apple data back to the database until I was sure I had enough oranges.

Let's show this in operation. In the morning, the farmer comes in and buys 50 oranges.

```
1> test_mnesia:start().
ok
2> test_mnesia:reset_tables().
{atomic, ok}
%% List the shop table
3> test_mnesia:demo(select_shop).
[{shop,orange,100,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,20,2.30000}]
%% The farmer buys 50 oranges
%% paying with 100 apples
4> test_mnesia:farmer(50).
{atomic,ok}
%% Print the shop table again
5> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

In the afternoon, the farmer wants to buy 100 more oranges (boy, does this guy love oranges).

```
6> test_mnesia:farmer(100).
{aborted,oranges}
7> test_mnesia:demo(select_shop).
[{shop,orange,50,3.80000},
 {shop,pear,200,3.60000},
 {shop,banana,420,4.50000},
 {shop,potato,2456,1.20000},
 {shop,apple,120,2.30000}]
```

When the transaction failed (when we called mnesia:abort(Reason)), the changes made by mnesia:write were undone. Because of this, the database state was restored to how it was before we entered the transaction.

Loading the Test Data

Now we know how transactions work, so we can look at the code for loading the test data.

The function test_mnesia:example_tables/0 is used to provide data to initialize the database tables. The first element of the tuple is the table name. This is followed by the table data in the order given in the original record definitions.

test_mnesia.erl
```erlang
example_tables() ->
    [%% The shop table
     {shop, apple,   20,   2.3},
     {shop, orange,  100,  3.8},
     {shop, pear,    200,  3.6},
     {shop, banana,  420,  4.5},
     {shop, potato,  2456, 1.2},
     %% The cost table
     {cost, apple,   1.5},
     {cost, orange,  2.4},
     {cost, pear,    2.2},
     {cost, banana,  1.5},
     {cost, potato,  0.6}
    ].
```

Following is the code that inserts data into Mnesia from the example tables. This just calls mnesia:write for each tuple in the list returned by example_tables/1.

test_mnesia.erl
```erlang
reset_tables() ->
    mnesia:clear_table(shop),
    mnesia:clear_table(cost),
    F = fun() ->
                foreach(fun mnesia:write/1, example_tables())
        end,
    mnesia:transaction(F).
```

The do() Function

The do() function called by demo/1 is slightly more complex.

test_mnesia.erl
```erlang
do(Q) ->
    F = fun() -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.
```

This calls qlc:e(Q) inside an Mnesia transaction. Q is a compiled QLC query, and qlc:e(Q) evaluates the query and returns all answers to the query in a list. The return value {atomic, Val} means that the transaction succeeded with value Val. Val is the value of the transaction function.

20.5 Storing Complex Data in Tables

One of the disadvantages of using a conventional DBMS is that there are a limited number of data types you can store in a table column. You can store an integer, a string, a float, and so on. But if you want to store a complex object, then you're in trouble. So, for example, if you're a Java programmer, storing a Java object in a SQL database is pretty messy.

Mnesia is designed to store Erlang data structures. In fact, you can store any Erlang data structure you want in an Mnesia table.

To illustrate this, we'll suppose that a number of architects want to store their designs in an Mnesia database. To start with, we must define a record to represent their designs.

test_mnesia.erl
```erlang
-record(design, {id, plan}).
```

Then we can define a function that adds some designs to the database.

test_mnesia.erl
```erlang
add_plans() ->
    D1 = #design{id   = {joe,1},
                 plan = {circle,10}},
    D2 = #design{id   = fred,
                 plan = {rectangle,10,5}},
    D3 = #design{id   = {jane,{house,23}},
                 plan = {house,
                         [{floor,1,
                           [{doors,3},
                            {windows,12},
                            {rooms,5}]},
                          {floor,2,
                           [{doors,2},
                            {rooms,4},
                            {windows,15}]}]}},
    F = fun() ->
              mnesia:write(D1),
              mnesia:write(D2),
              mnesia:write(D3)
        end,
    mnesia:transaction(F).
```

Now we can add some designs to the database.

```erlang
1> test_mnesia:start().
ok
2> test_mnesia:add_plans().
{atomic,ok}
```

Now we have some plans in the database. We can extract these with the following access function:

```
test_mnesia.erl
get_plan(PlanId) ->
    F = fun() -> mnesia:read({design, PlanId}) end,
    mnesia:transaction(F).

3> test_mnesia:get_plan(fred).
{atomic,[{design,fred,{rectangle,10,5}}]}
4> test_mnesia:get_plan({jane, {house,23}}).
{atomic,[{design,{jane,{house,23}},
                {house,[{floor,1,[{doors,3},
                                  {windows,12},
                                  {rooms,5}]},
                        {floor,2,[{doors,2},
                                  {rooms,4},
                                   {windows,15}]}]}}]}
```

As you can see, both the database key and the extracted record can be arbitrary Erlang terms.

In technical terms, we say there is no *impedance mismatch* between the data structures in the database and the data structures in our programming language. This means that inserting and deleting complex data structures into the database is very fast.

20.6 Table Types and Location

We can configure Mnesia tables in many different ways. First, tables can be in RAM or on disk (or both). Second, tables can be located on a single machine or replicated on several machines.

When we design our tables, we must think about the type of data we want to store in the tables. Here are the properties of the tables:

RAM tables

These are very fast. The data in them is *transient*, so it will be lost if the machine crashes or when you stop the DBMS.

Disk tables

Disk tables should survive a crash (provided the disk is not physically damaged).

When an Mnesia transaction writes to a table and the table is stored on disk, what actually happens is that the transaction data is first written to a disk log. This disk log grows continuously, and at regular intervals the information in the disk log is consolidated with the other data in the

Fragmented Tables

Mnesia supports "fragmented" tables (*horizontal partitioning* in database terminology). This is designed for implementing extremely large tables. The tables are split into fragments that are stored on different machines. The fragments are themselves Mnesia tables. The fragments can be replicated, have indexes, and so on, like any other table.

Refer to the Mnesia User's Guide for more details.

database, and the entry in the disk log is cleared. If the system crashes, then the next time the system is restarted, the disk log is checked for consistency, and any outstanding entries in the log are added to the database before the database is made available. Once a transaction has succeeded, the data should have been properly written to the disk log, and if the system fails after this, then when the system is next restarted, changes made in the transaction should survive the crash.

If the system crashes during a transaction, then the changes made to the database should be lost.

Before using a RAM table, you need to perform some experiments to see whether the entire table will fit into physical memory. If the RAM tables don't fit into physical memory, the system will page a lot, which will be bad for performance.

RAM tables are transient, so if we want to build a fault-tolerant application, we will need to replicate the RAM table on disk or replicate it on a second machine as a RAM or disk table, or both.

Creating Tables

To create a table, we call mnesia:create_table(Name, ArgS), where ArgS is a list of {Key,Val} tuples. create_table returns {atomic, ok} if the table was successfully created; otherwise, it returns {aborted, Reason}. Some of the most common arguments to create_table are as follows:

Name
> This is the name of the table (an atom). By convention, it is the name of an Erlang record—the table rows will be instances of this record.

{type, Type}
> This specifies the type of the table. Type is one of set, ordered_set, or bag. These types have the same meaning as described in Section 19.1, *Types of Table*, on page 306.

{disc_copies, NodeList}

> NodeList is a list of the Erlang nodes where disk copies of the table will be stored. When we use this option, the system will also create a RAM copy of the table on the node where we performed this operation.
>
> It is possible to have a replicated table of type disc_copies on one node and to have the same table stored as a different table type on a different node. This is desirable if we want the following:
>
> - Read operations to be very fast and performed from RAM
> - Write operations to be performed to persistent storage

{ram_copies, NodeList}

> NodeList is a list of the Erlang nodes where RAM copies of the table will be stored.

{disc_only_copies, NodeList}

> NodeList is a list of the Erlang nodes where disk-only copies of data are stored. These tables have no RAM replicas and are slower to access.

{attributes, AtomList}

> This is a list of the column names of the values in a particular table. Note that to create a table containing the Erlang record xxx, we can use the syntax {attributes, record_info(fields, xxx)} (alternatively we can specify an explicit list of record field names).

Note: There are more options to create_table than I have shown here. Refer to the manual page for mnesia for details of all the options.

Common Combinations of Table Attributes

In all the following, we'll assume that Attrs is an {attributes,...} tuple.

Here are some common table configuration options that cover the most common cases:

```
mnesia:create_table(shop, [Attrs])
```

- This makes a RAM-resident table on a single node.
- If the node crashes, the table will be lost.
- This is the fastest of all tables.
- The table must fit into memory.

```
mnesia:create_table(shop,[Attrs,{disc_copies,[node()]}])
```

- This makes a RAM-resident table and a disk copy on a single node.
- If the node crashes, the table will be recovered from disk.

- The has fast read access but slower write access.
- The table should fit into memory.

```
mnesia:create_table(shop, [Attrs,{disc_only_copies,[node()]}])
```

- This makes a disk-only copy on a single node.
- This is used for tables too large to fit in memory.
- This has slower access times than with RAM-resident replicas.

```
mnesia:create_table(shop,
  [Attrs,{ram_copies,[node(),someOtherNode()]}])
```

- This makes a RAM-resident table on two nodes.
- If both nodes crash, the table will be lost.
- The table must fit into memory.
- The table can be accessed on either node.

```
mnesia:create_table(shop,
  [Attrs, {disc_copies, [node(),someOtherNode()]}])
```

- This makes disk copies on several nodes.
- We can resume after a crash on any node.
- This survives failure of all nodes.
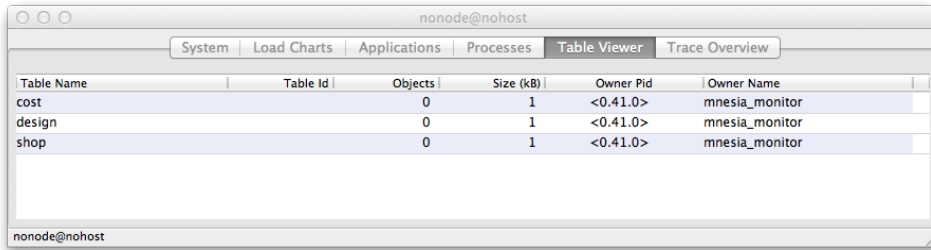
Table Behavior

When a table is replicated across several Erlang nodes, it is synchronized as far as possible. If one node crashes, the system will still work, but the number of replicas will be reduced. When the crashed node comes back online, it will resynchronize with the other nodes where the replicas are kept.

Note: Mnesia may become overloaded if the nodes running Mnesia stop functioning. If you are using a laptop that goes to sleep, when it restarts, Mnesia might become temporarily overloaded and produce a number of warning messages. We can ignore these messages.

20.7 The Table Viewer
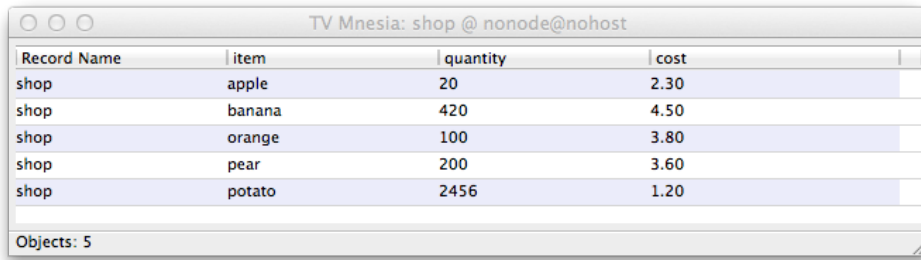
To see the data we have stored in Mnesia, we can use the table viewer that is built into the "observer" application. Start the observer with the command observer:start() and then click the Table Viewer tab. Now select View > Mnesia Tables in the control menu of the observer. You'll see a list of tables like the one shown here:

Click the shop entry, and a new window opens.

Using the observer, you can view tables, examine the state of the system, view processes, and so on.

20.8 Digging Deeper

I hope I've whetted your appetite for Mnesia. Mnesia is a very powerful DBMS. It has been in production use in a number of demanding telecom applications delivered by Ericsson since 1998.

Since this is a book about Erlang and not Mnesia, I can't really do any more than just give a few examples of the most common ways to use Mnesia. The techniques I've shown in this chapter are those I use myself. I don't actually use (or understand) much more than what I've shown you. But with what I've shown you, you can have a lot of fun and build some pretty sophisticated applications.

The main areas I've omitted are as follows:

Backup and recovery
  Mnesia has a range of options for configuring backup operations, allowing for different types of disaster recovery.

Dirty operations
  Mnesia allows a number of *dirty* operations (dirty_read, dirty_write, …). These are operations that are performed outside a transaction context. They are very dangerous operations that can be used if you know that your

application is single-threaded or under other special circumstances. Dirty operations are used for efficiency reasons.

SNMP tables

Mnesia has a built-in SNMP table type. This makes implementing SNMP management systems very easy.

The definitive reference to Mnesia is the Mnesia User's Guide available from the main Erlang distribution site. In addition, the examples subdirectory in the Mnesia distribution (/usr/local/lib/erlang/lib/mnesia-X.Y.Z/examples on my machine) has some Mnesia examples.

We've now completed the two chapters on data storage. In the next chapter, we will look at methods for debugging, tracing, and performance tuning.

Exercises

1. Imagine you're going to make a website where users can leave tips about good Erlang programs. Make a Mnesia database with three tables (users, tips, and abuse) to store all the data you need for the site. The users table should store user account data (name, email address, password, and so on). The tips table should store tips about valuable sites (something like site URL, description, date of review, and so on). The abuse table should store data to try to prevent abuse of the site (data on IP address of site visits, number of site visits, and so on).

   Configure the database to run on one machine with one RAM and one disk copy. Write routines to read, write, and list these tables.

2. Continue the previous exercise, but configure the database for two machines with replicated RAM and disk copies. Try making updates on one machine and then crashing the machine and checking that you can continue to access the database on the second machine.

3. Write a query to reject a tip if a user has submitted more than ten tips in a one-day period or logged in from more than three IP addresses in the last day.

   Measure how long time it takes to query the database.

Profiling, Debugging, and Tracing

In this chapter, we'll look at a number of techniques that you can use to tune your program, find bugs, and avoid errors.

Profiling

We use profiling for performance tuning to find out where the hot spots in our programs are. I think it's almost impossible to guess where the bottlenecks in our programs are. The best approach is to first write our programs, then confirm that they are correct, and finally *measure* to find out where the time goes. Of course, if the program is fast enough, we can omit the last step.

Coverage analysis

We use coverage analysis to count the number of times each line of code in our programs has been executed. Lines that have been executed zero times might indicate an error or dead code that you might be able to remove. Finding lines that are executed a large number of times might help you optimize your program.

Cross-referencing

We can use cross-referencing to find out whether we have any missing code and to find out who calls what. If we try to call a function that does not exist, then the cross-reference analysis will detect this. This is mostly useful for large programs with dozens of modules.

Compiler diagnostics

This section explains the compiler diagnostics.

Runtime error messages

The runtime system produces many different error messages. We'll explain what the most common error messages mean.

Debugging techniques

Debugging is split into two sections. First we'll look at some simple techniques, and second we'll look at the Erlang debugger.

Tracing

We can examine the behavior of a running system using process tracing. Erlang has advanced trace facilities that we can use to remotely observe the behavior of any process. We can observe the message passing between processes and find out which functions are being evaluated within a process, when processes are scheduled, and so on.

Test frameworks

There are a number of test frameworks for the automated testing of Erlang programs. We'll see a quick overview of these frameworks as well as references to how they can be obtained.

21.1 Tools for Profiling Erlang Code

The standard Erlang distribution comes with three profiling tools.

- cprof counts the number of times each function is called. This is a lightweight profiler. Running this on a live system adds from 5 to 10 percent to the system load.

- fprof displays the time for calling and called functions. Output is to a file. This is suitable for large system profiling in a lab or simulated system. It adds significant load to the system.

- eprof measures how time is used in Erlang programs. This is a predecessor of fprof, which is suitable for small-scale profiling.

Here's how you run cprof; we'll use it to profile the code that we wrote in Section 17.6, *A SHOUTcast Server*, on page 281:

```
1> cprof:start().        %% start the profiler
4501
2> shout:start().        %% run the application
<0.35.0>
3> cprof:pause().        %% pause the profiler
4844
4> cprof:analyse(shout). %% analyse function calls
{shout,232,
[{{shout,split,2},73},
{{shout,write_data,4},33},
{{shout,the_header,1},33},
{{shout,send_file,6},33},
{{shout,bump,1},32},
{{shout,make_header1,1},5},
```

```
{{shout,'-got_request_from_client/3-fun-0-',1},4},
{{shout,songs_loop,1},2},
{{shout,par_connect,2},2},
{{shout,unpack_song_descriptor,1},1},
...
5> cprof:stop().          %% stop the profiler
4865
```

In addition, cprof:analyse() analyzes all the modules for which statistics have been collected.

More details of cprof are available online.[1]

fprof[2] and eprof[3] are broadly similar to cprof. For details, consult the online documentation.

21.2 Testing Code Coverage

When we're testing our code, it's often nice to see not only which lines of code are executed a lot but also which lines are never executed. Lines of code that are never executed are a potential source of error, so it's really good to find out where these are. To do this, we use the program coverage analyzer.

Here's an example:

```
1> cover:start().         %% start the coverage analyser
{ok,<0.34.0>}
2> cover:compile(shout).  %% compile shout.erl for coverage
{ok,shout}
3> shout:start().         %% run the program
<0.41.0>
Playing:<<"title: track018 performer: .. ">>
4> %% let the program run for a bit
4>  cover:analyse_to_file(shout).  %% analyse the results
{ok,"shout.COVER.out"}            %% this is the results file
```

The results of this are printed to a file.

```
...
|  send_file(S, Header, OffSet, Stop, Socket, SoFar) ->
|       %% OffSet = first byte to play
|       %% Stop   = The last byte we can play
131..|      Need = ?CHUNKSIZE - byte_size(SoFar),
131..|      Last = OffSet + Need,
131..|      if
|        Last >= Stop ->
|             %% not enough data so read as much as possible and return
```

---

1. http://www.erlang.org/doc/man/cprof.html
2. http://www.erlang.org/doc/man/fprof.html
3. http://www.erlang.org/doc/man/fprof.html

```
0..|           Max = Stop - OffSet,
0..|           {ok, Bin} = file:pread(S, OffSet, Max),
0..|           list_to_binary([SoFar, Bin]);
|       true ->
131..|         {ok, Bin} = file:pread(S, OffSet, Need),
131..|         write_data(Socket, SoFar, Bin, Header),
131..|         send_file(S, bump(Header),
|                     OffSet + Need,  Stop, Socket, <<>>)
|      end.
...
```

On the left side of the file, we see the number of times each statement has been executed. The lines marked with a zero are particularly interesting. Since this code has never been executed, we can't really say whether our program is correct.

> ### The Best of All Test Methods?
>
> Performing a coverage analysis of our code answers the question, which lines of code are never executed? Once we know which lines of code are never executed, we can design test cases that force these lines of code to be executed.
>
> Doing this is a surefire way to find unexpected and obscure bugs in your program. Every line of code that has never been executed might contain an error. Forcing these lines to be executed is the best way I know to test a program.
>
> I did this to the original Erlang JAM[a] compiler. I think we got three bug reports in two years. After this, there were no reported bugs.
>
> ___
>
> a.   Joe's Abstract Machine (the first Erlang compiler)

Designing test cases that cause all the coverage counts to be greater than zero is a valuable method of systematically finding hidden faults in our programs.

21.3 Generating Cross-References

Running an occasional cross-reference check on your code when you are developing a program is a good idea. If there are missing functions, you will discover this before you run your program, not after.

You can generate cross-references using the xref module. xref works only if your code has been compiled with the debug_info flag set.

I can't show you the output of xref on the code accompanying this book, because the development is complete and there aren't any missing functions.

Instead, I'll show you what happens when I run a cross-reference check on the code in one of my hobby projects.

vsg is a simple graphics program that I might release one day. We'll do an analysis of the code in the vsg directory where I'm developing the program.

```
$ cd /home/joe/2007/vsg-1.6
$ rm *.beam
$ erlc +debug_info *.erl
$ erl
1> xref:d('.').
[{deprecated,[]},
{undefined,[{{new,win1,0},{wish_manager,on_destroy,2}},
{{vsg,alpha_tag,0},{wish_manager,new_index,0}},
{{vsg,call,1},{wish,cmd,1}},
{{vsg,cast,1},{wish,cast,1}},
{{vsg,mkWindow,7},{wish,start,0}},
{{vsg,new_tag,0},{wish_manager,new_index,0}},
{{vsg,new_win_name,0},{wish_manager,new_index,0}},
{{vsg,on_click,2},{wish_manager,bind_event,2}},
{{vsg,on_move,2},{wish_manager,bind_event,2}},
{{vsg,on_move,2},{wish_manager,bind_tag,2}},
{{vsg,on_move,2},{wish_manager,new_index,0}}]},
{unused,[{vsg,new_tag,0},
{vsg_indicator_box,theValue,1},
{vsg_indicator_box,theValue,1}]}]
```

xref:d('.') performs a cross-reference analysis of all the code in the current directory that has been compiled with the debug flag. It produces lists of deprecated, undefined, and unused functions.

Like most tools, xref has a large number of options, so reading the manual is necessary if you want to use the more powerful features that this program has.

21.4 Compiler Diagnostics

When we compile a program, the compiler provides us with helpful error messages if our source code is syntactically incorrect. Most of these are self-evident: if we omit a bracket, a comma, or a keyword, the compiler will give an error message with the filename and line number of the offending statement. The following are some errors we could see.

Head Mismatch

We'll get the following error if the clauses that make up a function definition do not have the same name and arity:

bad.erl
```
Line 1  foo(1,2) ->
    2        a;
    3  foo(2,3,a) ->
    4        b.
```

```
1> c(bad).
./bad.erl:3: head mismatch
```

Unbound Variables

Here's some code containing unbound variables:

bad.erl
```
Line 1  foo(A, B) ->
    2        bar(A, dothis(X), B),
    3        baz(Y, X).
```

```
1> c(bad).
./bad.erl:2: variable 'X' is unbound
./bad.erl:3: variable 'Y' is unbound
```

This means that in line 2 the variable X has no value. The error isn't actually on line 2 but is detected at line 2, which is the first occurrence of the unbound variable X. (X is also used on line 3, but the compiler reports only the first line where the error occurs.)

Unterminated String

If we forget the final quote mark in a string or atom, we'll get the following error message:

unterminated string starting with "..."

Sometimes finding the missing quote mark can be pretty tricky. If you get this message and really can't see where the missing quote is, then try placing a quote mark near to where you think the problem might be and recompile the program. This might produce a more precise diagnostic that will help you pinpoint the error.

Unsafe Variables

If we compile the following code:

bad.erl
```
Line 1  foo() ->
    2        case bar() of
    3            1 ->
    4                X = 1,
    5                Y = 2;
    6            2 ->
```

```
7              X = 3
8        end,
9        b(X).
```

we'll get the following warning:

```
1> c(bad).
./bad.erl:5: Warning: variable 'Y' is unused
{ok,bad}
```

This is just a warning, since Y is defined but not used. If we now change the program to the following:

bad.erl
```
Line 1  foo() ->
2        case bar() of
3            1 ->
4                X = 1,
5                Y = 2;
6            2 ->
7                X = 3
8        end,
9        b(X, Y).
```

we'll get the following error:

```
> c(bad).
./bad.erl:9: variable 'Y' unsafe in 'case' (line 2)
{ok,bad}
```

The compiler reasons that the program might take the second branch through the case expression (in which event the variable Y will be undefined), so it produces an "unsafe variable" error message.

Shadowed Variables

Shadowed variables are variables that hide the value of previously defined variables so that you can't use the values of the previously defined variables. Here's an example:

bad.erl
```
Line 1  foo(X, L) ->
2        lists:map(fun(X) -> 2*X end, L).
```

```
1> c(bad).
./bad.erl:1: Warning: variable 'X' is unused
./bad.erl:2: Warning: variable 'X' shadowed in 'fun'
{ok,bad}
```

Here, the compiler is worried that we might have made a mistake in our program. Inside the fun we compute 2*X, but which X are we talking about: the X that is the argument to the fun or the X that is the argument to foo?

If this happens, the best thing to do is to rename one of the Xs to make the warning go away. We could rewrite this as follows:

bad.erl
```
foo(X, L) ->
    lists:map(fun(Z) -> 2*Z end, L).
```

Now there is no problem if we want to use X inside the fun definition.

21.5 Runtime Diagnostics

If an Erlang process crashes, we might get an error message. To see the error message, some other process has to monitor the crashing process and print an error message when the monitored process dies. If we just create a process with spawn and the process dies, we won't get any error message. The best thing to do if we want to see all the error messages is always to use spawn_link.

The Stack Trace

Every time a process crashes that is linked to the shell, a stack trace will be printed. To see what's in the stack trace, we'll write a simple function with a deliberate error and call this function from the shell.

lib_misc.erl
```
deliberate_error(A) ->
    bad_function(A, 12),
    lists:reverse(A).

bad_function(A, _) ->
    {ok, Bin} = file:open({abc,123}, A),
    binary_to_list(Bin).

1> lib_misc:deliberate_error("file.erl").
** exception error: no match of right hand side value {error,badarg}
in function  lib_misc:bad_function/2 (lib_misc.erl, line 804)
in call from lib_misc:deliberate_error/1 (lib_misc.erl, line 800)
```

When we called lib_misc:deliberate_error("file.erl"), an error occurred and the system printed an error message followed by a stack trace. The error message is as follows:

```
** exception error: no match of right hand side value {error,badarg}
```

This comes from the following line:

```
{ok, Bin} = file:open({abc,123}, A)
```

Calling file:open/2 returned {error, badarg}. This was because {abc,123} is not a valid input value to file:open. When we try to match the return value with {ok, Bin}, we get a badmatch error, and the runtime system prints ** exception error ... {error, badarg}.

Following the error message is a stack trace. The stack trace starts with the name of the function where the error occurred. This is followed by a list of the names of the functions together with the module names and line numbers of the functions that the current function will return to when it completes. Thus, the error occurred in lib_misc:bad_function/2, which will return to lib_misc:deliberate_error/1, and so on.

Note that it is really only the top entries in the stack trace that are interesting. If the call sequence to the erroneous function involves a tail call, then the call won't be in the stack trace. For example, if we define the function deliberate_error1 as follows:

```
lib_misc.erl
deliberate_error1(A) ->
    bad_function(A, 12).
```

then when we call this and get an error, the function deliberate_error1 will be missing from the stack trace.

```
2> lib_misc:deliberate_error1("file.erl").
** exception error: no match of right hand side value {error,badarg}
in function  lib_misc:bad_function/2 (lib_misc.erl, line 804)
```

The call to deliberate_error1 is not in the trace since bad_function was called as the last statement in deliberate_error1 and will not return to deliberate_error1 when it completes but will return to the caller of deliberate_error1.

(This is the result of a *last-call* optimization; if the last thing executed in a function is a function call, that call is effectively replaced with a jump. Without this optimization, the *infinite loop* style of programming we use to code message reception loops would not work. However, because of this optimization, the calling function is effectively replaced in the call stack by the called function and hence becomes invisible in stack traces.)

21.6 Debugging Techniques

Debugging Erlang is pretty easy. That might surprise you, but this is a consequence of having single-assignment variables. Since Erlang has no pointers and no mutable state (with the exception of ETS tables and process dictionaries), finding out where things have gone wrong is rarely a problem. Once we

have observed that a variable has an incorrect value, it's relatively easy to find out when and where this happened.

I find a debugger is a very helpful tool when I write C programs, because I can tell it to watch variables and tell me when their value changes. This is often important because memory in C can be changed indirectly, via pointers. It can be hard to know just *where* a change to some chunk of memory came from. I don't feel the same need for a debugger in Erlang because we can't modify state through pointers.

Erlang programmers use a variety of techniques for debugging their programs. By far the most common technique is to just add print statements to the incorrect programs. This techniques fails if the data structures you are interested in become very large, in which case they can be dumped to a file for later inspection.

Some folks use the error logger to save error messages, and others write them in a file. Failing this, we can use the Erlang debugger or trace the execution of the program. Let's look at each of these techniques.

io:format Debugging

Adding print statements to the program is the most common form of debugging. You simply add io:format(…) statements to print the values of variables you are interested in at critical points in your program.

When debugging parallel programs, it's often a good idea to print messages immediately *before* you send a message to another process and immediately *after* you have received a message.

When I'm writing a concurrent program, I almost always start writing a receive loop like this:

```
loop(...) ->
    receive
        Any ->
            io:format("*** warning unexpected message:~p~n",[Any])
            loop(...)
    end.
```

Then, as I add patterns to the receive loop, I get warning messages printed if my process gets any message it doesn't understand. I also use spawn_link instead of spawn to make sure error messages are printed if my process exits abnormally.

I often use a macro NYI (not yet implemented), which I define like this:

```erlang
-define(NYI(X),(begin
                    io:format("*** NYI ~p ~p ~p~n",[?MODULE, ?LINE, X]),
                    exit(nyi)
                end)).
```

Then I might use this macro as follows:

```erlang
glurk(X, Y) ->
    ?NYI({glurk, X, Y}).
```

The body of the function glurk is not yet written, so when I call glurk, the program crashes.

```erlang
> lib_misc:glurk(1,2).
*** NYI lib_misc 83 {glurk,1,2}
** exited: nyi *
```

The program exits and an error message is displayed, so I know it's time to complete the implementation of my function.

Dumping to a File

If the data structure we're interested in is large, then we can write it to a file using a function such as dump/2.

```erlang
dump(File, Term) ->
    Out = File ++ ".tmp",
    io:format("** dumping to ~s~n",[Out]),
    {ok, S} = file:open(Out, [write]),
    io:format(S, "~p.~n",[Term]),
    file:close(S).
```

This prints a warning message to remind us that we have created a new file. It then adds a .tmp file extension to the filename (so we can easily delete all the temporary files later). Then it pretty-prints the term we are interested in to a file. We can examine the file in a text editor at a later stage. This technique is simple and particularly useful when examining large data structures.

Using the Error Logger

We can use the error logger and create a text file with debugging output. To do so, we create a configuration file such as the following:

```erlang
%% text error log
[ {kernel,
   [{error_logger,
     {file, "/Users/joe/error_logs/debug.log"}}]}].
```

Then we start Erlang with the following command:

```
erl -config elog5.config
```

Any error messages created by calling routines in the error_logger module, along with any error messages printed in the shell, will end up in the file specified in the configuration file.

21.7 The Erlang Debugger

The standard Erlang distribution contains a debugger. I'm not going to say a lot about it here, other than to tell you how to start it and to give you pointers to the documentation. Using the debugger once it has been started is pretty easy. You can inspect variables, single-step the code, set breakpoints, and so on.

Because we'll often want to debug several processes, the debugger can also spawn copies of itself, so we can have several debug windows, one for each process we are debugging.

The only tricky thing is getting the debugger started.

```
1> %% recompile lib_misc so we can debug it
1>  c(lib_misc, [debug_info]).
{ok, lib_misc}
2> im().  %% A window will pop up. Ignore it for now
<0.42.0>
3> ii(lib_misc).
{module,lib_misc}
4> iaa([init]).
true.
5> lib_misc:
...
```

Running this opens the window shown in Figure 1, *Debugger initial window, on page 351*.

In the debugger you can set breakpoints, examine variables, and so on.

All the commands without a module prefix (ii/1, iaa/1, and so on) are exported from the module i. This is the debugger/interpreter interface module. These routines are accessible from the shell without giving the module prefix.

The functions we called to get the debugger running do the following:

im()

   Start a new graphical monitor. This is the main window of the debugger. It displays the state of all the processes that the debugger is monitoring.
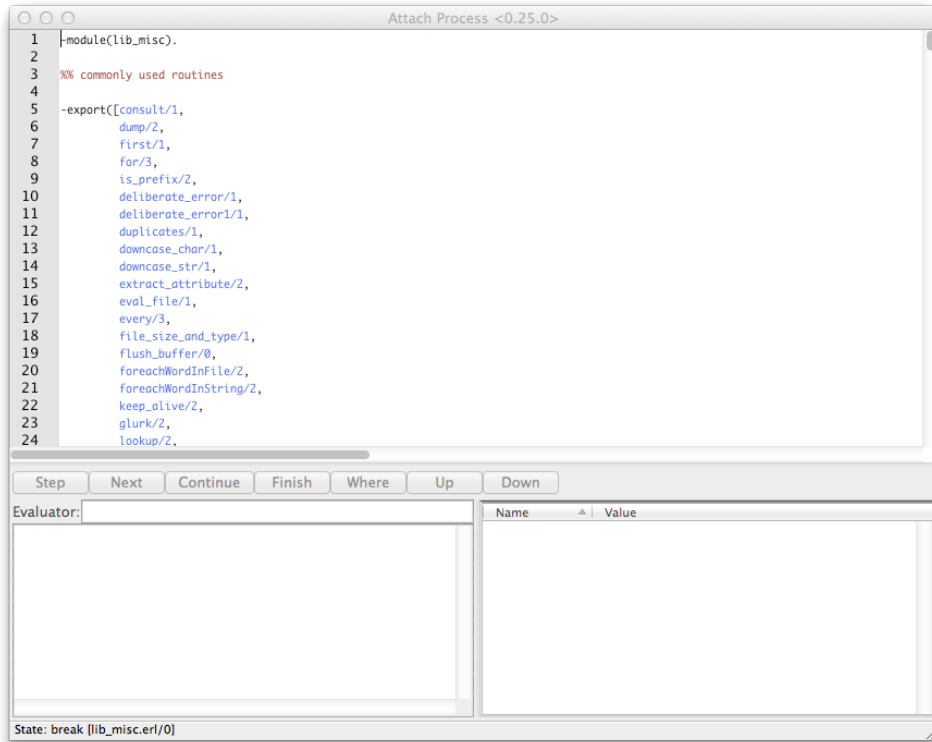
Figure 1—Debugger initial window

ii(Mod)

 Interpret the code in the module Mod.

iaa([init])

 Attach the debugger to any process executing interpreted code when that
 process is started.

To learn more about debugging, try these resources:

http://www.erlang.org/doc/apps/debugger/debugger.pdf

 The debugger reference manual is an introduction to the debugger, with
 screen dumps, API documentation, and more. It's a must-read for serious
 users of the debugger.

http://www.erlang.org/doc/man/i.html

 Here you can find the debugger commands that are available in the shell.

21.8 Tracing Messages and Process Execution

You can always trace a process without having to compile your code in a special way. Tracing a process (or processes) provides a powerful way of understanding how your system behaves and can be used to test complex systems without modifying the code. This is particularly useful in embedded systems or where you cannot modify the code being tested.

At a low level, we can set up a trace by calling a number of Erlang BIFs. Using these BIFs for setting up complex traces is difficult, so several libraries are designed to make this task easier.

We'll start by looking at the low-level Erlang BIFs for tracing and see how to set up a simple tracer; then we'll review the libraries that can provide a higher-level interface to the trace BIFs.

For low-level tracing, two BIFs are particularly important. erlang:trace/3 says, basically, "I want to monitor this process, so please send me a message if something interesting happens." erlang:trace_pattern defines what counts as being "interesting."

erlang:trace(PidSpec, How, FlagList)

> This starts tracing. PidSpec tells the system what to trace. How is a boolean that can turn the trace on or off. FlagList governs what is to be traced (for example, we can trace all function calls, all messages being sent, when garbage collections occur, and so on).

> Once we have called erlang:trace/3, the process that called this BIF will be sent trace messages when trace events occur. The trace events themselves are determined by calling erlang:trace_pattern/3.

erlang:trace_pattern(MFA, MatchSpec, FlagList)

> This is used to set up a *trace pattern*. If the pattern is matched, then the actions requested are performed. Here MFA is a {Module, Function, Args} tuple that says to which code the trace pattern applies. MatchSpec is a pattern that is tested every time the function specified by MFA is entered, and FlagList tells what to do if the tracing conditions are satisfied.

Writing match specifications for MatchSpec is complicated and doesn't really add much to our understanding of tracing. Fortunately, some libraries[4] make this easier.

4.　http://www.erlang.org/doc/man/ms_transform.html

Using the previous two BIFs, we can write a simple tracer. trace_module(Mod, Fun) sets up tracing on the module Mod and then evaluates Fun(). We want to trace all function calls and return values in the module Mod.

tracer_test.erl
```erlang
trace_module(Mod, StartFun) ->
    %% We'll spawn a process to do the tracing
    spawn(fun() -> trace_module1(Mod, StartFun) end).

trace_module1(Mod, StartFun) ->
    %% The next line says: trace all function calls and return
    %%                     values in Mod
    erlang:trace_pattern({Mod, '_','_'},
                         [{'_',[],[{return_trace}]}],
                         [local]),
    %% spawn a function to do the tracing
    S = self(),
    Pid = spawn(fun() -> do_trace(S, StartFun) end),
    %% setup the trace. Tell the system to start tracing
    %% the process Pid
    erlang:trace(Pid, true, [call,procs]),
    %% Now tell Pid to start
    Pid ! {self(), start},
    trace_loop().

%% do_trace evaluates StartFun()
%%    when it is told to do so by Parent
do_trace(Parent, StartFun) ->
    receive
        {Parent, start} ->
            StartFun()
    end.

%% trace_loop displays the function call and return values
trace_loop() ->
    receive
        {trace,_,call, X} ->
            io:format("Call: ~p~n",[X]),
            trace_loop();
        {trace,_,return_from, Call, Ret} ->
            io:format("Return From: ~p => ~p~n",[Call, Ret]),
            trace_loop();
        Other ->
            %% we get some other message - print them
            io:format("Other = ~p~n",[Other]),
            trace_loop()
    end.
```

Now we define a test case like this:

tracer_test.erl
```
test2() ->
    trace_module(tracer_test, fun() -> fib(4) end).

fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

Then we can trace our code.

```
1> c(tracer_test).
{ok,tracer_test}
2> tracer_test:test2().
<0.42.0>Call: {tracer_test,'-trace_module1/2-fun-0-',
[<0.42.0>,#Fun<tracer_test.0.36786085>]}
Call: {tracer_test,do_trace,[<0.42.0>,#Fun<tracer_test.0.36786085>]}
Call: {tracer_test,'-test2/0-fun-0-',[]}
Call: {tracer_test,fib,[4]}
Call: {tracer_test,fib,[3]}
Call: {tracer_test,fib,[2]}
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Call: {tracer_test,fib,[0]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 2
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 3
Call: {tracer_test,fib,[2]}
Call: {tracer_test,fib,[1]}
Return From: {tracer_test,fib,1} => 1
Call: {tracer_test,fib,[0]}
Return From: {tracer_test,fib,1} => 1
Return From: {tracer_test,fib,1} => 2
Return From: {tracer_test,fib,1} => 5
Return From: {tracer_test,'-test2/0-fun-0-',0} => 5
Return From: {tracer_test,do_trace,2} => 5
Return From: {tracer_test,'-trace_module1/2-fun-0-',2} => 5
Other = {trace,<0.43.0>,exit,normal}
```

Tracer output can be extremely fine-grained and is valuable for understanding
the dynamic behavior of a program. Reading code gives us a static picture of
the system. But observing the message flows gives us a view of the dynamic
behavior of the system.

Using the Trace Libraries

We can perform the same trace as the previous one using the library module
dbg. This hides all the details of the low-level Erlang BIFs.

```
tracer_test.erl
test1() ->
    dbg:tracer(),
    dbg:tpl(tracer_test,fib,'_',
            dbg:fun2ms(fun(_) -> return_trace() end)),
    dbg:p(all,[c]),
    tracer_test:fib(4).
```

Running this, we get the following:

```
1> tracer_test:test1().
(<0.34.0>) call tracer_test:fib(4)
(<0.34.0>) call tracer_test:fib(3)
(<0.34.0>) call tracer_test:fib(2)
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) call tracer_test:fib(0)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 2
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 3
(<0.34.0>) call tracer_test:fib(2)
(<0.34.0>) call tracer_test:fib(1)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) call tracer_test:fib(0)
(<0.34.0>) returned from tracer_test:fib/1 -> 1
(<0.34.0>) returned from tracer_test:fib/1 -> 2
(<0.34.0>) returned from tracer_test:fib/1 -> 5
```

This achieves the same as the previous section but with library code instead of using the trace BIFs. For fine-grained control, you'll probably want to write you own custom tracing code using the trace BIFs. For quick experiments, the library code is sufficient.

To learn more about tracing, you need to read three manual pages for the following modules:

- dbg provides a simplified interface to the Erlang trace BIFs.
- ttb is yet another interface to the trace BIFs. It is higher level than dbg.
- ms_transform makes match specifications for use in the tracer software.

21.9 Frameworks for Testing Erlang Code

For complex projects, you'll want to set up a test framework and integrate it with your build system. Here are a couple of frameworks that you might like to investigate:

Common Test Framework

> The Common Test Framework is part of the Erlang/OTP distribution. It provides a complete set of tools for automating tests. The common test framework is used to test the Erlang distribution itself and to test many Ericsson products.

Property-based testing

> Property-based testing is relatively new and an extremely good technique for shaking out hard-to-find bugs in your code. Instead of writing test cases, we describe the properties of the system in a form of predicate logic. The test tools generate random test cases that are consistent with the properties of the system and check that these properties are not violated.

> Two property-based tools exist for testing Erlang programs: QuickCheck, a commercial program supplied by a Swedish company called Quviq,[5] and proper,[6] which was inspired by QuickCheck.

Congratulations, you now know about programming sequential and concurrent programs, about files and sockets, about data storage and databases, and about debugging and testing your programs.

In the next chapter, we change gears. We talk about the Open Telecom Platform (OTP). This is a really weird name that reflects the history of Erlang. OTP is an application framework, or a set of programming patterns, that simplifies writing fault-tolerant distributed systems code. It has been battle-tested in a very large number of applications, so it provides a good starting point for your own projects.

Exercises

1. Create a new directory and copy the standard library module dict.erl to this directory. Add an error to dict.erl so it will crash if some particular line of code is executed. Compile the module.

2. We now have a broken module dict, but we probably don't yet *know* that it's broken, so we need to provoke an error. Write a simple test module that calls dict in various ways to see whether you can get dict to crash.

3. Use the coverage analyzer to check how many times each line in dict has been executed. Add more test cases to your test module, checking to see that you are covering all the lines of code in dict. The goal is to make sure

---

that every line of code in `dict` is executed. Once you know which lines are not being executed, it's often an easy task to work backward and figure out which lines of code in the test cases would cause a particular line of code to be executed.

Continue doing this until you manage to crash the program. Sooner or later this should happen, since by the time every line of code has been covered, you will have triggered the error.

4. Now we have an error. Pretend you don't know where the error is. Use the techniques in this chapter to find the error.

This exercise works even better if you don't know where the error is. Get a friend of yours to break a few of your modules. Run coverage tests on these modules to try to provoke the errors. Once you've provoked the errors, use debugging techniques to find what went wrong.

Introducing OTP

OTP stands for the Open Telecom Platform. The name is actually misleading, because OTP is far more general than you might think. It's an application operating system and a set of libraries and procedures used for building large-scale, fault-tolerant, distributed applications. It was developed at the Swedish telecom company Ericsson and is used within Ericsson for building fault-tolerant systems. The standard Erlang distribution contains the OTP libraries.

OTP contains a number of powerful tools—such as a complete web server, an FTP server, a CORBA ORB,[1] and so on—all written in Erlang. OTP also contains state-of-the-art tools for building telecom applications with implementations of H248, SNMP, and an ASN.1-to-Erlang cross-compiler (these are commonly used protocols in the telecommunications industry). I'm not going to talk about these here; you can find a lot more about these subjects at the Erlang website.[2]

If you want to program your own applications using OTP, then the central concept that you will find useful is the OTP *behavior*. A behavior encapsulates common behavioral patterns—think of it as an application framework that is parameterized by a *callback* module.

The power of OTP comes from the fact that properties such as fault tolerance, scalability, dynamic-code upgrade, and so on, can be provided by the behavior itself. In other words, the writer of the callback does not have to worry about things such as fault tolerance because this is provided by the behavior. For the Java-minded, you can think of a behavior as a J2EE container.

Put simply, the behavior solves the nonfunctional parts of the problem, while the callback solves the functional parts. The nice part about this is that the

---

1. http://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture
2. http://www.erlang.org/

nonfunctional parts of the problem (for example, how to do live code upgrades) are the same for all applications, whereas the functional parts (as supplied by the callback) are different for every problem.

In this chapter, we'll look at one of these behaviors, the gen_server module, in greater detail. But, before we get down to the nitty-gritty details of how the gen_server works, we'll first start with a simple server (the simplest server we can possibly imagine) and then change it in a number of small steps until we get to the full gen_server module. That way, you should be in a position to really understand how gen_server works and be ready to poke around in the gory details.

Here is the plan of this chapter:

1. Write a small client-server program in Erlang.
2. Slowly generalize this program and add a number of features.
3. Move to the real code.

22.1 The Road to the Generic Server

This is the most important section in the entire book, so read it once, read it twice, read it a hundred times—just make sure the message sinks in.

This section is about building abstractions; we'll look at a module called gen_server.erl. The gen server is one of the most commonly used abstractions in the OTP system, but a lot of folks never look inside gen_server.erl to see how it works. Once you understand how the gen server was constructed, you'll be able to repeat the abstraction process to build your own abstractions.

We're going to write four little servers called server1, server2, server3, server4, each slightly different from the last. server4 will be similar to the gen server in the Erlang distribution. The goal is to totally separate the nonfunctional parts of the problem from the functional parts of the problem. That last sentence probably didn't mean much to you now, but don't worry—it soon will. Take a deep breath.

Server 1: The Basic Server

The following is our first attempt. It's a little server that we can parameterize with a callback module.

server1.erl
```erlang
-module(server1).
-export([start/2, rpc/2]).
```

```erlang
start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.
loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            {Response, State1} = Mod:handle(Request, State),
            From ! {Name, Response},
            loop(Name, Mod, State1)
    end.
```

This small amount of code captures the quintessential nature of a server. Let's write a *callback* for server1. Here's a name server callback:

name_server.erl
```erlang
-module(name_server).
-export([init/0, add/2, find/1, handle/2]).
-import(server1, [rpc/2]).

%% client routines
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
find(Name)       -> rpc(name_server, {find, Name}).

%% callback routines
init() -> dict:new().
handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({find, Name}, Dict)       -> {dict:find(Name, Dict), Dict}.
```

This code actually performs two tasks. It serves as a callback module that is called from the server framework code, and at the same time, it contains the interfacing routines that will be called by the client. The usual OTP convention is to combine both functions in the same module.

Just to prove that it works, do this:

```erlang
1> server1:start(name_server, name_server).
true
2> name_server:add(joe, "at home").
ok
3> name_server:find(joe).
{ok,"at home"}
```

Now *stop and think*. The callback had no code for concurrency, no spawn, no send, no receive, and no register. It is pure sequential code—*nothing else. This means we can write client-server models without understanding anything about the underlying concurrency models.*

This is the *basic* pattern for all servers. Once you understand the basic *structure*, it's easy to "roll your own."

Server 2: A Server with Transactions

Here's a server that crashes the client if the query in the server results in an exception:

```erlang
server2.erl
-module(server2).
-export([start/2, rpc/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _:Why ->
                    log_the_error(Name, Request, Why),
                    %% send a message to cause the client to crash
                    From ! {Name, crash},
                    %% loop with the *original* state
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

This code provides "transaction semantics" in the server—it loops with the *original value* of State if an exception was raised in the handler function. But if the handler function succeeded, then it loops with the value of NewState provided by the handler function.

When the handler function fails, the client that sent the message that caused the failure is sent a message that causes it to crash. The client cannot proceed, because the request it sent to the server caused the handler function to crash. But any other client that wants to use the server will not be affected. Moreover, the state of the server is not changed when an error occurs in the handler.

Note that the callback module for this server is *exactly* the same as the callback module we used for server1. *By changing the server and keeping the callback module constant, we can change the nonfunctional behavior of the callback module.*

Note: The last statement wasn't strictly true. We have to make a small change to the callback module when we go from server1 to server2, and that is to change the name in the -import declaration from server1 to server2. Otherwise, there are no changes.

Server 3: A Server with Hot Code Swapping

Now we'll add hot code swapping. Most servers execute a fixed program, and if you want to modify the behavior of the server, you have to stop the server and then restart it with the modified code. When we want to change the behavior of our server, we don't stop it; we just send it a message containing the new code, and it picks up the new code and continues with the new code and the old session data. This process is called *hot code swapping*.

```erlang
server3.erl
-module(server3).
-export([start/2, rpc/2, swap_code/2]).
start(Name, Mod) ->
    register(Name,
             spawn(fun() -> loop(Name,Mod,Mod:init()) end)).
swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, Response} -> Response
    end.
loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallBackMod}} ->
            From ! {Name, ack},
            loop(Name, NewCallBackMod, OldState);
        {From, Request} ->
            {Response, NewState} = Mod:handle(Request, OldState),
            From ! {Name, Response},
            loop(Name, Mod, NewState)
    end.
```

If we send the server a swap code message, it changes the callback module to the new module contained in the message.

We can demonstrate this by starting server3 with a callback module and then dynamically swapping the callback module. We can't use name_server as the callback module because we hard-compiled the name of the server into the module. So, we make a copy of this, calling it name_server1 where we change the name of the server.

name_server1.erl
```erlang
-module(name_server1).
-export([init/0, add/2, find/1, handle/2]).
-import(server3, [rpc/2]).

%% client routines
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
find(Name)       -> rpc(name_server, {find, Name}).

%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle({find, Name}, Dict)       -> {dict:find(Name, Dict), Dict}.
```

First we'll start server3 with the name_server1 callback module.

```erlang
1> server3:start(name_server, name_server1).
true
2> name_server1:add(joe, "at home").
ok
3> name_server1:add(helen, "at work").
ok
```

Now suppose we want to find all the names that are served by the name server. There is no function in the API that can do this—the module name_server has only add and lookup access routines.

With lightning speed, we fire up our text editor and write a new callback module.

new_name_server.erl
```erlang
-module(new_name_server).
-export([init/0, add/2, all_names/0, delete/1, find/1, handle/2]).
-import(server3, [rpc/2]).

%% interface
all_names()     -> rpc(name_server, allNames).
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
delete(Name)    -> rpc(name_server, {delete, Name}).
find(Name)      -> rpc(name_server, {find, Name}).
```

```
%% callback routines
init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};
handle(allNames, Dict)           -> {dict:fetch_keys(Dict), Dict};
handle({delete, Name}, Dict)     -> {ok, dict:erase(Name, Dict)};
handle({find, Name}, Dict)       -> {dict:find(Name, Dict), Dict}.
```

We compile this and tell the server to swap its callback module.

```
4> c(new_name_server).
{ok,new_name_server}
5> server3:swap_code(name_server, new_name_server).
ack
```

Now we can run the new functions in the server.

```
6> new_name_server:all_names().
[joe,helen]
```

Here we *changed the callback module on the fly*—this is dynamic code upgrade, in action before your eyes, with no black magic.

Now stop and think again. The last two tasks we have done are generally considered to be pretty difficult, in fact, very difficult. Servers with "transaction semantics" are difficult to write, and servers with dynamic code upgrade are difficult to write, but this technique makes it easy.

This technique is extremely powerful. Traditionally we think of servers as programs with state that change state when we send them messages. The code in the servers is fixed the first time it is called, and if we want to change the code in the server, we have to stop the server and change the code, and then we can restart the server. In the examples we have given, the code in the server can be changed just as easily as we can change the state of the server. We use this technique a lot in products that are *never* taken out of service for software maintenance upgrades.

Server 4: Transactions and Hot Code Swapping

In the last two servers, code upgrade and transaction semantics were separate. Let's *combine* them into a single server. Hold onto your hats.

server4.erl
```
-module(server4).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
    register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).
swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).
```

```erlang
rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end.

loop(Name, Mod, OldState) ->
    receive
        {From, {swap_code, NewCallbackMod}} ->
            From ! {Name, ok, ack},
            loop(Name, NewCallbackMod, OldState);
        {From, Request} ->
            try Mod:handle(Request, OldState) of
                {Response, NewState} ->
                    From ! {Name, ok, Response},
                    loop(Name, Mod, NewState)
            catch
                _: Why ->
                    log_the_error(Name, Request, Why),
                    From ! {Name, crash},
                    loop(Name, Mod, OldState)
            end
    end.

log_the_error(Name, Request, Why) ->
    io:format("Server ~p request ~p ~n"
              "caused exception ~p~n",
              [Name, Request, Why]).
```

This server provides both hot code swapping and transaction semantics. Neat.

Server 5: Even More Fun

Now that we understand the idea of dynamic code change, we can have even more fun. Here's a server that does nothing at all until you tell it to *become* a particular type of server:

server5.erl
```erlang
-module(server5).
-export([start/0, rpc/2]).
start() -> spawn(fun() -> wait() end).
wait() ->
    receive
        {become, F} -> F()
    end.
rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.
```

If we start this and then send it a {become, F} message, it will become an F server by evaluating F(). We'll start it.

```
1> Pid = server5:start().
<0.57.0>
```

Our server does nothing and just waits for a become message.

Let's now define a server function. It's nothing complicated, just something to compute factorial.

```
my_fac_server.erl
-module(my_fac_server).
-export([loop/0]).

loop() ->
    receive
        {From, {fac, N}} ->
            From ! {self(), fac(N)},
            loop();
        {become, Something} ->
            Something()
    end.

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

Just make sure it's compiled, and then we can tell process <0.57.0> to become a factorial server.

```
2> c(my_fac_server).
{ok,my_fac_server}
3> Pid ! {become, fun my_fac_server:loop/0}.
{become,#Fun<my_fac_server.loop.0>}
```

Now that our process has become a factorial server, we can call it.

```
4> server5:rpc(Pid, {fac,30}).
265252859812191058636308480000000
```

Our process will remain a factorial server, until we send it a {become, Something} message and tell it to do something else.

As you can see from the previous examples, we can make a range of different types of servers, with different semantics and some quite surprising properties. This technique is almost too powerful. Used to its full potential, it can yield small programs of quite surprising power and beauty. When we make industrial-scale projects with dozens to hundreds of programmers involved, we might not actually want things to be too dynamic. We have to try to strike a balance between having something general and powerful and having

> **Erlang on PlanetLab**
>
> A few years ago, when I had my research hat on, I was working with PlanetLab. I had access to the PlanetLab network (a planet-wide research network (http://www.planet-lab.org)), so I installed "empty" Erlang servers on all the PlanetLab machines (about 450 of them). I didn't really know what I would do with the machines, so I just set up the server infrastructure to do something later.
>
> Once I had gotten this layer running, it was an easy job to send messages to the empty servers telling them to become real servers.
>
> The usual approach is to start (for example) a web server and then install web server plug-ins. My approach was to back off one step and install an empty server. Later the plug-in turns the empty server into a web server. When we're done with the web server, we might tell it to become something else.

something that is useful for commercial products. Having code that can morph into new versions while it runs is beautiful but terrible to debug if something goes wrong later. If we have made dozens of dynamic changes to our code and it then crashes, finding out exactly what went wrong is not easy.

The server examples in this section are actually not quite correct. They are written this way so as to emphasize the ideas involved, but they do have one or two extremely small and subtle errors. I'm not going to tell you immediately what they are, but I'll give you some hints at the end of the chapter.

The Erlang module gen_server is the kind of logical conclusion of a succession of successively sophisticated servers (just like the ones we've written so far in this chapter).

It has been in use in industrial products since 1998. Hundreds of servers can be part of a single product. These servers have been written by programmers using regular sequential code. All the error handling and all the non-functional behavior is factored out in the generic part of the server.

So now we'll take a great leap of imagination and look at the real gen_server.

22.2 Getting Started with gen_server

I'm going to throw you in at the deep end. Here's the simple three-point plan for writing a gen_server callback module:

1. Decide on a callback module name.
2. Write the interface functions.
3. Write the six required callback functions in the callback module.

This is really easy. Don't think—just follow the plan!

Step 1: Decide on the Callback Module Name

We're going to make a simple payment system. We'll call the module my_bank.

Step 2: Write the Interface Routines

We'll define five interface routines, all in the module my_bank.

start()
    Open the bank.

stop()
    Close the bank.

new_account(Who)
    Create a new account.

deposit(Who, Amount)
    Put money in the bank.

withdraw(Who, Amount)
    Take money out, if in credit.

Each of these results in exactly one call to the routines in gen_server, as follows:

my_bank.erl
```
start() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
stop()  -> gen_server:call(?MODULE, stop).

new_account(Who)       -> gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).
```

gen_server:start_link({local, Name}, Mod, ...) starts a *local* server. If the first argument was the atom global, it would start a global server that could be accessed on a cluster of Erlang nodes. The second argument to start_link is Mod and is the name of the callback module. The macro ?MODULE expands to the module name my_bank. We'll ignore the other arguments to gen_server:start_link for now.

gen_server:call(?MODULE, Term) is used for a remote procedure call to the server.

Step 3: Write the Callback Routines

Our callback module must export six callback routines: init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, and code_change/3.

To make life easy, we can use a number of templates to make a gen_server. Here's the simplest:

```
gen_server_template.mini
-module().
%% gen_server_mini_template
-behaviour(gen_server).
-export([start_link/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
        terminate/2, code_change/3]).

start_link() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
init([]) -> {ok, State}.

handle_call(_Request, _From, State) -> {reply, Reply, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

The template contains a simple skeleton that we can fill in to make our server. The keyword -behaviour is used by the compiler so that it can generate warning or error messages if we forget to define the appropriate callback functions. The macro ?SERVER needs to be defined as the name of the server in the start_link() function, because it is not defined by default.

Tip: If you're using Emacs, then you can pull in a gen_server template in a few keystrokes. If you edit in erlang-mode, then the Erlang > Skeletons menu offers a tab that creates a gen_server template. If you don't have Emacs, don't panic. I've included the template at the end of the chapter.

We'll start with the template and edit it a bit. All we have to do is get the arguments in the interfacing routines to agree with the arguments in the template.

The most important bit is the handle_call/3 function. We have to write code that matches the three query terms defined in the interface routines. That is, we have to fill in the dots in the following:

```
handle_call({new, Who}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
handle_call({add, Who, Amount}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
handle_call({remove, Who, Amount}, From, State} ->
    Reply  = ...
    State1 = ...
    {reply, Reply, State1};
```

The values of Reply in this code are sent back to the client as the return values of the remote procedure calls.

State is just a variable representing the global state of the server that gets passed around in the server. In our bank module, the state never changes; it's just an ETS table index that is a constant (although the content of the table changes).

When we've filled in the template and edited it a bit, we end up with the following code:

my_bank.erl
```erlang
init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                [] -> ets:insert(Tab, {Who,0}),
                       {welcome, Who};
                [_] -> {Who, you_already_are_a_customer}
            end,
    {reply, Reply, Tab};

handle_call({add,Who,X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                [] -> not_a_customer;
                [{Who,Balance}] ->
                    NewBalance = Balance + X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance}
            end,
    {reply, Reply, Tab};

handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
                [] -> not_a_customer;
                [{Who,Balance}] when X =< Balance ->
                    NewBalance = Balance - X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance};
                [{Who,Balance}] ->
                    {sorry,Who,you_only_have,Balance,in_the_bank}
            end,
    {reply, Reply, Tab};

handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

We start the server by calling gen_server:start_link(Name, CallBackMod, StartArgs, Opts); then the first routine to be called in the callback module is Mod:init(StartArgs), which must return {ok, State}. The value of State reappears as the third argument in handle_call.

Note how we stop the server. handle_call(stop, From, Tab) returns {stop, normal, stopped, Tab}, which stops the server. The second argument (normal) is used as the first argument to my_bank:terminate/2. The third argument (stopped) becomes the return value of my_bank:stop().

That's it, we're done. So, let's go visit the bank.

```
1> my_bank:start().
{ok,<0.33.0>}
2> my_bank:deposit("joe", 10).
not_a_customer
3> my_bank:new_account("joe").
{welcome,"joe"}
4> my_bank:deposit("joe", 10).
{thanks,"joe",your_balance_is,10}
5> my_bank:deposit("joe", 30).
{thanks,"joe",your_balance_is,40}
6> my_bank:withdraw("joe", 15).
{thanks,"joe",your_balance_is,25}
7> my_bank:withdraw("joe", 45).
{sorry,"joe",you_only_have,25,in_the_bank}
```

22.3 The gen_server Callback Structure

Now that we understand the idea, we'll take a more detailed look at the gen_server callback structure.

Starting the Server

The call gen_server:start_link(Name, Mod, InitArgs, Opts) starts everything. It creates a generic server called Name. The callback module is Mod. Opts controls the behavior of the generic server. Here we can specify the logging of messages, debugging of functions, and so on. The generic server starts by calling Mod:init(InitArgs).

The template entry for init is shown in Figure 2, *init template entry*, on page 373 (the full template can be found in Section A1.1, *The Generic Server Template, on page 471*):

In normal operation, we just return {ok, State}. For the meaning of the other arguments, consult the manual page for gen_server.

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Initializes the server
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                     ignore |
%%                     {stop, Reason}
%% @end
%%--------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.
```

Figure 2—init template entry

If {ok, State} is returned, then we have successfully started the server, and the initial state is State.

Calling the Server

To call the server, the client program calls gen_server:call(Name, Request). This results in handle_call/3 in the callback module being called.

handle_call/3 has the template entry as follows:

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling call messages
%%
%% @spec handle_call(Request, From, State) ->
%%                                  {reply, Reply, State} |
%%                                  {reply, Reply, State, Timeout} |
%%                                  {noreply, State} |
%%                                  {noreply, State, Timeout} |
%%                                  {stop, Reason, Reply, State} |
%%                                  {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.
```

Request (the second argument of gen_server:call/2) reappears as the first argument of handle_call/3. From is the PID of the requesting client process, and State is the current state of the client.

Normally we return {reply, Reply, NewState}. When this happens, Reply goes back to the client, where it becomes the return value of gen_server:call. NewState is the next state of the server.

The other return values, {noreply, ..} and {stop, ..}, are used relatively infrequently. no reply causes the server to continue, but the client will wait for a reply so the server will have to delegate the task of replying to some other process. Calling stop with the appropriate arguments will stop the server.

Calls and Casts

We've seen the interplay between gen_server:call and handle_call. This is used for implementing *remote procedure calls.* gen_server:cast(Name, Msg) implements a *cast*, which is just a call with no return value (actually just a message, but traditionally it's called a cast to distinguish it from a remote procedure call).

The corresponding callback routine is handle_cast; the template entry is like this:

```
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling cast messages
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
%%                                  {noreply, State, Timeout} |
%%                                  {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
    {noreply, State}.
```

The handler usually just returns {noreply, NewState}, which changes the state of the server, or {stop, …}, which stops the server.

Spontaneous Messages to the Server

The callback function handle_info(Info, State) is used for handling spontaneous messages to the server. Spontaneous messages are any messages that arrive at the server that were not sent by explicitly calling gen_server:call or gen_server:cast. For example, if the server is linked to another process and is trapping exits, then it might suddenly receive a unexpected {'EXIT', Pid, What} message. Alternatively, any process in the system that discovers the PID of the generic server can just send it a message. Any message like this ends up at the server as the value of Info.

The template entry for handle_info is as follows:

```erlang
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling all non call/cast messages
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.
```

The return values are the same as for handle_cast.

Hasta la Vista, Baby

The server can terminate for many reasons. One of the handle_Something routines might return a {stop, Reason, NewState}, or the server might crash with {'EXIT', reason}. In all of these circumstances, no matter how they occurred, terminate(Reason, NewState) will be called. Here's the template:

```erlang
%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any
%% necessary cleaning up. When it returns, the <mod>gen_server</mod> terminates
%% with Reason. The return value is ignored.
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
    ok.
```

This code can't return a new state because we've terminated. But knowing the state of the server when we terminated is extremely useful; we could store the state on disk, send it in a message to some other process, or discard it depending upon the application. If you want your server to be restarted in the future, you'll have to write an "I'll be back" function that is triggered by terminate/2.

Code Change

You can dynamically change the state of your server while it is running. This callback function is called by the release handling subsystem when the system performs a software upgrade.

This topic is described in detail in the section on release handling in the OTP systems documentation.[3]

```erlang
%%--------------------------------------------------------------------
%% @private
%% @doc
%% Convert process state when code is changed
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

22.4 Filling in the gen_server Template

Making an OTP gen_server is largely a matter of filling in a boilerplate template with some code of your own. The following is an example; the individual sections of the gen_server were listed in the previous section. The gen_server template itself is built into Emacs, but if you're not using Emacs, you can find the entire template in Section A1.1, *The Generic Server Template*, on page 471.

I've filled in the template to make a bank module called my_bank. This code is derived from the template. I've removed all the comments from the template so you can clearly see the structure of the code.

my_bank.erl
```erlang
-module(my_bank).

-behaviour(gen_server).
-export([start/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
-compile(export_all).
-define(SERVER, ?MODULE).

start() -> gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
stop()  -> gen_server:call(?MODULE, stop).

new_account(Who)      -> gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount)  -> gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).

init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
```

---

3. http://www.erlang.org/doc/pdf/otp-system-documentation.pdf.

```
    Reply = case ets:lookup(Tab, Who) of
               []  -> ets:insert(Tab, {Who,0}),
                      {welcome, Who};
               [_] -> {Who, you_already_are_a_customer}
            end,
    {reply, Reply, Tab};

handle_call({add,Who,X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
               []  -> not_a_customer;
               [{Who,Balance}] ->
                    NewBalance = Balance + X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance}
            end,
    {reply, Reply, Tab};

handle_call({remove,Who, X}, _From, Tab) ->
    Reply = case ets:lookup(Tab, Who) of
               []  -> not_a_customer;
               [{Who,Balance}] when X =< Balance ->
                    NewBalance = Balance - X,
                    ets:insert(Tab, {Who, NewBalance}),
                    {thanks, Who, your_balance_is,  NewBalance};
               [{Who,Balance}] ->
                    {sorry,Who,you_only_have,Balance,in_the_bank}
            end,
    {reply, Reply, Tab};

handle_call(stop, _From, Tab) ->
    {stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

22.5 Digging Deeper

The gen_server is actually rather simple. We haven't been through *all* the
interface functions in gen_server, and we haven't talked about all the arguments
to all the interface functions. Once you understand the basic ideas, you can
look up the details in the manual page for gen_server.

In this chapter, we looked only at the simplest possible way to use gen_server,
but this should be adequate for most purposes. More complex applications
often let gen_server reply with a noreply return value and delegate the real reply

to another process. For information about this, read the "Design Principles" documentation[4] and the manual pages for the modules sys and proc_lib.

This chapter introduced the idea of abstracting the behavior of a server into two components: a *generic* component that can be used for all servers and a *specific* component (or handler) that can be used to customize the generic component. The main benefit of this is that the code neatly factors into two parts. The generic component deals with many aspects of the concurrency and error handling, and the handler has only sequential code.

Following this, we introduced the first major behavior in the OTP system, the gen_server, and showed how it could be built from a fairly simple and easy-to-understand server in a number of small transformation steps.

The gen_server can be used for many purposes, but it is not a universal panacea. Sometimes the client-server interaction pattern of the gen_server feels awkward and does not fit naturally with your problem. If this happens, you should rethink the transformation steps needed to make the gen_server and modify them to the specific needs of your problem.

When we turn from individual servers to systems, we will need several servers; we want to monitor them, restarting failed servers and logging errors in a consistent manner. This is the subject of the next chapter.

Exercises

In these exercises, we'll make a server in the module job_centre, which uses gen_server to implement a job management service. The job center keeps a queue of jobs that have to be done. The jobs are numbered. Anybody can add jobs to the queue. Workers can request jobs from the queue and tell the job center that a job has been performed. Jobs are represented by funs. To do a job F, a worker must evaluate the function F().

1. Implement the basic job center functionality, with the following interface:

    job_centre:start_link() -> true.
       Start the job center server.

    job_centre:add_job(F) -> JobNumber.
       Add a job F to the job queue. Return an integer job number.

    job_centre:work_wanted() -> {JobNumber,F} | no.
       Request work. When a worker wants a job, it calls job_centre:work_wanted().
       If there are jobs in the queue, a tuple {JobNumber, F} is returned. The

---

4.   http://www.erlang.org/doc/pdf/design_principles.pdf

worker performs the job by evaluating F(). If there are no jobs in the queue, no is returned. Make sure the same job cannot be allocated to more than one worker at a time. Make sure that the system is fair, meaning that jobs are handed out in the order they were requested.

job_centre:job_done(JobNumber)

Signal that a job has been done. When a worker has completed a job, it must call job_centre:job_done(JobNumber).

2. Add a statistics call called job_centre:statistics() that reports the status of the jobs in the queue and of jobs that are in progress and that have been done.

3. Add code to monitor the workers. If a worker dies, make sure the jobs it was doing are returned to the pool of jobs waiting to be done.

4. Check for lazy workers; these are workers that accept jobs but don't deliver on time. Change the work wanted function to return {JobNumber, JobTime, F} where JobTime is a time in seconds that the worker has to complete the job by. At time JobTime - 1, the server should send a hurry_up message to the worker if it has not finished the job. And at time JobTime + 1, it should kill the worker process with an exit(Pid, youre_fired) call.

5. Optional: Implement a trade union server to monitor the rights of workers. Check that they are not fired without being sent a warning. Hint: use the process tracing primitives to do this.

Making a System with OTP

In this chapter, we're going to make a system that could function as the back end of a web-based company. Our company has two items for sale: prime numbers and areas. Customers can buy a prime number from us, or we'll calculate the area of a geometric object for them. I think our company has great potential.

We'll build two servers: one to generate prime numbers and the other to compute areas. To do this, we'll use the gen_server framework that we talked about in .

When we build the system, we have to think about errors. Even though we have thoroughly tested our software, we might not have caught all the bugs. We'll assume that one of our servers has a fatal error that crashes the server. In fact, we'll introduce a *deliberate error* into one of the servers that will cause it to crash.

When the server crashes, we'll need some mechanism to detect the fact that it has crashed and to restart it. For this we'll use the idea of a *supervision tree*. We'll create a supervisor that watches over our servers and restarts them if they crash.

Of course, if a server does crash, we'll want to know why it crashed so that we can fix the problem later. To log all errors, we'll use the OTP error logger. We'll show how to configure the error logger and how to generate error reports from the error logs.

When we're computing prime numbers and, in particular, large prime numbers, our CPU might overheat. To prevent this, we'll need to turn on a powerful fan. To do so, we'll need to think about *alarms*. We'll use the OTP event handling framework to generate and handle alarms.

All of these topics (creating a server, supervising a server, logging errors, and detecting alarms) are typical problems that have to be solved in any production system. So, even though our company might have a rather uncertain future, we can reuse the architecture here in many systems. In fact, this architecture *is* used in a number of commercially successful companies.

Finally, when everything works, we'll package all our code into an OTP *application*. This is a specialized way of grouping everything that has to do with a particular problem so that it can be started and stopped and managed by the OTP system itself.

The order in which this material is presented is slightly tricky since there are many circular dependencies between the different areas. Error logging is just a special case of event management. Alarms are just events and the error logger is a supervised process, but the process supervisor can call the error logger.

I'll try to impose some order here and present these topics in an order that makes some kind of sense. We'll do the following:

1. We'll look at the ideas used in a generic event handler.
2. We'll see how the error logger works.
3. We'll add alarm management.
4. We'll write two application servers.
5. We'll make a supervision tree and add the servers to it.
6. We'll package everything into an application.

23.1 Generic Event Handling

An *event* is just something that happens—something noteworthy that the programmer thinks somebody should do something about.

When we're programming and something noteworthy happens, we just send an event message to a registered process, like this:

```
RegProcName ! {event, E}
```

E is the event (any Erlang term). RegProcName is the name of a registered process.

We don't know (or care) what happens to the message after we have sent it. We have done our job and told somebody else that something has happened.

Now let's turn our attention to the process that receives the event messages. This is called an *event handler*. The simplest possible event handler is a "do nothing" handler. When it receives an {event, X} message, it does nothing with the event; it just throws it away.

Here's our first attempt at a generic event handler program:

```erlang
event_handler.erl
-module(event_handler).
-export([make/1, add_handler/2, event/2]).
%% make a new event handler called Name
%% the handler function is no_op -- so we do nothing with the event
make(Name) ->
    register(Name, spawn(fun() -> my_handler(fun no_op/1) end)).
add_handler(Name, Fun) -> Name ! {add, Fun}.

%% generate an event
event(Name, X) -> Name ! {event, X}.

my_handler(Fun) ->
    receive
        {add, Fun1} ->
            my_handler(Fun1);
        {event, Any} ->
            (catch Fun(Any)),
            my_handler(Fun)
    end.
no_op(_) -> void.
```

The event handler API is as follows:

event_handler:make(Name)

> Make a "do nothing" event handler called Name (an atom). This provides a place to send events to.

event_handler:event(Name, X)

> Send the event X to the event handler called Name.

event_handler:add_handler(Name, Fun)

> Add a handler Fun to the event handler called Name. Now when an event X occurs, the event handler will evaluate Fun(X).

Now we'll create an event handler and generate an error.

```
1> event_handler:make(errors).
true
2> event_handler:event(errors, hi).
{event,hi}
```

Nothing special happens, because we haven't installed a callback module in the event handler.

To get the event handler to do something, we have to write a callback module and install it in the event handler. Here's the code for an event handler call-back module:

```
motor_controller.erl
-module(motor_controller).
-export([add_event_handler/0]).

add_event_handler() ->
    event_handler:add_handler(errors, fun controller/1).
controller(too_hot) ->
    io:format("Turn off the motor~n");
controller(X) ->
    io:format("~w ignored event: ~p~n",[?MODULE, X]).
```

Once this has been compiled, it can be installed.

```
3> c(motor_controller).
{ok,motor_controller}
4> motor_controller:add_event_handler().
{add,#Fun<motor_controller.0.99476749>}
```

Now when we send events to the handler, they are processed by the motor_controller:controller/1 function.

```
5> event_handler:event(errors, cool).
motor_controller ignored event: cool
{event,cool}
6> event_handler:event(errors, too_hot).
Turn off the motor
{event,too_hot}
```

The point of this exercise was twofold. First we provided a name to which to send events; this was the registered process called errors. Then we defined a protocol to send events to this registered process. But we did not say what would happen to the message when it got there. In fact, all that happened was that we evaluated no_Op(X). Then at a later stage we installed a custom event handler. This essentially decouples the generation of an event with the processing of an event so we can decide at a later stage how we want to process the event without influencing the generation of the event.

The key point to note is that the event handler provides an infrastructure where we can install custom handlers.

The error logger infrastructure follows the event handler pattern. We can install different handlers in the error logger to get it to do different things. The alarm handling infrastructure also follows this pattern.

23.2 The Error Logger

The OTP system comes packaged with a customizable error logger. We can look at the error logger from three points of view. The *programmer view* concerns the function calls that programmers make in their code in order to log

Very Late Binding with "Change Your Mind"

Suppose we write a function that hides the event_handler:event routine from the programmer. For example, say we write the following:

lib_misc.erl
```
too_hot() ->
    event_handler:event(errors, too_hot).
```

Then we tell the programmer to call lib_misc:too_hot() in the code when things go wrong. In most programming languages, the call to the function too_hot will be statically or dynamically linked to the code that calls the function. Once it has been linked, it will perform a fixed job depending upon the code. If we change our mind later and decide that we want to do something else, we have no easy way of changing the behavior of the system.

The Erlang way of handling events is completely different. It allows us to decouple the generation of the event from the processing of the event. We can change the processing at any time we want by just sending a new handler function to the event handler. Nothing is statically linked, and the event handlers can be changed whenever you want.

Using this mechanism, we can build systems that *evolve with time* and that never need to be stopped to upgrade the code.

Note: This is not "late binding"—it's "very late binding, and you can change your mind later."

an error. The *configuration view* is concerned with where and how the error logger stores its data. The *report view* is concerned with the analysis of errors after they have occurred. We'll look at each of these in turn.

Logging an Error

As far as the programmer is concerned, the API to the error logger is simple. Here's a simple subset of the API:

-spec error_logger:error_msg(String) -> ok

Send an error message to the error logger.

```
1> error_logger:error_msg("An error has occurred\n").
=ERROR REPORT==== 15-Jul-2013::17:03:14 ===
An error has occurred
ok
```

-spec error_logger:error_msg(Format, Data) -> ok

Send an error message to the error logger. The arguments are the same as for io:format(Format, Data).

```
2> error_logger:error_msg("~s, an error has occurred\n", ["Joe"]).
=ERROR REPORT==== 15-Jul-2013::17:04:03 ===
Joe, an error has occurred
ok
```

-spec error_logger:error_report(Report) -> ok

> Send a standard error report to the error logger.

- -type Report = [{Tag, Data} | term() | string()].
- -type Tag = term().
- -type Data = term().

```
3> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).
=ERROR REPORT==== 15-Jul-2013::17:04:41 ===
tag1: data1
a_term
tag2: data
```

This is only a subset of the available API. Discussing this in detail is not particularly interesting. We'll use only error_msg in our programs anyway. The full details are in the error_logger manual page.

Configuring the Error Logger

We can configure the error logger in many ways. We can see all errors in the Erlang shell (this is the default if we don't do anything special). We can write all errors that are reported in the shell into a single formatted text file. Finally, we can create a *rotating log*. You can think of the rotating log as a large circular buffer containing messages produced by the error logger. As new messages come, they are appended to the end of the log, and when the log is full, the earliest entries in the log are deleted.

The rotating log is extremely useful. You decide how many files the log should occupy and how big each individual log file should be, and the system takes care of deleting old log files and creating new files in a large circular buffer. You can size the log to keep a record of the last few days of operations, which is usually sufficient for most purposes.

The Standard Error Loggers

When we start Erlang, we can give the system a *boot argument*.

$ erl -boot start_clean

> This creates an environment suited for program development. Only a simple form of error logging is provided. (The command erl with no boot argument is equivalent to erl -boot start_clean.)

$ erl -boot start_sasl

> This creates an environment suitable for running a production system. System Architecture Support Libraries (SASL) takes care of error logging, overload protection, and so on.

Log file configuration is best done from configuration files, because nobody can ever remember all the arguments to the logger. In the following sections, we'll look at how the default system works and then look at four specific configurations that change how the error logger works.

SASL with No Configuration

Here's what happens when we start SASL with no configuration file:

```
$ erl -boot start_sasl
Erlang R16B (erts-5.10.1) [source] [smp:2:2] ...

=PROGRESS REPORT==== 26-May-2013::12:48:37 ===
supervisor: {local,sasl_safe_sup}
   started: [{pid,<0.35.0>},
             {name,alarm_handler},
             {mfargs,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]
... many lines removed ...
Eshell V5.10.1  (abort with ^G)
```

Now we'll call one of the routines in error_logger to report an error.

```
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 26-May-2013::12:54:03 ===
This is an error
ok
```

Note the error is reported in the Erlang shell. Where the error is reported depends upon the error logger configuration.

Controlling What Gets Logged

The error logger produces a number of types of report.

Supervisor reports

> These are issued whenever an OTP supervisor starts or stops a supervised process (we'll talk about supervisors in Section 23.5, *The Supervision Tree, on page 396*).

Progress reports

> These are issued whenever an OTP supervisor starts or stops.

Crash reports

> These are issued when a process started by an OTP behavior terminates with an exit reason other than normal or shutdown.

These three reports are produced automatically without the programmer having to do anything.

In addition, we can explicitly call routines in the error_logger module to produce three types of log report. These let us log errors, warnings, and informational messages. These three terms have no semantic meaning; they are merely tags used by the programmer to indicate the nature of the entry in the error log.

Later, when the error log is analyzed, we can use these tags to help us decide which log entry to investigate. When we configure the error logger, we can choose to save only errors and discard all other types of entry. Now we'll write the configuration file elog1.config to configure the error logger.

elog1.config
```
%% no tty
[{sasl, [
        {sasl_error_logger, false}
        ]}].
```

If we start the system with this configuration file, we'll get only error reports and not progress reports and so on. All these error reports are only in the shell.

```
$ erl -boot start_sasl -config elog1
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 15-Jul-2013::11:53:08 ===
This is an error
ok
```

Text File and Shell

The next configuration file lists error reports in the shell, and all progress reports are saved in a file.

elog2.config
```
%% single text file - minimal tty

[{sasl, [
        %% All reports go to this file
        {sasl_error_logger, {file, "/Users/joe/error_logs/THELOG"}}
        ]}].
```

To test this, we start Erlang, generate an error message, and then look in the log file.

```
$ erl -boot start_sasl -config elog2
Erlang R16B (erts-5.10.1) [source] [smp:2:2] ...
Eshell V5.10.1  (abort with ^G)
1> error_logger:error_msg("This is an error\n").
=ERROR REPORT==== 26-May-2013::13:07:46 ===
This is an error
ok
```

If we now look in /Users/joe/error_logs/THELOG, we'll find it starts like this:

```
=PROGRESS REPORT==== 15-Jul-2013::11:30:55 ===
supervisor: {local,sasl_safe_sup}
   started: [{pid,<0.34.0>},
             {name,alarm_handler},
             {mfa,{alarm_handler,start_link,[]}},
             {restart_type,permanent},
             {shutdown,2000},
             {child_type,worker}]
...
```

This lists only progress reports, which otherwise would have appeared in the shell. The progress reports are for major events, such as starting and stopping applications. But the errors reported by error_logger:error_msg/1 are not saved in the log. For this we have to configure a rotating log.

Rotating Log and Shell

The following configuration gives us shell output plus a copy of everything that was written to the shell in a rotating log file.

```
elog3.config
%% rotating log and minimal tty
[{sasl, [
        {sasl_error_logger, false},
        %% define the parameters of the rotating log
        %% the log file directory
        {error_logger_mf_dir,"/Users/joe/error_logs"},
        %% # bytes per logfile
        {error_logger_mf_maxbytes,10485760}, % 10 MB
        %% maximum number of logfiles
        {error_logger_mf_maxfiles, 10}
       ]}].

erl -boot start_sasl -config elog3
Erlang R16B (erts-5.10.1) [source] [smp:2:2] ...
Eshell V5.10.1  (abort with ^G)
1> error_logger:error_msg("This is an error\n").
2>
=ERROR REPORT==== 26-May-2013::13:14:31 ===
This is an error
```

The log has a maximum size of 10MB and wraps around or "rotates" when it gets to 10MB. As you can imagine, this is a very useful configuration.

When we run the system, all the errors go into a rotating error log. Later in this chapter we'll see how to extract the errors from the log.

Production Environment

In a production environment, we are really interested only in errors and not progress or information reports, so we tell the error logger to report only errors. Without this setting, the system might get swamped with information and progress reports.

elog4.config
```erlang
%% rotating log and errors
[{sasl, [
        %% minimise shell error logging
        {sasl_error_logger, false},
        %% only report errors
        {errlog_type, error},
        %% define the parameters of the rotating log
        %% the log file directory
        {error_logger_mf_dir,"/Users/joe/error_logs"},
        %% # bytes per logfile
        {error_logger_mf_maxbytes,10485760}, % 10 MB
        %% maximum number of
        {error_logger_mf_maxfiles, 10}
        ]}].
```

Running this results in a similar output to the previous example. The difference is that only errors are reported in the error log.

Analyzing the Errors

Reading the error logs is the responsibility of the rb module. It has an extremely simple interface.

```erlang
$ erl -boot start_sasl -config elog3
...
1> rb:help().
Report Browser Tool - usage
===========================
2> rb:start()        - start the rb_server with default options
rb:start(Options)  - where Options is a list of:
                    {start_log, FileName}
                      - default: standard_io
                    {max, MaxNoOfReports}
                      - MaxNoOfReports should be an integer or 'all'
                      - default: all
    ...
```

```
    ... many lines omitted ...
    ...
3> rb:start([{max,20}]).
rb: reading report...done.
```

First we must start Erlang with the correct configuration file so that the error logs can be located; then we start the report browser by telling it how many log entries to read (in this case the last twenty). Now we'll list the entries in the log.

```
4> rb:list().
No              Type   Process        Date       Time
==              ====   =======        ====       ====
12          progress  <0.31.0> 2013-05-26 13:21:53
11          progress  <0.31.0> 2013-05-26 13:21:53
10          progress  <0.31.0> 2013-05-26 13:21:53
9           progress  <0.24.0> 2013-05-26 13:21:53
8              error  <0.25.0> 2013-05-26 13:23:04
7           progress  <0.31.0> 2013-05-26 13:23:58
6           progress  <0.31.0> 2013-05-26 13:24:13
5           progress  <0.31.0> 2013-05-26 13:24:13
4           progress  <0.31.0> 2013-05-26 13:24:13
3           progress  <0.31.0> 2013-05-26 13:24:13
2           progress  <0.24.0> 2013-05-26 13:24:13
1           progress  <0.31.0> 2013-05-26 13:24:17
ok
```

The error log entry message produced by calling error_logger:error_msg/1 ended up in the log as entry number 8. We can examine this as follows:

```
> rb:show(8).
ERROR REPORT  <0.44.0>                   2013-05-26 13:23:04
============================================================
This is an error
ok
```

To isolate a particular error, we can use commands such as rb:grep(RegExp), which will find all reports matching the regular expression RegExp. I don't want to go into all the details of how to analyze the error log. The best thing is to spend some time interacting with rb and seeing what it can do. Note that you never need to actually delete an error report, since the rotation mechanism will eventually delete old error logs.

If you want to keep all the error logs, you'll have to poll the error log at regular intervals and remove the information in which you are interested.

23.3 Alarm Management

When we write our application, we need only one alarm—we'll raise it when the CPU starts melting because we're computing a humongous prime (remember, we were making a company that sells prime numbers). This time we'll use the real OTP alarm handler (and not the simple one we saw at the start of this chapter).

The alarm handler is a callback module for the OTP gen_event behavior. Here's the code:

```erlang
my_alarm_handler.erl
-module(my_alarm_handler).
-behaviour(gen_event).

%% gen_event callbacks
-export([init/1, code_change/3, handle_event/2, handle_call/2,
         handle_info/2, terminate/2]).

%% init(Args) must return {ok, State}
init(Args) ->
    io:format("*** my_alarm_handler init:~p~n",[Args]),
    {ok, 0}.

handle_event({set_alarm, tooHot}, N) ->
    error_logger:error_msg("*** Tell the Engineer to turn on the fan~n"),
    {ok, N+1};
handle_event({clear_alarm, tooHot}, N) ->
    error_logger:error_msg("*** Danger over. Turn off the fan~n"),
    {ok, N};
handle_event(Event, N) ->
    io:format("*** unmatched event:~p~n",[Event]),
    {ok, N}.

handle_call(_Request, N) -> Reply = N, {ok, Reply,  N}.
handle_info(_Info, N)    -> {ok, N}.

terminate(_Reason, _N)   -> ok.
code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

This code is pretty similar to the callback code for gen_server, which we saw earlier in *Calling the Server*, on page 373. The interesting routine is handle_event(Event, State). This should return {ok, NewState}. Event is a tuple of the form {EventType, Event-Arg}, where EventType is set_event or clear_event and EventArg is a user-supplied argument. We'll see later how these events are generated.

Now we can have some fun. We'll start the system, generate an alarm, install an alarm handler, generate a new alarm, and so on.

```
$ erl -boot start_sasl -config elog3
1> alarm_handler:set_alarm(tooHot).
ok
=INFO REPORT==== 15-Jul-2013::14:20:06 ===
alarm_handler: {set,tooHot}
2> gen_event:swap_handler(alarm_handler,
                          {alarm_handler, swap},
                          {my_alarm_handler, xyz}).
*** my_alarm_handler init:{xyz,{alarm_handler,[tooHot]}}
3> alarm_handler:set_alarm(tooHot).
ok
=ERROR REPORT==== 15-Jul-2013::14:22:19 ===
*** Tell the Engineer to turn on the fan
4> alarm_handler:clear_alarm(tooHot).
ok
=ERROR REPORT==== 15-Jul-2013::14:22:39 ===
*** Danger over. Turn off the fan
```

This is what happened:

1. We started Erlang with -boot start_sasl. When we do this, we get a standard alarm handler. When we set or clear an alarm, nothing happens. This is similar to the "do nothing" event handler we discussed earlier.

2. When we set an alarm (line 1), we just get an information report. There is no special handling of the alarm.

3. We install a custom alarm handler (line 2). The argument to my_alarm_handler (xyz) has no particular significance; the syntax requires some value here, but since we don't use the value and we just used the atom xyz, we can identify the argument when it is printed.

   The ** my_alarm_handler_init: ... printout came from our callback module.

4. We set and clear a tooHot alarm (lines 3 and 4). This is processed by our custom alarm handler. We can verify this by reading the shell printout.

Reading the Log

Let's go back to the error logger to see what happened.

```
1> rb:start([{max,20}]).
rb: reading report...done.
2> rb:list().
No              Type    Process Date        Time
==              ====    ======= ====        ====
...
3        info_report    <0.29.0> 2013-07-30 14:20:06
2              error    <0.29.0> 2013-07-30 14:22:19
1              error    <0.29.0> 2013-07-30 14:22:39
```

```
3> rb:show(1).

ERROR REPORT       <0.33.0>                2013-07-30 14:22:39
=====================================================
*** Danger over. Turn off the fan
ok
4> rb:show(2).
ERROR REPORT       <0.33.0>                2013-07-30 14:22:19
=====================================================
*** Tell the Engineer to turn on the fan
```

So, we can see that the error logging mechanism works.

In practice, we would make sure the error log was big enough for several days or weeks of operation. Every few days (or weeks) we'd check the error logs and investigate all errors.

Note: The rb module has functions to select specific types of errors and to extract these errors to a file. So, the process of analyzing the error logs can be fully automated.

23.4 The Application Servers

Our application has two servers: a prime number server and an area server.

The Prime Number Server

Here's the prime number server. It has been written using the gen_server behavior (see Section 22.2, *Getting Started with gen_server*, on page 368). Note how it includes the alarm handling procedures we developed in the previous section.

prime_server.erl
```erlang
-module(prime_server).

-behaviour(gen_server).

-export([new_prime/1, start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

new_prime(N) ->
    %% 20000 is a timeout (ms)
    gen_server:call(?MODULE, {prime, N}, 20000).
```

```erlang
init([]) ->
    %% Note we must set trap_exit = true if we
    %% want terminate/2 to be called when the application
    %% is stopped

    process_flag(trap_exit, true),
    io:format("~p starting~n",[?MODULE]),
    {ok, 0}.

handle_call({prime, K}, _From, N) ->
    {reply, make_new_prime(K), N+1}.

handle_cast(_Msg, N)  -> {noreply, N}.

handle_info(_Info, N)  -> {noreply, N}.

terminate(_Reason, _N) ->
    io:format("~p stopping~n",[?MODULE]),
    ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

make_new_prime(K) ->
    if
        K > 100 ->
            alarm_handler:set_alarm(tooHot),
            N = lib_primes:make_prime(K),
            alarm_handler:clear_alarm(tooHot),
            N;
        true ->
            lib_primes:make_prime(K)
    end.
```

The Area Server

And now the area server. This is also written with the gen_server behavior. Note that writing a server this way is extremely quick. When I wrote this example, I cut and pasted the code in the prime server and turned it into an area server. This took only a few minutes.

The area server is not the most brilliant program in the world, and it contains a deliberate error (can you find it?). My not-so-cunning plan is to let the server crash and be restarted by the supervisor. And what's more, we'll get a report of all this in the error log.

```erlang
area_server.erl
-module(area_server).
-behaviour(gen_server).

-export([area/1, start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

area(Thing) ->
    gen_server:call(?MODULE, {area, Thing}).

init([]) ->
    %% Note we must set trap_exit = true if we
    %% want terminate/2 to be called when the application
    %% is stopped
    process_flag(trap_exit, true),
    io:format("~p starting~n",[?MODULE]),
    {ok, 0}.

handle_call({area, Thing}, _From, N) -> {reply, compute_area(Thing), N+1}.

handle_cast(_Msg, N)  -> {noreply, N}.

handle_info(_Info, N)  -> {noreply, N}.

terminate(_Reason, _N) ->
    io:format("~p stopping~n",[?MODULE]),
    ok.

code_change(_OldVsn, N, _Extra) -> {ok, N}.

compute_area({square, X})      -> X*X;
compute_area({rectongle, X, Y}) -> X*Y.
```

Now we've written our application code, complete with a little error. Now we have to set up a supervision structure to detect and correct any errors that might occur at runtime.

23.5 The Supervision Tree

A supervision tree is a tree of processes. The upper processes (supervisors) in the tree monitor the lower processes (workers) in the tree and restart the lower processes if they fail. There are two types of supervision tree. You can see them in the following figure:

one_for_one supervision
If one process crashes, it is restarted
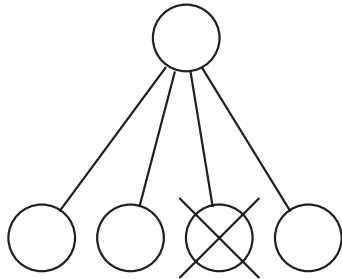
one_for_all supervision
If one process crashes, all are terminated
and then restarted
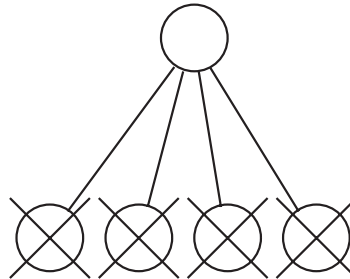
One-for-one supervision trees

In one-for-one supervision, if a worker fails, it is restarted by the supervisor.

One-for-all supervision trees

In one-for-all supervision, if any worker dies, then all the worker processes are killed (by calling the terminate/2 function in the appropriate callback module). Then all the worker processes are restarted.

Supervisors are created using the OTP *supervisor* behavior. This behavior is parameterized with a callback module that specifies the supervisor strategy and how to start the individual worker processes in the supervision tree. The supervisor tree is specified with a function of this form:

```
init(...) ->
    {ok, {
          {RestartStrategy, MaxRestarts, Time},
          [Worker1, Worker2, ...]
         }}.
```

Here RestartStrategy is one of the atoms one_for_one or one_for_all. MaxRestarts and Time specify a "restart frequency." If a supervisor performs more than MaxRestarts in Time seconds, then the supervisor will terminate all the worker processes and then itself. This is to try to stop the situation where a process crashes, is restarted, and then crashes for the same reason, on and on, in an endless loop.

Worker1, Worker2, and so on, are tuples describing how to start each of the worker processes. We'll see what these look like in a moment.

Now let's get back to our company and build a supervision tree.

The first thing we need to do is to choose a name for our company. Let's call it sellaprime. The job of the sellaprime supervisor is to make sure the prime and area servers are always running. To do this, we'll write yet another callback module, this time for gen_supervisor. Here's the callback module:

```
sellaprime_supervisor.erl
-module(sellaprime_supervisor).
-behaviour(supervisor).          % see erl -man supervisor
-export([start/0, start_in_shell_for_testing/0, start_link/1, init/1]).

start() ->
    spawn(fun() ->
                  supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = [])
          end).
start_in_shell_for_testing() ->
    {ok, Pid} = supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = []),
    unlink(Pid).
start_link(Args) ->
    supervisor:start_link({local,?MODULE}, ?MODULE, Args).
init([]) ->
    %% Install my personal error handler
     gen_event:swap_handler(alarm_handler,
                                     {alarm_handler, swap},
                                     {my_alarm_handler, xyz}),
    {ok, {{one_for_one, 3, 10},
          [{tag1,
            {area_server, start_link, []},
            permanent,
            10000,
            worker,
            [area_server]},
           {tag2,
            {prime_server, start_link, []},
            permanent,
            10000,
            worker,
            [prime_server]}
          ]}}.
```

The important part of this is the data structure returned by init/1.

```
sellaprime_supervisor.erl
{ok, {{one_for_one, 3, 10},
      [{tag1,
        {area_server, start_link, []},
        permanent,
        10000,
        worker,
        [area_server]},
       {tag2,
        {prime_server, start_link, []},
        permanent,
        10000,
        worker,
        [prime_server]}
      ]}}.
```

This data structure defines a supervision strategy. We talked about the supervision strategy and restart frequency earlier. Now all that remains are the start specifications for the area server and the prime number server.

The Worker specifications are tuples of the following form:

```
{Tag, {Mod, Func, ArgList},
      Restart,
      Shutdown,
      Type,
      [Mod1]}
```

These arguments mean the following:

Tag

> This is an atom tag that we can use to refer to the worker process later (if necessary).

{Mod, Func, ArgList}

> This defines the function that the supervisor will use to start the worker. It is used as arguments to apply(Mod, Fun, ArgList).

Restart = permanent | transient | temporary

> A permanent process will always be restarted. A transient process is restarted only if it terminates with a non-normal exit value. A temporary process is never restarted.

Shutdown

> This is a shutdown time. This is the maximum time a worker is allowed to take in terminating. If it takes longer than this, it will be killed. (Other values are possible—see the supervisor manual pages.)

Type = worker | supervisor

> This is the type of the supervised process. We can construct a tree of supervisors by adding supervisor processes in place of worker processes.

[Mod1]

> This is the name of the callback module if the child process is a supervisor or gen_server behavior callback module. (Other values are possible—see the supervisor manual page.)

These arguments look scarier than they actually are. In practice, you can cut and paste the values from the earlier area server code and insert the name of your module. This will suffice for most purposes.

23.6 Starting the System

Now we're ready for prime time. Let's launch our company. Off we go. We'll see who wants to buy the first prime number!

Let's start the system.

```
$ erl -boot start_sasl -config elog3
1> sellaprime_supervisor:start_in_shell_for_testing().
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
```

Now make a valid query.

```
2> area_server:area({square,10}).
100
```

Does the Supervision Strategy Work?

Erlang was designed for programming fault-tolerant systems. It was originally developed in the Computer Science Laboratory at the Swedish Telecom company Ericsson. Since then, the OTP group at Ericsson took over development aided by dozens of internal users. Using gen_server, gen_supervisor, and so on, Erlang has been used to build systems with 99.9999999 percent reliability (that's nine 9s). Used correctly, the error handling mechanisms can help make your program run forever (well, almost). The error logger described here has been run for years in live products.

Now make an invalid query.

```
3> area_server:area({rectangle,10,20}).
area_server stopping

=ERROR REPORT==== 15-Jul-2013::15:15:54 ===
** Generic server area_server terminating
** Last message in was {area,{rectangle,10,20}}
** When Server state == 1
** Reason for termination ==
** {function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
                     {area_server,handle_call,3},
                     {gen_server,handle_msg,6},
                     {proc_lib,init_p,5}]}
  area_server starting
  ** exited: {{function_clause,
                [{area_server,compute_area,[{rectangle,10,20}]},
                 {area_server,handle_call,3},
                 {gen_server,handle_msg,6},
                 {proc_lib,init_p,5}]},
                {gen_server,call,
                  [area_server,{area,{rectangle,10,20}}]}} **
```

Whoops—the area server crashed; we hit the deliberate error. The crash was detected by the supervisor, and the area server was restarted by the supervisor. All of this was logged by the error logger, and we get a printout of the error. A quick look at the error message shows what went wrong. The program crashed after trying to evaluate area_server:compute_area({rectangle,10,20}). This is shown in the first line of the function_clase error message. The error message is in the {Mod,Func,[Args]} format. If you look back a few pages to where the area computation was defined (in compute_area/1), you should be able to find the error.

After the crash, everything is back to normal, as it should be. Let's make a valid request this time.

```
4> area_server:area({square,25}).
625
```

We're up and running again. Now let's generate a little prime.

```
5> prime_server:new_prime(20).
Generating a 20 digit prime . ......
37864328602551726491
```

And let's generate a big prime.

```
6> prime_server:new_prime(120).
Generating a 120 digit prime
=ERROR REPORT==== 15-Jul-2013::15:22:17 ===
*** Tell the Engineer to turn on the fan
 ....................................

=ERROR REPORT==== 15-Jul-2013::15:22:20 ===
*** Danger over. Turn off the fan
765525474077993399589034417231006593110007130279318737419683
288059079481951097205184294443332300308877493399942800723107
```

Now we have a working system. If a server crashes, it is automatically restarted, and in the error log there will be information about the error. Let's now look at the error log.

```
1> rb:start([{max,20}]).
rb: reading report...done.
rb: reading report...done.
{ok,<0.53.0>}
2> rb:list().
No                      Process Date       Time
==              Type    ======= ====       ====
20          progress    <0.29.0> 2013-07-30 15:05:15
19          progress    <0.22.0> 2013-07-30 15:05:15
18          progress    <0.23.0> 2013-07-30 15:05:21
17   supervisor_report  <0.23.0> 2013-07-30 15:05:21
```

```
16              error     <0.23.0> 2013-07-30 15:07:07
15              error     <0.23.0> 2013-07-30 15:07:23
14              error     <0.23.0> 2013-07-30 15:07:41
13           progress     <0.29.0> 2013-07-30 15:15:07
12           progress     <0.29.0> 2013-07-30 15:15:07
11           progress     <0.29.0> 2013-07-30 15:15:07
10           progress     <0.29.0> 2013-07-30 15:15:07
9            progress     <0.22.0> 2013-07-30 15:15:07
8            progress     <0.23.0> 2013-07-30 15:15:13
7            progress     <0.23.0> 2013-07-30 15:15:13
6               error     <0.23.0> 2013-07-30 15:15:54
5        crash_report   area_server 2013-07-30 15:15:54
4   supervisor_report     <0.23.0> 2013-07-30 15:15:54
3            progress     <0.23.0> 2013-07-30 15:15:54
2               error     <0.29.0> 2013-07-30 15:22:17
1               error     <0.29.0> 2013-07-30 15:22:20
```

Something is wrong here. We have a crash report for the area server. To find
out what happened, we can look at the error report.

```
9> rb:show(5).
```

```
CRASH REPORT  <0.43.0>                        2013-07-30 15:15:54
=================================================================
Crashing process
pid                                                     <0.43.0>
registered_name                                       area_server
error_info
{function_clause,[{area_server,compute_area,[{rectangle,10,20}]},
                {area_server,handle_call,3},
                    {gen_server,handle_msg,6},
                    {proc_lib,init_p,5}]]}
initial_call
    {gen,init_it,
        [gen_server,
         <0.42.0>,
         <0.42.0>,
         {local,area_server},
         area_server,
         [],
         []]}
ancestors                         [sellaprime_supervisor,<0.40.0>]
messages                                                       []
links                                                  [<0.42.0>]
dictionary                                                     []
trap_exit                                                   false
status                                                    running
heap_size                                                     233
stack_size                                                     21
reductions                                                    199
ok
```

The printout {function_clause, compute_area, …} shows us exactly the point in the program where the server crashed. It should be an easy job to locate and correct this error. Let's move on to the next errors.

```
10> rb:show(2).

ERROR REPORT   <0.33.0>                     2013-07-30 15:22:17
================================================================
*** Tell the Engineer to turn on the fan
```

And.

```
10> rb:show(1).

ERROR REPORT   <0.33.0>                     2013-07-30 15:22:20
================================================================
*** Danger over. Turn off the fan
```

These were our fan alarms caused by computing too large of primes!

23.7 The Application

We're almost done. All we have to do now is write a file with the extension .app that contains information about our application.

```
sellaprime.app
%% This is the application resource file (.app file) for the 'base'
%% application.
{application, sellaprime,
 [{description, "The Prime Number Shop"},
  {vsn, "1.0"},
  {modules, [sellaprime_app, sellaprime_supervisor, area_server,
             prime_server, lib_lin, lib_primes, my_alarm_handler]},
  {registered,[area_server, prime_server, sellaprime_super]},
  {applications, [kernel,stdlib]},
  {mod, {sellaprime_app,[]}},
  {start_phases, []}
 ]}.
```

Then we have to write a callback module with the same name as the mod file in the previous file. This file is derived by filling in the template found in Section A1.3, *The Application Template,* on page 475.

```
sellaprime_app.erl
-module(sellaprime_app).
-behaviour(application).
-export([start/2, stop/1]).
start(_Type, StartArgs) ->
    sellaprime_supervisor:start_link(StartArgs).
stop(_State) ->
    ok.
```

This must export the functions start/2 and stop/1. Once we've done all of this, we can start and stop our application in the shell.

```
$ erl -boot start_sasl -config elog3
1> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.16.1"},
 {sasl,"SASL  CXC 138 11","2.3.1"},
 {stdlib,"ERTS  CXC 138 10","1.19.1"}]
2> application:load(sellaprime).
ok
3> application:loaded_applications().
[{sellaprime,"The Prime Number Shop","1.0"},
 {kernel,"ERTS  CXC 138 10","2.16.1"},
 {sasl,"SASL  CXC 138 11","2.3.1"},
 {stdlib,"ERTS  CXC 138 10","1.19.1"}]
4> application:start(sellaprime).
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
ok
5> application:stop(sellaprime).
prime_server stopping
area_server stopping

=INFO REPORT==== 26-May-2013::14:16:57 ===
application: sellaprime
exited: stopped
type: temporary
ok
6> application:unload(sellaprime).
ok
7> application:loaded_applications().
[{kernel,"ERTS  CXC 138 10","2.16.1"},
{sasl,"SASL  CXC 138 11","2.3.1"},
{stdlib,"ERTS  CXC 138 10","1.19.1"}]
```

This is now a fully fledged OTP application. In line 2 we loaded the application; this loads all the code but does not start the application. Line 4 started the application, and line 5 stopped the application. Note that we can see from the printout when the applications were started and stopped. The appropriate callback functions in the area server and prime number server were called. In Line 6 we unloaded the application. All the module code for the application is removed.

When we build complex systems using OTP, we package them as applications. This allows us to start, stop, and administer them uniformly.

Note that when we use init:stop() to close down the system, then all running applications will be closed down in an orderly manner.

```
$ erl -boot start_sasl -config elog3
1>  application:start(sellaprime).
*** my_alarm_handler init:{xyz,{alarm_handler,[]}}
area_server starting
prime_server starting
ok
2> init:stop().
ok
prime_server stopping
area_server stopping
$
```

The two lines following command 2 come from the area and prime number servers, which shows that the terminate/2 methods in the gen_server callback modules were called.

23.8 File System Organization

I haven't mentioned anything about the file system organization yet. This is deliberate—my intention is to confuse you with only one thing at a time.

Well-behaved OTP applications usually have the files belonging to different parts of the application in well-defined places. This is not a requirement; as long as all the relevant files can be found at runtime, it doesn't matter how the files are organized.

In this book I have put most of the demonstration files in the same directory. This simplifies the examples and avoids problems with search paths and interactions between the different programs.

The main files used in the sellaprime company are as follows:

File	*Content*
area_server.erl	Area server—a gen_server callback
prime_server.erl	Prime number server—a gen_server callback
sellaprime_supervisor.erl	Supervisor callback
sellaprime_app.erl	Application callback
my_alam_handler.erl	Event callback for gen_event
sellaprime.app	Application specification
elog4.config	Error logger configuration file

To see how these files and modules are used, we can look at the sequence of events that happens when we start the application.

1. We start the system with the following commands:

```
$ erl -boot start_sasl -config elog4.config
1> application:start(sellaprime).
...
```

The file sellaprime.app must be in the root directory where Erlang was started or in a subdirectory of this directory.

The application controller then looks for a {mod, ...} declaration in the sellaprime.app. This contains the name of the application controller. In our case, this was the module sellaprime_app.

2. The callback routine sellaprime_app:start/2 is called.

3. sellaprime_app:start/2 calls sellaprime_supervisor:start_link/2, which starts the sellaprime supervisor.

4. The supervisor callback sellaprime_supervisor:init/1 is called. This installs an error handler and returns a supervision specification. The supervision specification says how to start the area server and prime number server.

5. The sellaprime supervisor starts the area server and prime number server. These are both implemented as gen_server callback modules.

Stopping everything is easy. We just call application:stop(sellaprime) or init:stop().

23.9 The Application Monitor

The application monitor is a GUI for viewing applications. The command appmon:start() starts the application viewer. When you give this command, you'll see a window similar to Figure 3, *Application monitor initial window*. To see the applications, you have to click one of the applications. The application monitor view of the sellaprime application is shown in Figure 4, *The sellaprime application,* on page 407.
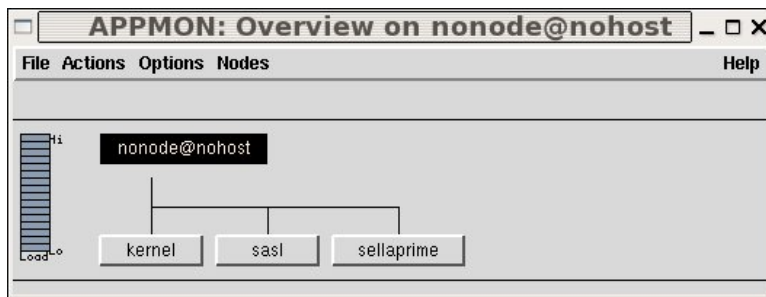
Figure 3—Application monitor initial window
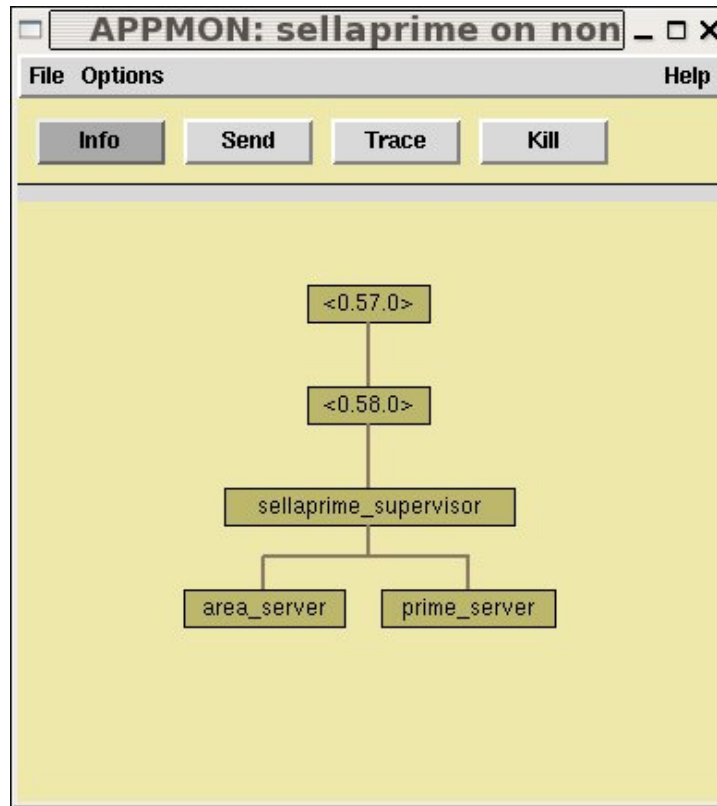
Figure 4—The sellaprime application

23.10 How Did We Make That Prime?

Easy. Here's the code:

lib_primes.erl
```erlang
Line 1  -module(lib_primes).
     -  -export([make_prime/1, is_prime/1, make_random_int/1]).
     -
     -  make_prime(1) ->
     5      lists:nth(random:uniform(4), [2,3,5,7]);
     -  make_prime(K) when K > 0 ->
     -      new_seed(),
     -      N = make_random_int(K),
     -      if N > 3 ->
     10             io:format("Generating a ~w digit prime ",[K]),
     -             MaxTries = N - 3,
     -             P1 = make_prime(MaxTries, N+1),
     -             io:format("~n",[]),
     -             P1;
```

```
15          true ->
-               make_prime(K)
-       end.
-
- make_prime(0, _) ->
20      exit(impossible);
- make_prime(K, P) ->
-       io:format(".",[]),
-       case is_prime(P) of
-           true  -> P;
25          false -> make_prime(K-1, P+1)
-       end.
-
- is_prime(D) when D < 10 ->
-       lists:member(D, [2,3,5,7]);
30 is_prime(D) ->
-       new_seed(),
-       is_prime(D, 100).
-
- is_prime(D, Ntests) ->
35      N = length(integer_to_list(D)) -1,
-       is_prime(Ntests, D, N).
-
- is_prime(0, _, _) -> true;
- is_prime(Ntest, N, Len) ->
40      K = random:uniform(Len),
-       %% A is a random number less than K
-       A = make_random_int(K),
-       if
-           A < N ->
45              case lib_lin:pow(A,N,N) of
-                   A -> is_prime(Ntest-1,N,Len);
-                   _ -> false
-               end;
-           true ->
50              is_prime(Ntest, N, Len)
-       end.
-
- %% make_random_int(N) -> a random integer with N digits.
- make_random_int(N) -> new_seed(), make_random_int(N, 0).
55
- make_random_int(0, D) -> D;
- make_random_int(N, D) ->
-       make_random_int(N-1, D*10 + (random:uniform(10)-1)).
```

make_prime(K) returns a prime of at least K digits. To do this, we use an algorithm based on Bertrand's postulate. Bertrand's postulate is that for every N > 3, there is a prime P satisfying N < P < 2N - 2. This was proved by Tchebychef in 1850, and Erdos improved the proof in 1932.

We first generate a random integer of N digits (lines 54 to 58); then we test N+1, N+2, and so on, for primality. This is done in the loop in lines 19 to 26.

is_prime(D) returns true if D is very likely to be prime and otherwise returns false. It makes use of Fermat's little theorem, which says that if N is a prime and if A < N, then A^N mod N = A. So, to test whether N is prime, we generate random values of A less than N and apply the Fermat test. If this fails, then N is not a prime. This is a probabilistic test, so the probability that N is prime increases every time we apply the test. This test is performed in lines 38 to 51.

It is time to make some primes.

```
1> lib_primes:make_prime(500).
Generating a 500 digit prime . ...........................
791015726987201027909055597115096126908592921342508297266 2439
125926314028552834613243970133079247710947860309449739469 6440
439969675871437494053122242294696670762292613938500209657 8309
062534166780603261012226023459181325555764028306928844115 1813
911078020075570667464760355151051540174212673823673149419 5650
557847449754525266671828097689040150301840652144065085734 9061
213980678938094352667372672691906693169783133618111423622 8904
018680428721980745461937400537776682710560368928381817300 7034
056505784153
```

We've now covered the basics of building an OTP application. OTP applications have a standardized file layout and are started and stopped in a regular manner. They usually consist of a gen_server supervised by a gen_supervisor together with some error logging code. On the Erlang distribution website, you can find full descriptions of all the OTP behaviors.

23.11 Digging Deeper

I've skipped over quite a lot of detail here, explaining just the principles involved. You can find the details in the manual pages for gen_event, error_logger, supervisor, and application.

OTP design principles are documented in more detail in the Erlang/OTP system documentation.[1]

At this stage, we have covered all the major topics needed to build a regular Erlang application that can fit into the OTP framework.

In the final part of the book, we will look outside the OTP framework, show some additional programming techniques, and build some complete example programs.

1.　http://www.erlang.org/doc/pdf/otp-system-documentation.pdf

Exercises

As you read through these exercises, don't get worried. By the time you get to the last problem in the list, you will be dealing with a pretty difficult problem. To solve this, you'll need to know about the OTP behaviors (this and the last chapter), understand how to use Mnesia (for data replication), and have a basic understanding of distributed Erlang (how to set up linked nodes).

No topic is, by itself, particularly complicated. But when we start combining simple things, the result can exhibit very complex behavior. Even if you don't solve these problems immediately, thinking about them will help you structure the solution to problems like this into manageable chunks. To build large fault-tolerant systems, we need to think about what the servers do, how to restart failed servers, how to load balance, how and where to replicate data, and so on. The exercises are arranged so as to guide you through this process.

1. Make a gen_server called prime_tester_server that tests if a given number is prime. You can use the is_prime/2 function in lib_primes.erl to do this (or implement a better prime number tester yourself). Add this to the supervisor tree in sellaprime_supervisor.erl.

2. Make a pool of ten prime tester servers. Make a queue server that queues requests until one of the prime tester servers become free. When a prime tester server becomes free, send it a request to test a number for primality.

3. Change the code in the prime server testers so that each prime server tester maintains its own queue of requests. Remove the queue server. Write a load balancer that keeps track of the work being done and requests to be done by the prime server testers. Requests to test new primes should now be sent to the load balancer. Arrange that the load balancer sends requests to the least loaded server.

4. Implement a supervisor hierarchy so that if any prime number tester server crashes, it should be restarted. If the load balancer crashes, crash all the prime number tester servers and restart everything.

5. Keep the data necessary to restart everything replicated on two machines.

6. Implement a restart strategy to restart everything if an entire machine crashes.

Part V

Building Applications

In this part, we'll look at some programming idioms that are commonly used when writing Erlang programs. We'll also see how to integrate third-party code into our applications. This way, we can build upon the work of others to obtain results far more quickly than if we had done all the work ourselves. We'll learn how to parallelize programs on a multi-core, and finally we'll solve Sherlock's last case.

Programming Idioms

In this chapter, we'll investigate some programming idioms and look at different techniques for structuring Erlang code. We'll start with an example that shows how we should perceive the programming world and the objects that we find in this world.

24.1 Maintaining the Erlang View of the World

The Erlang view of the world is that *everything is a process* and that processes can interact only by exchanging messages. Having such a view of the world imposes *conceptual integrity* on our designs, making them easier to understand.

Imagine we want to write a web server in Erlang. A user requests a page called hello.html from our web server. The simplest possible web server looks like this:

```erlang
web_server(Client) ->
   receive
       {Client, {get, Page}} ->
           case file:read(Page) of
               {ok, Bin} ->
                   Client ! {self(), {data, Bin}};
               {error, _} ->
                   Client ! {self(), error}
           end,
           web_server(Client)
   end.
```

But this code is simple only because all it does is receive and send Erlang terms. *But clients don't send us Erlang terms; they send us HTTP requests, which are far more complicated.* The HTTP requests come over TCP connections, and the requests might themselves be fragmented, all of which makes the server program far more complicated than the simple code shown previously.

To simplify things, we interpose a process called a *middle man* between the TCP driver that receives messages from the HTTP client and our Erlang server. The middle man parses the HTTP requests and turns them into Erlang messages. This is shown in the following figure. You can see why the translation process is called a middle man; it sits between the TCP driver and the web server.
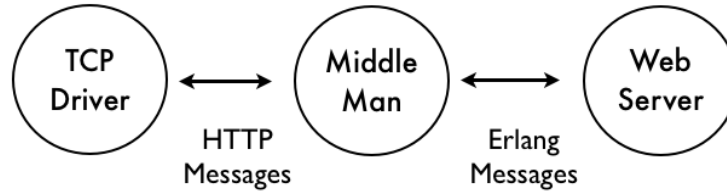
As far as the server is concerned, the objects in the external world only "speak" Erlang. Instead of having one process that does two things (handling HTTP requests and serving the requests), we now have two processes, each with a clearly defined role. The middle man knows only how to convert between HTTP and Erlang messages. The server knows nothing about the details of the HTTP protocol but just deals with pure Erlang messages. Breaking this into two processes not only makes the design clearer but has an additional benefit; it can increase concurrency. Both processes can execute in parallel.

The flow of messages concerned in satisfying an HTTP request is shown in Figure 5, *Web Server Protocol*, on page 415.

Exactly how the middle man works is not relevant to this discussion. All it has to do is parse incoming HTTP requests, convert them into Erlang terms, and convert outgoing Erlang terms into HTTP responses.

In our example we chose to abstract out a lot of the detail in the HTML request. HTML request headers contain a lot of additional information that we did not show here. As part of the design of the middle man, we have to decide exactly how much detail of the underlying protocol we want to expose to the Erlang application.

Suppose we want to extend this and respond to FTP requests for files or for files to be sent over an IRC channel. We can structure the processes in our system as shown in Figure 6, *Unified Messages*, on page 415.

HTTP, FTP, and IRC all use *entirely different protocols* for transferring files between machines. Actually, IRC doesn't support file transfer, but usually file transfer is supported by the Direct Client to Client (DCC) protocol, which most IRC clients support.

TCP Socket   **Middle Man**   **Web server**

```
GET /~joe/hello.html HTTP/1.1
User-Agent: ...
Host: www.sics.se
Accept: */*
```

{get,"hello.html"}

{data,<<"hello">>}

```
HTTP/1.1 200 OK
Date: Fri, 26 Feb 2013 13:20:06 GMT
Server: Erlang home brew
Accept-Ranges: bytes
Content-Length: 4898
Connection: close
Content-Type: text/html

hello
```

Figure 5—Web Server Protocol

HTTP
get request

HTTP
Middle
Man

Erlang
Messages

{get, File}

FTP
get request

FTP
Middle
Man

{get, File}

File
Server

IRC
get request

IRC
Middle
Man

{get, File}

Figure 6—Unified Messages

After the middle men have converted the external protocols to Erlang messages, a single Erlang server can be used as the back end to all these different protocols.

Using unified Erlang messaging really simplifies implementations. It has the following advantages:

- It abstracts out the difference between the different wire protocols (for example, the HTTP and FTP protocols).

- Erlang messages require no parser. The receiving process does not have to parse a message before it can process it. Contrast this with an HTTP server that has to parse all the messages it receives.

- Erlang messages can contain terms of arbitrary complexity. Contrast this with HTTP messages that have to be serialized into a flat form before being transmitted.

- Erlang messages can be sent over processor boundaries or be stored in a database in a simple universal serialization format.

24.2 A Multipurpose Server

Once we remove the idea that individual services need to have their different message formats, we can use uniform messaging to solve a range of problems. Here, for example, is a "multiserver":

multi_server.erl
```erlang
Line 1  -module(multi_server).
     -  -export([start/0]).
     -
     -  start() -> spawn(fun() -> multi_server() end).
     5
     -  multi_server() ->
     -      receive
     -          {_Pid, {email, _From, _Subject, _Text} = Email} ->
     -              {ok, S} = file:open("mbox", [write,append]),
    10              io:format(S, "~p.~n", [Email]),
     -              file:close(S);
     -          {_Pid, {im, From, Text}} ->
     -              io:format("Msg (~s): ~s~n",[From, Text]);
     -          {Pid, {get, File}} ->
    15              Pid ! {self(), file:read_file(File)};
     -          Any ->
     -              io:format("multi server got:~p~n",[Any])
     -      end,
     -      multi_server().
```

This code mimics the *essential behavior* of a number of well-known services.

In lines 8 to 11, it behaves like an email client.
> The *essential* job of an email client is to receive an email and store it on your computer, conventionally in a file called mbox. We receive a message, open a file called mbox, write the message to the file, and we're done.

In lines 12 to 13, it behaves like an instant messaging client.
> The *essential* job of an instant messaging client is to receive a message and tell the user. We tell the user by writing a message to the console.

In lines 12 to 13, it behaves like an FTP/RCP/HTTP server.
> The *essential* job of FTP server, an HTTP server, or any other file server is to transfer a file from a server to a client.

Looking at this code, we realize that we don't really need many different encodings of client-server requests and responses. One universal format suffices; Erlang terms are used in all messages.

All of this goodness works in a distributed environment because of the properties of two BIFs, term_to_binary(Term) and the inverse binary_to_term(Bin), which recovers the term.

In a distributed system, binary_to_term(Bin) can reconstruct any term from *the external representation of the term* stored in Bin. Usually Bin comes to the machine through a socket, though the details are not important here. binary_to_term simply reconstructs the term. In protocols such as HTTP, the input request must be *parsed*, which makes the entire process inherently inefficient.

We can achieve layers of encryption and compression by adding a symmetric pair of functions to do what we are interested in. Here's an example:

```
send1(Term) -> encrypt(compress(term_to_binary(Term))).

receive1(Bin) -> binary_to_term(decompress(decrypt(Bin))).
```

And if we want to send encrypted, compressed, mobile code over the network, we could do this:

```
send_code(Mod, Func, Args) ->
    encrypt(compress(term_to_binary({Mod,Func,Args}))).

receive_code(Bin) ->
   {Mod, Func, Args} = binary_to_term(decompress(decrypt(Bin))),
   apply(Mod, Func, Args).
```

Here we combine three ideas: using term_to_binary and its inverse to send terms across a network, using apply to evaluate code, and using symmetric pairs of functions to compress/decompress and encrypt/decrypt the data. Note that

compress/decompress and encrypt/decrypt are not Erlang BIFs but are just functions that are assumed to exist.

As far as the Erlang programmer is concerned, the world is a very nice place. Once the appropriate middle-men processes have been written, all external processes speak Erlang. This really simplifies complex systems, especially if a large number of different external protocols are used.

It's like a world where everybody speaks English (or Mandarin)—it's much easier to communicate.

24.3 Stateful Modules

Using a mechanism called *tuple modules*, we can arrange to encapsulate state together with a module name. We can use this mechanism for information hiding and creating adapter modules that hide the details of an interface from programs that use the interface. This is useful if we want to make a common interface to several dissimilar modules or mimic some of the features of object-oriented programming.

When we call X:Func(....), X does not have to be an atom. It can be a tuple. If we write X = {Mod, P1, P2, ..., Pn} and then call X:Func(A1, A2, ..., An), then what actually gets called is Mod:Func(A1, A2, ..., An, X). For example, the call {foo,1,2,3}:bar(a,b) gets converted to the call foo:bar(a,b,{foo,1,2,3}).

Using this mechanism, we can create "stateful" modules. We'll illustrate this first with a simple stateful counter and then move on to an example that creates an adapter module for two existing modules.

A Counter with State

To illustrate the idea of a tuple module, we'll start with a simple example of a counter that has a single state parameter, N, which represents the value of a counter. The code is as follows:

```
counter.erl
-module(counter).
-export([bump/2, read/1]).

bump(N, {counter,K}) ->  {counter, N + K}.
read({counter, N}) -> N.
```

We can test this code as follows. First we compile the module.

```
1> c(counter).
{ok,counter}
```

Then create an instance of a tuple module.

```
2> C = {counter,2}.
{counter, 2}
```

And we call get/0.

```
3> C:read().
2
```

Because C is a tuple, this gets converted to the call counter:read({counter,2}), which returns 2.

```
3> C1 = C:bump(3).
{counter, 5}
```

C:bump(3) gets converted to the call counter:bump(3, {counter, 2}) and so returns {counter, 5}.

```
4> C1:read().
5
```

Note how both the module name counter and the state variable are hidden from the calling code in the tuple C and C1.

24.4 Adapter Patterns

Suppose we have two or more libraries that do more or less the same thing but cannot decide which to use. These libraries might have a similar functional interface but different performance characteristics. As an example, consider key-value stores: one store might keep keys and values in memory, and another might keep keys in memory and values on disk. Yet another might keep keys with small values in memory but store large values on disk. Even for something as simple as a key-value store there are many variations on how the store is actually implemented.

Suppose we want to write some code that makes use of a key-value store. At the time when we write our application, we have to make a design decision and choose a particular key-value store from those available. At a much later date, some of our design decisions might prove to be mistaken, and we might want to change the back-end store. Unfortunately, if the APIs used to access the old and new stores are different, then we might have to make a large number of changes to our program.

This is where the *adapter* pattern comes in. Adapters are tuple modules that provide a uniform interface toward an application.

We'll illustrate this by building an adapter pattern that provides identical interfaces to a key-value store, which is implemented using the modules lists and dict. The interface to our adapter is as follows:

adapter_db1:new(Type :: dict | lists) -> Mod

> Creates a new key-value store of type Type. This returns a tuple module Mod.

Mod:store(Key, Val) -> Mod1

> Stores a Key, Value pair in the store. Mod is the old store; Mod1 is the new store.

Mod:lookup(Key) -> {ok, Val} | error

> Looks up Key in the store. This returns {ok, Val} if there is a value in the store; otherwise, it returns error.

To use this API, we write code such as the following:

```
M0 = adapter_db1:new(dict), ...
M1 = M0:store(Key1, Val2),
M2 = M1:store(Key2, Val2),
...
ValK = MK:lookup(KeyK),
```

If we want to use the lists implementation, we just change the line of code that created the module to Mod = adapter_db1:new(lists).

Just out of interest, we can compare this to the coding style we would have used if we had used the dict module. Using dict, we would have written something like this:

```
D0 = dict:new(),
D1 = dict:store(Key1, Val1, D0),
D2 = dict:store(Key2, Val2, D1),
...
ValK = dict:find(KeyK, Dk)
```

The code used to access the tuple module is somewhat shorter since we can hide all internals inside a single variable, Modi. Using dict requires two arguments: the name of the module and the dict structure itself.

Now let's write the adapter.

adapter_db1.erl
```
-module(adapter_db1).
-export([new/1, store/3, lookup/2]).

new(dict) ->
    {?MODULE, dict, dict:new()};
new(lists) ->
    {?MODULE, list, []}.

store(Key, Val, {_, dict, D}) ->
    D1 = dict:store(Key, Val, D),
```

```
    {?MODULE, dict, D1};
store(Key, Val, {_, list, L}) ->
    L1 = lists:keystore(Key, 1, L, {Key,Val}),
    {?MODULE, list, L1}.

lookup(Key, {_,dict,D}) ->
    dict:find(Key, D);
lookup(Key, {_,list,L}) ->
    case lists:keysearch(Key, 1, L) of
        {value, {Key,Val}} -> {ok, Val};
        false              -> error
    end.
```

This time our module is represented by a tuple of the form {adapter_db1, Type, Val}. If Type is list, then Val is a list, and if Type is dict, then Val is a dictionary.

We can write some simple code in a separate module to test the adapter.

```
adapter_db1_test.erl
-module(adapter_db1_test).
-export([test/0]).
-import(adapter_db1, [new/1, store/2, lookup/1]).
test() ->
    %% test the dict module
    M0 = new(dict),
    M1 = M0:store(key1, val1),
    M2 = M1:store(key2, val2),
    {ok, val1} = M2:lookup(key1),
    {ok, val2} = M2:lookup(key2),
    error = M2:lookup(nokey),
    %% test the lists module
    N0 = new(lists),
    N1 = N0:store(key1, val1),
    N2 = N1:store(key2, val2),
    {ok, val1} = N2:lookup(key1),
    {ok, val2} = N2:lookup(key2),
    error = N2:lookup(nokey),
    ok.
```

```
1> adapter_db1_test:test().
ok
```

The test succeeded. So, now we have achieved our goal of hiding two different modules with dissimilar interfaces behind an adapter module that provides a common interface to the two modules.

Adapters are useful for providing generic interfaces to preexisting code. The interface can remain constant, but the code behind the adapter can be changed to reflect different requirements.

24.5 Intentional Programming

Intentional programming is a name given to a style of programming where we can easily see what was intended by the programmer. The intention of the programmer should be obvious from the names of the functions involved and not be inferred by analyzing the structure of the code. This is best explained by an example. In the early days of Erlang, the library module dict exported a function lookup/2 that had the following interface:

```
lookup(Key, Dict) -> {ok, Value} | not_found
```

Given this definition, lookup could be used in three different contexts.

1. For *data retrieval*, we could write the following:

    ```
    {ok, Value} = lookup(Key, Dict)
    ```

    Here lookup is used to extract an item with a known key from the dictionary. If the key is not in the dictionary, not_found will be returned, a pattern match error will occur, and the program will raise an exception. The exit reason will be {badmatch, not_found}. This is a poor error message. A more informative error reason would be {bad_key, Key}.

2. For *searching*, we write the following:

    ```
    case lookup(Key, Dict) of
       {ok, Val} ->
          ... do something with Val ...
       not_found ->
          ... do something else ...
    end.
    ```

    We can see that the programmer did not know whether the key was in the dictionary since they wrote code that pattern matched the return value of lookup against both {ok, Val} and not_found. We can also see that something is done with Val (from the comment). We can infer from this code that the programmer wanted to search for a value in the dictionary. We search for something when we don't know where it is.

3. For *testing the presence of a key*, the following code fragment:

    ```
    case lookup(Key, Dict) of
        {ok, _} ->
           ... do something ...
        not_found ->
           ... do something else ...
    end.
    ```

tests to see whether a specific key is in the dictionary. We can infer this by noting that both possible return values from lookup are pattern matched, but the value of the item found is never used. We can see this since we pattern match {ok, _} and not {ok, Val} (as in the first example). Since we never use the value associated with the key, we can assume that lookup was called to test for the presence of a key.

The previous three examples overloaded the meaning of the function lookup. It was used for three different purposes: data retrieval, searching, and testing for the presence of a key.

Instead of guessing the programmer's intentions and analyzing the code, it is better to call a library routine that explicitly says which of the three alternatives is intended. dict exports three functions for this purpose.

```
dict:fetch(Key, Dict) = Val | EXIT
dict:search(Key, Dict) = {found, Val} | not_found.
dict:is_key(Key, Dict) = Boolean
```

These precisely express the intention of the programmer. No guesswork or program analysis is involved, and the names of the functions clearly tell us what the programmer intended. search is called when a key might be present in the dictionary, but it is not an error if the key is not present. fetch is called when a key must be in the dictionary, and it is an error if the key is not present. is_key is called to test whether the key is present in a dictionary.

Code written using lookup will be more difficult to understand and maintain than code using one of the (fetch, search, is_key) alternatives.

The most important idea that you should take away from this chapter is the idea of the middle man. The idea that everything in the external world should be modeled as an Erlang process is extremely important and is central to making components that smoothly fit together.

In the next chapter, we'll look at how to share our code and integrate our work with other people, and we'll look at a few third-party tools that are used in some of the examples in the book. We can solve problems quicker by using other people's code, and we can help other people by sharing our own code. If you help them, they will help you.

Exercises

1. Extend the adapter in adapter_db1 so that calling adapter_db1:new(persistent) creates a tuple module with a persistent data store.

2. Write a key-value store that stores small values in memory and large values on disk. Make an adapter module that makes this have the same interface as the previous adapter in this chapter.

3. Make a key-value store where some key-value pairs are persistent and others are transient. Calling put(Key, memory, Val) will store a Key, Val pair in memory. put(Key, disk, Val) should store the data on disk. Use a pair of processes to do this, one for the persistent store and one for the disk store. Reuse the earlier code in the chapter.

Third-Party Programs

This chapter talks about third-party programs, that is, Erlang programs that have been written and distributed by our users. The principal source for such programs is GitHub. In this chapter we'll look at three popular programs that are available from GitHub. We'll also see how to create and publicize a new project on GitHub and how to include a GitHub project in your own application. We'll take a look at the following:

Rebar. Rebar, written by Dave Smith, has become the de facto standard for managing Erlang projects. Using rebar, the user can create new projects, compile the projects, package them, and integrate them with other projects. Rebar is integrated with GitHub so users can easily fetch other rebar projects from GitHub and integrate them with their applications.

Bitcask. Bitcask, written by the folks at Basho,[1] is a persistent key-value disk store. It is fast and "crash friendly," which means it can recover quickly when restarted after a crash.

Cowboy. Cowboy, written by Loïc Hoguin, is a high-performance web server written in Erlang that is becoming popular for implementing embedded web servers. We used a cowboy server for the code developed in Chapter 18, *Browsing with Websockets and Erlang,* on page 287.

25.1 Making a Shareable Archive and Managing Your Code with Rebar

In this section we'll go through the steps necessary to make an open source Erlang project that we'll host on GitHub. I'll assume that you have an account on GitHub. We'll use rebar to manage the project.

---

1. http://basho.com

We'll do the following:

1. Install rebar.
2. Create a new project on GitHub.
3. Clone the project locally.
4. Add the project boilerplate code using rebar.
5. Use rebar to compile our project.
6. Upload our project to GitHub.

Installing Rebar

Rebar is available from https://github.com/basho/rebar. You should be able to find a prebuilt binary of rebar at https://github.com/rebar/rebar/wiki/rebar. To install rebar, make a copy of this file, change the file mode to executable, and put it somewhere in your path.

Having done this, you should test that you can run rebar.

```
$ rebar -V
rebar 2.0.0 R14B04 20120604_145614 git 0f24d93
```

Making a New Project on GitHub

Assume we want to make a new project called bertie (I name quite a few of my projects after the characters in Alexander McCall Smith's books). The first step is to create a new GitHub project. To do this, I log into my GitHub account and follow the instructions to create a new repository.

1. Click the "Create a new repo" icon on the top right of the login page toolbar (this is an icon that looks like a book with a plus on it).
2. Make it a public repository with a readme file.
3. Then click Create Repository.

Cloning the Project Locally

Back on my home machine, I have a single directory called ${HOME}/published where I keep all my shared projects. I move to my published directory and clone the GitHub repository.

```
$ cd ~/published
$ git clone git@github.com:joearms/bertie.git
Cloning into 'bertie'...
Identity added: /Users/joe/.ssh/id_rsa (/Users/joe/.ssh/id_rsa)
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

Now I usually check that I can write changes to the repository. So, I modify the readme file and push it back to the repository.

```
$ emacs -nw  README.md
$ git add README.md
$ git commit README.md
$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 326 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:joearms/bertie.git
6b9b6b9..6b0148e  master -> master
```

With this result I heave a sigh of relief and marvel at the mysteries of modern technology.

Now I have a local directory in ~/published/bertie that is synced to the GitHub repository at git@github.com:joearms/bertie.git.

Making an OTP Application

Now we change to the bertie directory and use rebar to create a standard OTP application.

```
$ cd ~/published/bertie
$ rebar create-app appid=bertie
==> bertie (create-app)
Writing src/bertie.app.src
Writing src/bertie_app.erl
Writing src/bertie_sup.erl
```

The command rebar create-app created the necessary boilerplate files and directory structure needed for a standard OTP application.

Now we'll add a single module bertie.erl to the ~/published/bertie/src directory.

```
-module(bertie).
-export([start/0]).

start() -> io:format("Hello my name is Bertie~n").
```

Then we use rebar to compile everything.

```
> rebar compile
==> bertie (compile)
Compiled src/bertie.erl
Compiled src/bertie_app.erl
Compiled src/bertie_sup.erl
```

We now have a complete program. What remains is to push it back to the repository.

```
$ git add src
$ git commit
$ git push
```

Publicizing Your Project

Now you have written your code and published on GitHub. The next step is to publicize it. The most obvious way to do this is with a short announcement on the Erlang mailing list[2] or to tweet about it with the hash tag #erlang.

All a user who wants to use your application has to do is download your application and run rebar compile to build your application.

25.2 Integrating External Programs with Our Code

We've been through the steps needed to publish your work on GitHub. Now we'll look at how to include other people's work in our projects. As an example, we'll integrate the code from bitcask into our bertie project.

We'll change bertie so that when it starts, it prints the number of times it has been started. So, for example, when it starts the tenth time, bertie will announce the following:

```
Bertie has been run 10 times
```

To achieve this, we'll store the number of times bertie has been run in a bitcask database. In bitcask, keys and values must be binaries. We'll choose the key to be the binary <<"n">> and the value to be term_to_binary(N), where N is the number of times bertie has been run. bertie.erl now reads as follows:

```
bertie/bertie.erl
-module(bertie).
-export([start/0]).

start() ->
    Handle = bitcask:open("bertie_database", [read_write]),
    N = fetch(Handle),
    store(Handle, N+1),
    io:format("Bertie has been run ~p times~n",[N]),
    bitcask:close(Handle),
    init:stop().

store(Handle, N) ->
    bitcask:put(Handle, <<"bertie_executions">>, term_to_binary(N)).
```

```
fetch(Handle) ->
    case bitcask:get(Handle, <<"bertie_executions">>) of
        not_found -> 1;
        {ok, Bin} -> binary_to_term(Bin)
    end.
```

To include bitcask in the bertie application, we create a "dependencies" file called rebar.config and store it in the top-level directory of the bertie project. rebar.config is as follows:

bertie/rebar.config
```
{deps, [
  {bitcask, ".*", {git, "git://github.com/basho/bitcask.git", "master"}}
]}.
```

I've also added a makefile.

bertie/Makefile
```
all:
        test -d deps || rebar get-deps
        rebar compile
        @erl -noshell -pa './deps/bitcask/ebin' -pa './ebin' -s bertie start
```

When we run the makefile the first time, we see the following:

```
$ ejoearm@ejoearm-eld:~/published/bertie$ make
==> bertie (get-deps)
Pulling bitcask from {git,"git://github.com/basho/bitcask.git","master"}
Cloning into 'bitcask'...
==> bitcask (get-deps)
Pulling meck from {git,"git://github.com/eproxus/meck"}
Cloning into 'meck'...
==> meck (get-deps)
rebar compile
==> meck (compile)
...
Bertie has been run 1 times
```

The command rebar get-deps fetched bitcask from GitHub and stored it in a sub-directory called deps. bitcask itself uses a program called meck for testing purposes. This is a so-called recursive dependency. Rebar recursively fetches any dependencies that bitcask might require and stores them in the deps subdirectory.

The makefile adds a -pa 'deps/bitcask/ebin' flag to the command line so that when the program is started, bertie can autoload the bitcask code.

Note: you can download the entire example from git://github.com/joearms/bertie.git. Assuming rebar has been installed, all you have to do is download the project and type make.

25.3 Making a Local Copy of the Dependencies

My bertie application made a local copy of bitcask in a local subdirectory of the bertie application. Sometimes several different applications want to make use of the same dependencies, in which case we create a dependency directory structure *outside* our application.

For my local projects I keep all my downloaded rebar dependencies in one place. I store all these dependencies under a top-level directory called ~joe/nobackup/erl_imports. My machine is organized so that any files under the nobackup directory are not subject to backup. Since the files I'm interested in are widely available on the Web, it doesn't seem necessary to create local backups.

The file ~joe/nobackup/erlang_imports/rebar.config lists all the dependencies I want to use and is as follows:

```
{deps, [
  {cowboy,  ".*", {git, "git://github.com/extend/cowboy.git", "master"}},
  {ranch,   ".*", {git, "git://github.com/extend/ranch.git", "master"}},
  {bitcask, ".*", {git, "git://github.com/basho/bitcask.git", "master"}}
  ]}.
```

To fetch the dependencies, we give the command rebar get-deps in the directory where we have stored the configuration file.

```
$ rebar get-deps
==> deps (get-deps)
Pulling cowboy from {git,"git://github.com/extend/cowboy.git","master"}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/cowboy/.git/
Pulling bitcask from {git,"git://github.com/basho/bitcask.git","master"}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/bitcask/.git/
==> cowboy (get-deps)
Pulling proper from {git,"git://github.com/manopapad/proper.git",{tag,"v1.0"}}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/proper/.git/
==> proper (get-deps)
==> bitcask (get-deps)
Pulling meck from {git,"git://github.com/eproxus/meck"}
Initialized empty Git repository in /Users/joe/nobackup/deps/deps/meck/.git/
==> meck (get-deps)
```

Rebar fetches the programs we specify in the configuration file and recursively fetches any programs that these programs depend upon.

One we have fetched the programs, we compile them with the command rebar compile.

```
$ rebar compile
... many lines of output ...
```

The final step is to tell Erlang where our dependencies are stored. This is done by moving the following lines of code to my ${HOME}/.erlang startup file:

```
%% Set paths to fix all dependencies
Home = os:getenv("HOME").
Dir = Home ++ "/nobackup/erlang_imports/deps",
{ok, L} = file:list_dir(Dir).
lists:foreach(fun(I) ->
                Path = Dir ++ "/" ++ I ++ "/ebin",
                code:add_path(Path)
         end, L).
```

25.4 Building Embedded Web Servers with Cowboy

Cowboy is a small, fast, modular HTTP server written in Erlang and available from https://github.com/extend/cowboy. It is supported by a company called Nine Nines.[3]

Cowboy is suitable for building embedded applications. There are no configuration files, and it produces no logs. Everything is controlled from Erlang.

We'll make a very simple web server that is started with the command simple_web_server:start(Port). This starts a web server listening for commands on Port and whose root directory is the directory where the program was started.

The main function that starts everything is as follows:

cowboy/simple_web_server.erl
```
Line 1  start(Port) ->
     -       ok = application:start(crypto),
     -       ok = application:start(ranch),
     -       ok = application:start(cowboy),
     5       N_acceptors = 10,
     -       Dispatch = cowboy_router:compile(
     -                       [
     -                        %% {URIHost, list({URIPath, Handler, Opts})}
     -                        {'_', [{'_', simple_web_server, []}]}
    10                       ]),
     -       cowboy:start_http(my_simple_web_server,
     -                           N_acceptors,
     -                           [{port, Port}],
     -                           [{env, [{dispatch, Dispatch}]}]
    15                          ).
```

Lines 2 to 4 start the OTP applications. Line 5 sets the number of "acceptors" that the web server has to 10. This means the web server uses ten parallel processes to accept HTTP connection requests. The number of simultaneous

---

3.   http://ninenines.eu

parallel sessions can be far greater than this. The variable Dispatch contains a list of "dispatcher patterns." A dispatch pattern maps URI paths onto the module names that will handle the individual requests. The pattern in line 9 maps all requests to the module simple_web_server.

cowboy_router:compile(Display) compiles the dispatcher information to create an efficient dispatcher, and cowboy:start_http/4 starts the web server.

The module names given in the dispatcher patterns must provide three call-back routines: init/3, handle/3, and terminate/2. In our case, there is only one handler module, which is called simple_web_server. First is init/3, which gets called when a fresh connection is made to the web server.

cowboy/simple_web_server.erl

```
init({tcp, http}, Req, _Opts) ->
    {ok, Req, undefined}.
```

init is called with three arguments. The first says what type of connection has been made to the server. In this case, it's an HTTP connection. The second argument is what cowboy calls a *request object*. The request object contains information about the request and will eventually contain information that is sent back to the browser. Cowboy provides a large set of functions for extracting information from the request object and for storing information in the request object that will be subsequently sent to the browser. The third argument to init (Opt) is whatever was given as the third argument in the dispatch tuple in the call to cowboy_router:compile/1.

init/3 conventionally returns the tuple {ok, Req, State}, which causes the web server to accept the connection. Req is the request object, and State is a private state associated with the connection. If the connection was accepted, then the HTTP driver will call the function handle/2 with the request object and state that was returned from the init function. handle/2 is as follows:

cowboy/simple_web_server.erl

```
Line 1  handle(Req, State) ->
     2      {Path, Req1} = cowboy_req:path(Req),
     3      Response = read_file(Path),
     4      {ok, Req2} = cowboy_req:reply(200, [], Response, Req1),
     5      {ok, Req2, State}.
```

handle calls cowboy_req:path(Req) (line 2) to extract the path to the resource that has been requested. So, for example, if the user requests a page from the address http://localhost:1234/this_page.html, then cowboy_req:path(Req) will return the path <<"/this_page.html">>. The path is represented as an Erlang binary.

The result of reading the file (Response) is packed into the request object by calling cowboy_req:reply/4 and becomes part of the return value from handle/2 (lines 4 and 5).

Reading the requested page is done by read_file/1.

cowboy/simple_web_server.erl
```erlang
read_file(Path) ->
    File = ["."|binary_to_list(Path)],
    case file:read_file(File) of
        {ok, Bin} -> Bin;
        _ -> ["<pre>cannot read:", File, "</pre>"]
    end.
```

Since we assumed that all files are served from the directory where the web server was started, we prepend a dot to the filename (which otherwise started with a forward slash) so that we read the correct file.

That's pretty much it. What happens now depends upon how the socket was established. If it was a keep-alive connection, then handle will be called again. If the connection is closed, then terminate/3 will be called.

cowboy/simple_web_server.erl
```erlang
terminate(_Reason, _Req, _State) ->
    ok.
```

Now that we've seen how to make a simple server, we'll play with the basic structures involved and make a more useful example.

We're going to program a JSON round-trip from the browser to Erlang and back again. This example is interesting because it shows how to interface a browser and Erlang. We start off with a JavaScript object in the browser. We send this to Erlang encoded as a JSON message. We decode this message in Erlang where it becomes an Erlang data structure and then send it back to the browser and turn it back into a JavaScript object. If everything goes well, the object will survive the round-trip and end up as it started.

We'll start with the Erlang code, making it slightly more general than in the previous example. We'll add a meta-call facility so that we can call an arbitrary Erlang function from the browser. When the browser requests a page with a URI of the form http://Host/cgi?mod=Modname&Func=Funcname, we want the function Mod:Func(Args) to be called in the Erlang web server. Args is assumed to be a JSON data structure.

The code to do this is as follows:

cowboy/cgi_web_server.erl
```erlang
handle(Req, State) ->
    {Path, Req1} = cowboy_req:path(Req),
    handle1(Path, Req1, State).
handle1(<<"/cgi">>, Req, State) ->
    {Args, Req1} = cowboy_req:qs_vals(Req),
    {ok, Bin, Req2} = cowboy_req:body(Req1),
    Val = mochijson2:decode(Bin),
    Response = call(Args, Val),
    Json = mochijson2:encode(Response),
    {ok, Req3} = cowboy_req:reply(200, [], Json, Req2),
    {ok, Req3, State};
handle1(Path, Req, State) ->
    Response = read_file(Path),
    {ok, Req1} = cowboy_req:reply(200, [], Response, Req),
    {ok, Req1, State}.
```

This is similar to the code shown earlier in the chapter. The only differences are that we have called cowboy_req:qs to decompose the query string and cowboy_req:body to extract the body of the HTTP request. We called encode and decode routines from the mochiweb2 library (available from https://github.com/mochi/mochiweb/) to convert between JSON strings and Erlang terms. The code to handle the call is as follows:

cowboy/cgi_web_server.erl
```erlang
call([{<<"mod">>,MB},{<<"func">>,FB}], X) ->
    Mod = list_to_atom(binary_to_list(MB)),
    Func = list_to_atom(binary_to_list(FB)),
    apply(Mod, Func, [X]).
```

And here's the echo code:

cowboy/echo.erl
```erlang
-module(echo).
-export([me/1]).

me(X) ->
    io:format("echo:~p~n",[X]),
    X.
```

We've finished with the Erlang code. Now for the corresponding code in the browser. This just needs a few lines of JavaScript and calls to the jQuery library.

cowboy/test2.html
```html
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<h1>Test2</h1>
<button id="button1">click</button>
```

```
<div id="result"></div>
<script>
$(document).ready(go);
var data = {int:1234,
            string:"abcd",
            array:[1,2,3,'abc'],
            map:{one:'abc', two:1, three:"abc"}};
function go(){
    $("#button1").click(test);
}
function test(){
    $.ajax({url:"cgi?mod=echo&func=me",
            type:'POST',
            data:JSON.stringify(data),
            success:function(str){
                var ret = JSON.parse(str);
                $("#result").html("<pre>"+
                                  JSON.stringify(ret, undefined, 4) +
                                  "</pre>");
            }});
}
</script>
```

Now we'll start the web server on port 1234; we can do this in the shell.

```
1> cgi_web_server:start(1234).
{ok,...}
```

Once the server has started, we can enter the address http://localhost:1234/test2.html in the browser, and we'll see a page with a button on it. When we click the button, the test is performed, and the browser displays the data sent back from Erlang (Figure 7, *JSON term sent from Erlang,* on page 436).

And here is what we see in the Erlang shell window:

```
> echo:{struct,[{<<"int">>,1234},
                {<<"string">>,<<"abcd">>},
                {<<"array">>,[1,2,3,<<"abc">>]},
                {<<"map">>,
                {struct,[{<<"one">>,<<"abc">>},
                        {<<"two">>,1},
                        {<<"three">>,<<"abc">>}]}}]]
```

This is the Erlang parse tree returned by mochijson2:decode/1. As you can see, the data is correctly transferred between the two systems.

Note: Not all JSON terms will survive a round-trip between the browser and Erlang. JavaScript has limited precision integers, and Erlang has bignums, so when dealing with large integers, we might run into problems. Similarly, floating-point numbers might lose precision in the conversion process.

Figure 7—JSON term sent from Erlang

As an alternative to starting the web server from the Erlang shell, we might want to start the server either from a makefile or from the command line. In this case, we need to add a routine to convert the arguments that Erlang receives from the shell (which is a list of atoms) into the form needed to start the server.

cowboy/cgi_web_server.erl
```
start_from_shell([PortAsAtom]) ->
    PortAsInt = list_to_integer(atom_to_list(PortAsAtom)),
    start(PortAsInt).
```

Then, for example, to start a server that listens to port 5000, we'd give the following command:

```
$ erl -s cgi_web_server start_from_shell 5000
```

This chapter showed how to use rebar for simple project management. We showed how to create a new project on GitHub and manage it with rebar and how to include projects from GitHub and include them in our own work. We gave a simple example of how to start building a specialized web server using

cowboy. The web server that was built for Chapter 18, *Browsing with Websockets and Erlang,* on page 287 is largely similar to the code shown in this chapter.

The second cowboy example converted JSON terms into Erlang structures using an encoding and decoding routine from mochiweb. When maps appear in Erlang version R17, I'll change the code in the book to reflect this.

In the next chapter, we'll take a look at multicore computers and explore some parallelization techniques for running code on multicore computers. When we run on multicore CPUs, concurrent programs become parallel programs and should just run faster—we'll see if this is true.

Exercises

1. Sign up for an account on GitHub and go through the steps at the start of this chapter to create your own project.

2. The second cowboy example might be unsafe. A user could request the execution of any Erlang module through the CGI call interface. Redesign the interface to allow calls only to a set of predefined modules.

3. Do a security audit of the cowboy example. There are a number of security problems with this code. For example, the value of the requested file is unchecked, so it is possible to access files outside the web server directory structure. Find and fix the security problems.

4. Any host can connect to the cowboy server. Change the code so as to only allow connections from hosts with known IP addresses. Store these hosts in some form of persistent database, for example Mnesia or bitcask. Keep track of how many times connections are made from a specific host. Make a blacklist of hosts that connect too frequently within a given time period.

5. Change the web server to allow dynamic recompilation of the modules that are called through the CGI interface. In our example, the module echo.erl had to be compiled before it could be called. When a module is called through the CGI interface, read the timestamp on the beam file and compare it with the timestamp on the corresponding .erl. Then recompile and reload the Erlang code if necessary.

6. Rebar is an excellent example of an Erlang program that is distributed as a "self-contained" binary. Copy the rebar executable to a scratch directory and rename it as rebar.zip. Rebar is actually a zip file. Unzip it and examine the contents. Make your own self-executing binary out of the cowboy example code.
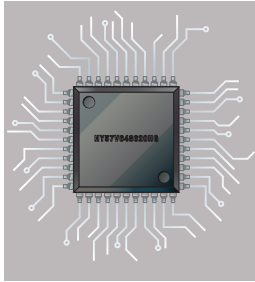
Programming Multicore CPUs

How can we write programs that run faster on a multicore CPU? It's all about mutable state and concurrency.

Back in the old days (twenty-odd years ago), there were two models of concurrency.

- Shared state concurrency
- Message passing concurrency

The programming world went one way (toward shared state). The Erlang community went the other way. (Few other languages followed the "message passing concurrency" road; some others were Oz and Occam.)

In message passing concurrency, there is no shared state. All computations are done in processes, and the *only* way to exchange data is through asynchronous message passing.

Why is this good?

Shared state concurrency involves the idea of "mutable state" (memory that can be changed)—all languages such as C, Java, C++, and so on, have the notion that there is this stuff called *state* and that we can change it.

This is fine as long as you have only *one* process doing the changing.

If you have multiple processes sharing and modifying the *same* memory, you have a recipe for disaster—madness lies here.

To protect against the simultaneous modification of shared memory, we use a locking mechanism. Call this a mutex, a synchronized method, or what you will, but it's still a lock.

If programs crash in the critical region (when they hold the lock), disaster results. All the other programs don't know what to do. If programs corrupt

the memory in the shared state, disaster will also happen. The other programs won't know what to do.

How do programmers fix these problems? With great difficulty. On a unicore processor, their program might just work—but on a multicore…disaster.

There are various solutions to this (transactional memory is probably the best), but these are at best kludges. At their worst, they are the stuff of nightmares.

Erlang has no mutable data structures (that's not quite true, but it's true enough).

- No mutable data structures = No locks.
- No mutable data structures = Easy to parallelize.

How do we do the parallelization? Easy. The programmer breaks up the solution of the problem into a number of parallel processes.

This style of programming has its own terminology; it's called *concurrency-oriented programming.*

26.1 Good News for Erlang Programmers

Here's the good news: your Erlang program might run *n* times faster on an *n*-core processor—*without any changes to the program.*

But you have to follow a simple set of rules.

If you want your application to run faster on a multicore CPU, you'll have to make sure that it has lots of processes, that the processes don't interfere with each other, and that you have no sequential bottlenecks in your program.

If instead you've written your code in one great monolithic clump of sequential code and never used spawn to create a parallel process, your program might not go any faster.

Don't despair. Even if your program started as a gigantic sequential program, several simple changes to the program will parallelize it.

In this chapter, we'll look at the following topics:

- What we have to do to make our programs run efficiently on a multicore CPU
- How to parallelize a sequential program
- The problem of sequential bottlenecks
- How to avoid side effects

When we've done this, we'll look at the design issues involved in a more complex problem. We'll implement a higher-order function called mapreduce

and show how it can be used to parallelize a computation. mapreduce is an abstraction developed by Google for performing parallel computations over sets of processing elements.

26.2 How to Make Programs Run Efficiently on a Multicore CPU

To run efficiently, we have to do the following:

- Use lots of processes
- Avoid side effects
- Avoid sequential bottlenecks
- Write "small messages, big computations" code

If we do all of these, our Erlang program should run efficiently on a multicore CPU.

Why Should We Care About Multicore CPUs?

You might wonder what all the fuss is about. Do we have to bother parallelizing our program so that it can run on a multicore? The answer is yes. Today, quad cores with hyperthreads are commonplace. Smartphones can have quadcores. Even my lowly MacBook Air has a dual core with hyperthreading, and my desktop has eight cores with hyperthreading.

Making a program go twice as fast on a dual-core machine is not that exciting (but it is a little bit exciting). But let's not delude ourselves. The clock speeds on dual-core processors are slower than on a single-core CPU, so the performance gains can be marginal.

Now although two times doesn't get me excited, ten times does, and a hundred times is really, really exciting. Modern processors are so fast that a single core can run 4 hyperthreads, so a 32-core CPU might give us an equivalent of 128 threads to play with. This means that a hundred times faster is within striking distance.

A factor of one hundred does make me excited.

All we have to do is write the code.

Use Lots of Processes

This is important—we have to keep the CPUs busy. All the CPUs should be busy all the time. The easiest way to achieve this is to have lots of processes.

When I say lots of processes, I mean lots in relation to the number of CPUs. If we have lots of processes, then we won't need to worry about keeping the CPUs busy. This appears to be a purely statistical effect. If we have a small number of processes, they might accidentally hog one of the CPUs; this effect

seems to go away if we have a large number of processes. If we want our programs to be future-proof, we should think that even though today's chips might have only a small number of CPUs, in the future we might have thousands of CPUs per chip.

Preferably the processes should do similar amounts of work. It's a bad idea to write programs where one process does a lot of work and the others do very little.

In many applications we get lots of processes "for free." If the application is "intrinsically parallel," then we don't have to worry about parallelizing our code. For example, if we're writing a messaging system that manages some tens of thousands of simultaneous connections, then we get the concurrency from the tens of thousands of connections; the code that handles an individual connection will not have to worry about concurrency.

Avoid Side Effects

Side effects prevent concurrency. Right in the beginning of the book we talked about "variables that do not vary." This is the key to understanding why Erlang programs can run faster on a multicore CPU than programs written in languages that can destructively modify memory.

In a language with shared memory and threads, a disaster might happen if two threads write to common memory at the same time. Systems with shared memory concurrency prevent this by locking the shared memory while the memory is being written to. These locks are hidden from the programmer and appear as mutex or synchronized methods in their programming languages. The main problem with shared memory is that one thread can corrupt the memory used by another thread. So, even if my program is correct, another thread can mess up my data structures and cause my program to crash.

Erlang does not have shared memory, so this problem does not exist. Actually, this is not quite true. There are only two ways to share memory, and the problem can be easily avoided. These two ways of sharing memory have to do with shared ETS or DETS tables.

Don't Use Shared ETS or DETS Tables

ETS tables can be shared by several processes. In Section 19.3, *Creating an ETS Table*, on page 309, we talked about the different ways of creating an ETS table. Using one of the options to ets:new, we could create a public table type. Recall that this did the following:

Create a public table. Any process that knows the table identifier can read and write this table.

This can be dangerous. It is safe only if we can guarantee the following:

- That only one process at a time writes to the table and that all other processes read from the table

- That the process that writes to the ETS table is correct and does not write incorrect data into the table

These properties cannot in general be guaranteed by the system but instead depend upon the program logic.

Note 1: Individual operations on ETS tables are atomic. What is not possible is performing a sequence of ETS operations as one atomic unit. Although we cannot corrupt the data in an ETS table, the tables can become logically inconsistent if several processes try to simultaneously update a shared table without coordinating their activities.

Note 2: The ETS table type `protected` is far safer. Only one process (the owner) can write to this table, but several processes can read the table. This property *is guaranteed* by the system. But remember, even if only one process can write to an ETS table, if this process corrupts the data in the table, all processes reading the table will be affected.

If you use the ETS table type `private`, then your programs will be safe. Similar observations apply to DETS. We can create a shared DETS table to which several different processes can write. This should be avoided.

Note: ETS and DETS were created in order to implement Mnesia and were not originally intended for stand-alone use. The intention is that application programs should use the Mnesia transaction mechanisms if they want to simulate shared memory between processes.

Avoid Sequential Bottlenecks

Once we've parallelized our program and have made sure we have lots of processes and no shared memory operations, the next problem to think about is sequential bottlenecks. Certain things are intrinsically sequential. If the "sequentialness" lies in the problem, we can't make it go away. Certain events happen in a certain sequential order, and no matter how we try, we can't change this order. We are born, we live, we die. We can't change the order. We can't do these things in parallel.

A sequential bottleneck is where several concurrent processes need access to a sequential resource. A typical example is I/O. Typically we have a single disk, and all output to the disk is ultimately sequential. The disk has one set of heads, not two, and we can't change that.

Every time we make a registered process, we are creating a potential sequential bottleneck. So, try to avoid the use of registered processes. If you do create a registered process and use it as a server, make sure that it responds to all requests as quickly as possible.

Often, the only solution to a sequential bottleneck is to change the algorithm concerned. There is no cheap and easy fix here. We have to change the algorithm from a nondistributed algorithm to a distributed algorithm. This topic (distributed algorithms) has a vast amount of research literature available but has had relatively little adoption in conventional programming language libraries. The main reason for this lack of adoption is that the need for such algorithms is not apparent until we try to program networked algorithms or multicore computers.

Programming computers that are permanently connected to the Internet and multicore CPUs will force us to dig into the research literature and implement some of these amazing algorithms.

A Distributed Ticket-Booking System

Suppose we have a single resource, a set of tickets to the next concert of the Strolling Bones. To guarantee that when you buy a ticket you actually get a ticket, we'd conventionally use a single agency that books all the tickets. But this introduces a sequential bottleneck.

The single agency bottleneck can be avoided by having two ticket agencies. At the start of sales, the first ticket agency is given all the even-numbered tickets, and the second agency is given all the odd-numbered tickets. This way, the agencies are guaranteed not to sell the same ticket twice.

If one of the agencies runs out of tickets, then it can request a bundle of tickets from the other agency.

I'm not saying this is a good method, because you might actually want to sit next to your friends when you go to a concert. But it does remove the bottleneck by replacing a single ticket office with two.

Replacing the single booking agency by n distributed agencies where n can vary with time and where the individual agencies can join and leave the network and crash at any time is an area of active research in distributed

computing. This research goes by the name *distributed hash tables*. If you Google this term, you'll find a vast array of literature on the subject.

26.3 Parallelizing Sequential Code

Recall the emphasis we made on list-at-a-time operations and in particular the function lists:map/2? map/2 is defined like this:

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

A simple strategy for speeding up our sequential programs would replace all calls to map with a new version of map that I'll call pmap, which evaluates all its arguments in parallel.

```
lib_misc.erl
pmap(F, L) ->
    S = self(),
    %% make_ref() returns a unique reference
    %%   we'll match on this later
    Ref = erlang:make_ref(),
    Pids = map(fun(I) ->
                       spawn(fun() -> do_f(S, Ref, F, I) end)
               end, L),
    %% gather the results
    gather(Pids, Ref).

do_f(Parent, Ref, F, I) ->
    Parent ! {self(), Ref, (catch F(I))}.
gather([Pid|T], Ref) ->
    receive
        {Pid, Ref, Ret} -> [Ret|gather(T, Ref)]
    end;
gather([], _) ->
    [].
```

pmap works like map, but when we call pmap(F, L), it creates one parallel process to evaluate each argument in L. Note that the processes that evaluate the arguments of L can complete in any order.

The selective receive in the gather function ensures that the order of the arguments in the return value corresponds to the ordering in the original list.

There is a slight semantic difference between map and pmap. In pmap, we use (catch F(H)) when we map the function over the list. In map, we just use F(H). This is because we want to make sure pmap terminates correctly in the case where the computation of F(H) raises an exception. In the case where no exceptions are raised, the behavior of the two functions is identical.

Important: This last statement is not strictly true. map and pmap will not behave the same way if they have side effects. Suppose F(H) has some code that modifies the process dictionary. When we call map, the changes to the process dictionary will be made in the process dictionary of the process that called map.

When we call pmap, each F(H) is evaluated in its own process, so if we use the process dictionary, the changes in the dictionary will not affect the process dictionary in the program that called pmap.

So, be warned: code that has side effects cannot be simply parallelized by replacing a call to map with pmap.

When Can We Use pmap?

Using pmap instead of map is not a general panacea for speeding up your programs. The following are some things to think about.

Granularity of Concurrency

Don't use pmap if the amount of work done in the function is small. Suppose we say this:

```
map(fun(I) -> 2*I end, L)
```

Here the amount of work done inside the fun is minimal. The overhead of setting up a process and waiting for a reply is greater than the benefit of using parallel processes to do the job.

Don't Create Too Many Processes

Remember that pmap(F, L) creates length(L) parallel processes. If L is very large, you will create a lot of processes. The best thing to do is create a *lagom* number of processes. Erlang comes from Sweden, and the word *lagom* loosely translated means "not too few, not too many, just about right." Some say that this summarizes the Swedish character.

Think About the Abstractions You Need

pmap might not be the right abstraction. We can think of many different ways of mapping a function over a list in parallel; we chose the simplest possible here.

The pmap version we used cared about the order of the elements in the return value (we used a selective receive to do this). If we didn't care about the order of the return values, we could write this:

```
lib_misc.erl
pmap1(F, L) ->
    S = self(),
    Ref = erlang:make_ref(),
    foreach(fun(I) ->
                    spawn(fun() -> do_f1(S, Ref, F, I) end)
            end, L),
    %% gather the results
    gather1(length(L), Ref, []).

do_f1(Parent, Ref, F, I) ->
    Parent ! {Ref, (catch F(I))}.

gather1(0, _, L) -> L;
gather1(N, Ref, L) ->
    receive
        {Ref, Ret} -> gather1(N-1, Ref, [Ret|L])
    end.
```

A simple change to this could turn this into a parallel foreach. The code is similar to the previous code, but we don't build any return value. We just note the termination of the program.

Another method would be to implement pmap using at most K processes where K is some fixed constant. This might be useful if we want to use pmap on very large lists.

Yet another version of pmap could map the computations not only over the processes in a multicore CPU but also over nodes in a distributed network.

I'm not going to show you how to do this here. You can think about this for yourself.

The purpose of this section is to point out that there is a large family of abstractions that can easily be built from the basic spawn, send, and receive primitives. You can use these primitives to create your own parallel control abstractions to increase the concurrency of your program.

As before, the absence of side effects is the key to increasing concurrency. Never forget this.

26.4 Small Messages, Big Computations

We've talked theory; now for some measurements. In this section, we'll perform two experiments. We'll map two functions over a list of a hundred elements, and we'll compare the time it takes with a parallel map vs. a sequential map.

We'll use two different problem sets. The first computes this:

```
L = [L1, L2, ..., L100],
map(fun lists:sort/1, L)
```

Each of the elements in L is a list of 1,000 random integers.

The second computes this:

```
L = [27,27,..., 27],
map(fun ptests:fib/1, L)
```

Here, L is a list of one hundred 27s, and we compute the list [fib(27), fib(27), …] one hundred times. (fib is the Fibonacci function.)

We'll time both these functions. Then we'll replace map with pmap and repeat the timings.

Using pmap in the first computation (sort) involves sending a relatively large amount of data (a list of 1,000 random numbers) in messages between the different processes. The sorting process is rather quick. The second computation involves sending a small request (to compute fib(27)) to each process, but the work involved in recursively computing fib(27) is relatively large.

Since there is little copying of data between processes in computing fib(27) and a relatively large amount of work involved, we would expect the second problem to perform better than the first on a multicore CPU.

To see how this works in practice, we need a script that automates our tests. But first we'll look at how to start SMP Erlang.

Running SMP Erlang

A symmetric multiprocessing (SMP) machine has two or more identical CPUs that are connected to a single shared memory. These CPUs may be on a single multicore chip, spread across several chips, or a combination of both. Erlang runs on a number of different SMP architectures and operating systems. The current system runs with Intel dual- and quad-core processors on motherboards with one or two processors. It also runs with Sun and Cavium processors. This is an area of extremely rapid development, and the number of supported operating systems and processors increases with every release of Erlang. You can find up-to-date information in the release notes of the current Erlang distribution. (Click the title entry for the latest version of Erlang in the download directory at http://www.erlang.org/download.html.)

Note: SMP Erlang has been enabled by default (that is, an SMP virtual machine is built by default) since Erlang version R11B-0. To force SMP Erlang to be built on other platforms, the --enable-smp-support flag should be given to the configure command.

SMP Erlang has two command-line flags that determine how it runs on a multicore CPU.

```
$erl -smp +S N
```

-smp

> Start SMP Erlang.

+S N

> Run Erlang with `N` schedulers. Each Erlang scheduler is a complete virtual machine that knows about all the other virtual machines. If this parameter is omitted, it defaults to the number of logical processors in the SMP machine.
>
> Why would we want to vary this? For a number of reasons:
>
> - When we do performance measurements, we want to vary the number of the schedulers to see the effect of running with a different number of CPUs.
>
> - On a single-core CPU, we can emulate running on a multicore CPU by varying `N`.
>
> - We might want to have more schedulers than physical CPUs. This can sometimes increase throughput and make the system behave in a better manner. These effects are not fully understood and are the subject of active research.

To perform our tests, we need a script to run the tests.

runtests
```sh
#!/bin/sh
echo "" >results
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16\
        17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

do
   echo $i
   erl -boot start_clean -noshell -smp +S $i \
       -s ptests tests $i >> results

done
```

This just starts Erlang with one to thirty-two different schedulers, runs a timing test, and collects all the timings into a file called results.

Then we need a test program.

ptests.erl
```erlang
-module(ptests).
-export([tests/1, fib/1]).
-import(lists, [map/2]).
-import(lib_misc, [pmap/2]).

tests([N]) ->
    Nsched = list_to_integer(atom_to_list(N)),
    run_tests(1, Nsched).

run_tests(N, Nsched) ->
    case test(N) of
        stop ->
            init:stop();
        Val ->
            io:format("~p.~n",[{Nsched, Val}]),
            run_tests(N+1, Nsched)
    end.
test(1) ->
    %% Make 100 lists
    %%   Each list contains 1000 random integers
    seed(),
    S = lists:seq(1,100),
    L = map(fun(_) -> mkList(1000) end, S),
    {Time1, S1} = timer:tc(lists,    map, [fun lists:sort/1, L]),
    {Time2, S2} = timer:tc(lib_misc, pmap, [fun lists:sort/1, L]),
    {sort, Time1, Time2, equal(S1, S2)};
test(2) ->
    %% L = [27,27,27,..] 100 times
    L = lists:duplicate(100, 27),
    {Time1, S1} = timer:tc(lists,    map, [fun ptests:fib/1, L]),
    {Time2, S2} = timer:tc(lib_misc, pmap, [fun ptests:fib/1, L]),
    {fib, Time1, Time2, equal(S1, S2)};
test(3) ->
    stop.

%% Equal is used to test that map and pmap compute the same thing
equal(S,S)   -> true;
equal(S1,S2) ->  {differ, S1, S2}.

%% recursive (inefficent) fibonacci
fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).

%% Reset the random number generator. This is so we
%% get the same sequence of random numbers each time we run
%% the program

seed() -> random:seed(44,55,66).
```

```
%% Make a list of K random numbers
%%     Each random number in the range 1..1000000
mkList(K) -> mkList(K, []).

mkList(0, L) -> L;
mkList(N, L) -> mkList(N-1, [random:uniform(1000000)|L]).
```

This runs map and pmap in the two different test cases. You can see the results in the following figure, where we have plotted the ratio of times taken by pmap and map.

Multicore Performance on a SUN Fire T2000 Server
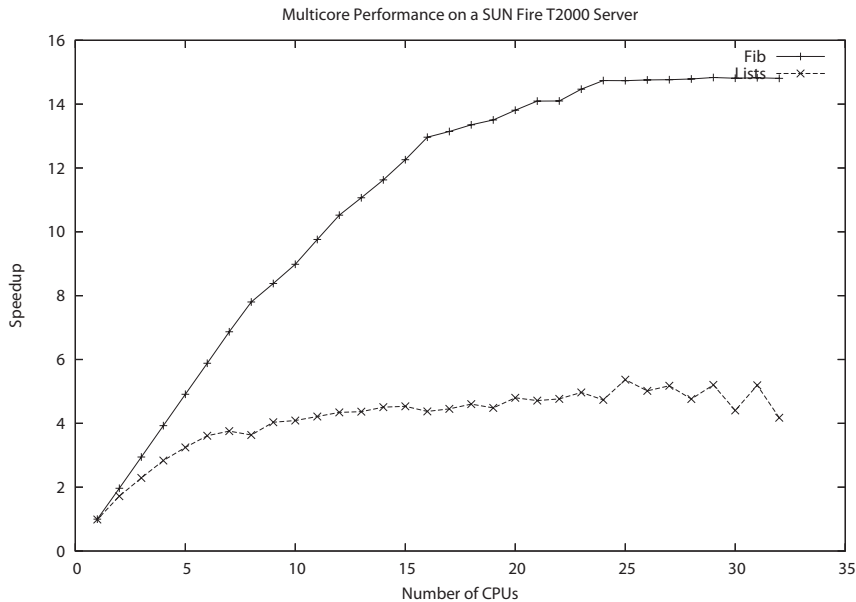
As we can see, CPU-bound computations with little message passing have linear speed-up, whereas lighter-weight computations with more message passing scale less well.

As a final note, we shouldn't read too much into these figures. SMP Erlang is undergoing daily changes, so what is true today may not be true tomorrow. All we can say is that we are very encouraged by our results. Ericsson is building commercial products that run almost twice as fast on dual-core processors, so we are very happy.

26.5 Parallelizing Computations with mapreduce

Now we're going to turn theory into practice. First we'll look at the higher-order function mapreduce; then we'll show how it can be used to parallelize a simple computation.

mapreduce

In the following figure, we can see the basic idea of mapreduce. We have a number of *mapping* processes, which produce streams of {Key, Value} pairs. The mapping processes send these pairs to a *reduce* process that merges the pairs, combining pairs with the same key.
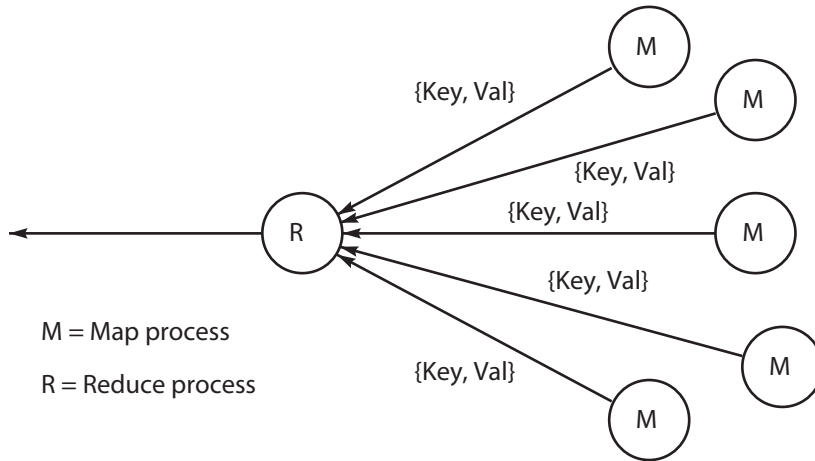
Warning: The word *map*, used in the context of mapreduce, is completely different from the map function that occurs elsewhere in this book.

mapreduce is a *parallel higher-order function*. Proposed by Jeffrey Dean and Sanjay Ghemawat of Google, it is said to be in daily use on Google clusters.

We can implement mapreduce in lots of different ways and with lots of different semantics—it's actually more like a family of algorithms than one particular algorithm.

mapreduce is defined as follows:

```
-spec mapreduce(F1, F2, Acc0, L) -> Acc
    F1 = fun(Pid, X) -> void
    F2 = fun(Key, [Value], Acc0) -> Acc
    L  = [X]
    Acc = X = term()
```

F1(Pid, X) is the mapping function.
    The job of F1 is to send a stream of {Key, Value} messages to the process Pid and then to terminate. mapreduce will spawn a fresh process for each value of X in the list L.

F2(Key, [Value], Acc0) -> Acc is the reduction function.

When all the mapping processes have terminated, the reduce process will have merged all the values for a particular key. mapreduce then calls F2(Key, [Value], Acc) for each of the {Key, [Value]} tuples it has collected. Acc is an accumulator whose initial value is Acc0. F2 returns a new accumulator. (Another way of describing this is to say that F2 performs a *fold* over the {Key, [Value]} tuples that it has collected.)

Acc0 is the initial value of the accumulator, used when calling F2.
L is a list of X values.

F1(Pid, X) will be called for each value of X in L. Pid is a process identifier of the reduce process; this is created by mapreduce.

mapreduce is defined in the module phofs (short for *parallel higher-order functions*).

phofs.erl
```erlang
-module(phofs).
-export([mapreduce/4]).
-import(lists, [foreach/2]).

%% F1(Pid, X) -> sends {Key,Val} messages to Pid
%% F2(Key, [Val], AccIn) -> AccOut
mapreduce(F1, F2, Acc0, L) ->
    S = self(),
    Pid = spawn(fun() -> reduce(S, F1, F2, Acc0, L) end),
    receive
        {Pid, Result} ->
            Result
    end.

reduce(Parent, F1, F2, Acc0, L) ->
    process_flag(trap_exit, true),
    ReducePid = self(),
    %% Create the Map processes
    %%   One for each element X in L
    foreach(fun(X) ->
                    spawn_link(fun() -> do_job(ReducePid, F1, X) end)
            end, L),
    N = length(L),
    %% make a dictionary to store the Keys
    Dict0 = dict:new(),
    %% Wait for N Map processes to terminate
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(F2, Acc0, Dict1),
    Parent ! {self(), Acc}.

%% collect_replies(N, Dict)
%%     collect and merge {Key, Value} messages from N processes.
```

```erlang
%%      When N processes have terminated return a dictionary
%%      of {Key, [Value]} tuples

collect_replies(0, Dict) ->
    Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} ->
            case dict:is_key(Key, Dict) of
                true ->
                    Dict1 = dict:append(Key, Val, Dict),
                    collect_replies(N, Dict1);
                false ->
                    Dict1 = dict:store(Key,[Val], Dict),
                    collect_replies(N, Dict1)
            end;
        {'EXIT', _,  _Why} ->
            collect_replies(N-1, Dict)
    end.

%% Call F(Pid, X)
%%   F must send {Key, Value} messsages to Pid
%%      and then terminate

do_job(ReducePid, F, X) ->
    F(ReducePid, X).
```

Before we go any further, we'll test `mapreduce` so as to be really clear about what it does.

We'll write a little program to count the frequencies of all words in the code directory that accompanies this book. Here is the program:

test_mapreduce.erl
```erlang
-module(test_mapreduce).
-compile(export_all).
-import(lists, [reverse/1, sort/1]).

test() ->
    wc_dir(".").

wc_dir(Dir) ->
    F1 = fun generate_words/2,
    F2 = fun count_words/3,
    Files = lib_find:files(Dir, "*.erl", false),
    L1 = phofs:mapreduce(F1, F2, [], Files),
    reverse(sort(L1)).

generate_words(Pid, File) ->
    F = fun(Word) -> Pid ! {Word, 1} end,
    lib_misc:foreachWordInFile(File, F).
```

```
count_words(Key, Vals, A) ->
    [{length(Vals), Key}|A].

1> test_mapreduce:test().
[{341,"1"},
 {330,"end"},
 {318,"0"},
 {265,"N"},
 {235,"X"},
 {214,"T"},
 {213,"2"},
 {205,"start"},
 {196,"L"},
 {194,"is"},
 {185,"file"},
 {177,"Pid"},
 ...
```

When I ran this, there were 102 Erlang modules in the code directory; mapreduce created 102 parallel processes, each sending a stream of pairs to the reduce process. This should run nicely on a hundred-core processor (if the disk can keep up).

When work in Erlang started in 1985, we had no idea that parallel computers would be commonplace and that computer clusters could be manufactured on a single chip. Erlang was designed for programming fault-tolerant clusters of computers. To achieve fault tolerance, you must have more than one machine, and the programs must run concurrently and in parallel.

When multicore CPUs arrived, we found that a lot of our programs just ran faster. The programs had been written for parallel execution, and all we needed was parallel hardware to run them on.

Erlang does not provide parallel abstractions like pmap and mapreduce; instead, it provides a small set of primitives (spawn, send, and receive) from which such abstractions can be built. This chapter showed how to build things such as pmap, and so on, from the underlying primitives.

At this point in the book we've covered a lot of ground. We've looked at the standard libraries and the major components of the OTP framework. We've looked at how to make a stand-alone system and how to build programs in distributed systems and on multicores.

In the next chapter, we'll launch straight into an application, using many of the techniques developed earlier in the book. We'll solve Sherlock's last case.

Exercises

Parallelizing computations can often dramatically speed up response times. We'll show this with the following exercises:

1. In section *Fetching Data from a Server*, on page 264, we wrote a program that fetched a web page. Change this program to issue a HEAD command instead of a GET command. We can issue an HTTP HEAD command to measure the response time of a website. In responding to a HEAD command, only the web page headers are returned and not the body. Write a function called web_profiler:ping(URL, Timeout) that measures the response time to a website address URL. This should return {time, T} or timeout.

2. Make a long list of websites called L. Time how long lists:map(fun(I) -> web_profiler:ping(URL, Timeout) end, L) takes. This might take quite a long time; the worst-case time is Timeout x length(L).

3. Repeat the last measurement, using pmap instead of map. Now all the HEAD requests should go out in parallel. So, the worst-case response time is just Timeout.

4. Store the results in a database and make a web interface to query the database. You can base your code for this on the database and web server code developed in Chapter 24, *Programming Idioms*, on page 413.

5. If you call pmap with an extremely long list of elements, you might create too many parallel processes. Write a function called pmap(F, L, Max) that computes the list [F(I) || I <- L] in parallel but is subject to the restriction that no more than Max parallel processes run simultaneously.

6. Write a version of pmap that works with distributed Erlang and distributes the work over several Erlang nodes.

7. Write a version of pmap that works with distributed Erlang, distributes the work over several Erlang nodes, and load balances the work between the nodes.

Sherlock's Last Case

"All we have is a fragment of the program," said Armstrong, "but we haven't a clue where it came from or what it's about."

"Show me," said Holmes.

"It's like this," said Armstrong, and he leaned over to Holmes and showed him a small scrap of paper filled with strange symbols, brackets, and arrows interleaved with English text.

"What is it?" asked Armstrong. "Lestrade said it was some kind of powerful black magic, come from the future."

"It's part of a computer program," said Holmes, puffing on his pipe. "As part of my experiments with time travel I've managed to teleport a computer and a large set of files from the future. In these files I discovered that there were 73,445 mails in the Erlang mailing list. I think that by comparing the text on your piece of paper with these mails that we will be able to discover the meaning of the strange symbols. But how? I took out my Stradivarius and played a piece by Paganini. Then it came to me. The most similar posting in the mailing list to the text on your scrap of paper must be that which maximizes the cosine similarity of the TF*IDF scores of the words in the document...."

"That's brilliant," said Armstrong, "but what's a TF*IFD score?"

"It's elementary, my dear Armstrong," said Holmes. "It's the term frequency times the inverse document frequency. I shall explain...."

27.1 Finding Similarities in Data

At the time of writing there were 73,445 mails to the Erlang mailing list.[1] This represents a vast store of knowledge that can be reused in many ways. We can use it to answer questions about Erlang or for inspiration.

Suppose you're writing a program and need help. This is where Sherlock comes in. Sherlock analyzes your program and then suggests the most similar posting from all previous mails in the Erlang list that might be able to help you.

So, here's the master plan. Step 1 is to download the entire Erlang mailing list and store it locally. Step 2 is to organize and parse all the mails. Step 3 is to compute some property of the mails that can be used for similarity searches. Step 4 is to query the mails to find the most similar mail to your program.

This is the core idea of this chapter. There are many variants. We could download all Erlang modules ever written and search them for similarities. Or we could take a large set of tweets or any large data set you're interested in.

Yet another variant could be used to help you classify data. Suppose you have just written a short note and want to store it in a file. Deciding on a filename or a directory in which to store the file is a difficult problem. Perhaps this is very similar to some other file on my disk that I wrote years ago, or perhaps it is similar to some other document written by some other person that I don't even know about. In both of these cases, I'd like to find the most similar document to my new document from an extremely large set of documents.

Sherlock finds similarities between things when we don't know what the similarities are. We use it to find similarities in the contributions to the Erlang mailing list.

Before we launch into the details of the implementation, we'll show Sherlock in action.

27.2 A Session with Sherlock

In this section, we'll take Sherlock for a test-drive. First we have to initialize Sherlock by fetching and analyzing data from the Erlang mailing lists archives. Once this is done, we'll be able to query the data in various ways.

---

1. http://erlang.org/pipermail/erlang-questions/

Fetching and Preprocessing the Data

Some of the commands in this section take a long time, so we do them only once. First we initialize the system.

```
1> sherlock:init().
Making ${HOME}/.sherlock/mails
sherlock_cache_created
```

This creates a directory structure under your home directory. It will fail if the environment variable HOME is not set. Sherlock stores all data in the directory structure under the ${HOME}/.sherlock top-level directory.

```
2> sherlock:fetch_index().
Written: /Users/joe/.sherlock/mails/questions.html
Written: /Users/joe/.sherlock/mails/questions.term
176 files must be fetched
```

fetch-index fetches the index to the mails in the Erlang mailing list. The index comes in an HTML file from which we can extract a list of all the filenames that store the mails.

```
3> sherlock:fetch_mails().
fetching:"http://erlang.org/pipermail/erlang-questions/1997-January.txt.gz"
written:/Users/joe/.sherlock/mails/cache/1997-January.txt.gz
fetching:"http://erlang.org/pipermail/erlang-questions/1997-May.txt.gz"
written:/Users/joe/.sherlock/mails/cache/1997-May.txt.gz
...
```

Once we know all the filenames, all we have to do is go grab them. The individual files have the year and month embedded in the filenames. We store all the files we've fetched in a directory called ${HOME}/.sherlock/mail/cache. This data should be kept for as long as we want to run the program. When I wrote this chapter and ran the previous command, I downloaded 176 files and 37MB of compressed mail data, which represented 73,445 individual mails.

We can ask Sherlock about the data that has been collected.

```
4> sherlock:mail_years().
["1997","1998","1999","2000","2001","2002","2003","2004",
"2005","2006","2007","2008","2009","2010","2011","2012",
"2013"]
```

This shows that we've managed to download mails from 1997 to 2013.

The next step is to partition the data into years. We create one new directory for each year. So, for example, the data for 2007 is stored in the directory ${HOME}/.sherlock/mail/2007, and so on. Then for each year we collect all the mails

for that year and write the results to the appropriate directory. Having done this, we parse and postprocess the data for each year.

```
5> sherlock:process_all_years().
Parsing mails for: 1997
Parsing: 1997-January.txt.gz
...
1997 had 3 mails
...
Parsing mails for: 1999
Parsing: 1999-January.txt.gz
Parsing: 1999-February.txt.gz
...
1999 had 803 mails
...
2009 had 7906 mails
...
73445 files in 215 seconds (341.6 files/second)
```

This takes a few minutes and results in 17MB of data. We have to do this only once.

Finding the Most Similar Mail to a Given File

Now we're ready to search the data. The query term is *the file I'm working on.* When I developed this program, I was writing a module called sherlock_tfidf. I was curious to know whether my search engine worked, so I entered the following query:

```
1> sherlock:find_mails_similar_to_file("2009", "./src/sherlock_tfidf.erl").
** searching for a mail in 2009 similar to the file:./src/sherlock_tfidf.erl
Searching for=[<<"idf">>,<<"word">>,<<"remove">>,<<"words">>,<<"tab">>,
<<"duplicates">>,<<"ets">>,<<"keywords">>,<<"bin">>,<<"skip">>,
<<"file">>,<<"index">>,<<"binary">>,<<"frequency">>,<<"dict">>]
7260 : 0.27 Word Frequency Analysis
7252 : 0.27 Word Frequency Analysis
7651 : 0.18 tab completion and word killing in the shell
4297 : 0.17 ets vs process-based registry + local vs global dispatch
5324 : 0.16 ets memory usage
5325 : 0.15 ets memory usage
1917 : 0.14 A couple of design questions
1860 : 0.12 leex and yecc spotting double newline
5361 : 0.11 dict slower than ets?
1991 : 0.11 Extending term external format to support  shared substructures
[7260,7252,7651,4297,5324,5325,1917,1860,5361,1991]
```

This query asked Sherlock to search in the 2009 mails for a posting that is similar to the content of the file sherlock_tfidf.erl.

The output is interesting. First, Sherlock thinks that sherlock_tfidf.erl is best described by the keywords idf, word, and so on. These are listed in the "searching for" line. Following this, Sherlock searches all mails posted in 2009 looking for these keywords. The output looks like this:

```
7260 : 0.27 Word Frequency Analysis
7252 : 0.27 Word Frequency Analysis
7651 : 0.18 tab completion and word killing in the shell
4297 : 0.17 ets vs process-based registry + local vs global dispatch
...
```

Each line has a mail index number followed by a similarity weight and a subject line. The similarity weight is a number from 0 to 1. A 1 means the documents are very similar. A 0 means there are no similarities. The highest-scoring mail is mail number 7260 with a similarity weight of 0.27. This might be interesting, so we'll take a look at it in more detail.

```
2> sherlock:print_mail("2009", 7260).
----
ID: 7260
Date: Fri, 04 Dec 2009 17:57:03 +0100
From: ...
Subject: Word Frequency Analysis
Hello!

I need to compute a word frequency analysis of a fairly large corpus. At
present I discovered the disco database
http://discoproject.org/

which seems to include a tf-idf indexer. What about couchdb? I found an
article that it fails rather quickly (somewhere between 100 and 1000
wikipedia text pages)
...
```

When I ran this the first time, I got very excited. The system had discovered a mail talking about TF*IDF indexing, but I hadn't told it to look for TF*IDF indexing. I'd just said to "find any mails that are similar to the code in my file."

So, now I know that mail 7260 is interesting. Perhaps there are some mails that are similar to this mail. We can ask Sherlock.

```
3> sherlock:find_mails_similar_to_mail("2009", "2009", 7260).
Searching for a mail in 2009 similar to mail number 7260 in 2009
Searching for=[<<"indexer">>,<<"analysis">>,<<"couchdb">>,<<"wortd">>,
<<"idf">>,<<"knuthellan.com">>,<<"frequncy">>,<<"corpus">>,
<<"dbm">>,<<"discoproject.org">>,<<"disco">>]
7252 : 0.84 Word Frequency Analysis
6844 : 0.21 couchdb in Karmic Koala
6848 : 0.21 couchdb in Karmic Koala
6847 : 0.20 couchdb in Karmic Koala
```

```
6849 : 0.19 couchdb in Karmic Koala
7264 : 0.17 Re: erlang search engine library?
6843 : 0.16 couchdb in Karmic Koala
2895 : 0.15 CouchDB integration
69 : 0.14 dialyzer fails when using packages and -r
[7252,6844,6848,6847,6849,7264,6843,2895,69]
```

This time I searched for a mail that is similar to mail 7260. Mail 7252 has the highest similarity score and turned out to be very similar to 7260, but mail 7264 turned out to be more interesting. The similarity score alone is not enough to determine which of the files is most interesting; the system only suggests files that might be similar. We actually have to look at the results and choose what we think is the most interesting post.

Staring with the code for sherlock_tfidf.erl, I discovered that the disco database has a TF*IDF indexer. A few queries later and I discovered a connection to couchdb. Sherlock finds all sort of interesting things for me to investigate.

Searching Mails for a Specific Author, Date, or Subject

We can also perform so-called *faceted* search on the data. A facet of a document is, for example, the username or the subject in a document. A faceted search[2] is a search within a specific field or set of fields. Parsed documents are represented as Erlang records. We can perform a specific search of all documents on any of these fields. Here's an example of a faceted search:

```
1> sherlock:search_mails_regexprs("2009", "*Armstrong*", "*Protocol*", "*").
946:   UBF and JSON Protocols
5994:  Message protocol vs. Function call API
Query took:23 ms #results=2
[946,5994]
```

This searches mails in 2009 for items whose author matches the regular expression *Armstrong*, with a subject line matching *Protocol* and with any content. There were two matches: mails 946 and 5994. We examine the first like so:

```
2> sherlock:print_mail("2009", 946).
----
ID: 946
Date: Sun, 15 Feb 2009 12:39:10 +0100
From: Joe Armstrong
Subject: UBF and JSON Protocols
For a long time I have been interested in describing protocols. In
2002 I published a contract system called UBF for defining protocols.
...
```

---

2.   http://en.wikipedia.org/wiki/Faceted_search

And so on. At this point, we can make more queries, searching for specific data or finding similar posts to previous posts.

Now that we've seen Sherlock in action, we'll take a look at the implementation.

27.3 The Importance of Partitioning the Data

I'll let you in on a secret now. Partitioning the data is the key to parallelizing the program. process_all_years(), which we used to build the data store, is defined like this:

```
process_all_years() ->
    [process_year(I) || I <- mail_years()].
```

process_year(Year) processes all the data for a specific year, and mail_years/0 returns a list of years.

To parallelize the program, all you have to do is change the definition of process_all_years and call pmap, which we talked about in Section 26.3, *Parallelizing Sequential Code,* on page 445. With this small change, our function looks like this:

```
process_all_years() ->
    lib_misc:pmap(fun(I) -> process_year(I) end, mail_years()).
```

There is an additional and less obvious benefit. *To test that our program works, we have to work with only one of the years.* If I had a seventeen-or-more-core monster machine, I could run all the year computations in parallel. There are seventeen years of data, so I'd need at least seventeen CPUs and a disk controller that allowed seventeen parallel input operations. Modern solid-state disks have multiple disk controllers, but I'm writing this on a machine with a conventional disk and a dual-core processor, so I can't expect a dramatic speedup if I parallelize the program.

I've been testing my program by analyzing the data for 2009. If I want to speed up the program, I need access to a monster machine, and I just parallelize the program by changing the top-level list comprehension into a pmap.

This form of parallelization is exactly the form used in a map-reduce architecture. To search the mail data quickly, we'd use seventeen machines, each looking at one year's worth of data. We dispatch the same query to all seventeen machines and then gather the results. This is the main idea in map-reduce, and it was discussed earlier in Section 26.5, *Parallelizing Computations with mapreduce,* on page 451.

You'll notice that all the examples in this chapter use 2009 as a base year. 2009 had sufficiently many mails to exercise all the software (there were 7,906

mails in 2009), but processing this number of mails didn't take an unreasonably long time, which is important when developing software, since the program has to be changed and run many times.

Because I've organized the data into independent collections of years, all I need to do is work on the code for one of these years. I know that my program can be trivially changed to run on a more powerful machine if this becomes necessary.

27.4 Adding Keywords to the Postings

If we look at the postings in the Erlang mailing list, you'll see they don't have any keywords. But even if they did, there is a more fundamental problem. Two different people might read the same document and disagree as to what keywords should be used to describe the document. So, if I do a keyword search, it won't work if the keywords that the author of the document chose were different from the keywords I chose to search the document with.

Sherlock computes a keyword vector for each posting in the mailing list. We can ask Sherlock what the derived keyword vector was for posting 946 in 2009.

```
1> sherlock:get_keyword_vector("2009", 946).
[{"protocols",0.6983839995773734},
{"json",0.44660371850799946},
{"ubf",0.38889626945542854},
{"widely",0.2507841072279312},
{"1.html",0.17649029959852883},
{"recast",0.17130091468424488},
...
```

The keyword vector is a list of the derived keywords together with the significance of the word in the document. The significance of the word in a document is actually the TF*IDF[3] weight of the word in the document. This is a number from 0 to 1, where 0 means the word is insignificant and 1 means the word is highly significant.

To compute the similarity between two documents, we compute the keyword vectors for each document, and then we compute the normalized cross product of the keyword vectors. This is called the *cosine similarity*[4] of the documents. If two documents have overlapping keywords, they are similar to a certain extent, and the cosine similarity is a measure of this.

Now we'll look at the details of the algorithm.

---

3. http://en.wikipedia.org/wiki/Tf-idf
4. http://en.wikipedia.org/wiki/Cosine_similarity

The Significance of a Word: The TF*IDF Weight

A commonly used measure of significance for words is the so-called TF*IDF weight. TF stands for *term frequency*, and IDF stands for *inverse document frequency*. Many search engines use the TF*IDF weights of words in a document to rank the significance of words and to find similar documents in a collection. In this section, we'll look at how TF*IDF weights are computed.

In the literature describing search methods, you'll find the word *corpus* used a lot. A corpus is a large set of reference documents. In our case, the corpus is the set of 73,445 mails to the Erlang mailing list.

The first thing we need to do to compute TF*IDF weights is to break the documents we are interested in into a sequence of words. We'll assume that words are sequences of alphabetic characters separated by nonalphabetic characters. Now suppose we find the word *socket* in a particular document; whether or not this word is significant depends upon the context of the word. We'll need to analyze a large number of documents before we can assess the significance of an individual word.

Suppose the word *socket* occurs in 1 percent of all the documents in the corpus. We can compute this by analyzing all the documents in the corpus and counting how many documents contain the word *socket*.

If we look at an individual document, it might also contain the word *socket*. The number of times a document contains a word divided by the total number of words in the document is called the *term frequency* of the word. So, if a document contains the word socket five times and has one hundred words, then the term frequency of the word socket is 5 percent. If the frequency of the word *socket* in the corpus is 1 percent, then the fact that our document uses the word 5 percent of the time is highly significant, so we might like to choose *socket* as a keyword that we could associate with the document. If the term frequency in our document was 1 percent, then it would have the same probability of occurrence as in the parent corpus and thus have little significance.

The *term frequency* (TF) of a word in a document is simply the number of times the word occurs in the document divided by the number of words in the document.

The *inverse document frequency* (IDF) of a word W is defined as $\log(Tot/N+1)$, where Tot is the total number of documents in the corpus and N is the number of documents that contain the word W.

For example, suppose we have a corpus of 1,000 documents. Assume the word *orange* occurs in twenty-five documents. The IDF of the word *orange* is

thus $\log(1000/26)$ (= 1.58). If the word *orange* occurs ten times in a document of a hundred words, then the TF of *orange* is $10/100 = (0.1)$, and the TM*IDF weight of the word is 0.158.

To find a set of keywords for a particular document, we compute the TF*IDF weights of each word in the document and then take words having the highest TF*IDF weights. We omit any words with very low TF*IDF weights.

With these preliminaries taken care of, we can now compute which words in a document are good keywords. This is a two-pass process. The first pass computes the IDF of each word in the corpus. The second pass computes the keywords in each document in the corpus.

Cosine Similarity: The Similarity of Two Weighted Vectors

Given two keyword vectors, we can compute the cosine similarity with a simple example. Assume we have two keyword vectors: K1 with keywords a, b, and c, and K2 with keywords a, b, and d. The keywords and their associated TF*IDF weights are as follows:

```
1> K1 = [{a,0.5},{b,0.1},{c,0.2}].
[{a,0.5},{b,0.1},{c,0.2}]
2> K2 = [{a,0.3},{b,0.2},{d,0.6}].
[{a,0.3},{b,0.2},{d,0.6}]
```

The cross product of K1 and K2 is the sum of the product of weights of entities with identical keywords.

```
3> Cross = 0.5*0.3 + 0.1*0.2.
0.16999999999999998.
```

To compute the cosine similarity, we divide the cross product by the norms of each of the vectors. The norm of a vector is the square root of the sum of the squares of the weights.

```
4> Norm1 = math:sqrt(0.5*0.5 + 0.1*0.1 + 0.2*0.2).
0.5477225575051662
Norm2 = math:sqrt(0.3*0.3 + 0.2*0.2 + 0.6*0.6).
0.7
```

The cosine similarity is the normalized cross product.

```
5> Cross/(Norm1*Norm2).
0.4433944513137058
```

This is baked into a library function.

```
6> sherlock_similar:cosine_similarity(K1, K2).
0.4433944513137058
```

The cosine similarity of two keyword vectors is a number from 0 to 1. 1 means the two vectors are identical. 0 means they have no similarities.

Similarity Queries

We've explained all the preliminaries to making a similarity query. First we compute the IDF for all words in the corpus; then we compute a keyword similarity vector for each document in the corpus. These computations can take a long time, but this doesn't matter since they have to be done only once.

To make a similarity query, we take the query document and compute its keyword vector using the IDF of the corpus. They we compute the cosine similarity of every document in the corpus and choose those values that have the largest cosine similarity coefficient.

We list the results, and users can select the documents that they find the most interesting. They can examine the documents and possibly search for more documents based on the results of the previous analyses.

27.5 Overview of the Implementation

All the code for Sherlock is stored in the code/sherlock directory in the code available from the book's home page.[5] This section provides a high-level overview of the implementation. The main stages in processing are as follows:

Initializing the data store
sherlock_mail:ensure_mail_root/0 is called at the very beginning. This routine ensures that the directory ${HOME}/.sherlock exists. We're going to store all data under a directory called ${HOME}/.sherlock/mails; we'll call this directory MAIL in the following text.

Fetching the mail index
The first step is to fetch the data at http://erlang.org/pipermail/erlang-questions/. This is done using the inets HTTP client, which is part of the Erlang distribution. sherlock_get_mails:get_index/1 fetches the mails, and sherlock_get_mails:parse_index/2 parses the mails. The HTML file retrieved from the server is stored in MAIL/questions.html, and the result of the parsing is in the file MAIL/questions.term.

Fetching raw data
sherlock_mails:fetch_all_mails/0 fetches all the mails. It starts by reading MAIL/questions.term; it fetches all the compressed mail files and stores them in the directory MAIL/cache.

---

5. http://pragprog.com/titles/jaerlang2/source_code

Partitioning the data

sherlock_mails:find_mail_years/0 analyzes the files in the mail cache and returns a list of the years for which mail has been recovered.

Processing the data for a given year

sherlock_mails:process_year(Year) does three things. It parses all the data for the given year, it computes the TF*IDF weights for all the posts in the year, and it adds synthetic keywords to each posting.

The following data files are created:

- MAIL/Year/parsed.bin contains a binary B where binary_to_term(B) contains a list of #post records.

- MAIL/Year/idf.ets is an ETS table storing tuples of the form {Word,Index,Idf}, where Word is a binary (one of the words in a mail), Index is an integer index, and Idf is the IDF weight of the word.

- MAIL/Year/mails.bin contains a binary B where binary_to_term(B) contains a list of #post records. This time the record has been augmented with the synthetic keywords.

- MAIL/Year/mails.list is a listing of the first few entries in mails.bin. This is used while developing the program; it's useful to examine the output occasionally and check that the results are as expected.

The work in this step is done in several different places. sherlock_tfidf.erl computes the TF*IDF weights. sherlock_mails:parse_mails/1 parses the compressed mail files. text_analyzers:standard_analyzer_factory/2 converts a binary into a list of words.

Performing similarity queries

sherlock_mails:find_mails_similar_to_binary/2 does most of the work. It reads MAIL/Year/idf.ets and uses this to compute the keyword vector of the binary (this is done by calling sherlock_tfidf:keywords_in_binary/2). It then iterates over all the entries in MAIL/Year/mails.bin, and each entry contains a keyword vector. sherlock_similar:cosine_similarity/2 computes the similarity of the keyword vectors, and sherlock_best.erl keeps a list of most significant posts.

Performing faceted searches

sherlock_mail:search_mails_regexp/4 iterates over the entries MAIL/Year/mails.bin and performs regular expression searches on the individual elements in the #post records.

27.6 Exercises

Future development of Sherlock is at GitHub.[6] If you have a nice solution to any of the following problems, make a fork of Sherlock, and send me a pull request. There are a lot of ways the Sherlock program can be improved, and as you can see from the list, there a lot more things that could be done.

Finding Similarities Between Modules

Sherlock finds similarities between postings in the Erlang mailing list. Add a facility to compare the similarities of a large collection of Erlang modules.

Finding the History of a Module

As well as defining the similarity between modules, we can define the *distance* between two modules. Once we have found a set of similar modules, try to derive the history of the modules. Have the modules been derived from each other by cutting and pasting code from each other? When you've found a set of similar modules, do they have a common ancestor?

Analyzing Data in Other Mailing Lists

Sherlock can extract data only from pipermail archives. Write code that can extract mails from other commonly used mailing lists or forum programs.

Beautiful Soup

Python has a program called Beautiful Soup[7] for writing screen-scrapers. Implement a version of beautiful soup in Erlang. Use it to collect all mails from some popular mailing lists.

Improving the Text Analyses

Improve the text analyzer. Take a look at the words it produces. Write custom word generators for Erlang that extract the text from different file types.

Faceted Search

Add additional facets to the data and improve the faceted search.

Make a Web Interface

Make a web interface to Sherlock.

---

6. https://github.com/joearms/sherlock
7. http://www.crummy.com/software/BeautifulSoup/

Grade the Mails

Try to grade the mails. The score of a mail could depend upon who wrote the mail and who commented on it and how many followups it generated.

Graph the Results

Make graphs of the similarity relationships computed from a specific query. Show this in a graphic display in the browser.

Turn the Collection of Mails into an Ebook

Generate in the *epub* format.

27.7 Wrapping Up

Thank you for reading this book. I hope you've enjoyed it. It's been a long journey; you've learned a lot of new things. The Erlang view of the world is very different from other programming languages. The biggest differences are in how we handle errors and in how we deal with concurrency.

Concurrency is *natural* in Erlang. The real world actually has independent things communicating through messages. I'm an ex-physicist—we perceive the world by receiving messages. Packets of light and sound carry information. All that we know about the world is what we've learned by receiving messages.

We don't have shared memory. I have my memory, you have yours, and I don't know what you think about anything. If I want to know what you think about something, then I have to ask you a question and wait for you to reply.

The model of programming that Erlang uses is very similar to how the world works. This makes programming easy. Many programmers have discovered this, as have many companies.

So, what now? Spread the word. Join the Erlang mailing list. Go to one of the many Erlang conferences that are organized around the world. Solve some interesting programs and try to make the world a better place.

OTP Templates

This appendix contains a full listing of the gen_server and the supervisor and application templates. These templates are built into Emacs mode.

A1.1 The Generic Server Template

gen_server_template.full

```
%%%-------------------------------------------------------------------
%%% @author some name <me@hostname.local>
%%% @copyright (C) 2013, some name
%%% @doc
%%%
%%% @end
%%% Created : 26 May 2013 by some name <me@hostname.local>
%%%-------------------------------------------------------------------
-module().

-behaviour(gen_server).

%% API
-export([start_link/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

%%%===================================================================
%%% API
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc
```

```erlang
%% Starts the server
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).


%%%===================================================================
%%% gen_server callbacks
%%%===================================================================


%%--------------------------------------------------------------------
%% @private
%% @doc
%% Initializes the server
%%
%% @spec init(Args) -> {ok, State} |
%%                     {ok, State, Timeout} |
%%                     ignore |
%%                     {stop, Reason}
%% @end
%%--------------------------------------------------------------------
init([]) ->
    {ok, #state{}}.


%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling call messages
%%
%% @spec handle_call(Request, From, State) ->
%%                                   {reply, Reply, State} |
%%                                   {reply, Reply, State, Timeout} |
%%                                   {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, Reply, State} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.


%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling cast messages
%%
%% @spec handle_cast(Msg, State) -> {noreply, State} |
```

```erlang
%%                                      {noreply, State, Timeout} |
%%                                      {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
    {noreply, State}.


%%--------------------------------------------------------------------
%% @private
%% @doc
%% Handling all non call/cast messages
%%
%% @spec handle_info(Info, State) -> {noreply, State} |
%%                                   {noreply, State, Timeout} |
%%                                   {stop, Reason, State}
%% @end
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
    {noreply, State}.


%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any
%% necessary cleaning up. When it returns, the gen_server terminates
%% with Reason. The return value is ignored.
%%
%% @spec terminate(Reason, State) -> void()
%% @end
%%--------------------------------------------------------------------
terminate(_Reason, _State) ->
    ok.


%%--------------------------------------------------------------------
%% @private
%% @doc
%% Convert process state when code is changed
%%
%% @spec code_change(OldVsn, State, Extra) -> {ok, NewState}
%% @end
%%--------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.


%%%===================================================================
%%% Internal functions
%%%===================================================================
```

A1.2 The Supervisor Template

```
supervisor_template.full
%%%-------------------------------------------------------------------
%%% @author some name   <me@hostname.local>
%%% @copyright (C) 2013, some name
%%% @doc
%%%
%%% @end
%%% Created : 26 May 2013 by some name <me@hostname.local>
%%%-------------------------------------------------------------------
-module().

-behaviour(supervisor).

%% API
-export([start_link/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).

%%%===================================================================
%%% API functions
%%%===================================================================

%%--------------------------------------------------------------------
%% @doc
%% Starts the supervisor
%%
%% @spec start_link() -> {ok, Pid} | ignore | {error, Error}
%% @end
%%--------------------------------------------------------------------
start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

%%%===================================================================
%%% Supervisor callbacks
%%%===================================================================

%%--------------------------------------------------------------------
%% @private
%% @doc
%% Whenever a supervisor is started using supervisor:start_link/[2,3],
%% this function is called by the new process to find out about
%% restart strategy, maximum restart frequency and child
%% specifications.
%%
%% @spec init(Args) -> {ok, {SupFlags, [ChildSpec]}} |
```

```
%%                      ignore |
%%                      {error, Reason}
%% @end
%%-----------------------------------------------------------------
init([]) ->
    RestartStrategy = one_for_one,
    MaxRestarts = 1000,
    MaxSecondsBetweenRestarts = 3600,

    SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},

    Restart = permanent,
    Shutdown = 2000,
    Type = worker,

    AChild = {'AName', {'AModule', start_link, []},
              Restart, Shutdown, Type, ['AModule']},

    {ok, {SupFlags, [AChild]}}.


%%%=================================================================
%%% Internal functions
%%%=================================================================
```

A1.3 The Application Template

```
application_template.full
%%%-----------------------------------------------------------------
%%% @author some name <me@hostname.local>
%%% @copyright (C) 2013, some name
%%% @doc
%%%
%%% @end
%%% Created : 26 May 2013 by some name <me@hostname.local>
%%%-----------------------------------------------------------------
-module().

-behaviour(application).

%% Application callbacks
-export([start/2, stop/1]).


%%%=================================================================
%%% Application callbacks
%%%=================================================================


%%-----------------------------------------------------------------
%% @private
%% @doc
%% This function is called whenever an application is started using
```

```erlang
%% application:start/[1,2], and should start the processes of the
%% application. If the application is structured according to the OTP
%% design principles as a supervision tree, this means starting the
%% top supervisor of the tree.
%%
%% @spec start(StartType, StartArgs) -> {ok, Pid} |
%%                                      {ok, Pid, State} |
%%                                      {error, Reason}
%%      StartType = normal | {takeover, Node} | {failover, Node}
%%      StartArgs = term()
%% @end
%%--------------------------------------------------------------------
start(_StartType, _StartArgs) ->
    case 'TopSupervisor':start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.


%%--------------------------------------------------------------------
%% @private
%% @doc
%% This function is called whenever an application has stopped. It
%% is intended to be the opposite of Module:start/2 and should do
%% any necessary cleaning up. The return value is ignored.
%%
%% @spec stop(State) -> void()
%% @end
%%--------------------------------------------------------------------
stop(_State) ->
    ok.


%%%===================================================================
%%% Internal functions
%%%===================================================================
```

A Socket Application

This appendix is devoted to the implementation of the library lib_chan, which was introduced in *Controlling Processes with lib_chan,* on page 224. The code for lib_chan implements an entire layer of networking on top of TCP/IP, providing both authentication and streams of Erlang terms. Once we understand the principles used in lib_chan, we should be able to tailor-make our own communication infrastructure and layer it on top of TCP/IP.

In its own right, lib_chan is a useful component for building distributed systems.

To make the appendix self-contained, there is a considerable amount of overlap with the material in *Controlling Processes with lib_chan,* on page 224.

The code in this appendix is some of the most complex code I've introduced so far, so don't worry if you don't understand it all on the first reading. If you just want to use lib_chan and don't care about how it works, read the first section and skip the rest.

A2.1 An Example

We'll start with a simple example that shows how to use lib_chan. We're going to create a simple server that can compute factorials and Fibonacci numbers. We'll protect it with a password.

The server will operate on port 2233.

We'll take four steps to create this server.

1. Write a configuration file.
2. Write the code for the server.
3. Start the server.
4. Access the server over the network.

Step 1: Write a Configuration File

Here's the configuration file for our example:

```
socket_dist/config1
{port, 2233}.
{service, math, password, "qwerty", mfa, mod_math, run, []}.
```

The configuration file has a number of service tuples of this form:

```
{service, <Name>, password, <P>, mfa, <Mod>, <Func>, <ArgList>}
```

The arguments are delimited by the atoms service, password, and mfa. mfa is short for *module, function, args*, meaning that the next three arguments are to be interpreted as a module name, a function name, and a list of arguments to some function call.

In our example, the configuration file specifies a service called *math* that will be available on port 2233. The service is protected by the password *qwerty*. It is implemented in a module called mod_math and will be started by calling mod_math:run/3. The third argument of run/3 will be [].

Step 2: Write the Code for the Server

The math server code looks like this:

```
socket_dist/mod_math.erl
-module(mod_math).
-export([run/3]).
run(MM, ArgC, ArgS) ->
    io:format("mod_math:run starting~n"
              "ArgC = ~p ArgS=~p~n",[ArgC, ArgS]),
    loop(MM).
loop(MM) ->
    receive
        {chan, MM, {factorial, N}} ->
            MM ! {send, fac(N)},
            loop(MM);
        {chan, MM, {fibonacci, N}} ->
            MM ! {send, fib(N)},
            loop(MM);
        {chan_closed, MM} ->
            io:format("mod_math stopping~n"),
            exit(normal)
    end.
fac(0) -> 1;
fac(N) -> N*fac(N-1).
fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

When a client connects to port 2233 and requests the service called *math*, lib_auth will authenticate the client and, if the password is correct, spawn a handler process by spawning the function mod_math:run(MM, ArgC, ArgS). MM is the PID of a *middle man*. ArgC comes from the client, and ArgS comes from the configuration file.

When the client sends a message X to the server, it will arrive as a {chan, MM, X} message. If the client dies or something goes wrong with the connection, the server will be sent a {chan_closed, MM} message. To send a message Y to the client, the server evaluated MM ! {send, Y}, and to close the communication channel, it evaluated MM ! close.

The math server is simple; it just waits for a {chan, MM, {factorial, N}} message and then sends the result to the client by evaluating MM ! {send, fac(N)}.

Step 3: Starting the Server

We start the server as follows:

```
1> lib_chan:start_server("./config1").
ConfigData=[{port,2233},{service,math,password,"qwerty",mfa,mod_math,run,[]}]
true
```

Step 4: Accessing the Server Over the Network

We can test this code on a single machine.

```
2> {ok, S} = lib_chan:connect("localhost",2233,math,
                              "qwerty",{yes,go}).

{ok,<0.47.0>}
3> lib_chan:rpc(S, {factorial,20}).
2432902008176640000
4> lib_chan:rpc(S, {fibonacci,15}).
610
4> lib_chan:disconnect(S).
close
```

A2.2 How lib_chan Works

lib_chan is built using code in four modules.

- lib_chan acts as a "main module." The only routines that the programmer needs to know about are the routines that are exported from lib_chan. The other three modules (discussed next) are used internally in the implementation of lib_chan.

- lib_chan_mm encodes and decodes Erlang messages and manages the socket communication.

- lib_chan_cs sets up the server and manages client connections. One of its primary jobs is to limit the maximum number of simultaneous client connections.

- lib_chan_auth contains code for simple challenge/response authentication.

lib_chan

lib_chan has the following structure:

```
-module(lib_chan).

start_server(ConfigFile) ->
    %% read configuration file - check syntax
    %% call start_port_server(Port, ConfigData)
    %% where Port is the required Port and ConfigData
    %% contains the configuration data

start_port_server(Port, ConfigData) ->
    lib_chan_cs:start_raw_server( ..
          fun(Socket) ->
              start_port_instance(Socket, ConfigData),
          end, ... )
    %% lib_chan_cs manages the connection
    %% when a new connection comes the fun which is an
    %% argument to start_raw_server will be called
start_port_instance(Socket, ConfigData) ->
    %% this is spawned when the client connects
    %% to the server. Here we setup a middle man,
    %% then perform authentication. If everything works call
    %% really_start(MM, ArgC, {Mod, Func, ArgS})
    %% (the last three arguments come from the configuration file

really_start(MM, ArgC, {Mod, Func, ArgS}) ->
    apply(Mod, Func, [MM, ArgC, ArgS]).

connect(Host, Port, Service, Password, ArgC) ->
    %% client side code
```

lib_chan_mm: The Middle Man

lib_chan_mm implements a middle man. It hides the socket communication from the applications, turning streams of data on TCP sockets into Erlang messages. The middle man is responsible for assembling the message (which might have become fragmented) and for encoding and decoding Erlang terms into streams of bytes that can be sent to and received from a socket.
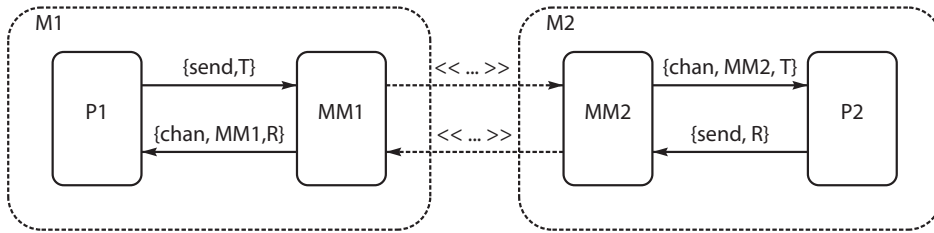
Figure 8—Socket communication with a middle man

Now is a good time to take a quick look at Figure 8, *Socket communication with a middle man,* on page 481, which shows our middle-man architecture. When a process P1 on a machine M1 wants to send a message T to a process P2 on a machine M2, it evaluates MM1 ! {send, T}. MM1 acts as a *proxy* for P2. Anything sent to MM1 is encoded and written to a socket and sent to MM2. MM2 decodes anything it receives on a socket and sends the message {chan, MM2, T} to P2.

On the machine M1, the process MM1 behaves as a proxy for P2, and on M2 the process MM2 behaves as a proxy for P1.

MM1 and MM2 are PIDs of middle-men processes. The middle-man process code looks something like this:

```
loop(Socket, Pid) ->
    receive
        {tcp, Socket, Bin} ->
            Pid ! {chan, self(), binary_to_term(Bin)},
            loop(Socket, Pid);
        {tcp_closed, Socket} ->
            Pid ! {chan_closed, self()};
        close ->
            gen_tcp:close(Socket);
        {send, T} ->
            gen_tcp:send(Socket, [term_to_binary(T)]),
            loop(Socket, Pid)
    end.
```

This loop is used as an interface between the world of socket data and the world of Erlang message passing. You can find the complete code for lib_chan_mm in *lib_chan_mm,* on page 490. This is slightly more complex than the code shown here, but the principle is the same. The only difference is that we've added some code for tracing messages and some interfacing routines.

lib_chan_cs

lib_chan_cs is responsible for setting up client and server communication. The two important routines that it exports are as follows:

start_raw_server(Port, Max, Fun, PacketLength)

> This starts a listener that listens for a connection on Port. At most, Max simultaneous sessions are allowed. Fun is a fun of arity 1; when a connection starts, Fun(Socket) is evaluated. The socket communication assumes a packet length of PacketLength.

start:raw_client(Host, Port, PacketLength) => {ok, Socket} | {error, Why}

> This tries to connect to a port opened with start_raw_server.

The code for lib_chan_cs follows the pattern described in *A Parallel Server*, on page 271, but in addition it keeps track of the maximum number of simultaneously open connections. This small detail, though conceptually simple, adds twenty-odd lines of rather strange-looking code that traps exits and so on. Code like this is a mess, but don't worry; it does its job and hides the complexity from the user of the module.

lib_chan_auth

This module implements a simple form of challenge/response authentication. Challenge/response authentication is based on the idea of a shared secret that is associated with the service name. To show how it works, we'll assume there is a service called math that has the shared secret qwerty.

If a client wants to use the service math, then the client has to prove to the server that they know the shared secret. This works as follows:

1. The client sends a request to the server saying it wants to use the math service.

2. The server computes a random string C and sends it to the client. This is the *challenge*. The string is computed by the function lib_chan_auth:make_challenge(). We can use it interactively to see what it does.

   ```
   1> C = lib_chan_auth:make_challenge().
   "qnyrgzqefvnjdombanrsmxikc"
   ```

3. The client receives this string (C) and computes a response (R) where R = MD5(C ++ Secret). R is computed using lib_chan_auth:make_response. Here's an example:

   ```
   2> R = lib_chan_auth:make_response(C, "qwerty").
   "e759ef3778228beae988d91a67253873"
   ```

4. The response is sent back to the server. The server receives the response and checks whether it is correct by computing the expected value of the response. This is done in lib_chan_auth:is_response_correct.

```
3> lib_chan_auth:is_response_correct(C, R, "qwerty").
true
```

A2.3 The lib_chan Code

Now it's time for the code.

lib_chan

socket_dist/lib_chan.erl
```erlang
-module(lib_chan).
-export([cast/2, start_server/0, start_server/1,
         connect/5, disconnect/1, rpc/2]).
-import(lists, [map/2, member/2, foreach/2]).
-import(lib_chan_mm, [send/2, close/1]).

%%----------------------------------------------------------------------
%% Server code

start_server() ->
    case os:getenv("HOME") of
        false ->
            exit({ebadEnv, "HOME"});
        Home ->
            start_server(Home ++ "/.erlang_config/lib_chan.conf")
    end.

start_server(ConfigFile) ->
    io:format("lib_chan starting:~p~n",[ConfigFile]),
    case file:consult(ConfigFile) of
        {ok, ConfigData} ->
            io:format("ConfigData=~p~n",[ConfigData]),
            case check_terms(ConfigData) of
                [] ->
                    start_server1(ConfigData);
                Errors ->
                    exit({eDaemonConfig, Errors})
            end;
        {error, Why} ->
            exit({eDaemonConfig, Why})
    end.

%% check_terms() -> [Error]
check_terms(ConfigData) ->
    L = map(fun check_term/1, ConfigData),
    [X || {error, X} <- L].
```

```erlang
check_term({port, P}) when is_integer(P)     -> ok;
check_term({service,_,password,_,mfa,_,_,_}) -> ok;
check_term(X) -> {error, {badTerm, X}}.

start_server1(ConfigData) ->
    register(lib_chan, spawn(fun() -> start_server2(ConfigData) end)).

start_server2(ConfigData) ->
    [Port] = [ P || {port,P} <- ConfigData],
    start_port_server(Port, ConfigData).

start_port_server(Port, ConfigData) ->
    lib_chan_cs:start_raw_server(Port,
                                 fun(Socket) ->
                                         start_port_instance(Socket,
                                                             ConfigData) end,
                                 100,
                                 4).

start_port_instance(Socket, ConfigData) ->
    %% This is where the low-level connection is handled
    %% We must become a middle man
    %% But first we spawn a connection handler

    S = self(),
    Controller = spawn_link(fun() -> start_erl_port_server(S, ConfigData) end),
    lib_chan_mm:loop(Socket, Controller).

start_erl_port_server(MM, ConfigData) ->
    receive
        {chan, MM, {startService, Mod, ArgC}} ->
            case get_service_definition(Mod, ConfigData) of
                {yes, Pwd, MFA} ->
                    case Pwd of
                        none ->
                            send(MM, ack),
                            really_start(MM, ArgC, MFA);
                        _ ->
                            do_authentication(Pwd, MM, ArgC, MFA)
                    end;
                no ->
                    io:format("sending bad service~n"),
                    send(MM, badService),
                    close(MM)
            end;
        Any ->
            io:format("*** ErL port server got:~p ~p~n",[MM, Any]),
            exit({protocolViolation, Any})
    end.
```

```erlang
do_authentication(Pwd, MM, ArgC, MFA) ->
    C = lib_chan_auth:make_challenge(),
    send(MM, {challenge, C}),
    receive
        {chan, MM, {response, R}} ->
            case lib_chan_auth:is_response_correct(C, R, Pwd) of
                true ->
                    send(MM, ack),
                    really_start(MM, ArgC, MFA);
                false ->
                    send(MM, authFail),
                    close(MM)
            end
    end.


%% MM is the middle man
%% Mod is the Module we want to execute ArgC and ArgS come from the client and
%% server respectively

really_start(MM, ArgC, {Mod, Func, ArgS}) ->
    %% authentication worked so now we're off
    case (catch apply(Mod,Func,[MM,ArgC,ArgS])) of
        {'EXIT', normal} ->
            true;
        {'EXIT', Why} ->
            io:format("server error:~p~n",[Why]);
        Why ->
            io:format("server error should die with exit(normal) was:~p~n",
                      [Why])
    end.

%% get_service_definition(Name, ConfigData)

get_service_definition(Mod, [{service, Mod, password, Pwd, mfa, M, F, A}|_]) ->
    {yes, Pwd, {M, F, A}};
get_service_definition(Name, [_|T]) ->
    get_service_definition(Name, T);
get_service_definition(_, []) ->
    no.

%%-----------------------------------------------------------------------
%% Client connection code
%% connect(...) -> {ok, MM} | Error

connect(Host, Port, Service, Secret, ArgC) ->
    S = self(),
    MM = spawn(fun() -> connect(S, Host, Port) end),
    receive
        {MM, ok} ->
```

```erlang
            case authenticate(MM, Service, Secret, ArgC) of
                ok    -> {ok, MM};
                Error -> Error
            end;
        {MM, Error} ->
            Error
    end.

connect(Parent, Host, Port) ->
    case lib_chan_cs:start_raw_client(Host, Port, 4) of
        {ok, Socket} ->
            Parent ! {self(), ok},
            lib_chan_mm:loop(Socket, Parent);
        Error ->
            Parent ! {self(),  Error}
    end.

authenticate(MM, Service, Secret, ArgC) ->
    send(MM, {startService, Service, ArgC}),
    %% we should get back a challenge or a ack or closed socket
    receive
        {chan, MM, ack} ->
            ok;
        {chan, MM, {challenge, C}} ->
            R = lib_chan_auth:make_response(C, Secret),
            send(MM, {response, R}),
            receive
                {chan, MM, ack} ->
                    ok;
                {chan, MM, authFail} ->
                    wait_close(MM),
                    {error, authFail};
                Other ->
                    {error, Other}
            end;
        {chan, MM, badService} ->
            wait_close(MM),
            {error, badService};
        Other ->
            {error, Other}
    end.

wait_close(MM) ->
    receive
        {chan_closed, MM} ->
            true
    after 5000 ->
            io:format("**error lib_chan~n"),
            true
    end.
```

```erlang
disconnect(MM) -> close(MM).

rpc(MM, Q) ->
    send(MM, Q),
    receive
        {chan, MM, Reply} ->
            Reply
    end.

cast(MM, Q) ->
    send(MM, Q).
```

lib_chan_cs

```erlang
socket_dist/lib_chan_cs.erl
-module(lib_chan_cs).
%% cs stands for client_server
-export([start_raw_server/4, start_raw_client/3]).
-export([stop/1]).
-export([children/1]).

%% start_raw_server(Port, Fun, Max, PacketLength)
%%   This server accepts up to Max connections on Port
%%   The *first* time a connection is made to Port
%%   Then Fun(Socket) is called.
%%   Thereafter messages to the socket result in messages to the handler.
%%   PacketLength is usually 0,1,2 or 4 (see the inet manual page for details).

%% tcp_is typically used as follows:
%% To setup a listener
%%   start_agent(Port) ->
%%     process_flag(trap_exit, true),
%%     lib_chan_server:start_raw_server(Port,
%%                                      fun(Socket) -> input_handler(Socket) end,
%%                                      15, 0).

start_raw_client(Host, Port, PacketLength) ->
    gen_tcp:connect(Host, Port,
                    [binary, {active, true}, {packet, PacketLength}]).

%% Note when start_raw_server returns it should be ready to
%% Immediately accept connections

start_raw_server(Port, Fun, Max, PacketLength) ->
    Name = port_name(Port),
    case whereis(Name) of
        undefined ->
            Self = self(),
            Pid = spawn_link(fun() ->
                                  cold_start(Self,Port,Fun,Max,PacketLength)
                             end),
```

```erlang
            receive
                {Pid, ok} ->
                    register(Name, Pid),
                    {ok, self()};
                {Pid, Error} ->
                    Error
            end;
        _Pid ->
            {error, already_started}
    end.

stop(Port) when integer(Port) ->
    Name = port_name(Port),
    case whereis(Name) of
        undefined ->
            not_started;
        Pid ->
            exit(Pid, kill),
            (catch unregister(Name)),
            stopped
    end.
children(Port) when integer(Port) ->
    port_name(Port) ! {children, self()},
    receive
        {session_server, Reply} -> Reply
    end.


port_name(Port) when integer(Port) ->
    list_to_atom("portServer" ++ integer_to_list(Port)).


cold_start(Master, Port, Fun, Max, PacketLength) ->
    process_flag(trap_exit, true),
    %% io:format("Starting a port server on ~p...~n",[Port]),
    case gen_tcp:listen(Port, [binary,
                               %% {dontroute, true},
                               {nodelay,true},
                               {packet, PacketLength},
                               {reuseaddr, true},
                               {active, true}]) of
        {ok, Listen} ->
            %% io:format("Listening to:~p~n",[Listen]),
            Master ! {self(), ok},
            New = start_accept(Listen, Fun),
            %% Now we're ready to run
            socket_loop(Listen, New, [], Fun, Max);
        Error ->
            Master ! {self(), Error}
    end.
```

```erlang
socket_loop(Listen, New, Active, Fun, Max) ->
    receive
        {istarted, New} ->
            Active1 = [New|Active],
            possibly_start_another(false,Listen,Active1,Fun,Max);
        {'EXIT', New, _Why} ->
            %% io:format("Child exit=~p~n",[Why]),
            possibly_start_another(false,Listen,Active,Fun,Max);
        {'EXIT', Pid, _Why} ->
            %% io:format("Child exit=~p~n",[Why]),
            Active1 = lists:delete(Pid, Active),
            possibly_start_another(New,Listen,Active1,Fun,Max);
        {children, From} ->
            From ! {session_server, Active},
            socket_loop(Listen,New,Active,Fun,Max);
        _Other ->
            socket_loop(Listen,New,Active,Fun,Max)
    end.

possibly_start_another(New, Listen, Active, Fun, Max)
  when pid(New) ->
    socket_loop(Listen, New, Active, Fun, Max);
possibly_start_another(false, Listen, Active, Fun, Max) ->
    case length(Active) of
        N when N < Max ->
            New = start_accept(Listen, Fun),
            socket_loop(Listen, New, Active, Fun,Max);
        _ ->
            socket_loop(Listen, false, Active, Fun, Max)
    end.

start_accept(Listen, Fun) ->
    S = self(),
    spawn_link(fun() -> start_child(S, Listen, Fun) end).

start_child(Parent, Listen, Fun) ->
    case gen_tcp:accept(Listen) of
        {ok, Socket} ->
            Parent ! {istarted,self()},               % tell the controller
            inet:setopts(Socket, [{packet,4},
                                   binary,
                                   {nodelay,true},
                                   {active, true}]),
            %% before we activate socket
            %% io:format("running the child:~p Fun=~p~n", [Socket, Fun]),
            process_flag(trap_exit, true),
            case (catch Fun(Socket)) of
                {'EXIT', normal} ->
                    true;
                {'EXIT', Why} ->
```

```
                            io:format("Port process dies with exit:~p~n",[Why]),
                            true;
                     _ ->
                            %% not an exit so everything's ok
                            true
                end
        end.
```

lib_chan_mm

```
%% Protocol
%%    To the controlling process
%%        {chan, MM, Term}
%%        {chan_closed, MM}
%%    From any process
%%        {send, Term}
%%        close

-module(lib_chan_mm).
%% TCP Middle man
%%    Models the interface to gen_tcp

-export([loop/2, send/2, close/1, controller/2, set_trace/2, trace_with_tag/2]).

send(Pid, Term)        -> Pid ! {send, Term}.
close(Pid)             -> Pid ! close.
controller(Pid, Pid1) -> Pid ! {setController, Pid1}.
set_trace(Pid, X)      -> Pid ! {trace, X}.

trace_with_tag(Pid, Tag) ->
    set_trace(Pid, {true,
                    fun(Msg) ->
                            io:format("MM:~p ~p~n",[Tag, Msg])
                    end}).

loop(Socket, Pid) ->
    %% trace_with_tag(self(), trace),
    process_flag(trap_exit, true),
    loop1(Socket, Pid, false).

loop1(Socket, Pid, Trace) ->
    receive
        {tcp, Socket, Bin} ->
            Term = binary_to_term(Bin),
            trace_it(Trace,{socketReceived, Term}),
            Pid ! {chan, self(), Term},
            loop1(Socket, Pid, Trace);
        {tcp_closed, Socket} ->
            trace_it(Trace, socketClosed),
            Pid ! {chan_closed, self()};
```

```erlang
        {'EXIT', Pid, Why} ->
            trace_it(Trace,{controllingProcessExit, Why}),
            gen_tcp:close(Socket);
        {setController, Pid1} ->
            trace_it(Trace, {changedController, Pid}),
            loop1(Socket, Pid1, Trace);
        {trace, Trace1} ->
            trace_it(Trace, {setTrace, Trace1}),
            loop1(Socket, Pid, Trace1);
        close ->
            trace_it(Trace, closedByClient),
            gen_tcp:close(Socket);
        {send, Term}  ->
            trace_it(Trace, {sendingMessage, Term}),
            gen_tcp:send(Socket, term_to_binary(Term)),
            loop1(Socket, Pid, Trace);
        UUg ->
            io:format("lib_chan_mm: protocol error:~p~n",[UUg]),
            loop1(Socket, Pid, Trace)
    end.
trace_it(false, _)     -> void;
trace_it({true, F}, M) -> F(M).
```

lib_chan_auth

```erlang
socket_dist/lib_chan_auth.erl
-module(lib_chan_auth).
-export([make_challenge/0, make_response/2, is_response_correct/3]).

make_challenge() ->
    random_string(25).
make_response(Challenge, Secret) ->
    lib_md5:string(Challenge ++ Secret).
is_response_correct(Challenge, Response, Secret) ->
    case lib_md5:string(Challenge ++ Secret) of
        Response -> true;
        _        -> false
    end.

%% random_string(N) -> a random string with N characters.
random_string(N) -> random_seed(), random_string(N, []).
random_string(0, D) -> D;
random_string(N, D) ->
    random_string(N-1, [random:uniform(26)-1+$a|D]).
random_seed() ->
    {_,_,X} = erlang:now(),
    {H,M,S} = time(),
    H1 = H * X rem 32767,
    M1 = M * X rem 32767,
    S1 = S * X rem 32767,
    put(random_seed, {H1,M1,S1}).
```

A Simple Execution Environment

In this appendix we'll build a simple execution environment (SEE) for running Erlang programs. The code is in an appendix rather than the main body of the book since it is rather different in character. All the code in the book is intended to be run in the standard Erlang/OTP distribution, whereas this code deliberately makes minimal use of Erlang library code and tries to use only the Erlang primitives.

When confronted with Erlang for the first time, it's often unclear what belongs to the language and what belongs to the operating environment. OTP provides a rich environment, akin to an operating system for running long-lived distributed Erlang applications. But exactly what functionality is provided by Erlang (the language) and OTP (the environment) is unclear.

SEE provides an environment that is "nearer to the metal," giving a much clearer distinction between what is provided by Erlang and what is provided by OTP. All that is provided by SEE is contained in a single module. OTP starts by loading 60-odd modules, and it's not immediately apparent how things work, but none of this is particularly complicated if you know where to look. The place to start is in the boot file. Start by looking at the boot file and then by reading the code in init.erl, and all will be revealed.

The SEE environment can be used for scripting, since it starts very quickly, or for embedding, since it is very small. To achieve this, you'll learn how Erlang starts and how the code autoloading system works.

When you start a "standard" Erlang system (with the shell command erl), sixty-seven modules get loaded and twenty-five processes are started, and only then your program gets to run. This takes about a second. If the program we want to execute doesn't need all the goodies provided by the standard system, we can cut this time down to a few tens of milliseconds.

Figuring out exactly what all these sixty-seven modules and processes do can be rather daunting. But there is a different and shorter path to enlightenment. SEE simplifies the system to the point where only one module needs to be studied in order to understand how code gets loaded into the system and how I/O services can be provided. SEE provides autoloading, generic server processes, and error handling all in one module.

Before we start on SEE, we'll gather some statistics on the OTP system that we'll use for reference later.

```
$ erl
1> length([I||{I,X} <- code:all_loaded(), X =/= preloaded]).
67
2> length(processes()).
25
3> length(registered()).
16
```

code:all_loaded() returns a list of all the modules that are currently loaded into the system. processes() is a BIF that returns a list of all processes known to the system, and registered() returns a list of all registered processes.

So, just starting the system loads sixty-seven modules and starts twenty-five processes, of which sixteen are registered processes. Let's see if we can reduce this number.

A3.1 How Erlang Starts

When Erlang starts, it reads a boot file and executes the commands that it finds in the boot file. Eight Erlang modules are preloaded. These are Erlang modules that have been compiled into C and linked into the Erlang virtual machine. These eight modules are responsible for booting the system. They include init.erl, which reads and evaluates commands in the boot file, and erl_prim_loader, which knows how to load code into the system.

The boot file contains a binary that was created by calling term_to_binary(Script), where Script is a tuple containing a boot script.

We're going to make a new boot file called see.boot and a script called see, which launches a program using the new boot file. The boot file will load a small number of modules, including see.erl, which contains our custom execution environment. The boot file and script are created by evaluating make_scripts/0.

see/see.erl
```
Line 1  make_scripts() ->
   -        {ok, Cwd} = file:get_cwd(),
   -        Script =
```

```
   -              {script,{"see","1.0"},
   5            [{preLoaded,preloaded()},
   -             {progress,preloaded},
   -             {path, [Cwd]},
   -             {primLoad,
   -              [lists,
  10             error_handler,
   -              see
   -             ]},
   -             {kernel_load_completed},
   -             {progress,kernel_load_completed},
  15           {progress,started},
   -             {apply,{see,main,[]}}
   -            ]},
   -       io:format("Script:~p~n",[Script]),
   -       file:write_file("see.boot", term_to_binary(Script)),
  20     file:write_file("see",[
   -                          "#!/bin/sh\nerl ",
   -                          %%" -init_debug ",
   -                          " -boot ", Cwd, "/see ",
   -                          "-environment `printenv` -load $1\n"]),
  25     os:cmd("chmod a+x see"),
   -       init:stop(),
   -       true.
```

Line 4 contains a string to identify the script. This is followed by a list of commands that are evaluated by init.erl. The first command is in line 5 where the tuple {preloaded,[Mods]} tells the system which modules have been preloaded. Next is a {progress, Atom} command. progress tuples are included for debugging purposes; they are printed if the -init_debug flag is included on the command line when Erlang is started.

The tuple {path, [Dirs]} (line 7) sets up a code loader path, and {primLoad, [Mods]} means load the modules in the module list. So, lines 8 to 12 tell the system to load three modules (lists, error_handler, and see) using the given code paths.

When we've loaded all the code, we get to the command {kernel_load_completed}. This means "we're open for business," and we hand over control to user code. After the kernel load complete command (but not before), we can evaluate user functions. This is done by calling apply. Finally, in line 16, we say {apply,{see,main,[]}}, which calls apply(see,main,[]).

make_scripts/0 must be evaluated from within the Erlang development environment since it calls functions in the modules code, filename, and file, which are not available to SEE programs.

Now we'll build the boot file and the start script.

```
$ erlc see.erl
$ erl -s see make_scripts
Eshell V5.9.3  (abort with ^G)
Script:{script,{"see","1.0"},
        [{preLoaded,[zlib,prim_file,prim_zip,prim_inet,erlang,
                     otp_ring0,init,erl_prim_loader]},
         {progress,preloaded},
         {path,["/Users/joe/projects_active/book/jaerlang2/Book/code/see"]},
         {primLoad,[lists,error_handler,see]},
         {kernel_load_completed},
         {progress,kernel_load_completed},
         {progress,started},
         {apply,{see,main,[]}}]}
```

A3.2 Running Some Test Programs in SEE

Once we've built the boot script, we can turn our attention to the programs that we'll run inside SEE. All our examples are normal Erlang modules that must export the function main().

The simplest of all programs is see_test1.

```
see/see_test1.erl
-module(see_test1).
-export([main/0]).
main() ->
    see:write("HELLO WORLD\n"),
    see:write(integer_to_list(see:modules_loaded()-8) ++ " modules loaded\n").
```

To run this, we first compile see_test1 using the standard compiler in the Erlang development environment. Once we have done this, then we can run our program.

```
$ erlc see_test1.erl
$ ./see see_test1
Hello world
```

We can time this as follows:

```
$ time ./see see_test1
HELLO WORLD

4 modules loaded
real    0m0.019s
user    0m0.016s
sys     0m0.000s
```

So, it took 0.019 seconds to load four modules (lists, error_handler, see, and see_test1) and run the program.

For comparison, we can time the equivalent Erlang program using the OTP system.

```
see/otp_test1.erl
-module(otp_test1).
-export([main/0]).

main() ->
    io:format("HELLO WORLD\n").
```

```
$ erlc otp_test1.erl
$ time erl -noshell -s otp_test1 main -s init stop
HELLO WORLD

real    0m1.127s
user    0m0.100s
sys     0m0.024s
```

SEE was fifty-nine times faster in starting and running our little program than OTP. It should be remembered that fast-starting OTP was never a design goal. Once started, OTP applications are expected to run *forever*, so shaving off a few milliseconds in the start time is irrelevant if the application subsequently runs forever.

Here are a few more simple programs. see_test2 tests that autoloading works.

```
see/see_test2.erl
-module(see_test2).
-export([main/0]).

main() ->
    erlang:display({about_to_call,my_code}),
    2000 = my_code:double(1000),
    see:write("see_test2 worked\n").
```

Where:

```
see/my_code.erl
-module(my_code).
-export([double/1]).

double(X) ->
    2*X.
```

Thus:

```
$ ./see see_test2
{about_to_call,my_code}
{new_error_handler,undefined_function,my_code,double,[1000]}
{error_handler,calling,my_code,double,[1000]}
see_test2 worked
```

We can see that the my_code module was correctly autoloaded. The other output is debug printout that comes from a custom code loader. This is produced by the module error_handler and is discussed later in this chapter.

see_test3 is an Erlang program that copies everything it sees on standard input to standard output. (This is how to write a Unix pipe process.)

```erlang
see/see_test3.erl
-module(see_test3).
-export([main/0]).
-import(see, [read/0, write/1]).

main() -> loop().

loop() ->
    case read() of
        eof ->
            true;
        {ok, X} ->
            write([X]),
            loop()
    end.
```

Here's an example:

```
$ cat see.erl | cksum
3915305815 9065
$ cat see.erl | ./see see_test3 see.erl | cksum
3915305815 9065
```

see_test4 tests error handling.

```erlang
see/see_test4.erl
-module(see_test4).
-export([main/0]).

main() ->
    see:write("I will crash now\n"),
    1 = 2,
    see:write("This line will not be printed\n").
```

Here's an example:

```
$ see see_test4
I will crash now
{stopping_system,{{badmatch,2},
  [{see_test4,main,0,[{file,"see_test4.erl"},{line,6}]}]}}
```

A3.3 The SEE API

see.erl is a single module that exports the following functions:

main()
>   Starts the system.

load_module(Mod)
>   Loads the module Mod.

log_error(Error)
>   Prints the value of Error on standard output.

make_server(Name, FunStart, FunHandler)
>   Creates a permanent server called Name. The initial state of the server is determined by evaluating FunStart(), and the "handler" function for the server is the fun FunHandler (we will say more about this later).

rpc(Name, Query)
>   Make a remote procedure call Query to the server Name.

change_behaviour(Name,FunHandler)
>   Change the behavior of the server called Name by sending it a new handler function FunHandler.

keep_alive(Name, Fun)
>   Make sure that there is always a registered process called Name. This process is started (or restarted) by evaluating Fun().

make_global(Name, Fun)
>   Make a global process with registered name Name. The process itself spawns the fun Fun().

on_exit(Pid, Fun)
>   Monitor the process Pid. If this process exits with reason {'EXIT', Why}, then evaluate Fun(Why).

on_halt(Fun)
>   Set up a condition such that if a request to stop the system is made, then Fun() will be evaluated. In the case where multiple funs have been specified, all these funs will be called.

stop_system(Reason)
>   Stop the system with the reason Reason.

every(Pid, Time, Fun)
>   As long as Pid has not terminated, evaluate Fun() every Time milliseconds.

lookup(Key,[{Key,Val}]) -> {found, Val} | not_found
    Look up a Key in a dictionary.

read() -> [string()] | eof
    Read the next line from standard input.

write([string()]) -> ok
    Write string to standard output.

env(Name)
    Return the value of the environment variable Name.

These functions can be used by simple Erlang programs.

A3.4 SEE Implementation Details

The shell script (see) that starts everything is as follows:

```
#!/bin/sh
erl -boot /home/joe/erl/example_programs-2.0/examples-2.0/see\
-environment `printenv` -load $1
```

The SEE Main Program

When the system is started, see:main() is evaluated, and when this terminates, the system will halt. see:main() is as follows:

see/see.erl
```
main() ->
    make_server(io,
                fun start_io/0, fun handle_io/2),
    make_server(code,
                const([lists,error_hander,see|preloaded()]),
                fun handle_code/2),
    make_server(error_logger,
                const(0),  fun handle_error_logger/2),
    make_server(halt_demon,
                const([]), fun handle_halt_demon/2),
    make_server(env,
                fun start_env/0, fun handle_env/2),
    Mod = get_module_name(),
    load_module(Mod),
    run(Mod).
```

This starts five servers (io, code…), loads the error handler, finds the name of the module that is to be run, loads this module, and then runs the code in the module. The run(Mod) spawn links Mod:main() and waits for it to terminate. When it terminates, stop_system is called.

```erlang
run(Mod) ->
    Pid = spawn_link(Mod, main, []),
    on_exit(Pid, fun(Why) -> stop_system(Why) end).
```

The function main/0 starts a lot of different servers. Before going into the details of how the individual servers work, we'll take a look at the generic framework that is used to build client-servers.

The Client-Server Model in SEE

To create a server, we call make_server(Name, Fun1, Fun2). Name is the global name of the server, and Fun1() is expected to return State1, the initial state of the server. Evaluating Fun2(State, Query) should return {Reply, State1} if called with a remote procedure call or simply State1 if called with a cast. make_server is as follows:

```erlang
make_server(Name, FunD, FunH) ->
    make_global(Name,
                fun() ->
                        Data = FunD(),
                        server_loop(Name, Data, FunH)
                end).
```

The server loop is simply as follows:

```erlang
server_loop(Name, Data, Fun) ->
    receive
        {rpc, Pid, Q} ->
            case (catch Fun(Q, Data)) of
                {'EXIT', Why} ->
                    Pid ! {Name, exit, Why},
                    server_loop(Name, Data, Fun);
                {Reply, Data1} ->
                    Pid ! {Name, Reply},
                    server_loop(Name, Data1, Fun)
            end;
        {cast, Pid, Q} ->
            case (catch Fun(Q, Data)) of
                {'EXIT', Why} ->
                    exit(Pid, Why),
                    server_loop(Name, Data, Fun);
                Data1 ->
                    server_loop(Name, Data1, Fun)
            end;
        {eval, Fun1} ->
            server_loop(Name, Data, Fun1)
    end.
```

To query the server, we use rpc (short for *remote procedure call*), which is as follows:

```
see/see.erl
rpc(Name, Q) ->
    Name ! {rpc, self(), Q},
    receive
        {Name, Reply} ->
            Reply;
        {Name, exit, Why} ->
            exit(Why)
    end.
```

Note how the code in the server loop and rpc interacts. The handler fun in the server is protected by a catch, and if an exception is raised in the server, a {Name,exit,Why} message is sent back to the client. If this message is received by the client, an exception is raised by evaluating exit(Why) in the client.

The net effect of this is to raise an exception in the client. Note that in the case where the client sent a query to the server that the server could not process, the server continues with its old state.

Thus, remote procedure calls function as *transactions* as far as the server is concerned. Either they work completely or the server is rolled back to the state it was in before the remote procedure call was made.

If we just want to send a message to the server and are not interested in the reply, we call cast/2.

```
see/see.erl
cast(Name, Q) ->
    Name ! {cast, self(), Q}.
```

We can change the behavior of a server by sending it a different fun to use in the server loop.

```
see/see.erl
change_behaviour(Name, Fun) ->
    Name ! {eval, Fun}.
```

Recall that when we started the server, the initial data structure is often a constant. We can define const(C), which returns a function, which when evaluated returns C.

```
see/see.erl
const(C) -> fun() -> C end.
```

Now we turn our attention to the individual servers.

The Code Server

The code server was started by evaluating the following:

```
see/see.erl
make_server(Name, FunD, FunH) ->
    make_global(Name,
                fun() ->
                        Data = FunD(),
                        server_loop(Name, Data, FunH)
                end).
```

load_module(Mod) is implemented as a remote procedure call to the code server.

```
load_module(Mod) ->
    rpc(code, {load, Mod}).
```

The global state of the code server is simply [Mod], that is, a list of all modules that have been loaded. (The initial value of this is [init, erl_prim_loader]. These modules are *preloaded* and compiled into the Erlang runtime system's kernel.)

The server handler function handle_code/see/2 is as follows:

```
handle_code(modules_loaded, Mods) ->
    {length(Mods), Mods};
handle_code({load, Mod}, Mods) ->
    case member(Mod, Mods) of
        true ->
            {already_loaded, Mods};
        false ->
            case primLoad(Mod) of
                {ok,Mod} ->
                    {{ok,Mod}, [Mod|Mods]};
                Error ->
                    {Error, Mods}
            end
    end.
```

And primLoad does the loading:

```
primLoad(Module) ->
    Str = atom_to_list(Module),
    case erl_prim_loader:get_file(Str ++ ".beam") of
        {ok, Bin, _FullName} ->
            case erlang:load_module(Module, Bin) of
                {module, Module} ->
                    {ok,Module};
                {module, _} ->
                    {error, wrong_module_in_binary};
                _Other ->
                    {error, {bad_object_code, Module}}
            end;
```

```
        _Error ->
            {error, {cannot_locate, Module}}
    end.
```

The Error Logger

log_error(What) logs the error What on standard output; this is implemented as a cast.

```
log_error(Error)  ->  cast(error_logger, {log, Error}).
```

The corresponding server handler function is as follows:

```
handle_error_logger({log, Error}, N) ->
    erlang:display({error, Error}),
    {ok, N+1}.
```

Note that the global state of the error handler is an integer N denoting the total number of errors that have occurred.

The Halt Demon

The halt demon is called when the system is halted. Evaluating on_halt(Fun) sets up a condition such that Fun() will be evaluated when the system is halted. Halting the system is done by calling the function stop_system().

```
on_halt(Fun)     -> cast(halt_demon,{on_halt,Fun}).
stop_system(Why) -> cast(halt_demon,{stop_system,Why}).
```

The server handler code for this is as follows:

```
handle_halt_demon({on_halt, Fun}, Funs) ->
    {ok, [Fun|Funs]};
handle_halt_demon({stop_system, Why}, Funs) ->
    case Why of
        normal -> true;
        _      -> erlang:display({stopping_system,Why})
    end,
    map(fun(F) -> F() end, Funs),
    erlang:halt(),
    {ok, []}.
```

The I/O Server

The I/O server allows access to STDIO. read() reads a line from standard input, and write(String) writes a string to standard output.

```
read()   -> rpc(io, read).
write(X) -> rpc(io, {write, X}).
```

The initial state of the I/O server is obtained by evaluating start_io().

```
start_io() ->
    Port = open_port({fd,0,1}, [eof, binary]),
    process_flag(trap_exit, true),
    {false, Port}.
```

And the I/O handler is as follows:

```
handle_io(read, {true, Port}) ->
    {eof, {true, Port}};
handle_io(read, {false, Port}) ->
    receive
        {Port, {data, Bytes}} ->
            {{ok, Bytes}, {false, Port}};
        {Port, eof} ->
            {eof, {true,Port}};
        {'EXIT', Port, badsig} ->
            handle_io(read, {false, Port});
        {'EXIT', Port, _Why} ->
            {eof, {true, Port}}
    end;
handle_io({write,X}, {Flag,Port}) ->
    Port ! {self(), {command, X}},
    {ok, {Flag, Port}}.
```

The state of the I/O server is {Flag, Port} where Flag is true if eof has been encountered; otherwise, it's false.

Environment Server

The function env(E) is used to find the value of the environment variable E.

```
env(Key) -> rpc(env, {lookup, Key}).
```

The server is as follows:

```
handle_env({lookup, Key}, Dict) ->
    {lookup(Key, Dict), Dict}.
```

The initial state of the server is found by evaluating the following:

```
start_env() ->
    Env = case init:get_argument(environment) of
              {ok, [L]} ->
                  L;
```

```
            error ->
                fatal({missing, '-environment ...'})
        end,
    map(fun split_env/1, Env).
split_env(Str) -> split_env(Str, []).
```

Support for Global Processes

We need a few routines for keeping processes alive and registering global names.

keep_alive(name, Fun) makes a registered process called Name. It is started by evaluating Fun(), and if the process dies, it is automatically restarted.

see/see.erl
```
keep_alive(Name, Fun) ->
    Pid = make_global(Name, Fun),
    on_exit(Pid,
            fun(_Exit) -> keep_alive(Name, Fun) end).
```

make_global(Name, Fun) checks whether there is a global process with the registered name Name. If there is no process, it spawns a process to evaluate Fun() and registers it with the name Name.

see/see.erl
```
make_global(Name, Fun) ->
    case whereis(Name) of
        undefined ->
            Self = self(),
            Pid = spawn(fun() ->
                                make_global(Self,Name,Fun)
                        end),
            receive
                {Pid, ack} ->
                    Pid
            end;
        Pid ->
            Pid
    end.
make_global(Pid, Name, Fun) ->
    case register(Name, self()) of
        {'EXIT', _} ->
            Pid ! {self(), ack};
        _ ->
            Pid ! {self(), ack},
            Fun()
    end.
```

Support for Processes

on_exit(Pid, Fun) links to Pid. If Pid dies with the reason Why, then Fun(Why) is evaluated.

see/see.erl
```erlang
on_exit(Pid, Fun) ->
    spawn(fun() ->
                  process_flag(trap_exit, true),
                  link(Pid),
                  receive
                      {'EXIT', Pid, Why} ->
                          Fun(Why)
                  end
          end).
```

every(Pid, Time, Fun) links to Pid; then every Time, Fun() is evaluated. If Pid exits, this process stops.

see/see.erl
```erlang
every(Pid, Time, Fun) ->
    spawn(fun() ->
                  process_flag(trap_exit, true),
                  link(Pid),
                  every_loop(Pid, Time, Fun)
          end).

every_loop(Pid, Time, Fun) ->
    receive
        {'EXIT', Pid, _Why} ->
            true
    after Time ->
            Fun(),
            every_loop(Pid, Time, Fun)
    end.
```

Utilities

get_module_name() gets the module name from the command line.

see/see.erl
```erlang
get_module_name() ->
    case init:get_argument(load) of
        {ok, [[Arg]]} ->
            module_name(Arg);
        error ->
            fatal({missing, '-load Mod'})
    end.
```

A3.5 How Code Gets Loaded in Erlang

The default mechanism for loading code into Erlang is to use a form of "on-demand" code loading. When code is first called and it is discovered that the code is missing, then the code is loaded.

This is what happens. Assume the function my_mod:myfunc(Arg1, Arg2, ... ArgN) is called but the code for this module has not yet been loaded. The system automatically converts this call into the following call:

error_module:undefined_function(mymod, myfunc, [Arg1, Arg2, ..., ArgN])

undefined_function is something like this:

```
undefined_function(Mod, Func, ArgList) ->
    case code:load_module(Mod) of
        {ok, Bin} ->
            erlang:load_module(Mod, Bin),
            apply(Mod, Func, ArgList);
        {error, _ } ->
            ...
    end
```

The undefined function handler delegates the task of finding the code for the module to the code handler. If the code handler can find the code, it loads the module and then calls apply(Mod, Func, ArgList), and the system continues as if the trap had not occurred.

The code in SEE to do this follows this pattern:

```
see/error_handler.erl
-module(error_handler).
-export([undefined_function/3,undefined_global_name/2]).
undefined_function(see, F, A) ->
    erlang:display({error_handler,undefined_function,
                    see,F,A}),
    exit(oops);
undefined_function(M, F, A) ->
    erlang:display({new_error_handler,undefined_function,M,F,A}),
    case see:load_module(M) of
        {ok, M} ->
            case erlang:function_exported(M,F,length(A)) of
                true ->
                    erlang:display({error_handler,calling,M,F,A}),
                    apply(M, F, A);
                false ->
                    see:stop_system({undef,{M,F,A}})
            end;
        {ok, _Other} ->
            see:stop_system({undef,{M,F,A}});
```

```
        already_loaded ->
            see:stop_system({undef,{M,F,A}});
        {error, What} ->
            see:stop_system({load,error,What})
    end.
undefined_global_name(Name, Message) ->
    exit({badarg,{Name,Message}}).
```

Note: The error handler in the Erlang system is significantly different from the code here. It does a lot more than the simple code handler here; it has to keep track of module versions and a few other things that make life complicated.

Once the error handler has been loaded by the boot script, autoloading will then work, using the code handling mechanisms in SEE and not the OTP system.

Exercises

1. SEE provides autoloading. But we could write an *even simpler* one without autoloading. Remove the code for autoloading.

2. You don't even need SEE for totally stand-alone applications. Write a minimal program that writes "Hello world" on standard output and terminates.

3. Write a minimal cat program that copies standard input to standard output (this is similar to see_test3.erl) but that starts without using SEE. Measure the time taken to copy a few large files. Compare this with some popular scripting languages.

Index

Put the "Fun" in Functional

Elixir puts the "fun" back into functional programming, on top of the robust, battle-tested, industrial-strength environment of Erlang, and Joe Kutner puts the fun back in staying fit.

You want to explore functional programming, but are put off by the academic feel (tell me about monads just one more time). You know you need concurrent applications, but also know these are almost impossible to get right. Meet Elixir, a functional, concurrent language built on the rock-solid Erlang VM. Elixir's pragmatic syntax and built-in support for metaprogramming will make you productive and keep you interested for the long haul. This book is *the* introduction to Elixir for experienced programmers.

Dave Thomas
(240 pages) ISBN: 9781937785581. $36
http://pragprog.com/book/elixir
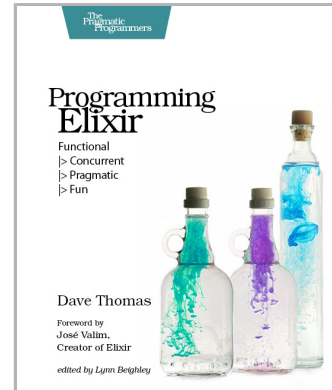
To keep doing what you love, you need to maintain your own systems, not just the ones you write code for. Regular exercise and proper nutrition help you learn, remember, concentrate, and be creative—skills critical to doing your job well. Learn how to change your work habits, master exercises that make working at a computer more comfortable, and develop a plan to keep fit, healthy, and sharp for years to come.

This book is intended only as an informative guide for those wishing to know more about health issues. In no way is this book intended to replace, countermand, or conflict with the advice given to you by your own healthcare provider including Physician, Nurse Practitioner, Physician Assistant, Registered Dietician, and other licensed professionals.

Joe Kutner
(254 pages) ISBN: 9781937785314. $36
*http://pragprog.com/book/jkthp*
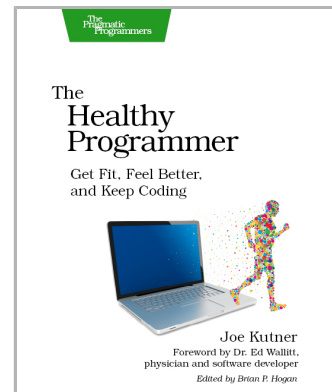
Long Live the Command Line!

Use tmux and vim for incredible mouse-free productivity.

Your mouse is slowing you down. The time you spend context switching between your editor and your consoles eats away at your productivity. Take control of your environment with tmux, a terminal multiplexer that you can tailor to your workflow. Learn how to customize, script, and leverage tmux's unique abilities and keep your fingers on your keyboard's home row.

Brian P. Hogan
(88 pages) ISBN: 9781934356968. $16.25
http://pragprog.com/book/bhtmux
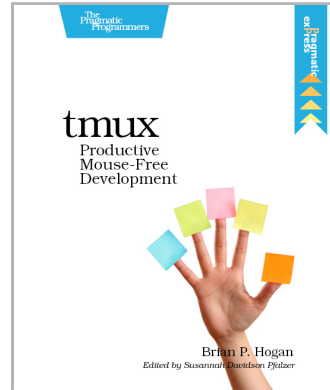
Vim is a fast and efficient text editor that will make you a faster and more efficient developer. It's available on almost every OS—if you master the techniques in this book, you'll never need another text editor. In more than 100 Vim tips, you'll quickly learn the editor's core functionality and tackle your trickiest editing and writing tasks.

Drew Neil
(346 pages) ISBN: 9781934356982. $29
*http://pragprog.com/book/dnvim*
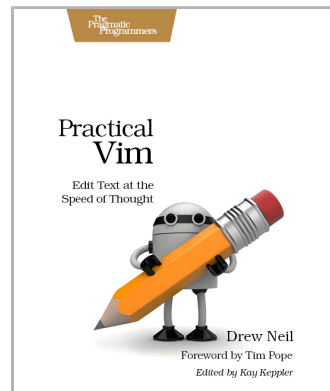
Seven Databases, Seven Languages

There's so much new to learn with the latest crop of NoSQL databases. And instead of learning a language a year, how about seven?

Data is getting bigger and more complex by the day, and so are your choices in handling it. From traditional RDBMS to newer NoSQL approaches, *Seven Databases in Seven Weeks* takes you on a tour of some of the hottest open source databases today. In the tradition of Bruce A. Tate's *Seven Languages in Seven Weeks*, this book goes beyond your basic tutorial to explore the essential concepts at the core of each technology.

Eric Redmond and Jim R. Wilson
(354 pages) ISBN: 9781934356920. $35
http://pragprog.com/book/rwdata
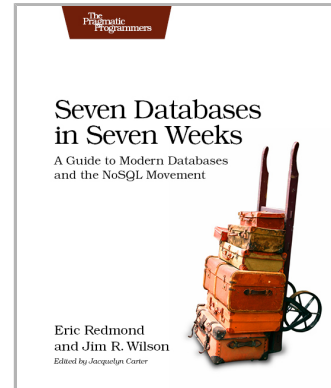
You should learn a programming language every year, as recommended by *The Pragmatic Programmer.* But if one per year is good, how about *Seven Languages in Seven Weeks*? In this book you'll get a hands-on tour of Clojure, Haskell, Io, Prolog, Scala, Erlang, and Ruby. Whether or not your favorite language is on that list, you'll broaden your perspective of programming by examining these languages side-by-side. You'll learn something new from each, and best of all, you'll learn how to learn a language quickly.

Bruce A. Tate
(330 pages) ISBN: 9781934356593. $34.95
*http://pragprog.com/book/btlang*
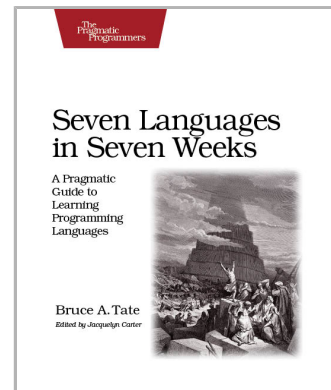
Web and Mobile Apps

Get a handle on responsive web apps and easier iOS mobile apps with Ruby.

With the advent of HTML5, front-end MVC, and Node.js, JavaScript is ubiquitous—and still messy. This book will give you a solid foundation for managing async tasks without losing your sanity in a tangle of callbacks. It's a fast-paced guide to the most essential techniques for dealing with async behavior, including PubSub, evented models, and Promises. With these tricks up your sleeve, you'll be better prepared to manage the complexity of large web apps and deliver responsive code.

Trevor Burnham
(104 pages) ISBN: 9781937785277. $17
*http://pragprog.com/book/tbajs*
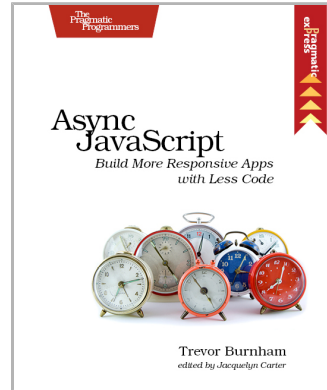
Make beautiful apps with beautiful code: use the elegant and concise Ruby programming language with RubyMotion to write truly native iOS apps with less code while having more fun. You'll learn the essentials of creating great apps, and by the end of this book, you'll have built a fully functional API-driven app. Whether you're a newcomer looking for an alternative to Objective-C or a hardened Rails veteran, RubyMotion allows you to create gorgeous apps with no compromise in performance or developer happiness.

Clay Allsopp
(112 pages) ISBN: 9781937785284. $17
*http://pragprog.com/book/carubym*
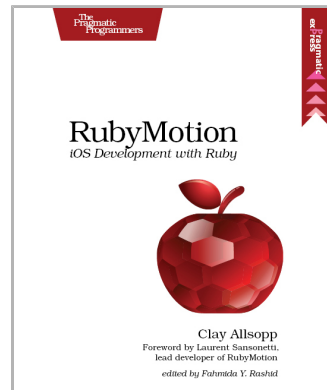
The Joy of Math and Programming

Rediscover the joy and fascinating weirdness of pure mathematics, or get your kids started programming in JavaScript.

Mathematics is beautiful—and it can be fun and exciting as well as practical. *Good Math* is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you've ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll
(282 pages) ISBN: 9781937785338. $34
http://pragprog.com/book/mcmath
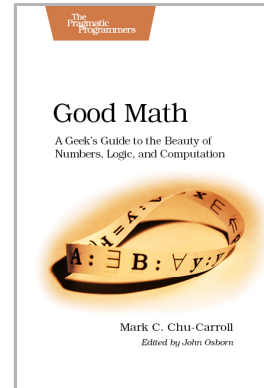
You know what's even better than playing games? Creating your own. Even if you're an absolute beginner, this book will teach you how to make your own online games with interactive examples. You'll learn programming using nothing more than a browser, and see cool, 3D results as you type. You'll learn real-world programming skills in a real programming language: JavaScript, the language of the web. You'll be amazed at what you can do as you build interactive worlds and fun games.

Chris Strom
(250 pages) ISBN: 9781937785444. $36
http://pragprog.com/book/csjava
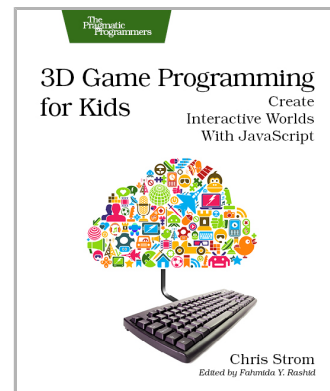
The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/book/jaerlang2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/book/jaerlang2*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764