8

FUNCTIONALLY SOLVING PROBLEMS

So we're ready to do something practical with all that Erlang juice we drank. In this chapter, we'll apply some of the techniques covered in previous chapters to solve some interesting problems.

The problems in this chapter were taken from Miran Lipovača's *Learn You a Haskell for Great Good!* (No Starch Press, 2011; available from *http://learnyouahaskell.com*). I decided to use the same problems so curious readers can compare solutions in Erlang and Haskell as they wish. If you do so, you might find the final results to be pretty similar for two languages with such different syntaxes. This is because once you understand functional concepts, you'll find that they're relatively easy to carry over to other functional languages.

Reverse Polish Notation Calculator

Most people have learned to write arithmetic expressions with the operators in between the numbers (`(2 + 2) / 5`). This is how most calculators let you insert mathematical expressions, and it's probably the notation you were taught in school. This notation has the downside of needing you to know about operator precedence. For example, multiplication and division are more important (have a higher precedence) than addition and subtraction.

In another notation, called *prefix notation* or *Polish notation*, the operator comes before the operands. Under this notation, `(2 + 2) / 5` becomes `(/ (+ 2 2) 5)`. If we decide to say `+` and `/` always take two arguments, then `(/ (+ 2 2) 5)` can simply be written as `/ + 2 2 5`.

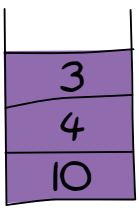
However, we will instead focus on *Reverse Polish notation (RPN)*, which is the opposite of prefix notation: the operator follows the operands. In RPN, our example is written as `2 2 + 5 /`. The expression `9 * 5 + 7` becomes `9 5 * 7 +`, and `10 * 2 * (3 + 4) / 2` is translated to `10 2 * 3 4 + * 2 /`. This notation was used a whole lot in early models of calculators, as it takes little memory to use. In fact, some people still carry around RPN calculators. We'll write one of these.

How RPN Calculators Work

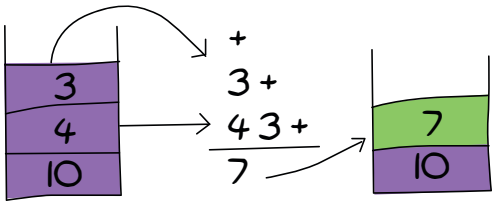
First, let's consider how to read RPN expressions. One way is to find the operators one by one, and then regroup them with their operands by arity:

```
10 4 3 + 2 * -
10 (4 3 +) 2 * -
10 ((4 3 +) 2 *) -
(10 ((4 3 +) 2 *) -)
(10 (7 2 *) -)
(10 14 -)
-4
```

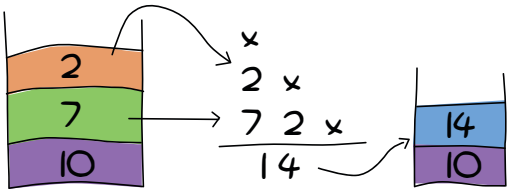
However, in the context of a computer or a calculator, a simpler way to read RPN expressions is to make a *stack* of all the operands as we see them. For example, in the mathematical expression `10 4 3 + 2 * -`, the first operand we see is `10`. We add that to the stack. Then there is `4`, so we also push that on top of the stack. In third place, we have `3`—let's push that one on the stack, too. Our stack should now look like this:



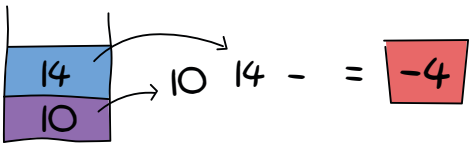
The next character to parse is +. That one is a function of arity 2. In order to use it, we will need to feed it two operands, which will be taken from the stack:



So we take that 7 and push it back on top of the stack (yuck, we don't want to keep these filthy numbers floating around!). The stack is now [7,10], and what's left of the expression is `2 * -`. We can take the 2 and push it on top of the stack. We then see *, which needs two operands to work. Again, we take them from the stack:



And we push 14 back on top of our stack. All that remains is -, which also needs two operands. Oh glorious luck! There are two operands left in our stack. Use them!



And so we have our result. This stack-based approach is relatively foolproof, and the low amount of parsing needed before starting to calculate results explains why it was a good idea for old calculators to use this approach.

Creating an RPN Calculator

Making our own RPN calculator in Erlang is not too hard once we've done the complex stuff. It turns out the tough part is figuring out what steps need to be done in order to get our end result, and we just did that. So let's get started by opening a file named *calc.erl*.

The first part to worry about is how we're going to represent a mathematical expression. To make things simple, we'll probably input them as a string: `"10 4 3 + 2 * -"`. This string has whitespace, which isn't part of our problem-solving process, but is necessary in order to use a

simple tokenizer. What would be usable then is a list of terms of the form `["10","4","3","+","2","*","-"]` after going through the tokenizer. It turns out the function `string:tokens/2` does just that:

```
1> string:tokens("10 4 3 + 2 * -", " ").
["10","4","3","+","2","*","-"]
```

This will be a good representation for our expression.

The next part to define is the stack. How are we going to do that? You might have noticed that Erlang's lists act a lot like stacks. Using the cons operator (`|`) in [*Head*|*Tail*] effectively produces the same behavior as pushing *Head* on top of a stack (*Tail*, in this case). Using a list for a stack will be good enough.

To read the expression, we just need to do the same thing as we did when solving the problem by hand. Read each value from the expression, and if it's a number, put it on the stack. If it's a function, pop all the values it needs from the stack, and then push the result back in. To generalize, we need to go over the whole expression as a loop only once and accumulate the results. Sounds like the perfect job for a fold!

What we need to plan for is the function that `lists:foldl/3` will apply on every operator and operand of the expression. This function, because it will be run in a fold, will need to take two arguments: the first one will be the element of the expression to work with, and the second one will be the stack.

We can start writing our code in the *calc.erl* file. First, we'll write the function responsible for all the looping and also the removal of spaces in the expression:

```
-module(calc).
-export([rpn/1]).

rpn(L) when is_list(L) ->
    [Res] = lists:foldl(fun rpn/2, [], string:tokens(L, " ")),
    Res.
```

Next, we'll implement `rpn/2`. Note that because each operator and operand from the expression ends up being put on top of the stack, the solved expression's result will be on that stack. We need to get that last value out of there before returning it to the user. This is why we pattern match over `[Res]` and return only `Res`.

Now to the harder part. Our `rpn/2` function will need to handle the stack for all values passed to it. The head of the function will probably look like `rpn(Op,Stack)`, and its return value will look like `[NewVal|Stack]`. When we get regular numbers, the operation will be as follows:

```
rpn(X, Stack) -> [read(X)|Stack].
```

Here, `read/1` is a function that converts a string to an integer or a floating-point value. Sadly, there is no built-in function to do this in Erlang (it just has functions that convert to only one or the other). So we'll add the function ourselves, like this:

```
read(N) ->
    case string:to_float(N) of
        {error,no_float} -> list_to_integer(N);
        {F,_} -> F
    end.
```

Here, `string:to_float/1` does the conversion from a string such as `"13.37"` to its numeric equivalent. However, if there is no way to read a floating-point value, it returns `{error,no_float}`. When that happens, we need to call `list_to_integer/1` instead.

Now let's get back to `rpn/2`. The numbers we encounter are all added to the stack. However, because our pattern matches on anything (see Chapter 5 for a discussion of pattern matching), operators will also get pushed on the stack. To avoid this, we'll put them all in preceding clauses. The first one we'll try this with is the addition:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn(X, Stack) -> [read(X)|Stack].
```

You can see that whenever we encounter the `"+"` string, we take two numbers from the top of the stack (`N1`,`N2`) and add them before pushing the result back onto that stack. This is exactly the same logic we applied when solving the problem by hand. Trying the program, we can see that it works:

```
1> c(calc).
{ok,calc}
2> calc:rpn("3 5 +").
8
3> calc:rpn("7 3 + 5 +").
15
```

The rest is trivial, as we just need to add all the other operators:

```
rpn("+", [N1,N2|S]) -> [N2+N1|S];
rpn("-", [N1,N2|S]) -> [N2-N1|S];
rpn("*", [N1,N2|S]) -> [N2*N1|S];
rpn("/", [N1,N2|S]) -> [N2/N1|S];
rpn("^", [N1,N2|S]) -> [math:pow(N2,N1)|S];
rpn("ln", [N|S])    -> [math:log(N)|S];
rpn("log10", [N|S]) -> [math:log10(N)|S];
rpn(X, Stack) -> [read(X)|Stack].
```

Note that functions that take only one argument such as logarithms need to pop only one element from the stack. It is left as an exercise for the

reader to add functions such as `sum` and `prod`, which return the sum of all the elements read so far and the products of all the elements, respectively. To help you out, they are already implemented in my version of *calc.erl*.

Testing the Code

To make sure this all works, we'll write some very simple unit tests. Erlang's `=` operator can act as an *assertion* function. Assertions should crash whenever they encounter unexpected values, which is exactly what we need. Of course, there are more advanced testing frameworks for Erlang, including EUnit and Common Test. We'll check them out in Chapters 25 and 28 but for now, the basic `=` will do the job.

```
rpn_test() ->
    5 = rpn("2 3 +"),
    87 = rpn("90 3 -"),
    -4 = rpn("10 4 3 + 2 * -"),
    -2.0 = rpn("10 4 3 + 2 * - 2 /"),
    ok = try
        rpn("90 34 12 33 55 66 + * - +")
    catch
        error:{badmatch,[_|_]} -> ok
    end,
    4037 = rpn("90 34 12 33 55 66 + * - + -"),
    8.0 = rpn("2 3 ^"),
    true = math:sqrt(2) == rpn("2 0.5 ^"),
    true = math:log(2.7) == rpn("2.7 ln"),
    true = math:log10(2.7) == rpn("2.7 log10"),
    50 = rpn("10 10 10 20 sum"),
    10.0 = rpn("10 10 10 20 sum 5 /"),
    1000.0 = rpn("10 10 20 0.5 prod"),
    ok.
```

The test function tries all operations. If no exception is raised, the tests are considered successful. The first four tests check that the basic arithmetic functions work correctly. In the fifth test, the `try ... catch` expects a `badmatch` error to be thrown because the expression can't work:

```
90 34 12 33 55 66 + * - +
90 (34 (12 (33 (55 66 +) *) -) +)
```

At the end of `rpn/1`, the values `-3947` and `90` are left on the stack because there is no operator to work on the `90` that hangs there. There are two possible ways to handle this problem: ignore it and take only the value on top of the stack (which would be the last result calculated), or crash because the arithmetic is wrong. Given that Erlang's policy is to let it crash, that's the path chosen here. The part that actually crashes is the `[Res]` in `rpn/1`. That one makes sure only one element—the result—is left in the stack.

The few tests that are of the form `true = FunctionCall1 == FunctionCall2` are there because you can't have a function call on the left-hand side of `=`. It still works as an assertion because we compare the comparison's result to `true`.

I've also added the test cases for the `sum` and `prod` operations, so you can test them after implementing these functions. If all tests are successful, you should see the following:

```
1> c(calc).
{ok,calc}
2> calc:rpn_test().
ok
3> calc:rpn("1 2 ^ 2 2 ^ 3 2 ^ 4 2 ^ sum 2 -").
28.0
```

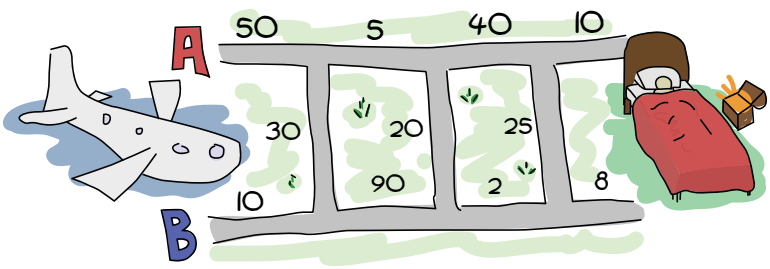
Here, `28.0` is indeed equal to `sum(1² + 2² + 3² + 4²) - 2`. Try as many calculations as you wish.

NOTE *One way to improve our calculator is to make sure it raises `badarith` errors when it crashes because of unknown operators or values left on the stack, rather than raising `badmatch` errors. It would certainly make debugging easier for the user of the `calc` module.*

Heathrow to London

Our next problem is also taken from *Learn You a Haskell*. You're on a plane due to land at Heathrow Airport in the next few hours. You need to get to London as fast as possible. Your rich uncle is dying, and you want to be the first one there to claim dibs on his estate.

There are two roads going from Heathrow to London, and a bunch of smaller streets linking them together. Because of speed limits and traffic, some parts of the roads and smaller streets take longer to travel than others. Before you land, you decide to maximize your chances by finding the optimal path to your uncle's house. Here's the map you've found on your laptop:



Having become a huge fan of Erlang after reading online books, you decide to solve the problem using that language. To make it easier to work with the map, you enter the following data in a file named *road.txt*:

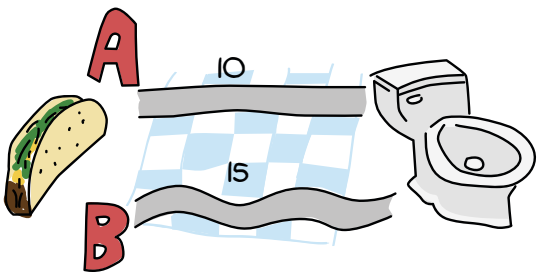
```
50
10
30
5
90
20
40
2
25
10
8
0
```

The path is laid out in the pattern `A1`, `B1`, `X1`, `A2`, `B2`, *`X2`*, `...`, `An`, `Bn`, *`Xn`*, where *`X`* is one of the roads joining the A side to the B side of the map. We insert a `0` as the last `X` segment, because no matter what we do, we're at our destination already. Data can probably be organized in tuples of three elements of the form `{A,B,X}`.

The next thing you realize is that it's worthless to try to solve this problem in Erlang when you don't even know how to solve it by hand. In order to do this, we'll use what recursion taught us.

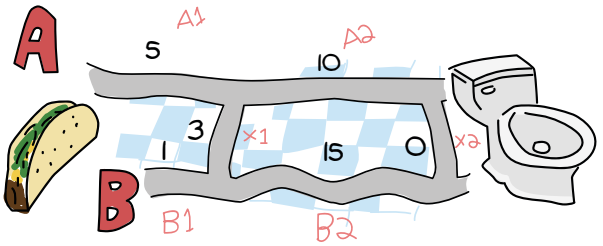
Solving the Problem Recursively

When writing a recursive function, the first thing to do is to find the base case. For the problem at hand, this would be if we had only one tuple to analyze; that is, if we only had to choose between `A`, `B` (and crossing `X`, which in this case is useless because we're at our destination):



Then the only choice is picking whether path A or path B is the shortest. By understanding how recursion works, we know that we should try to converge toward the base case. This means that on each step we'll take, we'll want to reduce the problem to choosing between A and B for the next step.

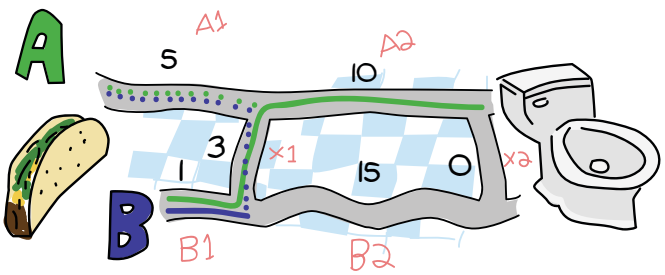
Let's extend our map and start over:



Ah! It gets interesting! How can we reduce the triple {5,1,3} to a strict choice between A and B? Let's see how many options are possible for A. To get to the intersection of A1 and A2 (we'll call this point A1), we can either take road A1 directly (5) or come from B1 (1) and then cross over X1 (3). In this case, the first option (5) is longer than the second one (4). For option A, the shortest path is [B, X]. So what are the options for B? We can either proceed from A1 (5) and then cross over X1 (3) or strictly take the path B1 (1).

So we now have a length 4 with the path [B, X] toward the first intersection A and a length 1 with the path [B] toward the intersection of B1 and B2. Now we must decide how to go to the second point A (the intersection of A2 and the endpoint or X2) and the second point B (intersection of B2 and X2). To make a decision, I suggest we do the same as before (and you don't have much choice but to obey, given that I'm the guy writing this book). Here we go!

We can get to the next point A by either taking the path A2 from [B, X], which gives us a length of 14 (14 = 4 + 10), or by taking B2 then X2 from [B], which gives us a length of 16 (16 = 1 + 15 + 0). In this case, the path [B, X, A] is better than [B, B, X].



We can also get to the next point B by either taking the path A2 from [B, X], and then crossing over X2 for a length of 14 (14 = 4 + 10 + 0), or by taking the road B2 from [B] for a length of 16 (16 = 1 + 15). Here, the best path is to pick the first option: [B, X, A, X].

When this whole process is complete, we're left with two paths: A or B, both of length 14. Either of them is the right one. The last selection will always have two paths of the same length, given the last X segment has a length of 0. By solving our problem recursively, we've made sure to always get the shortest path at the end. Not too bad, eh?

Subtly enough, we've given ourselves the basic logical parts we need to build a recursive function. We could implement it, but I promised we would have very few recursive functions to write ourselves. Instead, we'll use a fold.

NOTE *While I have shown folds being used and constructed with lists, folds represent a broader concept of iterating over a data structure with an accumulator. As such, folds can be implemented over trees, dictionaries, arrays, database tables, and so on. It is sometimes useful when experimenting to use abstractions like maps and folds because they make it easier to later change the data structure you use to work with your own logic.*

Writing the Code

So where were we? Ah, yes! We have the file we're going to feed as input ready. To do file manipulations, the `file` module is our best tool. It contains a lot of functions common to many programming languages in order to deal with files themselves (setting permissions, moving files around, renaming files, deleting files, and so on).

The `file` module also contains the usual functions to read and/or write from files, such as `file:open/2` and `file:close/1` to do as their names say (open and close files!), `file:read/2` to get the content of a file (either as string or a binary), `file:read_line/1` to read a single line, and `file:position/3` to move the pointer of an open file to a given position.

The module also contains a bunch of shortcut functions, such as `file:read_file/1` (opens and reads the contents as a binary), `file:consult/1` (opens and parses a file as Erlang terms), `file:pread/2` (changes the position and then reads content), and `file:pwrite/2` (changes the position and writes content).

With all these choices available, it's going to be easy to find a function to read our *road.txt* file. Because we know our road is relatively small, we'll call `file:read_file("road.txt").`:

```
1> {ok, Binary} = file:read_file("road.txt").
{ok,<<"50\r\n10\r\n30\r\n5\r\n90\r\n20\r\n40\r\n2\r\n25\r\n10\r\n8\r\n0\r\n">>}
2> S = string:tokens(binary_to_list(Binary), "\r\n\t ").
["50","10","30","5","90","20","40","2","25","10","8","0"]
```

Note that in this case, we added a space (`" "`) and a tab (`"\t"`) to the valid tokens, so the file could also have been written in the form `"50 10 30 5 90 20 40 2 25 10 8 0"`.

Given that list, we'll need to transform the strings into integers.

```
3> [list_to_integer(X) || X <- S].
[50,10,30,5,90,20,40,2,25,10,8,0]
```

Let's start a new module called *road.erl* and write down this logic:

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

main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    parse_map(Bin).

parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")].
```

The function `main/0` is responsible for reading the content of the file and passing it on to `parse_map/1`. Because we use the function `file:read_file/1` to get the contents of *road.txt*, the result we obtain is a binary. For this reason, we've made the function `parse_map/1` match on both lists and binaries. In the case of a binary, we just call the function again with the string being converted to a list (our function to split the string works only on lists).

The next step in parsing the map would be to regroup the data into the `{A,B,X}` form described earlier. Sadly, there's no simple generic way to pull elements from a list three at a time, so we'll need to pattern match our way in a recursive function in order to accomplish this:

```
group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).
```

That function works in a standard tail-recursive manner; there's nothing too complex going on here. We'll just need to call it by modifying `parse_map/1` a bit:

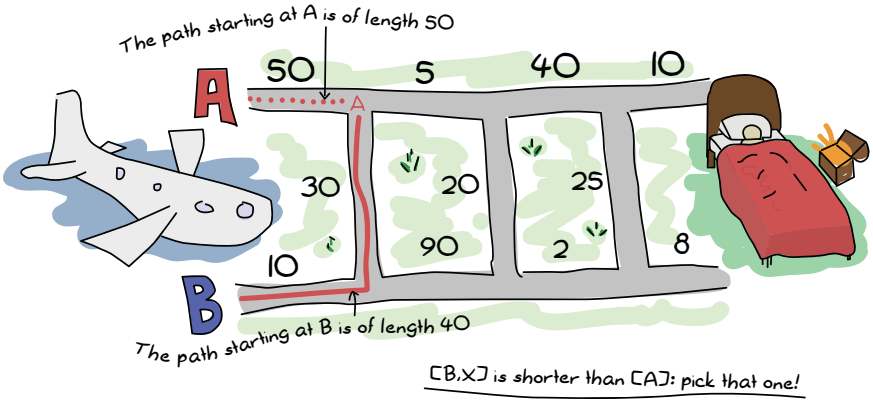
```
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")],
    group_vals(Values, []).
```

Let's try to compile it all and see if we now have a road that makes sense.

```
1> c(road).
{ok,road}
2> road:main().
[{50,10,30},{5,90,20},{40,2,25},{10,8,0}]
```

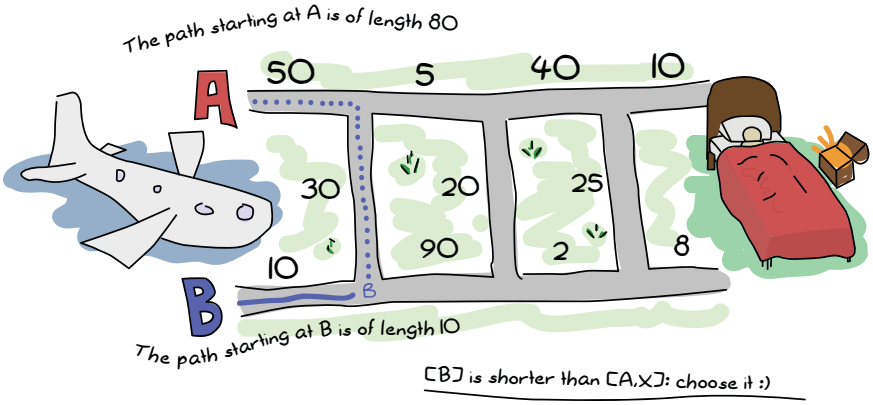
Ah yes, that looks right. We get the blocks we need to write our function that will then fit in a fold. For this to work, finding a good accumulator is necessary.

To decide what to use as an accumulator, the method I find the easiest to employ is to imagine myself in the middle of the algorithm while it runs. For this specific problem, we'll imagine that we're currently trying to find the shortest path of the second triple ({5,90,20}). To decide on which path is the best, we need to have the result from the previous triple. Luckily, we know how to get that, because we don't need an accumulator, and we have all that logic already worked out. So for A, we have the following:



And we take the shorter of these two paths.

For B, the choice is similar:



So now we know that the current best path coming from A is `[B, X]`. We also know it has a length of 40. For B, the path is simply `[B]`, and the length is 10. We can use this information to find the next best paths for A and B by reapplying the same logic, but counting the previous ones in the expression.

The other data we need is the path traveled so we can show it to the user. Given that we need two paths (one for A and one for B) and two accumulated lengths, our accumulator can take the form {{*DistanceA*, *PathA*}, {*DistanceB*, *PathB*}}. That way, each iteration of the fold has access to all the state, and we build it up to show it to the user in the end.

This gives us all the parameters our function will need: the {A,B,X} tuples and an accumulator of the form {{*DistanceA*,*PathA*}, {*DistanceB*,*PathB*}}.

We can put this into code in order to get our accumulator as follows:

```
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

Here, `OptA1` gets the first option for A (going through `A`), and `OptA2` gets the second one (going through `B` then `X`). The variables `OptB1` and `OptB2` get the similar treatment for point B. Finally, we return the accumulator with the paths obtained.

For the paths saved in this code, I decided to use the form [{x,X}] rather than [x] for the simple reason that it might be nice for the user to know the length of each segment. We're also accumulating the paths backward ({x,X} comes before {b,B}). This is because we're in a fold, which is tail recursive. The whole list is reversed given how we accumulate it, so we must put the last one traversed before the others.

Finally, we use `erlang:min/2` to find the shortest path. It might sound weird to use such a comparison function on tuples, but remember that every Erlang term can be compared to any other! Because the length is the first element of the tuple, we can sort them that way.

What's left to do is to stick that function into a fold:

```
optimal_path(Map) ->
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[]}, {0,[]}}, Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
                      hd(element(2,B)) =/= {x,0} -> B
                   end,
    lists:reverse(Path).
```

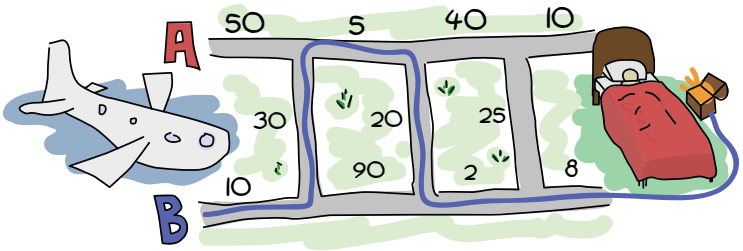
At the end of the fold, both paths should end up having the same distance, except one is going through the final {x,0} segment. The `if` looks at the last visited element of both paths and returns the one that doesn't go through {x,0}. Picking the path with the fewest steps (compare with `length/1`) would also work. Once the shortest path has been selected, it is reversed (it was built in a tail-recursive manner; you *must* reverse it). You can then display it to the world, or keep it secret and get your rich uncle's estate. To do that, we need to modify the `main` function to call `optimal_path/1`. Then it can be compiled.

```
main() ->
    File = "road.txt",
    {ok, Bin} = file:read_file(File),
    optimal_path(parse_map(Bin)).
```

And we can try it as follows:

```
1> c(road).
{ok,road}
2> road:main().
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

Oh, look! We have the right answer. Great job!
Or, to put it in a visual way:



Running the Program Without the Erlang Shell

You know what would be really useful? Being able to run our program from outside the Erlang shell. To do this, we'll need to change our `main` function again:

```
main([FileName]) ->
    {ok, Bin} = file:read_file(FileName),
    Map = parse_map(Bin),
    io:format("~p~n",[optimal_path(Map)]),
    erlang:halt().
```

The `main` function now has an arity of 1, needed to receive parameters from the command line. We've also added the function `erlang:halt/0`, which will shut down the Erlang VM after being called. We've wrapped the call to `optimal_path/1` into `io:format/2` because that's the only way to have the text visible outside the Erlang shell.

With all of this, your *road.erl* file should now look like this (minus comments):

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

main([FileName]) ->
    {ok, Bin} = file:read_file(FileName),
```

```
    Map = parse_map(Bin),
    io:format("~p~n",[optimal_path(Map)]),
    erlang:halt(0).

%% Transform a string into a readable map of triples.
parse_map(Bin) when is_binary(Bin) ->
    parse_map(binary_to_list(Bin));
parse_map(Str) when is_list(Str) ->
    Values = [list_to_integer(X) || X <- string:tokens(Str,"\r\n\t ")],
    group_vals(Values, []).

group_vals([], Acc) ->
    lists:reverse(Acc);
group_vals([A,B,X|Rest], Acc) ->
    group_vals(Rest, [{A,B,X} | Acc]).

%% Picks the best of all paths, woo!
optimal_path(Map) ->
    {A,B} = lists:foldl(fun shortest_step/2, {{0,[]}, {0,[]}}, Map),
    {_Dist,Path} = if hd(element(2,A)) =/= {x,0} -> A;
                      hd(element(2,B)) =/= {x,0} -> B
                   end,
    lists:reverse(Path).

%% actual problem solving
%% Change triples of the form {A,B,X}
%% where A,B,X are distances and a,b,x are possible paths
%% to the form {DistanceSum, PathList}.
shortest_step({A,B,X}, {{DistA,PathA}, {DistB,PathB}}) ->
    OptA1 = {DistA + A, [{a,A}|PathA]},
    OptA2 = {DistB + B + X, [{x,X}, {b,B}|PathB]},
    OptB1 = {DistB + B, [{b,B}|PathB]},
    OptB2 = {DistA + A + X, [{x,X}, {a,A}|PathA]},
    {erlang:min(OptA1, OptA2), erlang:min(OptB1, OptB2)}.
```

And we can run the code like this:

```
$ erlc road.erl
$ erl -noshell -run road main road.txt
[{b,10},{x,30},{a,5},{x,20},{b,2},{b,8}]
```

And yep, we get the right answer! That's pretty much all you need to do to get things to work, though you could also make yourself a bash/batch script to wrap the line into a single executable, or you could check out the `escript` command (which provides scripting support) to get similar results.

As you've seen with these two exercises, solving problems is much easier when you break them into small parts that you can solve individually before piecing everything together. It's also important not to dive right in before you fully understand the problem, since this will usually end up creating more work in the long run. Finally, a few tests are always appreciated. They allow you to make sure everything works initially and will return the same results down the road, even if you change the particulars of the implementation.

USING ESCRIPT

The Erlang `escript` command provides a simple way to run Erlang programs without starting the `erl` application directly. Basically, the command takes a module and allows you to interpret it without needing to compile it first.

The structure of the module remains similar to what you had before, but you need to change its head. Instead of having a `-module(`*`Name`*`)` attribute, the following is required:

```
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -pa 'ebin/' [Other erl Arguments]
main([StringArguments]) ->
    ...
```

The function `main/1` will automatically be called when you start the script, either as `./`*`script-name`*`.erl` or `escript` *`script-name`*`.erl` (the latter makes it easier to run on Windows). The module will run as a normal script.

If you want the benefits of `escript` without needing to interpret the code (which is slower) and would prefer compiling the code, just add the `-mode(compile).` module attribute somewhere in the file.

To find out more about `escript`, read the documentation that comes with Erlang, which is also available online at *http://erlang.org/doc/man/escript.html*.

9

A SHORT VISIT TO COMMON DATA STRUCTURES

Chances are that you now understand the functional subset of Erlang pretty well and could read many programs without a problem. However, I bet it's still a bit hard to think about how to build a real, useful program, even though Chapter 8 was about solving problems in a functional manner. Well, that's how I felt at this point in my Erlang studies—if you're doing better, congratulations!

So far, we've covered a bunch of topics, including most of the basic data types, the shell, how to write modules and functions (with recursion), different ways to compile, how to control the flow of the program, how to handle exceptions, and how to abstract away some common operations. We've also gone over how to store data with tuples, lists, and an incomplete implementation of a binary search tree. What we haven't talked about yet is the other data structures provided to the programmer in the Erlang standard library. This chapter fills that void, with information about records, key/value stores, sets, directed graphs, and queues.

Records

Records are, first of all, a hack. They are more or less an afterthought to the language and can have their share of inconveniences. However, they're still pretty useful when you have a small data structure and you want to access the attributes by name directly. Used this way, Erlang records are a lot like structs in C.

Defining Records

Records are declared as module attributes in the following manner:

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

-record(robot, {name,
                type=industrial,
                hobbies,
                details=[]}).
```

Here, we have a record representing robots with four fields: `name`, `type`, `hobbies`, and `details`. There are also default values for `type` and `details`, which are `industrial` and `[]`, respectively.

Here's how to create an instance of a record in the module `records`:

```
first_robot() ->
    #robot{name="Mechatron",
           type=handmade,
           details=["Moved by a small man inside"]}.
```

Let's try running the code:

```
1> c(records).
{ok,records}
2> records:first_robot().
{robot,"Mechatron",handmade,undefined,
       ["Moved by a small man inside"]}
```

Whoops! Here comes the hack! Erlang records are just syntactic sugar on top of tuples. Fortunately, there's a way to keep the illusion going. The Erlang shell has the command `rr(`*`Module`*`)`, which lets you load record definitions from *`Module`*. Try it with our `records` module.

```
3> rr(records).
[robot]
4> records:first_robot().
#robot{name = "Mechatron",type = handmade,
       hobbies = undefined,
       details = ["Moved by a small man inside"]}
```

NOTE *The `rr()` function can take more than a module name. It can also take a wildcard (like `rr("*")`) and a list as a second argument to specify which records to load.*

Ah, there! This makes it much easier to work with records. You'll notice that in `first_robot/0`, we did not define the `hobbies` field, and it has no default value in its declaration. By default, Erlang sets the value to `undefined` for you.

To see the behavior of the defaults we set in the `robot` definition, let's compile the following function:

```
car_factory(CorpName) ->
    #robot{name=CorpName, hobbies="building cars"}.
```

Now run it.

```
5> c(records).
{ok,records}
6> records:car_factory("Jokeswagen").
#robot{name = "Jokeswagen",type = industrial,
       hobbies = "building cars",details = []}
```

Now we have an industrial robot that likes to spend time building cars.

OTHER RECORD FUNCTIONS FOR THE ERLANG SHELL

Along with `rr()`, Erlang provides a few other functions to deal with records in the shell:

- Use `rd(`*`Name`*`, `*`Definition`*`)` to define a record in a manner similar to the `-record(`*`Name`*`, `*`Definition`*`)` function used in our module.
- Use `rf()` to "unload" all records.
- Use `rf(`*`Name`*`)` or `rf([`*`Names`*`])` to get rid of specific definitions.
- Use `rl()` to print all record definitions currently defined in the shell in a way that makes it easy to copy and paste them into the module. Use `rl(`*`Name`*`)` or `rl([`*`Names`*`])` to print only specific records.

Reading Values from Records

Simply writing records isn't very useful. We need a way to extract values from them. There are basically two ways to do this: with a special dot syntax or through pattern matching. Assuming you have the record definition for robots loaded, we'll take a look at the dot syntax first.

```
5> Crusher = #robot{name="Crusher", hobbies=["Crushing people","petting cats"]}.
#robot{name = "Crusher",type = industrial,
       hobbies = ["Crushing people","petting cats"],
       details = []}
```

```
6> Crusher#robot.hobbies.
["Crushing people","petting cats"]
```

Ugh—not a pretty syntax. This is due to the nature of records as tuples. Because they're just a kind of compiler trick, you need to include keywords to define which record goes with which variable; hence, the `#robot` part of `Crusher#robot.hobbies`. It's sad, but there's no way out of it. Worse than that, nested records can get pretty ugly:

```
7> NestedBot = #robot{details=#robot{name="erNest"}}.
#robot{name = undefined,type = industrial,
       hobbies = undefined,
       details = #robot{name = "erNest",type = industrial,
                        hobbies = undefined,details = []}}
8> (NestedBot#robot.details)#robot.name.
"erNest"
```

And no, the parentheses are not mandatory. You could also type `NestedBot#robot.details#robot.name`. For backward compatibility (with Erlang versions before R14A) and to suit my personal preferences, I tend to use the version with parentheses, because I think that they make the code more readable.

The following example further demonstrates the dependence of records on tuples.

```
9> #robot.type.
3
```

This outputs which element of the underlying tuple `type` is.

One redeeming feature of records is that you can use them in function heads to pattern match and also in guards. To see how this works, declare a new record at the top of the file, and then add the functions under the declaration.

```
-record(user, {id, name, group, age}).

%% Use pattern matching to filter.
admin_panel(#user{name=Name, group=admin}) ->
    Name ++ " is allowed!";
admin_panel(#user{name=Name}) ->
    Name ++ " is not allowed".

%% Can extend user without problem.
adult_section(U = #user{}) when U#user.age >= 18 ->
    %% Show stuff that can't be written in such a text.
    allowed;
adult_section(_) ->
    %% Redirect to Sesame Street site.
    forbidden.
```

The syntax to bind a variable to any field of a record is demonstrated in the `admin_panel/1` function (it's possible to bind variables to more than one field).

Regarding the `adult_section/1` function, note that you need to do *`SomeVar = #some_record{}`* in order to bind the whole record to a variable.

Then we do the compiling as usual.

```
10> c(records).
{ok,records}
11> rr(records).
[robot,user]
12> records:admin_panel(#user{id=1, name="ferd", group=admin, age=96}).
"ferd is allowed!"
13> records:admin_panel(#user{id=2, name="you", group=users, age=66}).
"you is not allowed"
14> records:adult_section(#user{id=21, name="Bill", group=users, age=72}).
allowed
15> records:adult_section(#user{id=22, name="Noah", group=users, age=13}).
forbidden
```

This shows that it's not necessary to match on all parts of the tuple, or even know how many there are when writing the function. We can match on only the age or the group, if that's what's needed, and forget about all the rest of the structure. If we were to use a normal tuple, the function definition might need to look a bit like `function({record, _, _, ICareAboutThis, _, _}) -> ...`. Then, whenever someone decided to add an element to the tuple, someone else (probably angry about it) would need to update all the functions where that tuple is used.

Updating Records

The following function illustrates how to update a record (they wouldn't be very useful otherwise).

```
repairman(Rob) ->
    Details = Rob#robot.details,
    NewRob = Rob#robot{details=["Repaired by repairman"|Details]},
    {repaired, NewRob}.
```

Now compile it.

```
16> c(records).
{ok,records}
17> records:repairman(#robot{name="Ulbert", hobbies=["trying to have feelings"]}).
{repaired,#robot{name = "Ulbert",type = industrial,
                 hobbies = ["trying to have feelings"],
                 details = ["Repaired by repairman"]}}
```

As you can see, the robot has been repaired. The syntax to update records is a bit special here. It looks like we're updating the record in place (`Rob#robot{`*`Field=NewValue`*`}`), but it's all compiler trickery to call the underlying `erlang:setelement/3` function.

Sharing Records

Because records can be useful and code duplication is annoying, Erlang programmers frequently share records across modules with the help of header files. Erlang *header files* are similar to their C counterparts. A header file is nothing but a snippet of code that gets added to the module as if it were written there in the first place.

Create a file named *records.hrl* with the following content:

```
%% This is a .hrl (header) file.
-record(included, {some_field,
                   some_default = "yeah!",
                   unimaginative_name}).
```

To include it in `records.erl`, just add the following line to the module:

```
-include("records.hrl").
```

And then add the following function to try it:

```
included() -> #included{some_field="Some value"}.
```

Now compile it as usual.

```
18> c(records).
{ok,records}
19> rr(records).
[included,robot,user]
20> records:included().
#included{some_field = "Some value",some_default = "yeah!",
          unimaginative_name = undefined}
```

Hooray! That's about it for records. As you've seen, their syntax is not pretty, and they're not much more than a hack, but they're relatively important for the maintainability of your code.

WARNING *You will often see open source software using the method shown here of having a project-wide* .hrl *file for records that are shared across all modules. While I felt obligated to document this use, I strongly recommend that you keep all record definitions local, within one module. If you want some other module to look at a record's innards, write functions to access its fields and keep its details as private as possible. This helps prevent name clashes, avoids problems when upgrading code, and just generally improves the readability and maintainability of your code.*

Key/Value Stores

Back in Chapter 5, we built a tree and then used it as a key/value store for an address book. That address book sucked. We couldn't delete or convert it to anything useful. It was a good demonstration of recursion, but not much more.

Now is the time to introduce you to a bunch of useful data structures and modules to store data under a certain key. I won't define what every function does, show entire examples, or go through all the modules, because you can easily find that information in Erlang's documentation. Consider me as "someone responsible for raising awareness about key/value stores in Erlang" (sounds like a good title—I just need one of those ribbons).

Stores for Small Amounts of Data

For storing small amounts of data, basically two types of data structures can be used: a *property list (proplist)* or an ordered *dictionary (orddict)*.

Proplists

A proplist is any list of tuples of the form [{*Key*,*Value*}]. Proplists are a weird kind of structure because that's the only rule that applies to them. In fact, the rules are so relaxed that the list can also contain Boolean values, integers, and whatever else you want. Here, we're interested in the idea of a tuple with a key and a value in a list.

To work with proplists, use the `proplists` module. It contains functions such as `proplists:delete/2`, `proplists:get_value/2`, `proplists:get_all_values/2`, `proplists:lookup/2`, and `proplists:lookup_all/2`. You can get their definitions from Erlang's documentation.

You'll notice there is no function to add or update an element of the list. This shows how loosely defined proplists are as a data structure. In fact, a proplist is more often appropriate when you need a list of properties. For example, we could describe a dog as the proplist [{`name, buddy`}, {`race, husky`}, `friendly`], where the value `friendly` is equivalent to {`friendly, true`}.

If you want to add an element to a proplist, you must use the cons operator to insert your element manually (`NewList = [NewElement|OldList]`). This works well even for updates, because the `proplists` module will look through the list in order and stop as soon as it finds a matching element. You can also use functions such as `lists:keyreplace/4` to update a proplist if you need to do it a lot, as this approach avoids making the proplist longer as time goes on. Using two modules for one small data structure is not the cleanest technique, but because proplists are so loosely defined, they're often used to deal with configuration lists.

Orddicts

If you want a more complete key/value store for small amounts of data, the `orddict` module is what you need. Orddicts are proplists with a taste for

formality. Each key can be there only once. The whole list is sorted so, on average, lookups are faster. The items need to respect a strict `{Key, Value}` structure. You're not expected to edit orddicts as lists, as with proplists, but to use the functional interface for all the operations you need.

Common functions for CRUD (Create, Read, Update, and Delete) usage include `orddict:store/3`, `orddict:find/2` (when you do not know whether the key is in the dictionaries), `orddict:fetch/2` (when you know it is there or that it must be there), and `orddict:erase/2`. You can create an orddict by using `orddict:new/0` or `orddict:from_list/1`. Again, you can look up these functions in the Erlang documentation.

WARNING *To create and manipulate the orddict, you might be tempted to manually modify the key/value list, but you should* always *use the functions provided by the* `orddict` *module to avoid ordering errors.*

Orddicts are generally a good compromise between complexity and efficiency for up to about 75 elements (see my benchmark, *keyval_benchmark.erl*, available with the rest of the code for this book). After that amount, you should switch to different key/value stores, such as the ones discussed next.

Larger Dictionaries: Dicts and GB Trees

Erlang provides two key/value structures to deal with larger amounts of data: *dictionaries (dicts)* and *general balanced (GB) trees*. Dicts have the same interface as orddicts: `dict:store/3`, `dict:find/2`, and `dict:fetch/2`, `dict:erase/2`. They also have every other function from `orddict`, such as `dict:map/2` and `dict:fold/2` (pretty useful to work on the whole data structure!). Dicts are thus very good choices for scaling up orddicts whenever it is needed.

GB trees, handled through the `gb_trees` module, have many more functions that give you more direct control over how the structure is to be used. There are basically two modes for `gb_trees`: the mode where you know your structure inside and out (I call this the *smart mode*), and the mode where you can't assume much about it (I call this the *naive mode*). In naive mode, the functions are `gb_trees:enter/2`, `gb_trees:lookup/2`, and `gb_trees:delete_any/2`. The related smart functions are `gb_trees:insert/3`, `gb_trees:get/2`, `gb_trees:update/3`, and `gb_trees:delete/2`. There is also `gb_trees:map/2`, which is a tree-based equivalent to `lists:map/2` (always a nice thing to have when you need it).

The disadvantage of naive functions over smart ones is that because GB trees are balanced trees, whenever you insert a new element (or delete a bunch of elements), the tree may need to balance itself. This can take time and memory (even in useless checks that end up changing nothing but seek

to make sure the tree is still balanced). The smart functions all assume that the key is present in the tree. This lets you skip all the safety checks and results in faster operations.

> **DON'T DRINK TOO MUCH KOOL-AID**
>
> What about code that requires data structures with only numeric keys? For that, most languages usually have arrays. Erlang has arrays, too. They allow you to access elements with numeric indices and to fold over the whole structure while possibly ignoring undefined slots. However, very few people use them.
>
> Erlang arrays, unlike their imperative counterparts, do not have such things as constant-time insertion or lookup. Instead, they are said to be *persistent*, as they allow no destructive updates. For this reason, they're usually slower than those in languages that support destructive assignment. People who know and use that type of array usually do so with a given set of algorithms and a precise style in mind. Erlang's arrays hardly allow that. They tend to sit in a dark corner, alone.
>
> Erlang programmers who need to do matrix manipulations and other jobs that require arrays tend to use concepts called *ports* to let other languages do the heavy lifting, or *C nodes*, *linked-in drivers*, and *native implemented functions (NIFs)*. See the Erlang documentation for more details.

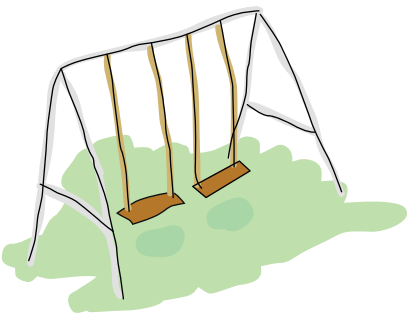
When should you use the `gb_trees` module rather than `dict` functions? Well, it's not a clear decision. As the benchmark module I wrote (*keyval_benchmark.erl*) shows, GB trees and dicts have somewhat similar performances in many respects. However, the benchmark demonstrates that dicts have the best read speeds, and the GB trees tend to be a little quicker on other operations.

Also note that while dicts have a fold function, GB trees don't. Instead, they have an iterator function, which returns a bit of the tree on which you can call `gb_trees:next(Iterator)` to get the following values in order. This means that you need to write your own recursive functions on top of using `gb_trees`, rather than using a generic fold. On the other hand, `gb_trees` lets you have quick access to the smallest and largest elements of the structure with `gb_trees:smallest/1` and `gb_trees:largest/1`. This is because a GB tree preserves the order of all elements inside of it, from the smallest to the largest. A dict, on the other hand, will not provide this ordering. As such, if you need to be able to traverse your key/value store in order, GB trees might be a good option.

So, your application's needs are what should govern which key/value store you choose. You'll need to consider factors such as how much data you have to store and what you need to do with it. Measure, profile, and benchmark to make sure.

NOTE *Some special key/value stores exist to deal with resources of different sizes. Such stores are ETS tables, Dets tables, and the Mnesia database. Their use is strongly related to the concepts of multiple processes and distribution, so we'll get to them in Chapter 25. I'm mentioning them now just to pique your curiosity and as a reference for those who are interested.*

A Set of Sets



If you've ever studied set theory in a mathematics class, you have an idea about what sets can do. If you haven't, you might want to skip this section.

Sets are groups of unique elements that you can compare and operate on—find which elements are in two groups, in none of them, in only one or the other, and so on. There are advanced operations that let you define relations and operate on these relations, and much more. I'm not going to dive into the theory here but just give you an idea of what is available.

Erlang has four main modules to deal with sets. This seems a bit weird at first, but it's because the implementers agreed that there was no "best" way to build a set. The four modules are as follows:

`ordsets`
: `ordsets` module sets are implemented as sorted lists. They're mainly useful for small sets, and are the slowest kind of set, but they have the simplest and most readable representation of all sets. Some of the many standard functions for them are `ordsets:new/0`, `ordsets:is_element/2`, `ordsets:add_element/2`, `ordsets:del_element/2`, `ordsets:union/1`, and `ordsets:intersection/1`.

`sets`
: `sets` (the module) is implemented on top of a structure similar to the one used by dicts. The `sets` module implements the same interface as `ordsets`, but its sets scale much better. Like dicts, they're especially good for read-intensive manipulations, such as checking whether some element is part of the set.

`gb_sets`
: `gb_sets` module sets are constructed above a GB tree structure similar to the one used in the `gb_trees` module. `gb_sets` is to `sets` what `gb_trees` is to `dict`: an implementation that is faster when considering operations other than reading, leaving you with more control. While `gb_sets`

implements the same interface as `sets` and `ordsets`, it adds more functions. As with `gb_trees`, we have smart versus naive functions, iterators, and quick access to the smallest and largest values.

`sofs`
: Sets of sets, created with the `sofs` module, are implemented with sorted lists, stuck inside a tuple with some metadata. This is the module to use if you want to have full control over relationships between sets and families, enforce set types, and so on. These sets are what you want if you need the mathematics concept of sets, rather than just groups of unique elements.

It's a bit confusing to have so many options available. Björn Gustavsson, from the Erlang/OTP team and programmer of Wings 3D, suggests using `gb_sets` in most circumstances, using `ordset` when you need a clear representation that you want to process with your own code, and using `sets` when you need the `=:=` operator (see *http://erlang.org/pipermail/erlang-questions/2010-March/050333.html*).

In any case, as with key/value stores, the best solution is usually to benchmark and see which approach best suits your application.

DON'T DRINK TOO MUCH KOOL-AID

While such a variety of sets can be seen as something great, some implementation details can be downright frustrating. As an example, `gb_sets`, `ordsets`, and `sofs` all use the `==` operator to compare values; if you have the numbers `2` and `2.0`, they'll be seen as the same number.

However, the `sets` module uses the `=:=` operator, which means you can't necessarily switch over every implementation as you wish. There are cases where you need one precise behavior, and at that point, you might lose the benefit of having multiple implementations.

Directed Graphs

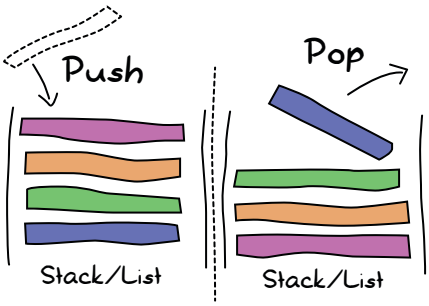
One other data structure intimately related to mathematics is the directed graph. Directed graphs in Erlang are implemented as two modules: `digraph` and `digraph_utils`. The `digraph` module basically allows the construction and modification of a directed graph—manipulating edges and vertices, finding paths and cycles, and so on. The `digraph_utils` module allows you to navigate a graph (postorder and preorder); test for cycles, arborescences, and trees; find neighbors; and so on.

Because directed graphs are closely related to set theory, the `sofs` module contains a few functions that let you convert families to directed graphs and directed graphs to families.

Because of the way the directed graphs modules are built, they aren't really appropriate without a good basic knowledge of either graphs or set theory. If you know your stuff and you are interested in learning more about these modules, you'll have no problem figuring them out by their standard documentation.

Queues

The `queue` module implements a double-ended first in, first out (FIFO) queue. Queues are implemented a bit as illustrated here: two lists (in this context, stacks) that allow you to both append and prepend elements rapidly.



Because a single list doesn't allow efficiently adding *and* removing elements from both ends at once (it's only fast to add and remove the head), the idea behind the `queue` module is that if you have two lists, then you can use one to add elements and one to remove elements. One of the lists then behaves as one end of the queue, where you push values, and the other list acts as the other end, where you pop them. When the latter is empty, you take the former and reverse it, and it becomes the new list to pop from. This allows an efficient queue implementation on the average of all operations over the life of the queue.

The `queue` module has different functions that are separated into three interfaces (or APIs) of varying complexity:

Original API

The original API contains the functions at the base of the queue concept. These include `new/0`, for creating empty queues; `in/2`, for inserting new elements; and `out/1`, for removing elements. It also has functions to convert to lists, reverse the queue, check if a particular value is part of the queue, and so on.

Extended API

The extended API mainly adds some introspection power and flexibility. It lets you do things such as look at the front of the queue without removing the first element (`get/1` or `peek/1`), remove elements without caring about them (`drop/1`), and so on. These functions are not essential to the concept of queues, but they're still useful in general.

Okasaki API

The Okasaki API is a bit weird. It's derived from Chris Okasaki's *Purely Functional Data Structures* (Cambridge University Press, 1999). The API provides operations similar to those available in the other APIs, but some of the function names are written backward, and the whole thing is relatively peculiar. Unless you have a specific reason for using this API, I wouldn't bother with it.

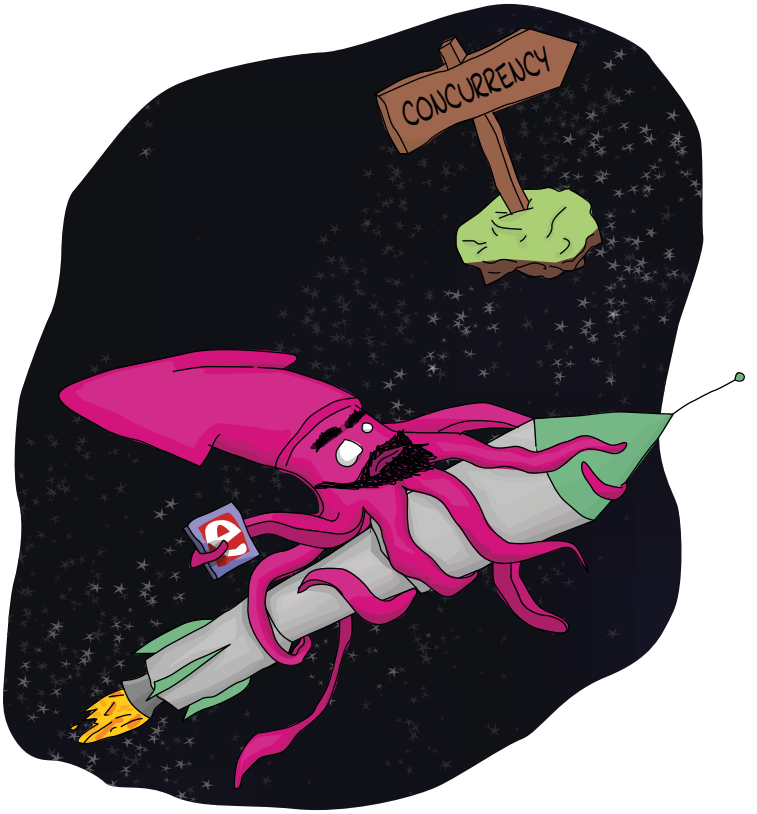
You'll generally want to use queues when you need to ensure that the first item ordered is indeed the first one processed. So far, the examples I've shown mainly used lists as accumulators that would then be reversed. In cases where you can't just do all the reversing at once, and elements are frequently added, the `queue` module is what you want. (Well, you should test and measure first. Always test and measure first!)

End of the Short Visit

That's about it for our trip through the most common data structures of Erlang. Thank you for having kept your arms inside the vehicle the whole time. There are a few more data structures available to solve different problems. Here, I've covered those that you're likely to encounter or need the most, given the strengths of general use cases of Erlang. I encourage you to explore the standard library and the extended one, too, to find more information.

You might be glad to learn that this completes our trip into sequential (functional) Erlang. I know a lot of people get into Erlang to see all the concurrency and processes and whatnot. This is understandable, given these are the areas where Erlang really shines. It offers supervision trees, fancy error management, distribution, and more. I know that I've been very impatient to write about these subjects, so I guess some readers are very impatient to read about them.

However, it makes more sense to be comfortable with functional Erlang before moving on to concurrent Erlang. Now we can focus on all the new concepts. Here we go!


CONCURRENCY

10

THE HITCHHIKER'S GUIDE TO CONCURRENCY

Far out in the uncharted backwaters of the unfashionable beginning of the 21st century lies a small subset of human knowledge. Within this subset of human knowledge is an utterly insignificant little discipline whose Von Neumann–descended architecture is so amazingly primitive that it is still thought that RPN calculators are a pretty neat idea.

This discipline has—or rather had—a problem, which was this: Most of the people studying it were unhappy for pretty much of the time when trying to write parallel software. Many solutions were suggested for this problem, but most of these were largely concerned with the handling of little pieces of logic called *locks* and *mutexes* and whatnot, which is odd because on the whole, it wasn't the small pieces of logic that needed parallelism.

And so the problem remained. Lots of people were mean, and most of them were miserable, even those with RPN calculators.

Many were increasingly of the opinion that they'd all made a big mistake in trying to add parallelism to their programming languages, and that no program should have ever left its initial thread.

NOTE *Parodying* The Hitchhiker's Guide to the Galaxy *is fun. Read the book if you haven't already. It's good!*

Don't Panic

Hi. Today (or whatever day you are reading this—even tomorrow), I'm going to tell you about concurrent Erlang. Chances are you've read about or dealt with concurrency before. You might also be curious about the emergence of multicore programming. Anyway, the probabilities are high that you're reading this book because of all the talk about concurrency going on these days.

A warning though: This chapter is mostly theory. If you have a headache, a distaste for programming language history, or a desire just to program, you might be better off skipping to the end of the chapter, or moving on to the next one (where more practical content is shown).

In the Introduction to this book, I explained that Erlang's concurrency is based on message passing and the actor model, using the example of people communicating with nothing but letters. We'll get to more details about concurrency later in this chapter, but first, it is important to define the difference between *concurrency* and *parallelism*.

In many places, both words refer to the same concept, but in the context of Erlang, *concurrency* refers to having many actors running independently but not necessarily all at the same time, while *parallelism* is having actors running at exactly the same time. This is how I'll use these terms in this text, but don't be surprised if other sources or people use the same terms to mean different things. There doesn't seem to be any consensus on these definitions in the computer science world.

Erlang had concurrency from the beginning, even when everything was done on a single core processor in the 1980s. Each Erlang process would have its own slice of time to run, much like desktop applications did before multicore systems. Parallelism was still possible back then; all you needed to do was to have a second computer running the code and communicating with the first one. Even then, only two actors could be run in parallel in this setup. Nowadays, multicore systems allow for parallelism on a single computer (some industrial chips have many dozens of cores), and Erlang takes full advantage of this possibility.

DON'T DRINK TOO MUCH KOOL-AID

The distinction between concurrency and parallelism is important to make, because many programmers hold the belief that Erlang was ready for multicore computers years before it actually was. Erlang was adapted to true *symmetric multiprocessing (SMP)* in the mid-2000s, and only got most of the implementation right with the R13B release of the language in 2009. Before that, SMP often needed to be disabled to avoid performance losses. Then to get parallelism on a multicore computer without SMP, you would need to start many instances of the VM.

An interesting fact is that because Erlang concurrency is all about isolated processes, it took no conceptual change at the language level to bring true parallelism to the language. All the changes were transparently done in the VM, away from the eyes of the programmers.

Concurrency Concepts

Back in the day, Erlang's development as a language was extremely quick, with frequent feedback from engineers working on telephone switches in Erlang itself. These interactions proved process-based concurrency and asynchronous message passing to be a good way to model the problems the engineers faced. Moreover, the telephony world already had a certain culture going toward concurrency before Erlang came to be. This was inherited from PLEX, a language created earlier at Ericsson, and AXE, a switch developed with it. Erlang followed this tendency and attempted to improve on previous tools available.

Erlang had a few requirements to satisfy before being considered good. The main ones were being able to scale up and support many thousands of users across many switches, and to achieve high reliability—to the point of never stopping the code.

Scalability

Some properties were seen as necessary to achieve scalability. Because users would be represented as processes that reacted only upon the occurrence of certain events (such as receiving a call or hanging up), an ideal system would support processes doing small computations, switching between them very quickly as events came through. To make the system efficient, it

made sense for processes to be started and destroyed very quickly. Having them be lightweight was mandatory to achieve this efficiency. It was also mandatory because you didn't want to have things like process pools (a fixed amount of processes you split the work among). Instead, it would be much easier to design programs that could use as many processes as they needed.

NOTE *Another important aspect of scalability is to be able to bypass your hardware's limitations. There are two ways to do this: make the hardware better or add more hardware. The first option is useful up to a certain point, after which it becomes extremely expensive. The second option is usually cheaper and requires you to add more computers to do the job. This is where distribution can be useful to have as a part of your language.*

Because telephony applications needed a lot of reliability, it was decided that the cleanest approach was to forbid processes from sharing memory. Shared memory could leave things in an inconsistent state after some crashes (especially on data shared across different nodes) and had some complications. Instead, processes should communicate by sending messages where all the data is copied. This might end up being slower but safer.

Fault Tolerance

The first writers of Erlang always kept in mind that failure is common. You can try to prevent bugs all you want, but most of the time, some will still creep in. And even if by some miracle your code doesn't have any bugs, nothing can stop the eventual hardware failure. Therefore, the idea is to find good ways to handle errors and problems, rather than trying to prevent them all.

It turns out that taking the design approach of multiple processes with message passing was a good idea, because error handling could be grafted onto it relatively easily. Take lightweight processes (made for quick restarts and shutdowns) as an example. Some studies proved that the main sources of downtime in large-scale software systems are intermittent or transient bugs (see *http://dslab.epfl.ch/pubs/crashonly/*). Also, there's a principle that says that errors that corrupt data should cause the faulty part of the system to die as fast as possible in order to avoid propagating errors and bad data to the rest of the system.

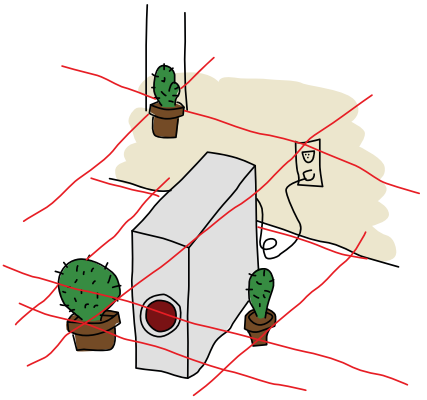
Another concept here is that a system can terminate in many different ways, two of which are clean shutdowns and crashes (terminating with an unexpected error).

Here, the worst case is obviously the crash. A safe solution would be to make sure all crashes are the same as clean shutdowns. This can be done through practices such as *shared-nothing* (all memory is separated for subparts of the system) and single assignment (which can further isolate a process's memory), avoiding locks (if certain data was locked during a crash, it would keep other processes from accessing the data or leave it in an inconsistent state), and other safeguards, which were all part of Erlang's design.

The ideal solution in Erlang is thus to kill processes as fast as possible to avoid data corruption and transient bugs. Lightweight processes are a key element in this. Further error-handling mechanisms are also part of the language to allow processes to monitor other processes (which are described in Chapter 12), in order to know when processes die and to decide what to do about it.

Assuming that restarting processes quickly is enough to deal with crashes, the next problem is handling hardware failures. How do you make sure your program keeps running when someone kicks the computer it's running on? Although a fancy defense mechanism consisting of laser detection and strategically placed cacti could do the job for a while, it would not last forever. The solution is simply to have your program running on more than one computer at once—something that's necessary for scaling anyway. This is another advantage of independent processes with no communication channel outside message passing. You can have them working the same way whether they're local or on a different computer, making fault tolerance through distribution nearly transparent to the programmer.

Being distributed has direct consequences on how processes can communicate with each other. One of the biggest hurdles of distribution is that you can't assume that because a node (a remote computer) was there when you made a function call, it will still be there for the whole transmission of the call, or that it will even execute the call correctly. Someone tripping over a cable or unplugging the machine would leave your application hanging. Or maybe it would make it crash. Who knows?

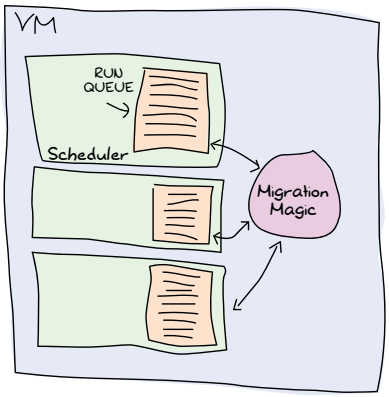


Well, it turns out the choice of asynchronous message passing was a good design pick there, too. Under the processes-with-asynchronous-messages model, messages are sent from one process to a second one and stored in a *mailbox* inside the receiving process until they are taken out to be read. It's important to mention that messages are sent without even checking if the receiving process exists, because it would not be useful to do so. As implied in the previous paragraph, it's impossible to know if a process will crash between the time a message is sent and received. And if the message is received, it's impossible to know whether the message will be acted upon or if the receiving process will die before that. Asynchronous messages allow safe remote function calls because there is no assumption about what will happen; the programmer is the one to know. If you need to have a confirmation of delivery, you must send a second message as a reply to the original process. This message will have the same safe semantics, and so will any program or library you build on this principle.

Concurrency Implementation

So now you know why it was decided that lightweight processes with asynchronous message passing were the approach to take for Erlang. But how could Erlang's implementers make this work?

First of all, the operating system can't be trusted to handle the processes. Operating systems have many different ways to handle processes, and their performance varies a lot. Most, if not all, of them are too slow or heavy for what is needed by standard Erlang applications. By handling processes in the VM, the Erlang implementers kept control of optimization and reliability. Nowadays, Erlang's processes take about 300 words of memory each and can be created in a matter of microseconds—not something currently doable on major operating systems.



To handle all these potential processes your programs could create, the VM starts one thread per core that acts as a *scheduler*. Each of these schedulers has a *run queue*, or a list of Erlang processes on which to spend a slice of time. When one of the schedulers has too many tasks in its run queue, some tasks are migrated to another queue. This means that each Erlang VM takes care of doing all the load balancing, and the programmer doesn't need to worry about it. The VM also does some other optimizations, such as limiting the rate at which messages can be sent to overloaded processes in order to regulate and distribute the load.

All the hard stuff is in there, managed for you. That is what makes it easy to go parallel with Erlang. Going parallel means your program should go twice as fast if you add a second core, four times faster if there are four cores, and so on, right? It depends. Such a phenomenon is named *linear scaling* in relation to speed gain versus the number of cores or processors (see the graph in the next section). In real life, there is no such thing as a free lunch (well, maybe at funerals, but someone, somewhere, still has to pay).

Not Entirely Unlike Linear Scaling

The difficulty of obtaining linear scaling is not due to the language itself, but rather to the nature of the problems to solve. Problems that scale very well are often said to be *embarrassingly parallel*. If you look up "embarrassingly parallel problems" on the Internet, you're likely to find examples such as ray-tracing (a method to create 3D images), brute-forcing searches in cryptography, and weather prediction.

From time to time, messages pop up in IRC channels, forums, and mailing lists asking if Erlang could be used to solve that kind of problem, or if it could be used to program on a graphical processing unit (GPU).

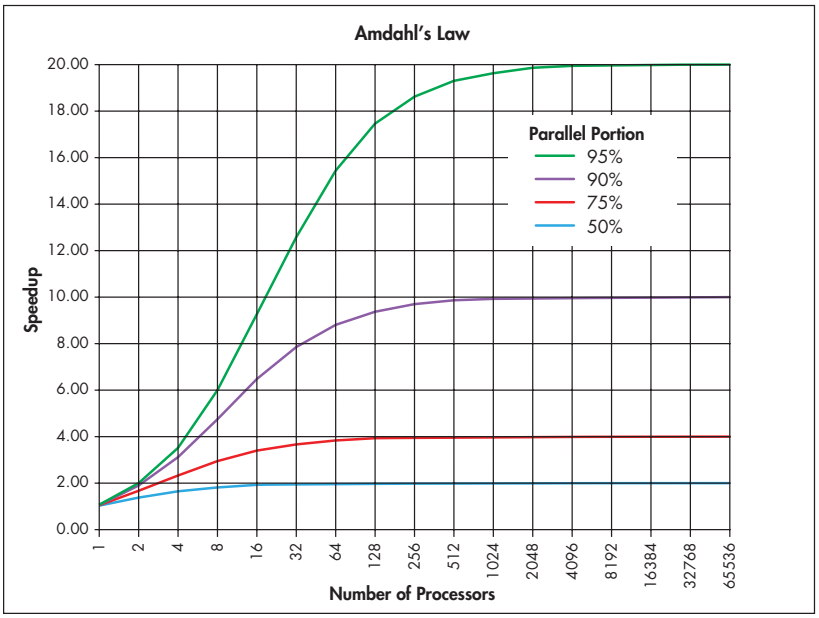
The answer is almost always no. The reason is relatively simple: All these problems usually involve numerical algorithms with a lot of data crunching. Erlang is not very good at this.

Erlang's embarrassingly parallel problems are present at a higher level. Usually, they have to do with concepts such as chat servers, phone switches, web servers, message queues, web crawlers, or any other application where the work done can be represented as independent logical entities (actors, anyone?). This kind of problem can be solved efficiently with close-to-linear scaling.

Many problems will never show such scaling properties. In fact, you need only one centralized sequence of operations to lose it all. *Your parallel program goes only as fast as its slowest sequential part.* An example of that phenomenon is observable any time you go to a mall. Hundreds of people can be shopping at once, rarely interfering with each other. Then once it's time to pay, queues form as soon as there are fewer cashiers than there are customers ready to leave. It would be possible to add cashiers until one exists for each customer, but then you would need a door for each customer, because the shoppers couldn't get inside or outside the mall all at once.

To put this another way, even though customers could pick each of their items in parallel and take as much time to shop whether they're alone or one of a thousand in the store, they would still need to wait to pay. Therefore, their shopping experience could never be shorter than the time it takes them to wait in the queue and pay.

A generalization of this principle is called *Amdahl's law*. It indicates how much of a speedup you can expect your system to have when you add parallelism to it, and in what proportion:



(Adapted from an image created by Daniel; used under a Creative Commons license. Original can be found at http://en.wikipedia.org/wiki/File:AmdahlsLaw.svg.)

According to Amdahl's law, code that is 50 percent parallel can never get faster than twice what it was before, and code that is 95 percent parallel can theoretically be expected to be about 20 times faster if you add enough processors. What's interesting to see on this graph is how getting rid of the last few sequential parts of a program allows a relatively huge theoretical speedup compared to removing as much sequential code in a program that is not very parallel to begin with.

DON'T DRINK TOO MUCH KOOL-AID

Parallelism is *not* the answer to every problem. In some cases, going parallel will even slow down your application. This can happen when your program is 100 percent sequential but still uses multiple processes.

One of the best examples of this is the *ring benchmark*. A ring benchmark is a test where many thousands of processes will pass a piece of data to one after the other in a circular manner. Think of it as a game of telephone. In this benchmark, only one process at a time does something useful, but the Erlang VM still spends time distributing the load across cores and giving every process its share of time.

This plays against many common hardware optimizations and makes the VM spend time doing useless stuff.

This load distribution often makes purely sequential applications run much slower on many cores than on a single one. If this kind of algorithm is central to your system, disabling SMP (`$ erl -smp disable`) might be a good idea. However, in other cases, sequential algorithms that aren't central to the execution of the whole program will usually be drowned by other events. In these cases, disabling SMP shouldn't have a big impact.

So Long and Thanks for All the Fish!

Of course, this chapter would not be complete if it didn't show the three primitives required for concurrency in Erlang: spawning new processes, sending messages, and receiving messages. In practice, more mechanisms are required for making really reliable applications; but for now, these three will suffice.

Spawning Processes

I've skirted around the issue a lot but have yet to explain what a process really is. It's actually nothing but a function. A process runs a function, and once it's finished, it disappears. Technically, a process also has some hidden state (such as a mailbox for messages), but functions are our focus for now.

To start a new process, Erlang provides the function `spawn/1`, which takes a single function and runs it:

```
1> F = fun() -> 2 + 2 end.
#Fun<erl_eval.20.67289768>
2> spawn(F).
<0.44.0>
```

The result of `spawn/1` (`<0.44.0>`) is called a *process identifier*, often just written as *pid* (the form I'll use), *Pid*, or *PID* by the Erlang community. The pid is an arbitrary value representing any process that exists (or might have existed) at some point in the VM's life. It is used as an address to communicate with the process.

You'll notice that we can't see the result of function `F`. We get only its pid. That's because processes do not return anything.

How can we see the result of `F` then? Well, there are two ways. The easiest one is to just output whatever we get:

```
3> spawn(fun() -> io:format("~p~n",[2 + 2]) end).
4
<0.46.0>
```

This isn't practical for a real program, but it is useful for seeing how Erlang dispatches processes. Fortunately, using `io:format/2` is enough to let us experiment. We'll quickly start 10 processes and pause each of them for a while with the help of the function `timer:sleep/1`, which takes an integer value *N* and waits for *N* milliseconds before resuming. After the delay, the value present in the process is output:

```
4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.
#Fun<erl_eval.6.13229925>
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,10)].
[<0.273.0>,<0.274.0>,<0.275.0>,<0.276.0>,<0.277.0>,
 <0.278.0>,<0.279.0>,<0.280.0>,<0.281.0>,<0.282.0>]
2
1
4
3
5
8
7
6
10
9
```

The order doesn't make sense. Welcome to parallelism! Because the processes are running at the same time, the ordering of events isn't guaranteed anymore. That's because the Erlang VM uses many tricks to decide

which process to run, making sure each gets a good share of time. Many Erlang services are implemented as processes, including the shell you're typing in. Your processes must be balanced with those the system itself needs, and this might be the cause of the weird ordering.

> **SYMMETRIC MULTIPROCESSING AND YOU**
>
> The results are similar whether or not SMP is enabled. To prove this, you can just test it by starting the Erlang VM with `$ erl -smp disable`.
>
> To see if your Erlang VM runs with or without SMP support, start a new VM without any options and look for the first line output. If you can spot the text `[smp:2:2]`, it means you're running with SMP enabled, and that you have two run queues running on two cores. If you don't see it, that means you're running with SMP disabled.
>
> The `[smp:2:2]` means that two cores are available, with two schedulers (each having a run queue). In earlier versions of Erlang, you could have multiple schedulers, but with only one shared run queue for all of them. Since R13B, there is one run queue per scheduler, which allows for better parallelism.

To prove the shell itself is implemented as a regular process, let's use the BIF `self/0`, which returns the pid of the current process:

```
6> self().
<0.41.0>
7> exit(self()).
** exception exit: <0.41.0>
8> self().
<0.285.0>
```

And the pid changes because the process has been restarted.

The next concern is how to send messages around, because no one wants to be stuck with outputting the resulting values of processes all the time, and then entering them by hand in other processes (at least, I know I don't).

Sending Messages

The next primitive required to do message passing is the operator `!`, also known as the *bang* symbol. On the left-hand side, it takes a pid; on the right-hand side, it takes any Erlang term. The term is then sent to the process represented by the pid, which can access it. Here's an example:

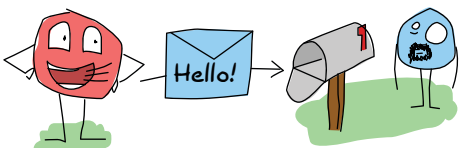
```
9> self() ! hello.
hello
```

The message has been put in the process's mailbox, but it hasn't been read yet. The second `hello` shown here is the return value of the send function. This means it is possible to send the same message to many processes by doing this:

```
10> self() ! self() ! double.
double
```

This is equivalent to `self() ! (self() ! double)`.

Something to note about a process's mailbox is that the messages are kept in the order they are received. Every time a message is read, it is taken out of the mailbox. Again, this is a bit similar to the analogy of people writing letters.



To see the contents of the current mailbox, you can use the `flush()` command while in the shell:

```
11> flush().
Shell got hello
Shell got double
Shell got double
ok
```

The `flush/0` function is just a shortcut that outputs received messages. This means we still can't bind the result of a process to a variable, but at least we know how to send it from a process to another one and check if it has been received.

Receiving Messages

Sending messages that no one will read is as useful as writing emo poetry (in other words, not very useful). This is why we need the `receive` expression. Rather than playing too long in the shell, we'll write a short program about dolphins to demonstrate how receiving messages works. Here's our new program:

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

dolphin1() ->
    receive
        do_a_flip ->
            io:format("How about no?~n");
        fish ->
            io:format("So long and thanks for all the fish!~n");
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
     end.
```

As you can see, `receive` is syntactically similar to `case ... of`. In fact, the patterns work exactly the same way, except they bind variables coming from messages rather than the expression between `case` and `of`. The `receive` expressions can also have guards. Here's their general syntax:

```
receive
    Pattern1 when Guard1 -> Expr1;
    Pattern2 when Guard2 -> Expr2;
    Pattern3 -> Expr3
end
```

Knowing this, we can now compile the module, run it, and start communicating with dolphins:

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin = spawn(dolphins, dolphin1, []).
<0.40.0>
13> Dolphin ! "oh, hello dolphin!".
Heh, we're smarter than you humans.
"oh, hello dolphin!"
14> Dolphin ! fish
fish
```

Here, we introduce a new way of spawning with `spawn/3`. Rather than taking a single function, `spawn/3` takes the module, function, and its arguments as its own arguments. Once the function is running, the following events take place:

1. The function hits the `receive` expression. Given that the process's mailbox is empty, our dolphin waits until it gets a message.
2. The message `"oh, hello dolphin!"` is received. The function tries to pattern match against `do_a_flip`. This fails, and so the pattern `fish` is tried, and this also fails. Finally, the message meets the catchall clause (`_`) and matches.
3. The process outputs the message "Heh, we're smarter than you humans."

Note that if the first message we sent works, the second provokes no reaction whatsoever from the process `<0.40.0>`. This is due to the fact that once our function output `Heh, we're smarter than you humans.`, it terminated, and so did the process. We'll need to restart the dolphin:

```
8> f(Dolphin).
ok
9> Dolphin = spawn(dolphins, dolphin1, []).
<0.53.0>
10> Dolphin ! fish.
So long and thanks for all the fish!
fish
```

And this time, the `fish` message works.

Wouldn't it be useful to be able to receive a reply from the dolphin rather than needing to use `io:format/2`? Of course it would! (Why am I even asking?)

I mentioned earlier in this chapter that the only way to know if a process has received a message is to send a reply. Our dolphin process will need to know who to reply to. This works in the same way as it does with the postal service. If we want someone to answer our letter, we need to add our address. In Erlang terms, this is done by packaging a process's pid in a tuple, given that messages are otherwise anonymous. The end result is a message that looks a bit like {*Pid, Message*}. Let's create a new dolphin function that will accept such messages:

```
dolphin2() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?";
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n")
    end.
```

As you can see, rather than accepting `do_a_flip` and `fish` for messages, we now require a variable `From`. That's where the pid will go.

```
11> c(dolphins).
{ok,dolphins}
12> Dolphin2 = spawn(dolphins, dolphin2, []).
<0.65.0>
13> Dolphin2 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
14> flush().
Shell got "How about no?"
ok
```

It seems to work pretty well. We can receive replies to messages we sent (we need to add an address to each message, a bit like an e-mail's Reply To field), but we still need to start a new process for each call. Recursion is the way to solve this problem. We just need the function to call itself so it never ends and always expects more messages. Here's a `dolphin3/0` function that puts this in practice:

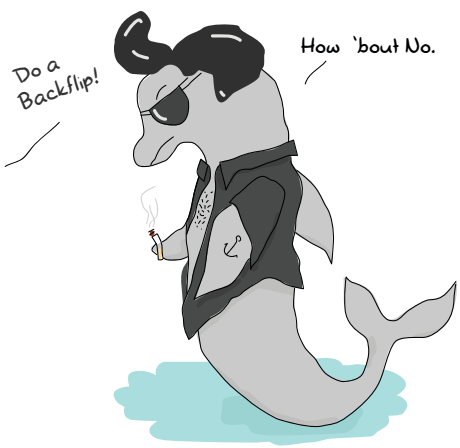
```
dolphin3() ->
    receive
        {From, do_a_flip} ->
            From ! "How about no?",
            dolphin3();
```

```
        {From, fish} ->
            From ! "So long and thanks for all the fish!";
        _ ->
            io:format("Heh, we're smarter than you humans.~n"),
            dolphin3()
    end.
```

Here, the catchall clause and the `do_a_flip` clause both loop with the help of `dolphin3/0`. Note that the function will not blow the stack because it is tail recursive. As long as only these messages are sent, the dolphin process will loop indefinitely. However, if we send the `fish` message, the process will stop:

```
15> Dolphin3 = spawn(dolphins, dolphin3, []).
<0.75.0>
16> Dolphin3 ! Dolphin3 ! {self(), do_a_flip}.
{<0.32.0>,do_a_flip}
17> flush().
Shell got "How about no?"
Shell got "How about no?"
ok
18> Dolphin3 ! {self(), unknown_message}.
Heh, we're smarter than you humans.
{<0.32.0>,unknown_message}
19> Dolphin3 ! Dolphin3 ! {self(), fish}.
{<0.32.0>,fish}
20> flush().
Shell got "So long and thanks for all the fish!"
ok
```

And that's it for *dolphins.erl*. As you see, it does respect our expected behavior of replying once for every message and continuing execution afterwards, except for the `fish` call. The dolphin got fed up with our crazy human antics and left us for good.



There you have it. This is the core of all of Erlang's concurrency. We've covered processes and basic message passing. There are more concepts to understand in order to make truly useful and reliable programs. We'll look at some of these in the following chapters.

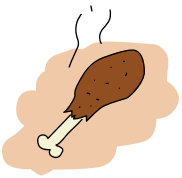
11

MORE ON MULTIPROCESSING

The examples shown in Chapter 10 were suitable for demonstrative purposes, but they won't take you very far in your own projects. It's not that the examples were bad; it's just that there's no huge advantage to processes and actors if they're just functions with messages. To reap the benefits, we need to be able to hold state in a process.

In this chapter, we will apply the concurrency concepts and primitives to practical examples that are able to hold state.

State Your State



Let's first create a function in a new *kitchen.erl* module that will let a process act like a refrigerator. The process will allow two operations: storing food in the fridge and taking food from the fridge. It should only be possible to take food that has been stored beforehand, and only as many times as it was stored. The following function can act as the base for our process:

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

fridge1() ->
    receive
        {From, {store, _Food}} ->
            From ! {self(), ok},
            fridge1();
        {From, {take, _Food}} ->
            %% uh....
            From ! {self(), not_found},
            fridge1();
        terminate ->
            ok
    end.
```

Something's wrong here. When we ask to store the food, the process should reply with `ok`, but there is nothing actually storing the food; `fridge1()` is called, and then the function starts from scratch, without state. Also, when we call the process to take food from the fridge, there is no state to take it from, and so the only reply is `not_found`. In order to store and take food items, we'll need to add state to the function.

With the help of recursion, the state of a process can be held entirely in the parameters of the function. In the case of our fridge process, one possibility would be to store all the food as a list, and then look in that list when someone needs to eat something:

```
fridge2(FoodList) ->
    receive
        {From, {store, Food}} ->
            From ! {self(), ok},
            fridge2([Food|FoodList]);
        {From, {take, Food}} ->
            case lists:member(Food, FoodList) of
                true ->
                    From ! {self(), {ok, Food}},
                    fridge2(lists:delete(Food, FoodList));
```

```
                false ->
                    From ! {self(), not_found},
                    fridge2(FoodList)
            end;
        terminate ->
            ok
    end.
```

Notice that `fridge2/1` takes one argument, `FoodList`. You can see that when we send a message that matches `{From, {store, Food}}`, the function will add `Food` to `FoodList` before recursing. Once that recursive call is made, it will then be possible to retrieve the same item. In fact, we've implemented that here.

The function uses `lists:member/2` to check whether `Food` is part of `FoodList`. Depending on the result, the item is sent back to the calling process (and removed from `FoodList`) or `not_found` is sent:

```
1> c(kitchen).
{ok,kitchen}
2> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.51.0>
3> Pid ! {self(), {store, milk}}.
{<0.33.0>,{store,milk}}
4> flush().
Shell got {<0.51.0>,ok}
ok
```

Storing items in the fridge seems to work. Now let's try to store something else and then take it from the fridge:

```
5> Pid ! {self(), {store, bacon}}.
{<0.33.0>,{store,bacon}}
6> Pid ! {self(), {take, bacon}}.
{<0.33.0>,{take,bacon}}
7> Pid ! {self(), {take, turkey}}.
{<0.33.0>,{take,turkey}}
8> flush().
Shell got {<0.51.0>,ok}
Shell got {<0.51.0>,{ok,bacon}}
Shell got {<0.51.0>,not_found}
ok
```

As expected, we can take bacon from the fridge because we have put it in there first (along with the milk and baking soda), but the fridge process has no turkey to find when we request some. This is why we get the last `{<0.51.0>,not_found}` message. More interestingly, because of the way mailboxes work, it is guaranteed that even if a thousand people suddenly reached for the last piece of turkey in the fridge at the same time, only one of them could get it.

We Love Messages, But We Keep Them Secret

Something annoying with the previous example is that the programmer who's going to use the fridge must know about the protocol that has been invented for that process. That's a useless burden. A good way to solve this is to abstract messages away with the help of functions dealing with receiving and sending them:

```
store(Pid, Food) ->
    Pid ! {self(), {store, Food}},
    receive
        {Pid, Msg} -> Msg
    end.

take(Pid, Food) ->
    Pid ! {self(), {take, Food}},
    receive
        {Pid, Msg} -> Msg
    end.
```

Now the interaction with the process is much cleaner:

```
9> c(kitchen).
{ok,kitchen}
10> f().
ok
11> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
<0.73.0>
12> kitchen:store(Pid, water).
ok
13> kitchen:take(Pid, water).
{ok,water}
14> kitchen:take(Pid, juice).
not_found
```

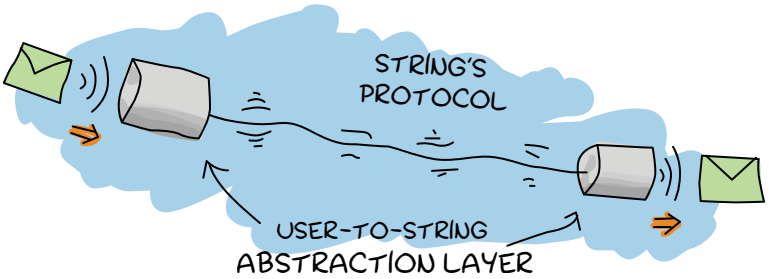
We don't need to care about how the messages work anymore. If you want to send `self()` or a precise atom like `take` or `store`, you just need a pid and to know which functions to call. This hides all of the dirty work and makes it easier to build on the fridge process.

Now let's hide that whole part about needing to spawn a process. We dealt with hiding messages, but then we still expect the user to handle the creation of the process. Let's add the following `start/1` function:

```
start(FoodList) ->
    spawn(?MODULE, fridge2, [FoodList]).
```

Here, `?MODULE` is a macro that returns the current module's name. At first glance, it might not seem like there are any advantages to writing such a function, but there are. The essential advantage is consistency with the calls to `take/2` and `store/2`—everything about the fridge process is now handled

by the kitchen module. If we wanted to add logging when the fridge process is started or start a second process (say a freezer), that would be really easy to do inside our start/1 function. However, if the spawning is left for the user to do through spawn/3, then every place that starts a fridge now needs to add the new calls. That's prone to errors, and errors suck.



Let's see this function put to use:

```
15> f().
ok
16> c(kitchen).
{ok,kitchen}
17> Pid = kitchen:start([rhubarb, dog, hotdog]).
<0.84.0>
18> kitchen:take(Pid, dog).
{ok,dog}
19> kitchen:take(Pid, dog).
not_found
```

Yay! The dog has gotten out of the fridge, and our abstraction is complete!

Time Out

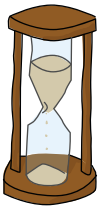
Let's try a test with the help of the command pid(A,B,C), which lets us change the three integers A, B and C into a pid. Here, we'll deliberately feed kitchen:take/2 a fake pid:

```
20> kitchen:take(pid(0,250,0), dog).
```

Whoops. The shell is frozen. This happened because of how take/2 was implemented. To understand what goes on, let's first revisit what happens in the normal case:

1. A message to store food is sent from you (the shell) to the fridge process.
2. Your process switches to receive mode and waits for a new message.
3. The fridge stores the item and sends ok to your process.
4. Your process receives it and moves on with its life.

And here's what happens when the shell freezes:



1. A message to store food is sent from you (the shell) to an unknown process.
2. Your process switches to receive mode and waits for a new message.
3. The unknown process either doesn't exist or doesn't expect such a message and does nothing with it.
4. Your shell process is stuck in receive mode.

That's annoying, especially because there is no error handling possible here. Nothing illegal happened; the program is just waiting—forever, which is a *deadlock*. In general, anything dealing with asynchronous operations (how message passing is done in Erlang) needs a way to give up after a certain period of time if it gets no sign of receiving data. A web browser does this when a page or image takes too long to load, and you do it when someone takes too long to answer the phone or is late for a meeting. Erlang certainly has an appropriate mechanism for handling timeouts, and it's part of the `receive` construct:

```
receive
    Match -> Expression1
after Delay ->
    Expression2
end.
```

The part between `receive` and `after` is exactly the same as what you've seen so far. The `after` part will be triggered if the `Delay` (in milliseconds) has passed without receiving a message that matches the *`Match`* pattern. When this happens, *`Expression2`* is executed.

We'll write two new interface functions, `store2/2` and `take2/2`, which will act exactly like `store/2` and `take/2`, except that they will stop waiting after three seconds:

```
store2(Pid, Food) ->
    Pid ! {self(), {store, Food}},
    receive
        {Pid, Msg} -> Msg
    after 3000 ->
        timeout
    end.

take2(Pid, Food) ->
    Pid ! {self(), {take, Food}},
    receive
        {Pid, Msg} -> Msg
    after 3000 ->
        timeout
    end.
```

When it takes too long, we return `timeout`. This doesn't tell us how to deal with the fact that something took too long, and the message might come back later to haunt us in unexpected ways, but at least we won't be deadlocked if the other process is dead. Something called a *monitor* will help us make this type of code more robust, as you'll see in Chapter 12; but for now, you can just unfreeze the shell by pressing CTRL-G and try the new interface functions:

```
User switch command
--> i
--> s
--> c
Eshell V5.7.5  (abort with ^G)
1> c(kitchen).
{ok,kitchen}
2> kitchen:take2(pid(0,250,0), dog).
timeout
```

And now it works.

NOTE *I said that `after` takes only milliseconds as a value, but it is actually possible to use the atom `infinity`. While this is not useful in many cases (you might as well just remove the `after` clause altogether), it is sometimes used when the programmer can submit the wait time to a function where receiving a result is expected. That way, if the programmer really wants to wait forever, he can.*

Timers have uses other than just giving up after too long. One such use is in the implementation of the `timer:sleep/1` function, which we used in Chapter 10. Here's how it is implemented (let's put it in a new *multiproc.erl* module):

```
sleep(T) ->
    receive
    after T -> ok
    end.
```

In this specific case, no message will ever be matched in the `receive` part of the construct because there is no pattern. Instead, the `after` part of the construct will be called once the delay `T` has passed.

Another special case is when the timeout is at 0:

```
flush() ->
    receive
        _ -> flush()
    after 0 ->
        ok
    end.
```

When that happens, the Erlang VM will try to find a message that fits one of the available patterns. In the preceding case, anything matches. As long as there are messages, the `flush/0` function will recursively call itself until the mailbox is empty. After that, the `after 0 -> ok` part of the code is executed, and the function returns.

Selective Receives

Erlang's "flushing" concept makes it possible to implement a selective receive, which can give a priority to the messages you receive by nesting calls:

```
important() ->
    receive
        {Priority, Message} when Priority > 10 ->
            [Message | important()]
    after 0 ->
        normal()
    end.

normal() ->
    receive
        {_, Message} ->
            [Message | normal()]
    after 0 ->
        []
    end.
```

This function will build a list of all messages, placing those with a priority above 10 first:

```
1> c(multiproc).
{ok,multiproc}
2> self() ! {15, high}, self() ! {7, low}, self() ! {1, low}, self() ! {17, high}.
{17,high}
3> multiproc:important().
[high,high,low,low]
```

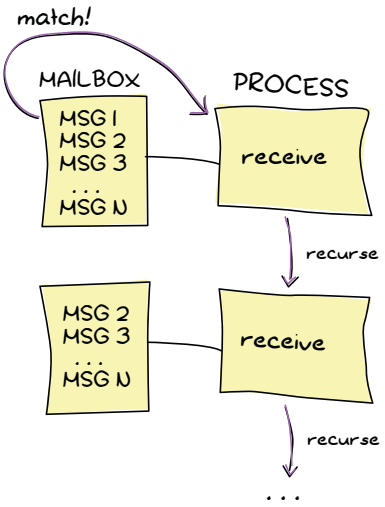
Because we used the `after 0` bit, every message will be obtained until no messages remain, but the process will try to grab all those with a priority above 10 before even considering the other messages, which are accumulated in the `normal/0` call. This practice is called a *selective receive*. If it looks interesting, be aware that it is sometimes unsafe due to the way it's handled by Erlang.

The Pitfalls of Selective Receives

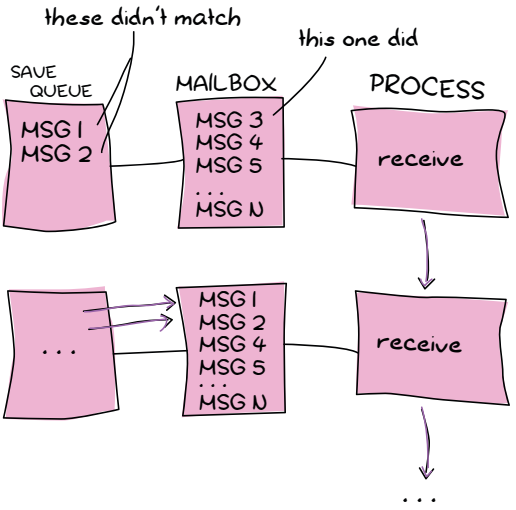
When messages are sent to a process, they're stored in the mailbox until the process reads them and they match a pattern there, even if the process that originally sent them has died since then. The messages are stored in

the order they were received. This means every time you enter a `receive` to match a message, the mailbox is scanned, beginning with the first (and oldest) message received.

That oldest message is then tried against every pattern of the `receive` until one of them matches. When it does, the message is removed from the mailbox, and the code for the process executes normally until the next `receive`. When this next `receive` is executed, the VM will look for the oldest message currently in the mailbox (the one after the one you removed), and so on.



When there is no way to match a given message, it is put in a *save queue*, and the next message is tried. If the second message matches, the first message is put back on top of the mailbox to be retried later.



This lets you care only about the messages that are useful. Ignoring some messages to handle them later in the manner described is the essence of selective receives. While they're useful, the problem with selective receives is that if your process has a lot of messages you never care about, reading useful messages will actually take longer and longer (and the processes will grow in size, too).

In the last illustration of matching messages, imagine we want the 367th message, but the first 366 messages are junk ignored by our code. To get the 367th message, the process needs to try to match those 366 junk messages. Once it has done that, and the messages have all been put in the queue, the 367th message is taken out, and the first 366 are put back on top of the mailbox. The next useful message could be burrowed much deeper and take even longer to be found.

This kind of receive is a frequent cause of performance problems in Erlang. If your application is running slowly and you know there are a lot of messages going around, this could be the cause. If such selective receives are effectively causing a massive slowdown in your code, the first thing to do is to ask yourself why you are getting messages you do not want. Are the messages sent to the right processes? Are the patterns correct? Are the messages formatted incorrectly? Are you using one process where there should be many? Answering one or many of these questions could solve your problem.

CONVERSATIONAL OPTIMIZATIONS

Since R14A, a new optimization has been added to Erlang's compiler. It simplifies selective receives in very specific cases of back-and-forth communications between processes. An example of such a function is `optimized/1` in *multiproc.erl*.

To make this optimization work, a reference must be created (either by using `make_ref()` or by starting a monitor, as described in Chapter 12) in a function and then sent in a message. In the same function, a selective receive is then made. If no message can match unless it contains the same reference, the compiler automatically makes sure the VM will skip messages received before the creation of that reference.

Note that you shouldn't try to coerce your code to fit such optimizations. The Erlang developers only look for patterns that are frequently used and then make them faster. If you write idiomatic code, optimizations should come to you, not the other way around.

More Mailbox Pitfalls

Because of the risks of having useless messages polluting a process's mailbox, Erlang programmers sometimes take a defensive measure against such events. A standard defense might look like this:

```
receive
    Pattern1 -> Expression1;
    Pattern2 -> Expression2;
    Pattern3 -> Expression3;
    ...
    PatternN -> ExpressionN;
    Unexpected ->
        io:format("unexpected message ~p~n", [Unexpected])
end.
```

This makes sure that any message will match at least one clause. The *Unexpected* variable will match anything, take the unexpected message out of the mailbox, and show a warning. Depending on your application, you might want to store the message in some kind of logging facility where you will be able to find information about it later on. If the messages are going to the wrong process, it would be a shame to lose them for good and have a hard time finding out why that other process doesn't receive what it should, given that's pretty much guaranteed to be a bug.

In cases where you do need to work with a priority in your messages and can't use such a catchall clause, a smarter way to handle them is to implement a *min-heap* (see *https://secure.wikimedia.org/wikipedia/en/wiki/Min-heap*) or use the `gb_trees` module (discussed in Chapter 9) and dump every received message in it (make sure to put the priority number first in the key so it gets used for sorting the messages). Then you can just search for the `smallest` or `largest` element in the data structure according to your needs.

In most cases, this technique should let you receive messages with a priority more efficiently than selective receives. However, it could slow you down if most messages you receive have the highest priority possible. As usual, the trick is to profile and measure before optimizing.

Now that we've covered how to hold state in processes, the next step is to do efficient error handling with multiple processes, which is the topic of Chapter 12.

12

ERRORS AND PROCESSES

In most languages, exceptions are managed from within the execution flow of the program, the way we've done it with `try ... catch` in previous examples. The problem with this very common approach is that your regular code needs to handle outstanding errors on every level, or you just delegate the burden of making things safe to the layer above it until you end up having the eternal top-level `try ... catch`, which catches everything but doesn't know anything about it. It's more complex than that in the real world, but that's generally what it looks like. Erlang supports this model too, as you've already seen.

However, Erlang also supports a different level of exception handling that allows you to move the handling of exceptions outside the normal flow of execution of the program, into a different, concurrent process. This usually leads to very clean code, where only the "happy case" is considered. In this chapter, we discuss the basic tools that make this possible: links, monitors, and named processes. We'll also cover some general practices that make the use of these tools more efficient.

Links

A *link* is a specific kind of relationship that can be created between two processes. When that relationship is set up and one of the processes dies from an unexpected throw, error, or exit (see Chapter 7), the other linked process also dies, binding their separate life cycles into a single, related one.

This is a useful concept from the perspective of failing as soon as possible to stop errors. If the process that has an error crashes, but those that depend on it continue to run, then all these depending processes must deal with a dependency disappearing. Letting them die and then restarting the whole group is usually an acceptable alternative. Links let us do exactly this.

To set a link between two processes, Erlang has the primitive function `link/1`, which takes a pid as an argument. When called, the function will create a link between the current process and the one identified by the pid. To get rid of a link, use `unlink/1`.

When one of the linked processes crashes, a special kind of message is sent, with information relative to what happened. No such message is sent if the process dies of natural causes (read: is done running its functions).

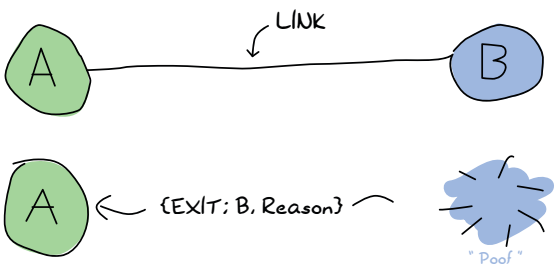
Let's look at how this new function works, as part of the *linkmon.erl* file.

```
myproc() ->
    timer:sleep(5000),
    exit(reason).
```

If you try the following calls (and wait 5 seconds between each `spawn` command), you should see the shell crashing for `reason` only when a link has been set between the two processes:

```
1> c(linkmon).
{ok,linkmon}
2> spawn(fun linkmon:myproc/0).
<0.52.0>
3> link(spawn(fun linkmon:myproc/0)).
true
** exception error: reason
```

Here's a picture of how it works:



However, this {'EXIT', B, Reason} message cannot be caught with a try ... catch as usual. Other mechanisms need to be used to do this, as discussed in "It's a Trap" on page 164.

NOTE *If you wanted to kill another process from the shell, you could use the function exit/2, which makes use of these mechanisms to kill processes. It is called this way: exit(Pid, Reason). Try it if you wish.*

Links are used to establish larger groups of processes that should all die together. Here's an example:

```
chain(0) ->
    receive
        _ -> ok
    after 2000 ->
        exit("chain dies here")
    end;
chain(N) ->
    Pid = spawn(fun() -> chain(N-1) end),
    link(Pid),
    receive
        _ -> ok
    end.
```

This function will take an integer N, start N processes that are linked together. To pass the N-1 argument to the next "chain" process (which calls spawn/1), the example wraps the call inside an anonymous function so it doesn't need arguments anymore. Calling spawn(?MODULE, chain, [N-1]) would have done a similar job.

Here, we'll have many processes linked together, dying as each of their successors exits.

```
4> c(linkmon).
{ok,linkmon}
5> link(spawn(linkmon, chain, [3])).
true
** exception error: "chain dies here"
```

And as you can see, the shell does receive the death signal from some other process. Here's a drawn representation of the spawned processes and links going down:

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] == *dead*
*dead, error message shown*
[shell] <-- restarted
```

After the process running `linkmon:chain(0)` dies, the error is propagated down the chain of links until the shell process itself dies because of it. The crash could have happened in any of the linked processes. Because links are bidirectional, you need only one of them to die for the others to follow suit.

NOTE *Links cannot be stacked. If you call `link/1` fifteen times for the same two processes, only one link will still exist between those processes, and a single call to `unlink/1` will be enough to tear it down.*

Note that `link(spawn(Function))` or `link(spawn(M,F,A))` happen in more than one step. In some cases, it is possible for a process to die before the link has been set up and then provoke unexpected behavior. For this reason, the function `spawn_link/1-3` has been added to the language. It takes the same arguments as `spawn/1-3`, creates a process, and links it as if `link/1` had been there, except it's all done as an atomic operation (the operations are combined as a single one, which can either fail or succeed, but nothing else). This is generally considered safer, and you save a set of parentheses, too.

It's a Trap!

Error propagation across processes is done through a process similar to message passing, but with a special type of message called *signals*. Exit signals are "secret" messages that automatically act on processes, killing them.

I have mentioned many times that in order to be reliable, an application needs to be able to both kill and restart a process quickly. Right now, links can serve to do the killing part. What's missing is the restarting. To restart a process, we first need a way to know that it died. This can be done by adding a layer on top of links (the delicious frosting on the cake) with a concept called *system processes*.

System processes are basically normal processes, except they can convert exit signals to regular messages. This is done by calling `process_flag(trap_exit, true)` in a running process. Nothing speaks as much as an example. Let's just redo the chain example with a system process at the beginning.

```
1> process_flag(trap_exit, true).
true
2> spawn_link(fun() -> linkmon:chain(3) end).
<0.49.0>
3> receive X -> X end.
{'EXIT',<0.49.0>,"chain dies here"}
```

Ah! Now things get interesting. To get back to our drawings, what happens is now more like this:

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] <-- {'EXIT,Pid,"chain dies here"} -- *dead*
[shell] <-- still alive!
```

And this is the mechanism that allows for a quick restart of processes. By writing programs using system processes, it's easy to create a process whose only role is to check if something dies and then restart it whenever it fails. We'll cover more of this in Chapter 13, when we really apply these techniques.

Old Exceptions, New Concepts

Let's return to the exception functions introduced in Chapter 7 and see how they behave around processes that trap exits. First, we'll set the bases to experiment without a system process. We'll look at the results of uncaught throws, errors, and exits in neighboring processes.

Exceptions and Traps

There's a load of reasons why processes ususally die. Let's look at a few of them and what the reasons look like when exits are trapped.

Exception source: `spawn_link(fun() -> ok end)`

Untrapped result: Nothing

Trapped result: `{'EXIT', <0.61.0>, normal}`

The process exited normally, without a problem. Note that this looks a bit like the result of `catch exit(normal)`, except a pid is added to the tuple to identify which process failed.

Exception source: `spawn_link(fun() -> exit(reason) end)`

Untrapped result: `** exception exit: reason`

Trapped result: `{'EXIT', <0.55.0>, reason}`

The process has terminated for a custom reason. If there is no trapped exit, the process crashes. While trapping exits, you get a message.

Exception source: `spawn_link(fun() -> exit(normal) end)`

Untrapped result: Nothing

Trapped result: `{'EXIT', <0.58.0>, normal}`

This successfully emulates a process terminating normally. In some cases, you might want to kill a process as part of the normal flow of a program, without anything exceptional going on. This is the way to do it.

Exception source: `spawn_link(fun() -> 1/0 end)`

Untrapped result:
`Error in process <0.44.0> with exit value: {badarith, [{erlang, '/', [1,0]}]}`

Trapped result: `{'EXIT', <0.52.0>, {badarith, [{erlang, '/', [1,0]}]}}`

The error (`{badarith, Reason}`) is never caught by a `try ... catch` block and bubbles up into an `'EXIT'`. At this point, it behaves exactly the same as `exit(reason)` does, but with a stack trace giving more details about what happened.

Exception source: `spawn_link(fun() -> erlang:error(reason) end)`

Untrapped result:
`Error in process <0.47.0> with exit value: {reason, [{erlang, apply, 2}]}`

Trapped result: `{'EXIT', <0.74.0>, {reason, [{erlang, apply, 2}]}}`

This is pretty much the same as with `1/0`. That's normal—`erlang:error/1` is meant to allow you to do just that.

Exception source: `spawn_link(fun() -> throw(rocks) end)`

Untrapped result:
`Error in process <0.51.0> with exit value: {{nocatch, rocks},`
`[{erlang, apply, 2}]}`

Trapped result: `{'EXIT', <0.79.0>, {{nocatch, rocks}, [{erlang, apply, 2}]}}`

Because the `throw` is never caught by a `try ... catch`, it bubbles up into an error, which in turn bubbles up into an `EXIT`. Without trapping exits, the process fails. While trapping exits, it deals with the error just fine.

And that's about it for usual exceptions. Things are normal, and everything goes fine. Exceptional stuff happens, and processes die and different signals are sent around.

exit/2 Changes Everything

Then there's `exit/2`. This one is the Erlang process equivalent of a gun. It allows a process to kill another one from a distance, safely. The following are some of the possible calls.

Exception source: `exit(self(), normal)`

Untrapped result: `** exception exit: normal`

Trapped result: `{'EXIT', <0.31.0>, normal}`

When not trapping exits, `exit(self(), normal)` acts the same as `exit(normal)`. Otherwise, you receive a message with the same format you would have received by listening to links from foreign processes dying.

Exception source: `exit(spawn_link(fun() -> timer:sleep(50000) end), normal)`

Untrapped result: Nothing

Trapped Result: Nothing

This basically is a call to `exit(Pid, normal)`. This command doesn't do anything useful, because a process cannot be remotely killed with the reason `normal` as an argument.

Exception source: `exit(spawn_link(fun() -> timer:sleep(50000) end), reason)`

Untrapped result: `** exception exit: reason`

Trapped result: `{'EXIT', <0.52.0>, reason}`

This is the foreign process terminating for `reason` itself. It looks the same as if the foreign process called `exit(reason)` on itself.

Exception source: `exit(spawn_link(fun() -> timer:sleep(50000) end), kill)`

Untrapped result: `** exception exit: killed`

Trapped result: `{'EXIT', <0.58.0>, killed}`

Surprisingly, the message gets changed from the dying process to the spawner. The spawner now receives `killed` instead of `kill`. That's because `kill` is a special exit signal, as explained in the next section.

Exception source: `exit(self(), kill)`

Untrapped result: `** exception exit: killed`

Trapped result: `** exception exit: killed`

Oops, look at that. It seems like this one is actually impossible to trap. The following exception doesn't make it easier.

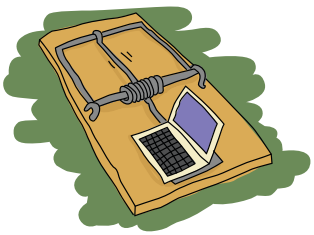
Exception source: `spawn_link(fun() -> exit(kill) end)`

Untrapped result: `** exception exit: killed`

Trapped result: `{'EXIT', <0.67.0>, kill}`

Now that's getting confusing. When another process kills itself with `exit(kill)`, and we don't trap exits, our own process dies with the reason `killed`. However, when we trap exits, things don't happen that way.

Killing Me (Not So) Softly



While you can trap most exit reasons, there are situations where you might want to brutally murder a process. Maybe one of your processes is trapping exits but is also stuck in an infinite loop, never reading any message. The `kill` reason acts as a special signal that

can't be trapped. This ensures any process you terminate with it will really be dead. Usually, `kill` is a bit of a last resort to apply when everything else has failed.

As the `kill` reason can never be trapped, it needs to be changed to `killed` when other processes receive the message. If it weren't changed, every other process linked to it would in turn die for the same `kill` reason, and would in turn kill its neighbors, and so on. A death cascade would ensue.

This also explains why `exit(kill)` looks like `killed` when received from another linked process (the signal is modified so it doesn't cascade), but still looks like `kill` when trapped locally.

If you find this all confusing, don't worry. Many programmers feel the same way. Exit signals are a bit of a strange beast. Luckily, there aren't other special cases than the ones described here. Once you understand these, you can understand most of Erlang's concurrent error management without a problem.

Monitors

Maybe murdering processes isn't what you want. Maybe you don't feel like taking the world down with you once you're gone. Maybe you're more of a stalker. In that case, *monitors* might be what you want, given that they don't kill processes. Monitors are a special type of link, with two differences:

- They are unidirectional.
- You can have many of them between two processes (they *stack* and they have an *identity*).

Monitors are useful when a process wants to know what's going on with a second process, but neither of them is really vital to each other. They're also useful for stacking references that are individually identifiable. This might seem useless at first, but it's great for writing libraries that need to know what's going on with other processes. Why aren't links appropriate for this? Because links do not stack, a library setting up a link and then removing it afterward might be playing with important links unrelated to it. Monitors (and stacking) allow library programmers to separate their use of monitoring from other, unrelated ones. Since each monitor has a unique identity, it is possible to choose which one to listen to or to manipulate.

Links are more of an organizational construct than monitors are. When you design the architecture of your application, you determine which process will do which jobs, and what will depend on what. Some processes will supervise others, some couldn't live without a twin process, and so on. This structure is usually something fixed and known in advance. Links are useful in this case, but should not necessarily be used outside it.

But what happens if you have two or three different libraries that you call and they all need to know whether a process is alive? If you were to use links for this, you would quickly hit a problem whenever you needed to unlink a process. Links aren't stackable, so the moment you unlink one, you unlink them all and mess up all the assumptions made by the other libraries. So you need stackable links, and monitors are your solution, since they can be removed individually. Plus, being unidirectional is handy in libraries because other processes shouldn't need to be aware of those libraries.

So what does a monitor look like? To see, let's set one up. The function is `erlang:monitor/2`, where the first argument is always the atom `process` and the second one is the pid.

```
1> erlang:monitor(process, spawn(fun() -> timer:sleep(500) end)).
#Ref<0.0.0.77>
2> flush().
Shell got {'DOWN',#Ref<0.0.0.77>,process,<0.63.0>,normal}
ok
```

Every time a process you monitor goes down, you will receive such a message, in the form {`'DOWN'`, *`MonitorReference`*, `process`, *`Pid`*, *`Reason`*}. The reference is there to allow you to demonitor the process. Remember that monitors are stackable, so it's possible to take more than one down. References allow you to track each of them in a unique manner. Also note that as with links, there is an atomic function to spawn a process while monitoring it: `spawn_monitor/1-3`.

```
3> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.73.0>,#Ref<0.0.0.100>}
4> erlang:demonitor(Ref).
true
5> Pid ! die.
die
6> flush().
ok
```

In this case, we demonitored the other process before it crashed, so we had no trace of it dying. The function `demonitor/2` also exists and gives a little more information. The second parameter can be a list of options. Only two exist: `info` and `flush`.

```
7> f().
ok
8> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.35.0>,#Ref<0.0.0.35>}
9> Pid ! die.
die
10> erlang:demonitor(Ref, [flush, info]).
false
11> flush().
ok
```

The `info` option tells you if a monitor existed when you tried to remove it. This is why line 10 returned `false`. Using `flush` as an option removes the `DOWN` message from the mailbox if it existed, resulting in `flush()` finding nothing in the current process's mailbox.

Naming Processes

With links and monitors covered, there is another problem still left to be solved: What do we do when we detect that a process we rely on has died? Let's use the following functions of the *linkmon.erl* module:

```
start_critic() ->
    spawn(?MODULE, critic, []).

judge(Pid, Band, Album) ->
    Pid ! {self(), {Band, Album}},
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.

critic() ->
    receive
        {From, {"Rage Against the Turing Machine", "Unit Testify"}} ->
            From ! {self(), "They are great!"};
        {From, {"System of a Downtime", "Memoize"}} ->
            From ! {self(), "They're not Johnny Crash but they're good."};
        {From, {"Johnny Crash", "The Token Ring of Fire"}} ->
            From ! {self(), "Simply incredible."};
        {From, {_Band, _Album}} ->
            From ! {self(), "They are terrible!"}
    end,
    critic().
```

Now we'll just pretend we're going around stores, shopping for music. There are a few albums that sound interesting, but we're never quite sure. We decide to call our friend, the critic.

```
1> c(linkmon).
{ok,linkmon}
2> Critic = linkmon:start_critic().
<0.47.0>
3> linkmon:judge(Critic, "Genesis", "The Lambda Lies Down on Broadway").
"They are terrible!"
```

Because of a solar storm (I'm trying to find something realistic here), the connection is dropped.

```
4> exit(Critic, solar_storm).
true
5> linkmon:judge(Critic, "Genesis", "A trick of the Tail Recursion").
timeout
```

This is annoying. We can no longer get criticism for the albums. To keep the critic alive, we'll write a basic "supervisor" process whose only role is to restart the critic when it goes down.

```
start_critic2() ->
    spawn(?MODULE, restarter, []).

restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    receive
        {'EXIT', Pid, normal} -> % not a crash
            ok;
        {'EXIT', Pid, shutdown} -> % manual termination, not a crash
            ok;
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

Here, the restarter will be its own process. It will in turn start the critic's process, and if it ever dies of an abnormal cause, `restarter/0` will loop and create a new critic. Note that we added a clause for `{'EXIT', Pid, shutdown}` as a way to manually kill the critic if we ever need to.

The problem with our approach is that there is no way to find the pid of the critic, and thus we can't call him to get his opinion. One of the solutions Erlang provides is to give names to processes. The act of giving a name to a process allows you to replace the unpredictable pid with an atom. This atom can then be used exactly as a pid when sending messages.

To give a process a name, use the function `erlang:register(Name,Pid)`. If the process dies, it will automatically lose its name. Alternatively, you can use `unregister/1` to do it manually. You can get a list of all registered processes with `registered/0`, or a more detailed one with the shell command `regs()`. We can rewrite the `restarter/0` function as follows:

```
restarter() ->
    process_flag(trap_exit, true),
    Pid = spawn_link(?MODULE, critic, []),
    register(critic, Pid),
    receive
        {'EXIT', Pid, normal} -> % not a crash
            ok;
        {'EXIT', Pid, shutdown} -> % manual termination, not a crash
            ok;
```

```
        {'EXIT', Pid, _} ->
            restarter()
    end.
```

As you can see, `register/2` will always give our critic the name `critic`, no matter what the pid is. Then we need to remove the need to pass in a pid from the abstraction functions. Let's try this:

```
judge2(Band, Album) ->
    critic ! {self(), {Band, Album}},
    Pid = whereis(critic),
    receive
        {Pid, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.
```

Here, the line `Pid = whereis(critic)` is used to find the critic's pid in order to pattern match against it in the `receive` expression. We want to match with this pid because it makes sure we will match on the right message. (There could be 500 of them in the mailbox as we speak!) This can be the source of a problem though. This code assumes that the critic's pid will remain the same between the first two lines of the function. However, it is completely plausible the following will happen:

```
1. critic ! Message
                          2. critic receives
                          3. critic replies
                          4. critic dies
5. whereis fails
                          6. critic is restarted
7. code crashes
```

This is also a possibility:

```
1. critic ! Message
                          2. critic receives
                          3. critic replies
                          4. critic dies
                          5. critic is restarted
6. whereis picks up
   wrong pid
7. message never matches
```

Things could go wrong in a different process and make another process have problems if we don't do things correctly. In this case, the value of the `critic` atom can be seen from multiple processes. This is known as *shared state*. The problem here is that the value of `critic` can be accessed

and modified by different processes at virtually the same time, resulting in inconsistent information and software errors. The common term for such things is a *race condition*.

Race conditions are particularly dangerous because they depend on the timing of events. In pretty much every concurrent and parallel language out there, this timing depends on unpredictable factors, such as how busy the processor is, where the processes go, and what data is being processed by your program.

> **DON'T DRINK TOO MUCH KOOL-AID**
>
> You might have heard that Erlang is usually free of race conditions or deadlocks and makes parallel code safe. This is true in many circumstances, but only because message passing through a mailbox forces some ordering of events and because the language seriously restricts how much shared state you can have. Generally, you should never assume your code is entirely free of race conditions.
>
> Named processes are only one example of the multiple ways in which parallel code can go wrong.
>
> Other examples include when accessing files on the computer (to modify them) and when updating the same database records from many different processes.

Luckily for us, it's relatively easy to fix the sample code if we don't assume the named process remains the same. Instead, we'll use references (created with `make_ref()`) as unique values to identify messages and make sure we receive the correct messages from the right process. We'll need to rewrite the `critic/0` function into `critic2/0` and `judge/3` into `judge2/2`.

```
judge2(Band, Album) ->
    Ref = make_ref(),
    critic ! {self(), Ref, {Band, Album}},
    receive
        {Ref, Criticism} -> Criticism
    after 2000 ->
        timeout
    end.

critic2() ->
    receive
        {From, Ref, {"Rage Against the Turing Machine", "Unit Testify"}} ->
            From ! {Ref, "They are great!"};
        {From, Ref, {"System of a Downtime", "Memoize"}} ->
            From ! {Ref, "They're not Johnny Crash but they're good."};
        {From, Ref, {"Johnny Crash", "The Token Ring of Fire"}} ->
            From ! {Ref, "Simply incredible."};
        {From, Ref, {_Band, _Album}} ->
            From ! {Ref, "They are terrible!"}
    end,
    critic2().
```

And then change `restarter/0` to fit by making it spawn `critic2/0` rather than `critic/0`.

Now the other functions should keep working fine, and the users won't see a difference. Well, they will because we renamed functions and changed the number of parameters, but they won't know what implementation details were changed and why it was important. All they will see is that their code got simpler and they no longer need to send a pid around function calls. Here's an example:

```
6> c(linkmon).
{ok,linkmon}
7> linkmon:start_critic2().
<0.55.0>
8> linkmon:judge2("The Doors", "Light my Firewall").
"They are terrible!"
9> exit(whereis(critic), kill).
true
10> linkmon:judge2("Rage Against the Turing Machine", "Unit Testify").
"They are great!"
```

And now, even though we killed the critic, a new one instantly came back to solve our problems. That's the usefulness of named processes. Had we tried to call `linkmon:judge/2` without a registered process, a bad argument error would have been thrown by the `!` operator inside the function, making sure that processes that depend on named ones can't run without them.

In Chapter 13, we'll put concurrent programming with Erlang into practice by writing a real application.

NAME WHAT'S WORTH NAMING

Remember that atoms can be used in a limited (though high) number. You should never create dynamic atoms. This means naming processes should be reserved for important services unique to an instance of the VM and processes that should be there for the whole time your application runs.

If you need named processes but they are transient or none of them can be unique to the VM, it may mean they need to be represented as a group instead. Linking and restarting them together if they crash might be the sane option, rather than trying to use dynamic names.

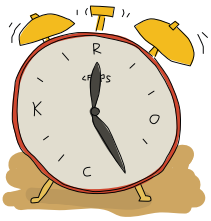
13

DESIGNING A CONCURRENT APPLICATION

All is fine and dandy. You understand the concepts. But then again, all we've had since the beginning of the book were toy examples: calculators, trees, Heathrow to London, and so on. It's time for something more fun and educational. In this chapter, we'll write a small application in concurrent Erlang. The application will be small and line-based, but still useful and moderately extensible.

I'm a somewhat disorganized person. I'm lost with homework, things to do around the apartment, this book, work, meetings, appointments, and so on. I end up having a dozen lists everywhere, listing tasks I still forget to do. I hope that you also sometimes need reminders of what to do (but you don't have a mind that wanders as much as mine does), because we're going to write one of those event reminder applications that prompt you to do stuff and remind you about appointments.

Understanding the Problem



The first step is to know what the hell we're doing. "A reminder app," you say. "Of course," I say. But there's more. How do we plan on interacting with the software? What do we want it to do for us? How do we represent the program with processes? How do we know what messages to send?

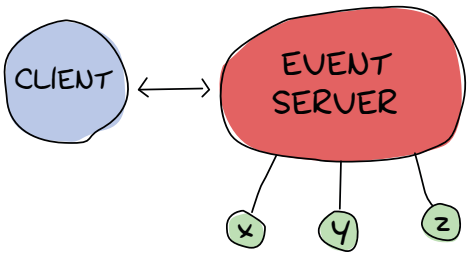
As the quote goes, "Walking on water and developing software from a specification are easy if both are frozen" (Edward V. Berard). So let's set up a spec and stick to it.

Our little piece of software will allow us to do the following:

- Add an event. Events contain a deadline (the time to warn), an event name, and a description.
- Show a warning when the time for our event has come.
- Cancel an event by name.
- Interact with the software via the command line, although it could be extended to allow other means (such as a GUI, web page access, instant messaging software, or e-mail).

This application will not have persistent disk storage. It's not needed to demonstrate the architectural concepts we'll cover in this chapter. But I will show you where it could be inserted if you wanted to add it for a real application, and also point to a few helpful functions. Given we have no persistent storage, we must be able to update the code while it is running.

Here's the structure of the program we'll build, where the client, event server, x, y, and z are all processes:



The event server has these tasks:

- Accept subscriptions from clients
- Forward notifications from event processes to each of the subscribers
- Accept messages to add events (and start the x, y, and z processes needed)
- Accept messages to cancel an event and subsequently kill the event processes

The event server can be terminated by a client, and it can have its code reloaded via the shell.

The client has these tasks:

- Subscribe to the event server and receive notifications as messages
- Ask the server to add an event with all its details
- Ask the server to cancel an event
- Monitor the server (to know if it goes down)
- Shut down the event server if needed

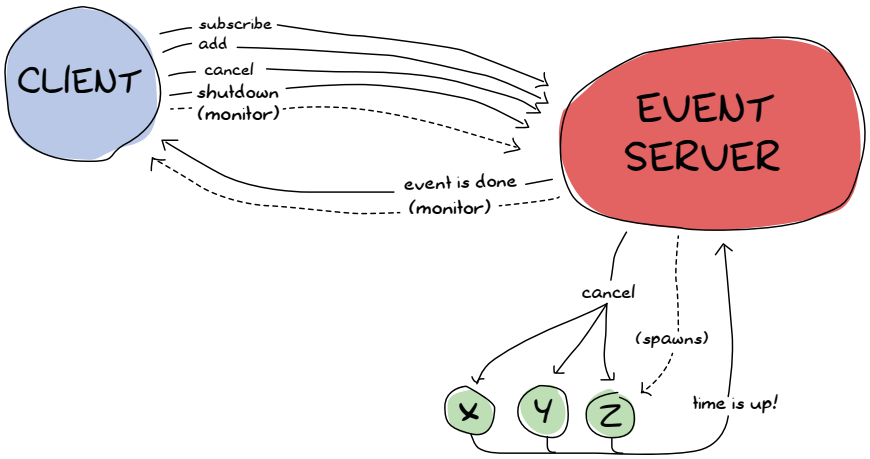
It should be easy to design a bunch of clients all subscribing to the event server. Each of these could potentially be a gateway to the different interaction points (GUI, web page, instant messaging software, email, and so on).

The x, y, and z processes represent a notification waiting to fire (they're basically just timers linked to the event server). They have the following tasks:

- Send a message to the event server when the time is up
- Receive a cancellation message and die

Note that all clients (instant messaging, mail, and others that are not implemented in this example) are notified about all events, and a cancellation is not something to warn the clients about. Here, the software is written for you and me, and it's assumed only one user will run it.

Here's a more complex graph with all the possible messages:

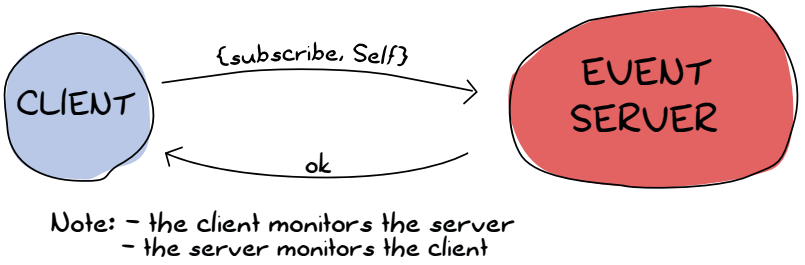


This represents every process we'll have. By drawing all the arrows there and saying they're messages, we've written a high-level protocol, or at least its skeleton.

In a real-world application, using one process per event to be reminded of would likely be overkill and hard to scale. However, since you are going to be the sole user of the application, this is good enough. A different approach could be using functions such as `timer:send_after/2-3` to avoid spawning too many processes.

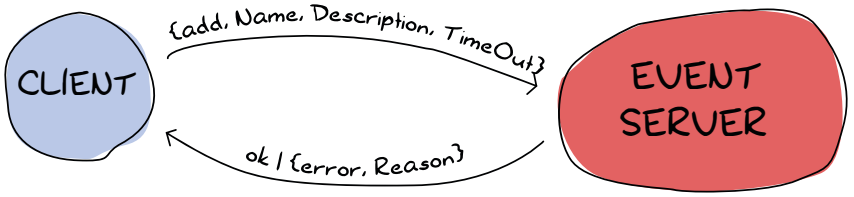
Defining the Protocol

Now that we know what each component needs to do and what it should communicate, it's a good idea to make a list of all messages that will be sent and specify what they will look like. Let's start with the communication between the client and the event server:

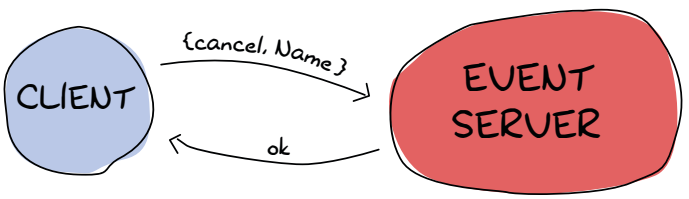


Here, we're using two monitors because there is no obvious dependency between the client and the server. Of course, the client doesn't work without the server, but the server can live without a client. A link could have done the job right here, but because we want our system to be extensible with many clients, we can't assume other clients will all want to crash when the server dies. Nor can we assume the client can really be turned into a system process and trap exits in case the server dies.

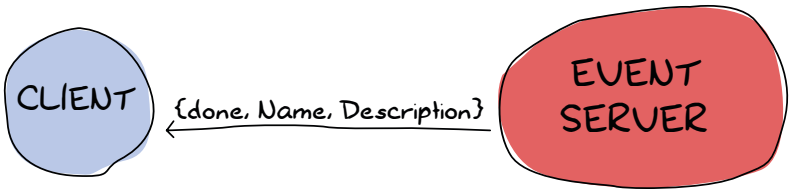
Now to the next message set:



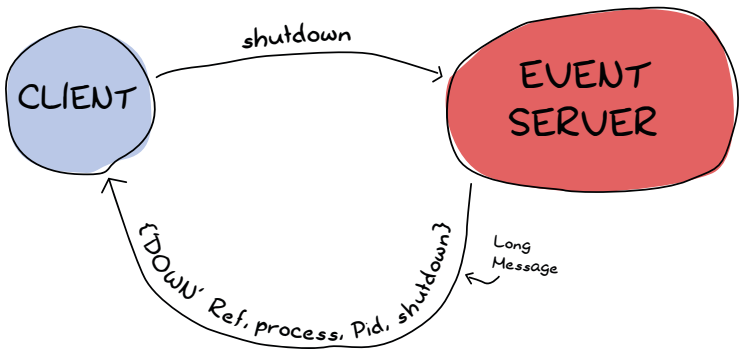
This adds an event to the event server. A confirmation is sent back under the form of the `ok` atom, unless something goes wrong (maybe the `TimeOut` is in the wrong format). The inverse operation, removing events, can be done as follows:



The event server can then later send a notification when the event is due:



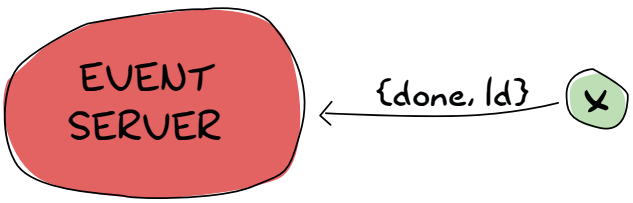
Then we need only the two following special cases for when we want to shut the server down or when it crashes:



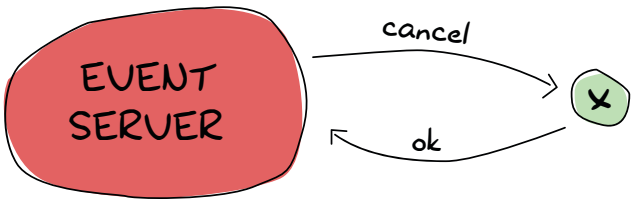
No direct confirmation is sent when the server dies because the monitor will already warn us of that. That's pretty much all that will happen between the client and the event server.

Now we need to deal with the messages between the event server and the event processes themselves. Something to note here before we start is that it would be very useful to have the event server linked to the events. This is because we want all events to die if the server does; they make no sense without it.

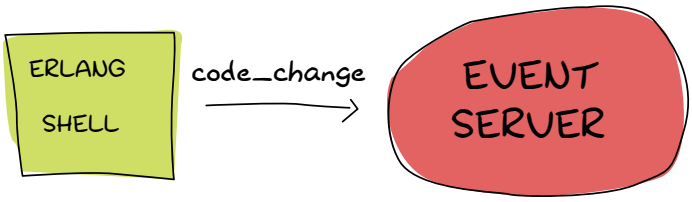
When the event server starts the events, it gives each of them a special identifier (the event's name). Once one of these event's time has come, it needs to send a message saying so:



On the other hand, the event needs to watch for cancel calls from the event server:



One last message will be needed for our protocol—the one that lets us upgrade the server:



No reply is necessary. When we actually program this feature, you'll see this makes sense.

Having both the protocol defined and the general idea of how our process hierarchy will look in place, we can actually start working on the project.

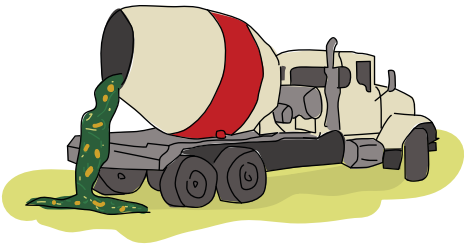
Lay Them Foundations

To begin, we should lay down a standard Erlang directory structure, which looks like this:

```
ebin/
include/
priv/
src/
```

These directories store files as follows:

- The *ebin/* directory is where files will go once they are compiled.
- The *include/* directory is used to store *.hrl* files that are to be included by other applications (the private *.hrl* files are usually kept inside the *src/* directory).
- The *priv/* directory is used for executables that might need to interact with Erlang, such as specific drivers and whatnot. We won't actually use that directory for this project.
- The *src/* directory is where all *.erl* files stay.



In standard Erlang projects, this directory structure can vary a little. A *conf/* directory can be added for specific configuration files, *doc/* for documentation, and *lib/* or *deps/* for third-party libraries required for your application to run. Erlang products on the market often use different directory names, but the four in our structure usually stay the same, given that they're part of the standard OTP practices.

An Event Module

We'll start with the event module because it's the one with the fewest dependencies. We should be able to run it without needing to implement the event server or client functions.

Navigate to the *src/* directory and start an *event.erl* module, which will implement the `x`, `y`, and `z` events for the application.

Before we begin writing any code, I have to mention that the protocol is incomplete. It helps represent what data will be sent from process to process, but not the intricacies: how the addressing works, whether we use references or names, and so on. Most messages will be wrapped in the form {*Pid*, *Ref*, *Message*}, where *Pid* is the sender and *Ref* is a unique message identifier to help determine which reply came from which sender. If we were to send many messages before looking for replies, we would not know which reply went with which message without a reference.

Events and Loops

The core of the processes that will run *event.erl*'s code will be the function `loop/1`, which, if you remember the protocol, will look a bit like the following skeleton:

```
loop(State) ->
    receive
        {Server, Ref, cancel} ->
            ...
    after Delay ->
            ...
    end.
```

This shows the timeout we need to support to announce an event has come to term and the way a server can call for the cancellation of an event. You'll notice a `State` variable in the loop. The `State` variable will need to contain data such as the timeout value (in seconds) and the name of the event (in order to send the message `{done, Id}`). It will also need to know the event server's pid in order to send it notifications.

This is all stuff that's fit to be held in the loop's state. So let's declare a `state` record at the top of the file:

```
-module(event).
-compile(export_all).
-record(state, {server,
                name="",
                to_go=0}).
```

Note that `-compile(export_all).` is used to avoid needing to modify lists of exported functions all the time. Once the development of the module is done, replacing it with a real sequence of `-export([...]).` is recommended.

With this state defined, it should be possible to refine the loop a bit more:

```
loop(S = #state{server=Server}) ->
    receive
        {Server, Ref, cancel} ->
            Server ! {Ref, ok}
    after S#state.to_go*1000 ->
        Server ! {done, S#state.name}
    end.
```

Here, the multiplication by a thousand is to change the `to_go` value from seconds to milliseconds. You could alternatively call `timer:seconds/1`, which converts seconds to milliseconds, to get the same result.

DON'T DRINK TOO MUCH KOOL-AID

Language wart ahead! We need to bind the variable `Server` in the function head because it's used in pattern matching in the receive section. Remember that records are hacks! The expression `S#state.server` is secretly expanded to `element(2, S)`, which isn't a valid pattern to match.

This still works fine for `S#state.to_go` after the `after` part, because that one can be an expression left to be evaluated later.

Now let's test the loop:

```
6> c(event).
{ok,event}
7> rr(event, state).
[state]
8> spawn(event, loop, [#state{server=self(), name="test", to_go=5}]).
<0.60.0>
9> flush().
ok
```

```
10> flush().
Shell got {done,"test"}
ok
11> Pid = spawn(event, loop, [#state{server=self(), name="test", to_go=500}]).
<0.64.0>
12> ReplyRef = make_ref().
#Ref<0.0.0.210>
13> Pid ! {self(), ReplyRef, cancel}.
{<0.50.0>,#Ref<0.0.0.210>,cancel}
14> flush().
Shell got {#Ref<0.0.0.210>,ok}
ok
```

First, we import the record from the event module with `rr(Mod)`. Then we spawn the event loop with the shell as the server (`self()`). This event should fire after 5 seconds. The ninth expression was run after 3 seconds, and the tenth one after 6 seconds. You can see we did receive the `{done, "test"}` message on the second try.

Right after that, we try the cancel feature (with an ample 500 seconds to type it). We created the reference, sent the message, and got a reply with the same reference, so we know the `ok` we received was coming from this process and not any other on the system.

The cancel message is wrapped with a reference, but the `done` message is not, simply because we don't expect it to come from anywhere specific (anyplace will do; we won't match on the receive), nor should we want to reply to it.

Let's try another test. What about an event happening next year?

```
15> spawn(event, loop, [#state{server=self(), name="test", to_go=365*24*60*60}]).
<0.69.0>
16>
=ERROR REPORT==== DD-MM-YYYY::HH:mm:SS ===
Error in process <0.69.0> with exit value: {timeout_value,[{event,loop,1}]}
```

Ouch. It seems like we hit an implementation limit. It turns out Erlang's timeout value is limited to about 50 days in milliseconds. It might not be significant, but I'm showing this error for three reasons:

- It bit me in the ass when writing the module and testing it, halfway through the chapter.
- Erlang is certainly not perfect for every task. What we're seeing here is the consequences of using timers in ways not intended by the implementers.
- It's not really a problem. We can work around it.

The fix we'll apply for this problem is to write a function that splits the timeout value into many parts if turns out to be too long. This will require some support from the `loop/1` function, too. So the way to split the time is to

divide it in equal parts of 49 days (because the limit is about 50), and then put the remainder with all these equal parts. The sum of the list of seconds should now be the original time:

```
%% Because Erlang is limited to about 49 days (49*24*60*60*1000) in
%% milliseconds, the following function is used.
normalize(N) ->
    Limit = 49*24*60*60,
    [N rem Limit | lists:duplicate(N div Limit, Limit)].
```

The function `lists:duplicate/2` will take a given expression as a second argument and reproduce it as many times as the value of the first argument (`[a,a,a] = lists:duplicate(3, a)`). If we were to send `normalize/1` the value `98*24*60*60+4`, it would return `[4,4233600,4233600]`.

The `loop/1` function should now look like this to accommodate the new format:

```
%% Loop uses a list for times in order to go around the ~49 days limit
%% on timeouts.
loop(S = #state{server=Server, to_go=[T|Next]}) ->
    receive
        {Server, Ref, cancel} ->
            Server ! {Ref, ok}
    after T*1000 ->
        if Next =:= [] ->
            Server ! {done, S#state.name};
           Next =/= [] ->
            loop(S#state{to_go=Next})
        end
    end.
```

This takes the first element of the `to_go` list and waits for its whole duration. When this is done, the next element of the timeout list is verified. If it's empty, the timeout is over and the server is notified. Otherwise, the loop keeps going with the rest of the list until it's finished.

You can test the revised loop. It should work as normal, but now support years and years of timeout.

Adding An Interface

It would be very annoying to need to manually call something like `event:normalize(N)` every time an event process is started, especially since our work-around shouldn't be of concern to programmers using our code. The standard way to do this is to instead have an `init` function handle all initialization of data required for the loop function to work well. While we're at it, we'll add the standard `start` and `start_link` functions.

```
start(EventName, Delay) ->
    spawn(?MODULE, init, [self(), EventName, Delay]).
```

```
start_link(EventName, Delay) ->
    spawn_link(?MODULE, init, [self(), EventName, Delay]).

%%% event's innards
init(Server, EventName, Delay) ->
    loop(#state{server=Server,
                name=EventName,
                to_go=normalize(Delay)}).
```

The interface is now much cleaner. Before testing, though, it would be nice to have the only message we can send, cancel, also have its own interface function.

```
cancel(Pid) ->
    %% Monitor in case the process is already dead.
    Ref = erlang:monitor(process, Pid),
    Pid ! {self(), Ref, cancel},
    receive
        {Ref, ok} ->
            erlang:demonitor(Ref, [flush]),
            ok;
        {'DOWN', Ref, process, Pid, _Reason} ->
            ok
    end.
```

Oh, a new trick! Here, we're using a monitor to see if the process is there. If the process is already dead, we avoid useless waiting time and return `ok` as specified in the protocol. If the process replies with the reference, then we know it will soon die, so we remove the reference to avoid receiving them when we no longer care about them. Note that we also supply the `flush` option, which will purge the `DOWN` message if it was sent before we had the time to demonitor.

Let's test these functions:

```
17> c(event).
{ok,event}
18> f().
ok
19> event:start("Event", 0).
<0.103.0>
20> flush().
Shell got {done,"Event"}
ok
21> Pid = event:start("Event", 500).
<0.106.0>
22> event:cancel(Pid).
ok
```

And it works!

The last thing annoying with the event module is that we need to input the time left in seconds. It would be much better if we could use a standard

format such as Erlang's `datetime` (`{{Year, Month, Day}, {Hour, Minute, Second}}`). Just add the following function, which will calculate the difference between the current time on your computer and the delay you inserted.

```
time_to_go(TimeOut={{_,_,_}, {_,_,_}}) ->
    Now = calendar:local_time(),
    ToGo = calendar:datetime_to_gregorian_seconds(TimeOut) -
           calendar:datetime_to_gregorian_seconds(Now),
    Secs = if ToGo > 0  -> ToGo;
              ToGo =< 0 -> 0
    end,
    normalize(Secs).
```

Oh yeah, the `calendar` module has pretty funky function names. This calculates the number of seconds between now and when the event is supposed to fire. If the event is in the past, we instead return 0 so it will notify the server as soon as it can. Now fix the `init` function to call this function instead of `normalize/1`. You can also rename `Delay` variables to say `DateTime` if you want the names to be more descriptive.

```
init(Server, EventName, DateTime) ->
    loop(#state{server=Server,
                name=EventName,
                to_go=time_to_go(DateTime)}).
```

Now that the event module is finished, we can take a break. Start a new event, go drink a pint (half liter) of milk/beer, and come back just in time to see the event message coming in.

The Event Server

Let's deal with the event server. According to the protocol, its skeleton should look a bit like this:

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

loop(State) ->
    receive
        {Pid, MsgRef, {subscribe, Client}} ->
            ...
        {Pid, MsgRef, {add, Name, Description, TimeOut}} ->
            ...
        {Pid, MsgRef, {cancel, Name}} ->
            ...
        {done, Name} ->
            ...
        shutdown ->
            ...
```

```
        {'DOWN', Ref, process, _Pid, _Reason} ->
            ...
        code_change ->
            ...
        Unknown ->
            io:format("Unknown message: ~p~n",[Unknown]),
            loop(State)
    end.
```

You'll notice calls that require replies are wrapped with the same {*Pid*, *Ref*, *Message*} format as earlier.

The server will need to keep two things in its state: a list of subscribing clients and a list of all the event processes it spawned.

The protocol says that when an event is done, the event server should receive {done, *Name*}, but send {done, *Name*, *Description*}. The idea here is to have as little traffic as necessary and have the event processes care only about what is strictly required. So here is the list of clients and list of events:

```
-record(state, {events,    %% list of #event{} records
                clients}). %% list of Pids

-record(event, {name="",
                description="",
                pid,
                timeout={{1970,1,1},{0,0,0}}}).
```

And the loop now has the record definition in its head:

```
loop(S = #state{}) ->
    receive
        ...
    end.
```

It would be nice if both events and clients were orddicts. We're unlikely to have many hundreds of them at once. As you'll recall from Chapter 9, orddicts fit that need very well. We'll write an `init` function to handle this.

```
init() ->
    %% Loading events from a static file could be done here.
    %% You would need to pass an argument to init telling where the
    %% resource to find the events is. Then load it from here.
    %% Another option is to just pass the events straight to the server
    %% through this function.
    loop(#state{events=orddict:new(),
                clients=orddict:new()}).
```

With the skeleton and initialization complete, we'll implement each message one by one.

Handling Messages

The first message is the one about subscriptions. We want to keep a list of all subscribers because when an event is done, we need to notify them. Also, our protocol mentions that we should monitor them. It makes sense because we don't want to hold onto crashed clients and send useless messages for no reason. The code should look like this:

```
{Pid, MsgRef, {subscribe, Client}} ->
    Ref = erlang:monitor(process, Client),
    NewClients = orddict:store(Ref, Client, S#state.clients),
    Pid ! {MsgRef, ok},
    loop(S#state{clients=NewClients});
```

This section of `loop/1` starts a monitor and stores the client information in the orddict under the key `Ref`. The reason for this is simple: The only other time we'll need to fetch the client ID will be if we receive a monitor's `EXIT` message, which will contain the reference (which will let us get rid of the `orddict`'s entry).

The next message we care about is the one where we add events. Now, it is possible to return an error status. The only validation we'll do is to check the timestamps we accept. While it's easy to subscribe to the {{*Year,Month,Day*}, {*Hour,Minute,seconds*}} layout, we need to make sure we don't do things like accept events on February 29 when we're not in a leap year, or on any other date that doesn't exist. Moreover, we don't want to accept impossible date values such as "5 hours, minus 1 minute and 75 seconds." A single function can take care of validating all of that.

The first building block we'll use is the function `calendar:valid_date/1`. As its name says, this function checks if the date is valid. Sadly, the weirdness of the `calendar` module doesn't stop at funky names; there is actually no function to confirm that {H,M,S} has valid values. We'll need to implement that one, too, following the funky naming scheme.

```
valid_datetime({Date,Time}) ->
    try
        calendar:valid_date(Date) andalso valid_time(Time)
    catch
        error:function_clause -> %% not in {{D,M,Y},{H,Min,S}} format
            false
    end;
valid_datetime(_) ->
    false.

valid_time({H,M,S}) -> valid_time(H,M,S).
valid_time(H,M,S) when H >= 0, H < 24,
                       M >= 0, M < 60,
                       S >= 0, S < 60 -> true;
valid_time(_,_,_) -> false.
```

The valid_datetime/1 function can now be used in the part where we try to add the message.

```
{Pid, MsgRef, {add, Name, Description, TimeOut}} ->
    case valid_datetime(TimeOut) of
        true ->
            EventPid = event:start_link(Name, TimeOut),
            NewEvents = orddict:store(Name,
                                      #event{name=Name,
                                             description=Description,
                                             pid=EventPid,
                                             timeout=TimeOut},
                                      S#state.events),
            Pid ! {MsgRef, ok},
            loop(S#state{events=NewEvents});
        false ->
            Pid ! {MsgRef, {error, bad_timeout}},
            loop(S)
    end;
```

If the time is valid, we spawn a new event process, and then store its data in the event server's state before sending a confirmation to the caller. If the timeout is wrong, we notify the client, rather than having the error pass silently or crashing the server. Additional checks could be added for name clashes or other restrictions. (Just remember to update the protocol documentation!)

The next message defined in our protocol is the one where we cancel an event. Canceling an event never fails on the client side, so the code is simpler there. Just check whether the event is in the process's state record. If it is, use the event:cancel/1 function we defined to kill it and send ok. If it's not found, tell the user everything went okay anyway—the event is not running, and that's what the user wanted.

```
{Pid, MsgRef, {cancel, Name}} ->
    Events = case orddict:find(Name, S#state.events) of
                 {ok, E} ->
                     event:cancel(E#event.pid),
                     orddict:erase(Name, S#state.events);
                 error ->
                     S#state.events
             end,
    Pid ! {MsgRef, ok},
    loop(S#state{events=Events});
```

So now all voluntary interaction coming from the client to the event server is covered. Let's deal with the stuff that's going between the server and the events themselves. There are two messages to handle: canceling

the events (which is done) and the events timing out. That message is simply {done, Name}:

```
{done, Name} ->
     case orddict:find(Name, S#state.events) of
        {ok, E} ->
              send_to_clients({done, E#event.name, E#event.description},
             S#state.clients),
             NewEvents = orddict:erase(Name, S#state.events),
             loop(S#state{events=NewEvents});
        error ->
            %% This may happen if we cancel an event and
            %% it fires at the same time.
           loop(S)
     end;
```

The function send_to_clients/2 does as its name says and is defined as follows:

```
send_to_clients(Msg, ClientDict) ->
    orddict:map(fun(_Ref, Pid) -> Pid ! Msg end, ClientDict).
```

That should be it for most of the loop code. What's left is the handling of different status messages: clients going down, shutdown, code upgrades, and so on. Here they come:

```
shutdown ->
    exit(shutdown);
{'DOWN', Ref, process, _Pid, _Reason} ->
    loop(S#state{clients=orddict:erase(Ref, S#state.clients)});
code_change ->
    ?MODULE:loop(S);
Unknown ->
    io:format("Unknown message: ~p~n",[Unknown]),
    loop(S)
```

The first case (shutdown) is pretty explicit. You get the kill message; let the process die. If you wanted to save state to disk, that could be a possible place to do it. If you wanted safer save/exit semantics, this could be implemented on every add, cancel, or done message. Loading events from disk could then be done in the init function, spawning them as they come.

The 'DOWN' message's actions are also simple enough. It means a client died, so we remove it from the client list in the state.

Unknown messages will just be shown with io:format/2 for debugging purposes, although a real production application would likely use a dedicated logging module. Otherwise, all that useful information would be wasted in output that no one ever looks for in production.

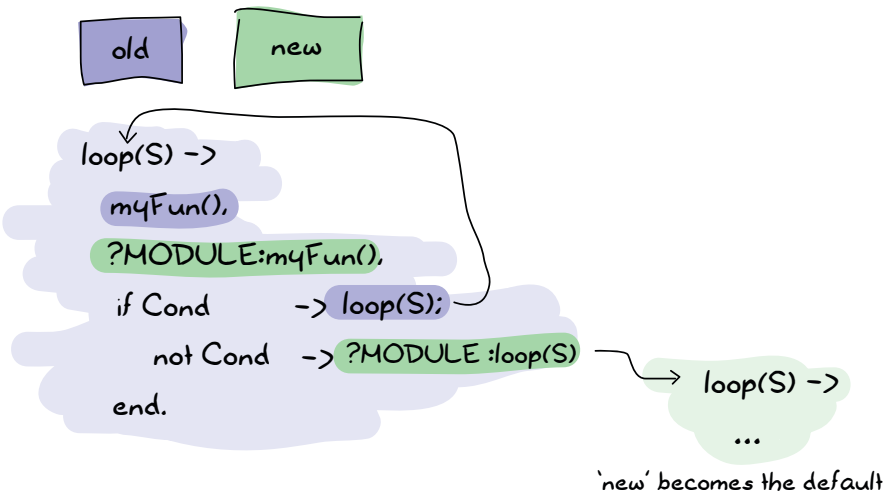
Next comes the code change message. This one is interesting enough to have its own section.

Hot Code Loving

In order to do hot code loading, Erlang has the *code server*. The code server is basically a VM process in charge of an ETS table (an in-memory database table, native to the VM, discussed later in Chapter 25). The code server can hold two versions of a single module in memory, and both versions can run at once. A new version of a module is loaded automatically when compiling it with `c(Module)`, loading with `l(Module)`, or loading it with one of the many functions of the `code` module, which you can read about in the Erlang documentation.

An important concept to understand is that Erlang has both local and external calls. Local calls are those function calls you can make with functions that might not be exported. They have the format *`Name(Args)`*. An external call can be done only with exported functions and has the form *`Module:Function(Args)`*. The precise name for an external call is *fully qualified call*.

When there are two versions of a module loaded in the VM, all local calls are done through the currently running version in a process. However, fully qualified calls are *always* done on the newest version of the code available in the code server. Then, if local calls are made from within the fully qualified one, they are in the new version of the code.



Given that every process/actor in Erlang needs to do a recursive call in order to change state, it is possible to load entirely new versions of an actor by having an external recursive call.

NOTE *If you load a third version of a module while a process still runs with the first one, that process will be killed by the VM, which assumes it was an orphan process without a supervisor or a way to upgrade itself. If no one runs the oldest version, it is simply dropped and the newest ones are kept instead.*

There are ways to bind your code to a system module that will send messages whenever a new version of a module is loaded. By doing this, you can trigger a module reload only when receiving such a message, and always do it with a code upgrade function, say `MyModule:Upgrade(CurrentState)`, which will then be able to transform the state data structure according to the new version's specification. This "subscription" handling is done automatically by the OTP framework, which we'll start studying in Chapter 14. For the reminder application, we won't use the code server and will instead use a custom `code_change` message from the shell, doing very basic reloading. That's pretty much all you need to know to do hot code loading. Nevertheless, here's a more generic example:

```
-module(hotload).
-export([server/1, upgrade/1]).

server(State) ->
    receive
        update ->
            NewState = ?MODULE:upgrade(State),
            ?MODULE:server(NewState);  %% Loop in the new version of the module.
        SomeMessage ->
            %% Do something here.
            server(State)  %% Stay in the same version no matter what.
    end.

upgrade(OldState) ->
%% Transform and return the state here.
```

As you can see, our `?MODULE:loop(S)` fits this pattern.

I Said, Hide Your Messages

Hide messages! If you expect people to build on your code and processes, you must hide the messages in interface functions. Here's what we used for the `evserv` module:

```
start() ->
    register(?MODULE, Pid=spawn(?MODULE, init, [])),
    Pid.

start_link() ->
    register(?MODULE, Pid=spawn_link(?MODULE, init, [])),
    Pid.

terminate() ->
    ?MODULE ! shutdown.
```

We registered the server module because, for now, we should have only one running at a time. If you were to expand the reminder application to support many users, it would be a decent idea to instead register the names

with the `global` module, and it would be even better to use the `gproc` library. For the sake of this example app, what we have here will be enough.

NOTE *The `gproc` library is a process dictionary for Erlang, which provides a number of useful features beyond what the built-in dictionary has, such as he use of any term as an alias, multiple names for a process, waiting for registration of other processes, atomic name giveaway, and counters. It's available from* http://github.com/uwiger/gproc.

The first message we wrote is the next we should abstract away: how to subscribe. The little protocol or specification we wrote earlier called for a monitor, so this one is added there. At any point, if the reference returned by the subscribe message is in a `DOWN` message, the client will know the server has gone down.

```
subscribe(Pid) ->
    Ref = erlang:monitor(process, whereis(?MODULE)),
    ?MODULE ! {self(), Ref, {subscribe, Pid}},
    receive
        {Ref, ok} ->
            {ok, Ref};
        {'DOWN', Ref, process, _Pid, Reason} ->
            {error, Reason}
    after 5000 ->
        {error, timeout}
    end.
```

The next message to abstract away is the event adding:

```
add_event(Name, Description, TimeOut) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
    receive
        {Ref, Msg} -> Msg
    after 5000 ->
        {error, timeout}
    end.
```

Note that we forward the `{error, bad_timeout}` message that could be received to the client. We could have also decided to crash the client by raising `erlang:error(bad_timeout)`. Whether crashing the client or forwarding the error message is the thing to do is still debated in the community. Here's the alternative crashing function:

```
add_event2(Name, Description, TimeOut) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {add, Name, Description, TimeOut}},
    receive
        {Ref, {error, Reason}} -> erlang:error(Reason);
        {Ref, Msg} -> Msg
    after 5000 ->
```

```
        {error, timeout}
    end.
```

Then there's event cancellation, which just takes a name:

```
cancel(Name) ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, {cancel, Name}},
    receive
        {Ref, ok} -> ok
    after 5000 ->
        {error, timeout}
    end.
```

Last of all is a small nicety provided for the client—a function used to accumulate all messages during a given period of time. If messages are found, they're all taken, and the function returns as soon as possible.

```
listen(Delay) ->
    receive
        M = {done, _Name, _Description} ->
            [M | listen(0)]
    after Delay*1000 ->
        []
    end.
```

This is mostly useful when working with applications where the client polls for updates, whereas applications that are always listening can use a push-based mechanism, and thus would not need such a function.

A Test Drive

You should now be able to compile the application and give it a test run. To make things a bit simpler, we'll write a specific Erlang makefile to build the project. Open a file named *Emakefile* and put it in the project's base directory. The file contains Erlang terms and gives the Erlang compiler the recipe to cook wonderful and crispy *.beam* files.

```
{'src/*', [debug_info,
           {i, "src"},
           {i, "include"},
           {outdir, "ebin"}]}.
```

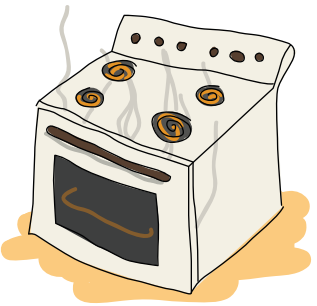
This tells the compiler to add `debug_info` to the files (this is rarely an option you want to give up), to look for header files in the *src/* and *include/* directories to help compile modules in *src/*, and to output them in *ebin/*.

Go to your command line and run `erl -make` from the project's base directory, and the files should all be compiled and put inside the *ebin/* directory for you. Start the Erlang shell by enterng `erl -pa ebin/`. The

`-pa directory` option tells the Erlang VM to add that path to the places it can look for modules.

Another option is to start the shell as usual and call `make:all([load])`. This will look for a file named *Emakefile* in the current directory, recompile it (if it changed), and load the new files.

You should now be able to track thousands of events. Try it out.



```
1> evserv:start().
<0.34.0>
2> evserv:subscribe(self()).
{ok,#Ref<0.0.0.31>}
3> evserv:add_event("Hey there", "test", FutureDateTime).
ok
4> evserv:listen(5).
[]
5> evserv:cancel("Hey there").
ok
6> evserv:add_event("Hey there2", "test", NextMinuteDateTime).
ok
7> evserv:listen(2000).
[{done,"Hey there2","test"}]
```

This works nicely. Writing any client should now be simple enough, given the few basic interface functions we have created.

Adding Supervision

In order to make our example a more stable application, we should write a "restarter," as we did in Chapter 12. Open a file named *sup.erl* where our supervisor will be:

```
-module(sup).
-export([start/2, start_link/2, init/1, loop/1]).

start(Mod,Args) ->
    spawn(?MODULE, init, [{Mod, Args}]).

start_link(Mod,Args) ->
    spawn_link(?MODULE, init, [{Mod, Args}]).

init({Mod,Args}) ->
    process_flag(trap_exit, true),
    loop({Mod,start_link,Args}).

loop({M,F,A}) ->
    Pid = apply(M,F,A),
    receive
```

```
        {'EXIT', _From, shutdown} ->
            exit(shutdown); % will kill the child too
        {'EXIT', Pid, Reason} ->
            io:format("Process ~p exited for reason ~p~n",[Pid,Reason]),
            loop({M,F,A})
    end.
```

This is somewhat similar to the restarter from Chapter 12, although this one is a tad more generic. It can take any module, as long as it has a `start_link` function. It will restart the process it watches indefinitely, unless the supervisor itself is terminated with a shutdown exit signal. Here it is in use:

```
1> c(evserv), c(sup).
{ok,sup}
2> SupPid = sup:start(evserv, []).
<0.43.0>
3> whereis(evserv).
<0.44.0>
4> exit(whereis(evserv), die).
true
Process <0.44.0> exited for reason die
5> exit(whereis(evserv), die).
Process <0.48.0> exited for reason die
true
6> exit(SupPid, shutdown).
true
7> whereis(evserv).
undefined
```

As you can see, killing the supervisor will also kill its child.

NOTE *We'll explore much more advanced and flexible supervisors in Chapter 18. Those are the ones people are thinking of when they mention supervision trees. The supervisor demonstrated here is only the most basic form that exists and is not exactly fit for production environments compared to the real thing.*

Namespaces (or Lack Thereof)

Because Erlang has a flat module structure (there is no hierarchy), some applications may have naming conflicts among their modules. One example of this is the frequently used `user` module that almost every project attempts to define at least once. This clashes with the `user` module shipped with Erlang. You can test for any clashes with the function `code:clash/0`.

Because of the potential for conflicts, the common pattern is to prefix every module name with the name of your project. In this case, our reminder application's modules should be renamed to `reminder_evserv`, `reminder_sup`, and `reminder_event`.

Some programmers then decide to add a module, named after the application itself, which wraps common calls that programmers could make when using their own application. Examples of calls could be functions such as starting the application with a supervisor, subscribing to the server, and adding and canceling events. It's important to be aware of other namespaces, too, such as registered names that must not clash, database tables, and so on.

That's pretty much it for a very basic concurrent Erlang application. This one showed we could have a bunch of concurrent processes without thinking too hard about it: supervisors, clients, servers, processes used as timers (and we could have thousands of them), and so on. There's no need to synchronize them, no locks, and no real main loop. Message passing has made it simple to compartmentalize our application into a few modules with separated concerns and tasks.

The basic calls inside *evserv.erl* could now be used to construct clients that could interact with the event server from somewhere outside the Erlang VM and make the program truly useful.

Before doing that, though, I suggest you read up on the OTP framework. The next few chapters will cover some of its building blocks, which allow for much more robust and elegant applications. A huge part of Erlang's power comes from using the OTP framework. It's a carefully crafted and well-engineered tool that any self-respecting Erlang programmer must know.