Extracted from:

Programming Erlang

Software for a Concurrent World

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Joe Armstrong

Chapter 2

Getting Started

2.1 Overview

As with every learning experience, you'll pass through a number of stages on your way to Erlang mastery. Let's look at the stages we cover in this book and the things you'll experience along the way.

Stage 1: I'm Not Sure...

As a beginner, you'll learn how to start the system, run commands in the shell, compile simple programs, and become familiar with Erlang. (Erlang is a small language, so this won't take you long.)

Let's break this down into smaller chunks. As a beginner, you'll do the following:

- Make sure you have a working Erlang system on your computer.
- Learn to start and stop the Erlang shell.
- Discover how to enter expressions into the shell, evaluate them, and understand the results.
- See how to create and modify programs using your favorite text editor.
- Experiment with compiling and running your programs in the shell.

Stage 2: I'm Comfortable with Erlang

By now you'll have a working knowledge of the language. If you run into language problems, you'll have the background to make sense of Chapter 5, *Advanced Sequential Programming*, on page 88.

At this stage you'll be familiar with Erlang, so we'll move on to more interesting topics:

- You'll pick up more advanced uses of the shell. The shell can do a lot more than we let on when you were first learning it. (For example, you can recall and edit previous expressions. This is covered in Section 6.5, *Command Editing in the Erlang Shell*, on page 132.)
- You'll start learning the libraries (called *modules* in Erlang). Most of the programs I write can be written using five modules: lists, io, file, dict, and gen_tcp; therefore, we'll be using these modules a lot throughout the book.
- As your programs get bigger, you'll need to learn how to automate compiling and running them. The tool of choice for this is make. We'll see how to control the process by writing a makefile. This is covered in Section 6.4, *Automating Compilation with Makefiles*, on page 129.
- The bigger world of Erlang programming uses an extensive library collection called OTP.[1] As you gain experience with Erlang, you'll find that knowing OTP will save you lots of time. After all, why reinvent the wheel if someone has already written the functionality you need? We'll learn the major OTP *behaviors*, in particular gen_server. This is covered in Section 16.2, *Getting Started with gen_server*, on page 303.
- One of the main uses of Erlang is writing distributed programs, so now is the time to start experimenting. You can start with the examples in Chapter 10, *Distributed Programming*, on page 177, and you can extend them in any way you want.

Stage 2.5: I May Learn Some Optional Stuff

You don't have to read every chapter in this book the first time through.

Unlike most of the languages you have probably met before, Erlang is a concurrent programming language—this makes it particularly suited for writing distributed programs and for programming modern multicore and SMP[2] computers. Most Erlang programs will just run faster when run on a multicore or SMP machine.

Erlang programming involves using a programming paradigm that I call *concurrency-oriented programming* (COP).

1. Open Telecom Platform.
2. Symmetric multiprocessing.

When you use COP, you break down problems and identify the natural concurrency in their solutions. This is an essential first step in writing any concurrent program.

Stage 3: I'm an Erlang Master

By now you've mastered the language and can write some useful distributed programs. But to achieve true mastery, you need to learn even more:

- Mnesia. The Erlang distribution comes complete with a built-in fast, replicated database called Mnesia. It was originally designed for telecom applications where performance and fault tolerance are essential. Today it is used for a wide range of nontelecom applications.
- Interfacing to code written in other programming languages, and using *linked-in drivers*. This is covered in Section 12.4, *Linked-in Drivers*, on page 223.
- Full use of the OTP behaviors-building supervision trees, start scripts, and so on. This is covered in Chapter 18, *Making a System with OTP*, on page 337.
- How to run and optimize your programs for a *multicore* computer. This is covered in Chapter 20, *Programming Multicore CPUs*, on page 369.

The Most Important Lesson

There's one rule you need to remember throughout this book: programming is fun. And I personally think programming distributed applications such as chat programs or instant messaging applications is a lot more fun than programming conventional sequential applications. What you can do on one computer is limited, but what you can do with networks of computers becomes unlimited. Erlang provides an ideal environment for experimenting with networked applications and for building production-quality systems.

To help you get started with this, I've mixed some real-world applications in among the technical chapters. You should be able to take these applications as starting points for your own experiments. Take them, modify them, and deploy them in ways that I hadn't imagined, and I'll be very happy.

2.2 Installing Erlang

Before you can do anything, you have to make sure you have a functioning version of Erlang on your system. Go to a command prompt, and type erl:

```
$ erl
Erlang (BEAM) emulator version 5.5.2 [source] ... [kernel-poll:false]

Eshell V5.5.2  (abort with ^G)
1>
```

On a Windows system, the command erl works only if you have installed Erlang and changed the PATH environment variable to refer to the program. Assuming you've installed the program in the standard way, you'll invoke Erlang through the Start > All Programs > Erlang OTP menu. In Appendix B, on page 398, I'll describe how I've rigged Erlang to run with MinGW and MSYS.

Note: I'll show the banner (the bit that says "Erlang (BEAM) ... (abort with ^G)") only occasionally. This information is useful only if you want to report a bug. I'm just showing it here so you won't get worried if you see it and wonder what it is. I'll leave it out in most of the examples unless it's particularly relevant.

If you see the shell banner, then Erlang is installed on your system. Exit from it (press Ctrl+G, followed by the letter Q, and then hit Enter or Return).[3] Now you can skip ahead to Section 2.3, *The Code in This Book*, on page 25.

If instead you get an error saying erl is an unknown command, you'll need to install Erlang on your box. And that means you'll need to make a decision—do you want to use a prebuilt binary distribution, use a packaged distribution (on OS X), build Erlang from the sources, or use the Comprehensive Erlang Archive Network (CEAN)?

Binary Distributions

Binary distributions of Erlang are available for Windows and for Linux-based operating systems. The instructions for installing a binary system are highly system dependent. So, we'll go through these system by system.

3. Or give the command q() in the shell.

Windows

You'll find a list of the releases at http://www.erlang.org/download.html. Choose the entry for the latest version, and click the link for the Windows binary—this points to a Windows executable. Click the link, and follow the instructions. This is a standard Windows install, so you shouldn't have any problems.

Linux

Binary packages exist for Debian-based systems. On a Debian-based system, issue the following command:

```
> apt-get install erlang
```

Installing on Mac OS X

As a Mac user, you can install a prebuilt version of Erlang using the MacPorts system, or you can build Erlang from source. Using MacPorts is marginally easier, and it will handle updates over time. However, MacPorts can also be somewhat behind the times when it comes to Erlang releases. During the initial writing up this book, for example, the MacPorts version of Erlang was two releases behind the then current version. For this reason, I recommend you just bite the bullet and install Erlang from source, as described in the next section. To do this, you'll need to make sure you have the developer tools installed (they're on the DVD of software that came with your machine).

Building Erlang from Source

The alternative to a binary installation is to build Erlang from the sources. There is no particular advantage in doing this for Windows systems since each new release comes complete with Windows binaries and all the sources. But for Mac and Linux platforms, there can be some delay between the release of a new Erlang distribution and the availability of a binary installation package. For any Unix-like OS, the installation instructions are the same:

1. Fetch the latest Erlang sources.[4] The source will be in a file with a name such as otp_src_R11B-4.tar.gz (this file contains the fourth maintenance release of version 11 of Erlang).

4. From http://www.erlang.org/download.html.

2. Unpack, configure, make, and install as follows:

```
$ tar -xzf otp_src_R11B-4.tar.gz
$ cd otp_src_R11B-4
$ ./configure
$ make
$ sudo make install
```

Note: You can use the command ./configure --help to review the available configuration options before building the system.

Use CEAN

The Comprehensive Erlang Archive Network (CEAN) is an attempt to gather all the major Erlang applications in one place with a common installer. The advantage of using CEAN is that it manages not only the basic Erlang system but a large number of packages written in Erlang. This means that as well as being able to keep your basic Erlang installation up-to-date, you'll be able to maintain your packages as well.

CEAN has precompiled binaries for a large number of operating systems and processor architectures. To install a system using CEAN, go to http://cean.process-one.net/download/, and follow the instructions. (Note that some readers have reported that CEAN might not install the Erlang compiler. If this happens to you, then start the Erlang shell and give the command cean:install(compiler). This will install the compiler.)

2.3 The Code in This Book

Most of the code snippets we show come from full-length, running examples, which you can download.[5] 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):

Download 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 PDF version of this book and your PDF viewer supports hyperlinks, you can click the bar, and the code should appear in a browser window.

5. From http://pragmaticprogrammer.com/titles/jaerlang/code.html.

2.4 Starting the Shell

Now let's get started. We can interact with Erlang using an interactive tool called *the shell*. Once we've started the shell, we can type expressions, and the shell will display their values.

If you've installed Erlang on your system (as described in Section 2.2, *Installing Erlang*, on page 23), then the Erlang shell, erl, will also be installed. To run it, open a conventional operating system command shell (cmd on Windows or a shell such as bash on Unix-based systems). At the command prompt, start the Erlang shell by typing erl:

```
❶ $ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1  (abort with ^G)
❷ 1> % I'm going to enter some expressions in the shell ..
❸ 1> 20 + 30.
❹ 50
❺ 2>
```

Let's look at what we just did:

❶ 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.

❷ The shell printed the prompt 1>, and then we typed a comment. 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.

❸ The shell repeated the prompt 1> since we hadn't entered a complete command. At this point we entered the expression 20 + 30, followed by a period and a carriage return. (Beginners often forget to enter the period. Without it, Erlang won't know that we've finished our expression, and we won't see the result displayed.)

❹ The shell evaluated the expression and printed the result (50, in this case).

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

Have you tried running the shell on your system? If not, please stop and try it now. If you just read the text without typing in the commands, you might think that you understand what is happening, but you will not

have transferred this knowledge from your brain to your fingertips—programming is not a spectator sport. Just like any form of athletics, you have to practice a lot.

Enter the expressions in the examples exactly as they appear in the text, and then try experimenting with the examples and changing them a bit. If they don't work, stop and ask yourself what went wrong. Even an experienced Erlang programmer will spend a lot of time interacting with the shell.

As you get more experienced, you'll learn that the shell is a really powerful tool. Previous shell commands can be recalled (with Ctrl+P and Ctrl+N) and edited (with emacs-like editing commands). This is covered in Section 6.5, *Command Editing in the Erlang Shell*, on page 132. 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 an 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.

Warning: You can't type everything you read in this book into the shell. In particular, you can't type the code that's listed in the Erlang program files into the shell. The syntactic forms in an .erl file are *not* expressions and are not understood by the shell. The shell can evaluate only Erlang expressions and doesn't understand anything else. In particular, you can't type module annotations into the shell; these are things that start with a hyphen (such as -module, -export, and so on).

The remainder of this chapter is in the form of a number of short dialogues with the Erlang shell. A lot of the time I won't explain all the details of what is going on, since this would interrupt the flow of the text. In Section 5.4, *Miscellaneous Short Topics*, on page 100, I'll fill in the details.

2.5 Simple Integer Arithmetic

Let's evaluate some arithmetic expressions:

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

Important: You'll see that this dialogue starts at command number 1 (that is the shell printed, 1>). This means we have started a new Erlang

Is the Shell Not Responding?

If the shell didn't respond after you typed a command, then you might have forgotten to end the command with a period followed by carriage return (called *dot-whitespace*).

Another thing 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.

If things go really wrong and the system won't respond at all, then just press Ctrl+C (on Windows, Ctrl+Break). You'll see the following output:

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

Now just press A to abort the current Erlang session.

Advanced: You can start and stop multiple shells. See Section 6.7, *The Shell Isn't Responding*, on page 135 for details.

shell. Every time you see a dialogue 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 means the shell session is continued from the previous examples so you don't have to start a new shell.

Note: If you're going to type these examples into the shell as you read the text (which is *absolutely* the best way to learn), then you might like to take a quick peek at Section 6.5, *Command Editing in the Erlang Shell*, on page 132.

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.

Variable Notation

Often we will want to talk about the values of particular variables. For this I'll use the notation Var ↦ Value, so, for example, A ↦ 42 means that the variable A has the value 42. When there are several variables, I'll write {A ↦ 42, B ↦ true ... }, meaning that A is 42, B is true, and so on.

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

```
3> 123456789 * 987654321 * 112233445566778899 * 998877665544332211.
136695602603218099859661988989257616966134279099353 41
```

You can enter integers in a number of ways.[6] Here's an expression that uses base 16 and base 32 notation:

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

2.6 Variables

How can you store the result of a command so that you can use it later? That's what variables are for. Here's an example:

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

What's happening here? First, we assign a value to the variable X; then, the shell prints the value of the variable.

Note: 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
```

6. See Section 5.4, *Integers*, on page 113.

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. X isn't a math variable: it's like a pigeon hole/little box....

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.

However, if you try to assign a different value to the variable X, you'll get a somewhat brutal error message:

```
4> X = 1234.
=ERROR REPORT==== 11-Sep-2006::20:32:49 ===
Error in process <0.31.0> with exit value:
  {{badmatch,1234},[{erl_eval,expr,3}]}

** exited: {{badmatch,1234},[{erl_eval,expr,3}]} **
```

What on Earth is going on here? Well, to explain it, I'm going to have to shatter two assumptions you have about the simple statement X = 1234:

- First, X is not a variable, at least not in the sense that you're used to in languages such as Java and C.
- Second, = is not an assignment operator.

This is probably one of the trickiest areas when you're new to Erlang, so let's spend a couple of pages digging deeper.

Variables That Don't Vary

Erlang has *single assignment variables*. As the name suggests, single assignment variables 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. All variables start off unbound.

When Erlang sees a statement such as X = 1234, it binds the variable X to the value 1234. Before being bound, X could take any value: it's just an empty hole waiting to be filled. However, once it gets a value, it holds on to it forever.

At this point, you're probably wondering why we use the name variable. 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 = ...
  ```
  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.

  Note: The use of ellipses (...) in Erlang code examples just means "code I'm not showing."

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 different.

Pattern Matching

In most languages, = denotes an assignment statement. In Erlang, however, = denotes 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).

Now 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.

Then, if at a later stage we say X = AnotherExpression, then this will succeed only if SomeExpression and AnotherExpression are identical. Here's an example of this:

```
Line 1  1> X = (2+4).
        6
        2> Y = 10.
        10
      5 3> X = 6.
        6
        4> X = Y.
        =ERROR REPORT==== 27-Oct-2006::17:25:25 ===
        Error in process <0.32.0> with exit value:
     10    {{badmatch,10},[{erl_eval,expr,3}]}
        5> Y = 10.
        10
        6> Y = 4.
        =ERROR REPORT==== 27-Oct-2006::17:25:46 ===
     15 Error in process <0.37.0> with exit value:
          {{badmatch,4},[{erl_eval,expr,3}]}
        7> Y = X.
        =ERROR REPORT==== 27-Oct-2006::17:25:57 ===
        Error in process <0.40.0> with exit value:
     20   {{badmatch,6},[{erl_eval,expr,3}]}
```

Here's what happened: In line 1 the system evaluated the expression 2+4, and the answer was 6. So after this line, the shell has the following set of bindings: {X ↦ 6}. After line 3 has been evaluated, we have the bindings {X ↦ 6, Y ↦ 10}.

Now we come to line 5. Just before we evaluate the expression, we know that X ↦ 6, so the match X = 6 succeeds.

When we say X = Y in line 7, our bindings are {X ↦ 6, Y ↦ 10}, and therefore the match fails and an error message is printed.

Expressions 4 to 7 either succeed or fail depending upon the values of X and Y. Now is a good time to stare hard at these and make sure you really understand them before going any further.

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.

Why Does Single Assignment Make My Programs Better?

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.

Let's see what can happen when you're allowed to change a variable. Let's define a variable X as follows:

```
1> X = 23.
23
```

Now we can use X in computations:

```
2> Y = 4 * X + 3.
95
```

Now suppose we could *change* the value of X (horrors):

```
3> X = 19.
```

Fortunately, Erlang doesn't allow this. The shell complains like crazy and says this:

```
=ERROR REPORT==== 27-Oct-2006::13:36:24 ===
Error in process <0.31.0> with exit value:
  {{badmatch,19},[{erl_eval,expr,3}]}
```

This just means that X cannot be 19 since we've already said it was 23.

But just suppose we could do this; then the value of Y would be wrong in the sense that we can no longer interpret statement 2 as an equation. Moreover, if X could change its value at many different points in the program and something goes wrong, it might be difficult saying which particular value of X had caused the failure and at exactly which point in the program it had acquired the wrong value.

In Erlang, variable values cannot be changed after they have been set. This 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 you discover that your program is incorrect is that a variable has an unexpected value. If this is the case, then you have to discover exactly the point in your program where the variable acquired the incorrect value. If this variable changed values many

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 nonmutable state.

Much later in the book we'll look at how to program multicore CPUs. When it comes to programming multicore CPUs, the consequences of having nonmutable 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.

times and at many different points in your program, then finding out exactly which of these changes was incorrect can be extremely difficult.

In Erlang there is no such problem. A variable can be set only once and thereafter never changed. So once we know which variable is incorrect, we can *immediately* infer the place in the program where the variable became bound, and this must be where the error occurred.

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

2.7 Floating-Point Numbers

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

```
1> 5/3.
1.66667
2> 4/2.
2.00000
3> 5 div 3.
1
```

```
4> 5 rem 3.
2
5> 4 div 2.
2
6> Pi = 3.14159.
3.14159
7> R = 5.
5
8> Pi * R * R.
78.5397
```

Don't get confused here. 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.

"/" always returns a float; thus, 4/2 evaluates to 2.0000 (in the shell). N div M and N rem M are used for integer division and remainder; thus, 5 div 3 is 1, and 5 rem 3 is 2.

Floating-point numbers must have a decimal point followed by at least one decimal digit. When you divide two integers with "/", the result is automatically converted to a floating-point number.

2.8 Atoms

In Erlang, atoms are used to represent different non-numerical constant values.

If you're used to enumerated types in C or Java, then you will already have used something very similar to atoms whether you realize it or not.

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 you want to write a program that manipulates days of the week. How would you represent a day in Erlang? Of course, you'd use one of the atoms monday, tuesday, ...

Atoms start with lowercase letters, followed by a sequence of alphanumeric characters or the underscore (_) or at (@) sign.[7] 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.

The value of an atom is just the atom. So if you 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 talking 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.

7. You might find that a period (.) can also be used in atoms—this is an unsupported extension to Erlang.

2.9 Tuples

Suppose you want to group a fixed number of items into a single entity. For this you'd use a *tuple*. You can create a tuple by enclosing the values you want to represent in curly brackets and separating them with commas. So, for example, if you want to represent someone's name and height, you 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;
```

You'd access the fields in a C struct using the dot operator. So to set the x and y values in Point, you 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, the fields of a tuple have no names. Since the tuple itself just contains 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.[8]

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}}.
```

8. This way of tagging a tuple is not a language requirement but is a recommended style of programming.

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 you 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 you try to create a data structure with an undefined variable, then you'll get an error. So in the next line, if we try to use the variable Q that is undefined, 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 that 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 you 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 you had entered something like this:

```
5> {point, C, C} = Point.
=ERROR REPORT==== 28-Oct-2006::17:17:00 ===
Error in process <0.32.0> with exit value:
{{badmatch,{point,10,45}},[{erl_eval,expr,3}]}
```

What happened? The pattern {point, C, C} does not match {point, 10, 45}, since C cannot be simultaneously 10 and 45. Therefore, the pattern matching fails,[9] and the system prints an error message.

If you have a complex tuple, then you 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 you want to extract values.[10]

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

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

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

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

9. For readers familiar with Prolog: Erlang considers nonmatching a failure and does not backtrack.

10. This method of extracting variables using pattern matching is called *unification* and is used in many functional and logic programming languages.

And finally we'll print out 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.

2.10 Lists

We use lists to store variable numbers of things: things you want to buy at the store, the names of the planets, the results returned by your prime factors function, and so on.

We create a list by enclosing the list elements in square brackets and separating them with commas. Here's how we could create a shopping list:

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

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,[11] with head H and tail T. The vertical bar | separates the head of a list from its tail. [] is the empty list.

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]. For example:

```
3> 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 in the shop, and we have our shopping list ThingsToBuy1—the first thing we do is unpack the list into its head and tail:

```
4> [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:

```
5> [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}].

11. 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.

2.11 Strings

Strictly speaking, there are no strings in Erlang. *Strings* are really just lists of integers. Strings are enclosed in double quotation marks ("), so, for example, we can write this:

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

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

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

When the shell prints the value of a list it prints the list as a string, but only if all the integers in the list represent printable characters:

```
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.

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"
```

Character Sets Used in Strings

The characters in a string represent Latin-1 (ISO-8859-1) character codes. For example, the string containing the Swedish name Håkan will be encoded as [72,229,107,97,110].

Note: If you enter [72,229,107,97,110] as a shell expression, you might not get what you expect:

```
1> [72,229,107,97,110].
"H\345kan"
```

What has happened to "Håkan"—where did he go? This actually has nothing to do with Erlang but with the locale and character code settings of your terminal.

As far as Erlang is concerned, a string is a just a list of integers in some encoding. If they happen to be printable Latin-1 codes, then they should be displayed correctly (if your terminal settings are correct).

2.12 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.[12] The third column of the table, marked *Result*, shows whether the pattern matched the term and, if so, the variable bindings that were created. Look through these examples, and make sure you really understand them:

Pattern	**Term**	**Result**
`{X,abc}`	`{123,abc}`	*Succeeds* X ↦ 123
`{X,Y,Z}`	`{222,def,"cat"}`	*Succeeds* X ↦ 222, Y ↦ def, Z ↦ "cat"
`{X,Y}`	`{333,ghi,"cat"}`	*Fails*—the tuples have different shapes
`X`	`true`	*Succeeds* X ↦ true
`{X,Y,X}`	`{{abc,12},42,{abc,12}}`	*Succeeds* X ↦ {abc,12}, 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* H ↦ 1, T ↦ [2,3,4,5]
`[H|T]`	`"cat"`	*Succeeds* H ↦ 99, T ↦ "at"
`[A,B,C|T]`	`[a,b,c,d,e,f]`	*Succeeds* A ↦ a, B ↦ b, C ↦ c, 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.

12. A *term* is just an Erlang data structure.

For 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, so we can step up the tempo and see how to define functions and modules. Let's see how in the next chapter.

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

Programming Erlang
http://pragmaticprogrammer.com/titles/jaerlang
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragmaticprogrammer.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragmaticprogrammer.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://pragmaticprogrammer.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/jaerlang.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com